HONEYWELL

MULTICS SUBROUTINES AND I/O MODULES

SOFTWARE

MULTICS SUBROUTINES AND I/O MODULES

SUBJECT

Description of Multics Subroutines and I/O Modules

SPECIAL INSTRUCTIONS

This publication supersedes the previous edition of the manual, Order No. AG93-04, dated February 1983, and its addendum AG93-04A, dated December 1983.

Change bars in the margins indicate technical changes to existing material. See "Significant Changes" in the Preface for a list of new subroutines and I/O modules. New subroutines and I/O modules are not identified by change bars.

SOFTWARE SUPPORTED

Multics Software Release 11.0

ORDER NUMBER

AG93-05

Feburary 1985

Honeywell

PREFACE

This document contains a description of Subroutines and I/O Modules provided as part of the standard Multics system. The subroutines can be called from PL/I programs to perform system-provided applications and supervisory functions. The I/O modules can be invoked by calling the iox_ subroutine to interface directly with the Multics I/O System when performing I/O operations.

The information is organized into three sections. Section 1 contains a list of the subroutine repertoire, arranged functionally. Section 2 contains descriptions of the Multics subroutines arranged in alphabetical order. Section 3 contains the descriptions of the I/O modules, also arranged in alphabetical order.

Significant Changes in AG93-05A

The following subroutines are new and do not contain change bars:

condition_	ocu_
datebin_	pascal_util_
enter_abs_request_	print_data_
find_bit_	rcp_
find_char_	reversion_
heap_manager_	translate_bytes_to_hex9_

The following subroutines have been complete rewritten to document additional capabilities; therefore, they contain no change bars:

check_star_name_
match_star_name_

The following subroutine was affected by the C software:

cu_

The following entry points have been moved from the obsolete section to the active section:

cu_$decode_entry_value

The following entry points have been moved from the active section to the obsolete section:

ipc_$create_ev_chn
ipc_$decl_event_call_chn

The following entry points were accidentally left out of the manual for MR11.0:

sort_items_$char
sort_items_$fixed_bin
sort_items_$float_bin
sort_items_$general
sort_items_$varying_char

Extensive information was accidentally left out from the following I/O modules for MR11.0:

tape_ansi_
tape_ibm_

CONTENTS

SECTION 1

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.

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

Storage System, Pathname Manipulation
Storage System, Access Control and Rings of Protection
Storage System, Segment Manipulation
Storage System, Directory Manipulation
Storage System, Links and Search Facility
Storage System, Multisegment Files (MSFS)
Area Management
Clock and Timer Procedures
Command Environment Utility Procedures
Subsystem Environment Utility Procedures
Input/Output System Procedures
Error Handling Procedures
Data Type Conversion Procedures
Condition Mechanism
Object Segment Manipulation
Process Synchronization
Resource Control Package (RCP)
Run Units
Data Management
System Metering
Miscellaneous Procedures

Storage System, Pathname Manipulation

absolute_pathname_
: converts a relative or absolute pathname into an absolute pathname.

change_default_wdir_
: changes the user's current default working directory.

change_wdir_
: changes the user's current working directory.

check_star_name_
: verifies formation of entrynames according to star name rules.

expand_pathname_
: converts a relative or absolute pathname into a directory name and entryname.

find_include_file_
: locates an include file via system include file search files.

find_source_file_
  finds a file given a pathname and an optional suffix.
get_default_wdir_
  returns pathname of user's current default working directory.
get_equal_name_
  constructs target name by substituting from entryname into equal name.
get_pdir_
  returns pathname of process directory.
get_shortest_path_
  shortens pathnames by replacing each directory level with the shortest name on the directory.
get_wdir_
  returns pathname of current working directory.
hcs_$get_link_target
  returns the target pathname of a link.
hcs_$fs_get_path_name
  returns pathname for a segment specified by segment number.
hcs_$star_
  returns storage system type and all names that match entryname according to star name rules.
match_star_name_
  compares entryname with starname.
nd_handler_
  resolves name duplication.
pathname_
  constructs pathnames and archive component pathnames.
search_paths_
  enables users to manipulate search lists and search segments, and to return directory names in which a specified entry can be found.
suffixed_name_
  aids in processing suffixed names.

Storage System, Access Control and Rings of Protection

aim_check_
  determines relationship between two access attributes.
aim_util_
  manipulates AIM access classes and authorizations.
check_gate_access_
  differentiates between not finding the gate and not having access.
compute_common_aim_ceiling_
  computes the maximum athorization or access class which is in common between two Multics systems given the definitions of their AIM attributes.
convert_aim_attributes_
  converts representation of process'/segment's access authorization/class into character string of defined form.
convert_authorization_
  converts an authorization back and forth between its binary and character-string representation.
copy_acl_
  copies the ACL from one segment, MSF, or directory to another.

cross_ring_io_$allow_cross
: allows use of an I/O switch via cross_ring_ attachments from an outer ring.

cu_$level_get
: obtains current ring validation level.

cu_$level_set
: sets current ring validation level.

cv_dir_mode_
: converts a character string containing access modes for directories into a bit string used by the ACL entries.

cv_mode_
: converts a character string containing access modes for segments into a bit string used by the ACL entries.

cv_userid_
: converts a character string containing an abbreviated User_id into one containing all three components.

fs_util_$add_acl_entries
: used to add to the Access Control List of an entry.

fs_util_$add_extended_acl_entries
: used to add to the Extended Access Control List of standard entry.

fs_util_$delete_acl_entries
: deletes a member of an entry's Access Control List.

fs_util_$get_ring_brackets
: returns the ring brackets of an entry.

fs_util_$get_user_access_modes
: returns the user's effective access mode and extended access mode on an entry.

fs_util_$list_acl
: list the components of an entry's Access Control List.

fs_util_$list_extended_acl
: returns the contents of the Extended Access Control List of a standard entry.

fs_util_$replace_acl
: used to replaced Access Control List components for an entry.

fs_util_$replace_extended_acl
: used to replace Extended Access Control List components for a standard entry.

fs_util_$set_ring_brackets
: sets the ring brackets for an entry.

get_authorization_
: returns authorization value of the process.

get_group_id_
: returns access control name of current user.

get_initial_ring_
: obtains a process' initial ring number.

get_max_authorization_
: returns maximum authorization value of the process.

get_privileges_
: returns process' access privileges.

get_process_authorization_
: returns the process's current authorization.

get_ring_
: returns number of current protection ring.

get_system_aim_attributes_
: returns a structure describing the AIM attributes defined on the host system.

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_$add_dir_inacl_entries
: adds specified access modes to initial ACL for directories.

hcs_$add_inacl_entries
: adds specified access modes to initial ACL for segments.

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_$delete_dir_inacl_entries
: deletes specified entries from initial ACL for directories.

hcs_$delete_inacl_entries
: deletes specified entries from initial ACL for segments.

hcs_$fs_get_mode
: returns access control mode for a given segment relative to the current validation level.

hcs_$get_access_class
: returns access class for a directory.

hcs_$get_access_class_seg
: returns access class for a segment.

hcs_$get_dir_ring_brackets
: returns ring brackets for specified subdirectory.

hcs_$get_ring_brackets
: returns ring brackets for specified segment.

hcs_$get_user_effmode
: returns a user's effective access mode to a branch.

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_$list_dir_inacl
: returns all or part of initial ACL for directories.

hcs_$list_inacl
: returns all or part of initial ACL for segments.

hcs_$replace_acl
: replaces one ACL on a segment with another.

hcs_$replace_dir_acl
: replaces one ACL on a directory with another.

hcs_$replace_dir_inacl
: replaces initial ACL with user-provided one for directories.

hcs_$replace_inacl
: replaces initial ACL with user-provided one for segments.

hcs_$set_dir_ring_brackets
: sets ring brackets for specified directory.

hcs_$set_ring_brackets
: sets ring brackets for specified segment.

msf_manager_$acl_add
: adds the specified access modes to the ACL of the multisegment file.

msf_manager_$acl_delete
deletes ACL entries from the ACL of a multisegment file.

msf_manager_$acl_list
returns the access control list (ACL) of a multisegment file.

msf_manager_$acl_replace
replaces the ACL of a multisegment file.

read_allowed_
determines if AIM allows read operations on object given process' authorization and object's access class.

read_write_allowed_
determines if AIM allows read/write operations on object given process' authorization and object's access class.

ring0_get_
supplies name, segment number, and entry point information about ring 0 segments.

ring_zero_peek_
copies information out of an inner-ring segment.

translate_aim_attributes_
translates the AIM attributes in an authorizatin or access class from one system's definition to another system's definition where possible.

write_allowed_
determines if AIM allows write operations on object given process' authorization and object's access class.

Storage System, Segment Manipulation

adjust_bit_count_
sets bit count of a segment to last nonzero character.

archive_
accesses, lists, or obtains information about archive components.

delete_
deletes segments.

dl_handler_
issues queries for situations involving deletion.

dump_segment_
prints a dump formatted the same way as the dump_segment command.

find_include_file_
locates an include file via system include file search rules.

find_source_file_
finds a file given a pathname and an optional suffix.

fs_util_$chname_file
changes the name of an entry.

fs_util_$copy
used to copy an entry.

fs_util_$delentry_file
deletes the name of an entry.

fs_util_$get_bit_count
returns the number of useful bits in an entry.

fs_util_$get_max_length
returns the maximum length setting for an entry.

fs_util_$get_switch
returns the value of a storage system switch for an entry.

fs_util_$get_type
returns the type of a specified entry.

fs_util_$list_switches
: returns a list of switches supported by the entry type.

fs_util_$list_switches_for_type
: returns a list of switches for a particular type of entry.

fs_util_$make_entry
: constructs an entry variable to a specified suffix support subroutine entry for a specified extended entry.

fs_util_$make_entry_for_type
: constructs an entry variable to a specified suffix support subroutine entry for a specified extened entry.

fs_util_$set_bit_count
: sets the number of bits considered useful for an entry.

fs_util_$set_max_length
: sets the maximum length that a particular entry can be.

fs_util_$set_switch
: sets the value of a storage system switch for an entry.

fs_util_$suffix_info
: returns information about an entry's type.

fs_util_$suffix_info_for_type
: returns information about the characteristics of an entry that is of a given type.

hcs_$append_branch
: creates a segment and initializes its ACL.

hcs_$change_bc
: provides an indivisible method of changing the bitcount of a segment.

hcs_$change_bc_seg
: provides an indivisible method of changing the bitcount of a segment.

hcs_$chname_file
: changes the entryname on a specified entry.

hcs_$chname_seg
: changes the entryname on a segment, given a pointer to the segment.

hcs_$create_branch_
: creates a segment, sets a number of attributes.

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_$fs_move_file
: moves contents of one segment to another, given pathnames of the segments.

hcs_$fs_move_seg
: moves contents of one segment to another, given pointers to the segments.

hcs_$get_author
: returns author of segment.

hcs_$get_bc_author
: returns bit count author of a segment.

hcs_$get_max_length
: returns maximum length of segment in words, given directory name and entryname.

hcs_$get_max_length_seg
: returns maximum length of segment in words, given a pointer to a segment.

hcs_$get_safety_sw_seg
returns safety switch value of segment.

hcs_$get_uid_file
returns the unique identifier of a storage system entry.

hcs_$get_uid_seg
returns the unique identifier associated with a segment.

hcs_$initiate
when given a pathname and a reference name, makes known the segment defined by the pathname initiates the given reference name, and increments the count of initiated reference names for the segment.

hcs_$initiate_count
when given a pathname and a reference name, causes the segment defined by the pathname to be made known and the given reference name initiated.

hcs_$make_entry
makes a segment known and returns the value of a specified entry point.

hcs_$make_ptr
makes a segment known and returns a pointer to a specified entry point.

hcs_$make_seg
creates a new segment, makes it known to the process and returns a pointer.

hcs_$set_bc
sets the bit count and bit count author of a segment.

hcs_$set_bc_seg
sets the bit count and bit count author of a segment, given a pointer to the segment.

hcs_$set_entry_bound
sets entry point bound of segment.

hcs_$set_entry_bound_seg
sets entry point bound of segment.

hcs_$set_max_length
sets maximum length of segment.

hcs_$set_max_length_seg
sets maximum length of segment.

hcs_$set_safety_sw
sets safety switch of segment.

hcs_$set_safety_sw_seg
sets safety switch of segment.

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

hcs_$status_long
returns most user-accessible information about an entry.

hcs_$status_minf
returns the bit count and entry type, given the name of a directory and an entry.

hcs_$status_mins
returns the bit count and entry type, given a pointer to the segment.

hcs_$truncate_file
truncates a file or segment to a given length, given a pathname.

hcs_$truncate_seg
truncates a file or segment to a given length, given a pointer.

initiate_file_
contains entry points for making a segment or archive component known with a null reference name.

mhcs_$get_seg_usage
returns the number of page faults taken on a segment since its creation.

nd_handler_
resolves name duplication.

pascal_util_
provides interfaces for establishing and removing an on unit for the current procedure.

qedx_
provides a subroutine interface to the Multics qedx Editor for use by subsystems wishing to edit arbitrary strings of ASCII text.

sort_seg_
provides entry points for sorting segments and character strings.

term_
removes a segment from the address space, unsnapping any subroutine linkage to it.

terminate_file_
performs common operations often necessary after a program finishes using a segment.

tssi_$clean_up_segment
is used by cleanup procedures in the translator.

tssi_$finish_segment
makes a segment unknown, sets bit count and ACL.

tssi_$get_segment
prepares a segment for use as output from the translator.

Storage System, Directory Manipulation

change_default_wdir_
changes the user's current default working directory.

change_wdir_
changes user's current working directory.

copy_dir_
copies a subtree from one point in the hierarchy to another.

delete_
deletes directories.

dl_handler_
issues queries for situations involving deletion.

fs_util_$chname_file
changes the name of an entry.

fs_util_$delentry_file
deletes the name of an entry.

fs_util_$get_bit_count
returns the number of useful bits in an entry.

fs_util_$get_switch
returns the value of a storage system switch for an entry.

fs_util_$get_type
    returns the type of a specified entry.

fs_util_$list_switches
    returns a list of switches supported by the entry type.

fs_util_$list_switches_for_type
    returns a list of switches for a particular type of entry.

fs_util_$make_entry
    constructs an entry variable to a specified suffix support subroutine entry for a specified extended entry.

This page intentionally left blank.

fs_util_$make_entry_for_type
constructs an entry variable to a specified suffix support subroutine entry for a specified extened entry.
fs_util_$set_bit_count
sets the number of bits considered useful for an entry.
fs_util_$set_switch
sets the value of a storage system switch for an entry.
fs_util_$suffix_info
returns information about an entry's type.
fs_util_$suffix_info_for_type
returns information about the characteristics of an entry that is of a given type.
get_default_wdir_
returns pathname of user's current default working directory.
get_pdir_
returns pathname of process directory.
get_wdir_
returns pathname of current working directory.
hcs_$append_branchx
creates a directory and initializes its ACL.
hcs_$append_link
creates a link to a directory.
hcs_$chname_file
changes the entryname on a specified entry.
hcs_$create_branch_
creates a directory, sets a number of attributes.
hcs_$get_author
returns author of a directory.
hcs_$get_bc_author
returns bit count author of a directory.
hcs_$get_uid_file
returns the unique identifier of a storage system entry.
hcs_$get_safety_sw
returns safety switch value of directory.
hcs_$quota_move
moves all or part of quota between two directories.
hcs_$quota_read
returns record quota and accounting information for directory.
hcs_$set_bc
sets multisegment file indicator for a directory.
hcs_$set_safety_sw
sets safety switch of directory.
hcs_$status_
returns various items of information about a specified directory entry.
hcs_$status_long
returns most user-accessible information about an entry.
hcs_$status_minf
returns the bit count and entry type, given the name of a directory and an entry.
list_dir_info_
lists the values in an entry in a directory information segment.
mdc_
provides entrypoints for master directory manipulation.
nd_handler_
resolves name duplication.

sweep_disk_
walks a given subroutine over a subtree of the directory heirarchy.

Storage System, Links and Search Facility

cv_entry_
converts a virtual entry to an entry value.
cv_ptr_
converts a virtual pointer to a pointer value.
delete_
unlinks links.
get_entry_name_
returns associated name of externally defined location or entry point in segment.
get_external_variable_
obtains the location and size of an external variable.
hcs_$append_link
creates a link to a directory.
hcs_$fs_get_refname
returns a reference name for a segment specified by segment number.
hcs_$fs_get_seg_ptr
returns a segment number for a segment specified by a reference name.
hcs_$get_author
returns author of a link.
hcs_$get_search_rules
returns user's current search rules.
hcs_$get_system_search_rules
prints site-defined search rule keywords.
hcs_$initiate_search_rules
allows user to specify search rules.
hcs_$make_entry
makes a segment known and returns the value of a specified entry point.
hcs_$make_ptr
makes a segment known and returns a pointer to a specified entry point.
search_paths_
enables users to manipulate search lists and search segments, and to return directory names in which a specified entry can be found.
set_ext_variable_
allows the caller to look up an external variable by name.

Storage System, Multisegment Files (MSFs)

msf_manager_
provides the means for multisegment files to create, access, and delete components, truncate the file and control access.
tssi_$clean_up_file
is used by cleanup procedures in the translator.
tssi_$finish_file
makes a MSF unknown, sets bit count and ACL.
tssi_$get_file
prepares a MSF for use as output from the translator.

Area Management

area_info_
: returns information about an area.

cu_$grow_stack_frame
: allows caller to allocate temporary storage.

cu_$shrink_stack_frame
: allows caller to deallocate temporary storage.

define_area_
: initializes a region of storage as an area.

get_external_variable_
: obtains the location and size of an external variable.

get_system_free_area_
: returns pointer to system free area for calling ring.

get_temp_segment_
: acquires a single temporary segment in the process directory.

get_temp_segments_
: acquires temporary segments in the process directory.

release_area_
: cleans up an area.

release_temp_segment_
: returns the temporary segment acquired by get_temp_segment_ to the free pool.

release_temp_segments_
: returns temporary segments to the free pool.

set_ext_variable_
: allows the caller to look up an external variable by name.

ssu_$get_area
: obtains an area for use by a subsystem invocation.

ssu_$get_temp_segment
: obtains a temporary segment for use by a subsystem invocation.

ssu_$release_area
: releases an area previously obtained by a call to ssu_$get_area.

ssu_$release_temp_segment
: releases a temporary segment previously acquired by a call to ssu_$get_temp_segment.

translator_temp_
: provides a temporary storage management facility for translators.

value_
: reads and maintains value segments containing name-value pairs across process boundaries.

Clock and Timer Procedures

clock_
: reads the system 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.

cv_fstime_
: returns a Multics clock value.

date_time_
: converts binary time to an ASCII string.

decode_clock_value_
converts a binary time value into an ASCII string.

encode_clock_value_
converts a month, day, year, hour, minute, second, microsecond, and time zone into a system clock reading.

hcs_$get_process_usage
retrieves system resource usage information.

request_id_
used by the absentee facility, I/O daemons, and other queue-driven facilities.

timer_manager_
allows user process interruption after specified amount of CPU or real-time passes.

total_cpu_time_
returns total CPU time used by this process.

virtual_cpu_time_
returns virtual CPU time used by this process.

Command Environment Utility Procedures

abbrev_
subroutine interface to the abbrev command.

ask_
flexible terminal-input facility for numbers and strings.

command_query_
asks questions.

cu_$af_arg_count
returns to caller number of arguments passed by its caller.

cu_$af_arg_count_rel
same as hcs_$af_arg_count but for any argument list.

cu_$af_arg_ptr
returns a pointer to the character-string argument specified by the argument number.

cu_$af_arg_ptr_rel
permits referencing of arguments in any specified argument list.

cu_$af_return_arg
makes available the return argument of an active function.

cu_$af_return_arg_rel
same as hcs_$af_return_arg but for any argument list.

cu_$arg_count
returns number of arguments supplied to the called procedure.

cu_$arg_list_ptr
returns a PL/I pointer to the argument list of its caller.

cu_$arg_ptr
returns a pointer to a specified argument in current argument list.

cu_$arg_ptr_rel
permits referencing of arguments in any specified argument list.

cu_$caller_ptr
allows a routine to obtain a pointer to its caller.

cu_$cp
calls the command processor to execute a command line.

cu_$evaluate_active_string
expands an active string.

cu_$get_command_processor
returns entry value of procedure invoked by cu_$cp.

cu_$get_evaluate_active_string
returns entry value of procedure currently being invoked by call to cu_$evaluate_active_string.

cu_$get_ready_mode
returns value of static ready mode.

cu_$get_ready_procedure
returns entry value of ready procedure.

cu_$ready_proc
used to call ready procedure.

cu_$reset_command_processor
resets procedure invoked by calls to cu_$cp.

cu_$reset_evaluate_active_string
resets procedure invoked by calls to cu_$evaluate_active_string.

cu_$reset_ready_procedure
resets procedure invoked by calls to cu_$ready_proc.

cu_$set_command_processor
allows a subsystem developer to replace the standard command processor with a different procedure.

cu_$set_evaluate_active_string
allows a subsystem developer to replace the standard active string evaluator with a different procedure.

cu_$set_ready_mode
returns value of internal static ready flags.

cu_$set_ready_procedure
allows user to change his ready procedure.

cu_$stack_frame
returns a pointer to the stack frame of its caller.

cu_$stack_frame_size
returns the size in words of the stack frame of the caller.

decode_descriptor_
extracts information from argument descriptors.

find_bit_
performs common bit string search operations.

find_char_
performs the function of the PL/I search and verify builtin functions.

get_process_id_
returns identification of current process.

get_temp_segment_
acquires a single temporary segment in the process directory.

get_temp_segments_
acquires temporary segments in the process directory.

hcs_$history_regs_get
returns current state of per-process history register switch.

hcs_$history_regs_set
controls state of per-process history register switch.

lex_string_
parses ASCII character strings.

read_password_
reads user's password from the terminal.

release_temp_segment_
returns the temporary segment acquired by get_temp_segment_ to the free pool.

release_temp_segments_
: returns temporary segments to the free pool.

requote_string_
: doubles all quotes within a character string and returns the result enclosed in quotes.

search_paths_
: enables users to manipulate search lists and search segments, and to return directory names in which a specified entry can be found.

terminate_process_
: terminates the process in which it is called.

Subsystem Environment Utility Procedures

qedx_
: provides a subroutine interface to the Multics qedx Editor for use by subsystems wishing to edit arbitrary strings of ASCII text.

search_paths_
: enables users to manipulate search lists and search segments, and to return directory names in which a specified entry can be found.

ssu_$abort_line
: prints an error message and aborts the execution of the current subsystem request line.

ssu_$abort_subsystem
: aborts the current invocation of a subsystem, optionally printing an error message.

ssu_$add_dir_info
: adds a new directory to the list of info directories being searched by this subsystem invocation.

ssu_$add_request_table
: adds a new request table to the list of request tables being searched by this subsystem invocation.

ssu_$apply_request_util
: a utility procedure for implementing subsystem "apply" requests.

ssu_$arg_count
: determines how many arguments a subsystem request received.

ssu_$arg_list_ptr
: gets a pointer to a subsystem request's argument list.

ssu_$arg_ptr
: is used by a procedure implementing a subsystem request to access its arguments.

ssu_$create_invocation
: creates an invocation of a subsystem.

ssu_$delete_info_dir
: deletes a directory from the list of info directories being searched.

ssu_$delete_request_table
: deletes a request table from the list of tables being searched.

ssu_$destroy_invocation
: destroys a subsystem invocation.

ssu_$evaluate_active_string
: interprets a single active request string in a subsystem.

ssu_$execute_line
: interprets a single request line.

ssu_$execute_start_up
: executes the current subsystem's start_up exec_com.

ssu_$execute_string
: executes a request string, usually expressed as an in-line constant or character string variable.

ssu_$get_area
: obtains an area for use by a subsystem invocation.

This page intentionally left blank.

ssu_$get_debug_mode
gets the current state of subsystem debug mode.

ssu_$get_default_procedure
gets the default value for a replaceable procedure value.

ssu_$get_default_rp_options
returns the default request processor options for the current subsystem.

ssu_$get_ec_search_list
returns the name of the search list currently being used to find subsystem exec_com files.

ssu_$get_ec_subsystem_ptr
returns the pointer currently used to implement the "referencing_dir" rule in the search list for subsystem exec_coms.

ssu_$get_ec_suffix
returns the suffix currently being used for subsystem exec_com files.

ssu_$get_info_ptr
gets the info_ptr for this subsystem invocation.

ssu_$get_invocation_count
determines the invocation index of the current subsystem invocation.

ssu_$get_level_n_sci_ptr
examines the state of other invocations of the subsystem by returning pointers for the other invocation.

ssu_$get_prev_sci_ptr
examines the state of other invocations of the subsystem by returning pointers for the immediately previous invocation.

ssu_$get_procedure
gets the current value for a replaceable procedure value in the specified subsystem invocation.

ssu_$get_prompt
gets the string currently being used as a prompt.

ssu_$get_prompt_mode
gets the current state of the prompting mode.

ssu_$get_ready_mode
determines the current state of ready processing.

ssu_$get_request_name
determines the primary name of the subsystem request currently being executed.

ssu_$get_reqeust_processor_options
returns the request processor options presently in effect for the current subsystem.

ssu_$get_subsystem_and_request_name
acquires a string identifying the subsystem and the current request.

ssu_$get_subsystem_name
determines the name of the subsystem owning the specified invocation.

ssu_$get_subsystem_version
determines the version number of the subsystem.

ssu_$get_temp_segment
obtains a temporary segment for use by the current subsystem invocation.

ssu_$list_info_dirs
lists the info directories currently in use by this subsystem invocation.

ssu_$list_request_tables
lists the request tables currently in use by this subsystem invocation.

ssu_$listen
implements the subsystem listener.

ssu_$print_blast
prints a "blast" message announcing a new version of the subsystem.

ssu_$print_message
prints informational, warning, or nonfatal error messages.

ssu_$record_usage
makes an entry in the usage segment to record a use of the subsystem.

ssu_$release_area
releases an area previously obtained by a call to ssu_$get_area.

ssu_$release_temp_segment
releases a temporary segment previously obtained by a call to ssu_$get_temp_segment.

ssu_$reset_procedure
resets a replaceable procedure in the current subsystem to its default value.

ssu_$reset_request_processor options
resets the request processor options presently in effect to their default values.

ssu_$return_arg
is used by a subsystem request procedure to determine whether it has been invoked as an active request.

ssu_$set_debug_mode
sets debug mode for the subsystem.

ssu_$set_ec_search_list
sets the name of the search list used to find subsystem exec_com files.

ssu_$set_ec_subsystem_ptr
sets the directory used to implement the "referencing_dir" rule in the search list for subsystem exec_coms.

ssu_$set_ec_suffix
sets the suffix for subsystem exec_com files.

ssu_$set_info_dirs
sets the list of info directories searched by this subsystem invocation.

ssu_$set_info_ptr
sets the info_ptr for this subsystem invocation.

ssu_$set_procedure
sets the current value of a replaceable procedure in this subsystem invocation.

ssu_$set_prompt
sets the prompt string for the subsystem.

ssu_$set_prompt_mode
sets the prompting mode for the subsystem.

ssu_$set_ready_mode
turns ready message processing in the subsystem listener on or off.

ssu_$set_request_processor_options
changes the request processor options presently in effect.

ssu_$set_request_tables
sets the list of request tables searched by the subsystem.

ssu_$standalone_invocation
creates a "standalone" subsystem invocation for use by Multics commands/active functions which can also be used as subsystem requests.

sort_seg_
provides entry points for sorting segments and character strings.

cb_menu_
    allows a COBOL program to use the Multics menu facility (menu_).
cb_window_
    is the basic video interface subroutine used by COBOL to create/destroy/change windows.
convert_dial_message_
    controls dialed terminals.
cross_ring_io_$allow_cross
    allows use of an I/O switch via cross_ring_ attachments from an outer ring.
dial_manager_
    interfaces to the answering service dial facility.
display_file_value_
    outputs information about a file on a user-supplied switch.
dprint_
    adds print, punch or plot requests to the specified queue.
find_partition_
    obtains information about a disk partition located on some mounted storage system disk.
format_document_
    fills and adjusts text.
ft_menu_
    allows a FORTRAN program to use the Multics menu facility (menu_).
ft_window_
    is the basic video interface subroutine to be used by FORTRAN to create/destroy/change windows.
get_line_length_
    returns the line length of an I/O switch.
hcs_$force_write
    writes pages from memory to disk.
hphcs_$read_partition
    reads words of data from a specified disk partition on some mounted physical storage disk.
hphcs_$write_partition
    writes words of data into a specified disk partition on some mounted physical storage-system disk.
ioa_
    produces formatted printed output.
iod_info_
    extracts information from the I/O daemon tables for commands and subroutines submitting I/O daemon requests.
iox_
    interfaces with the Multics I/O system.
menu_
    provides menu display and selection services.
mode_string_
    manipulates mode strings; can parse, analyze, and create them.
phcs_$read_disk_label
    reads the label of a storage-system disk volume.
pl1_io_
    extracts information about PL/I files.
shcs_$set_force_write_limit
    fixes limit on number of pages to be written to disk.

timed_io_
: performs I/O operations and returns an error code if it cannot complete its operation within the time specified.

ttt_info_
: extracts information from the terminal type table (TTT).

vfile_status_
: returns information about a storage system file supported by the vfile_ I/O module.

video_data_
: is a data segment containing information about the video system.

video_utils_
: provides interfaces for activating and de-activating the video system.

window_
: provides a terminal interface to video terminal operations.

Error Handling Procedures

active_fnc_err_
: prints formatted error message and signals active_function_error condition.

com_err_
: prints a standard status message for command errors.

command_query_
: asks questions.

condition_
: establishes a handler for a condition in the calling block activation.

convert_status_code_
: returns short and long status messages for given status code.

cu_$cl
: reenters command level.

cu_$get_cl_intermediary
: returns procedure invoked by cu_$cl.

cu_$reset_cl_intermediary
: resets procedure invoked by calls to cu_$cl.

cu_$set_cl_intermediary
: sets procedure invoked by cu_$cl.

cv_error_
: converts an error name to an error code.

dl_handler_
: issues queries for situations involving deletion.

find_bit_
: performs common bit string search operations.

find_char_
: performs the function of the PL/I search and verify builtin functions.

hcs_$get_page_trace
: retrieves trace of process' page faults from the supervisor.

lex_error_
: generates compiler-style error messages.

nd_handler_
: resolves name duplication.

print_cobol_error_
: prints error messages produced by COBOL programs.

reversion_
: causes the handler currently established for the given condition in the calling block activation to be disestablished.

ssu_$abort_line
: prints an error message and aborts the execution of the current subsystem request line.

ssu_$abort_subsystem
: aborts the current invocation of a subsystem, optionally printing an error message.

sub_err_
: reports errors detected by other subroutines.

Data Type Conversion Procedures

add_bit_offset_
: returns pointer to bit relative to bit referenced by input pointer.

add_char_offset_
: returns pointer to character relative to character referenced by input pointer.

arithmetic_to_ascii_
: formats any arithmetic value.

ascii_to_bcd_
: performs isomorphic (one-to-one reversible) conversion from ASCII to BCD.

ascii_to_ebcdic_
: performs conversion from ASCII to EBCDIC.

assign_
: assigns specified source value to specified target performing required conversion.

bcd_to_ascii
: performs isomorphic (one-to-one reversible) conversion from BCD to ASCII.

bit_offset_
: returns bit offset of pointer.

char_offset_
: returns character offset of pointer.

char_to_numeric_
: converts user-supplied string to a numeric type.

convert_date_to_binary_
: converts ASCII string to binary clock reading.

cv_bin_
: converts binary representation of an integer to 12-character ASCII string.

cv_dec_
: converts an ASCII representation of a decimal integer to fixed bin(35).

cv_dec_check_
: same as cv_dec_ except that a code is returned indicating the possibility of a conversion error.

cv_dir_mode_
: converts a character string containing access modes for directories into a bit string used by the ACL entries.

cv_mode_
: converts a character string containing access modes for segments into a bit string used by the ACL entries.

cv_entry_
: converts a virtual entry to an entry value.

cv_float_
converts an ASCII representation of a floating point number and returns a single precision floating point representatio5

cv_float_double_
converts an ASCII representation of a floating point number and returns a double precision floating point representation.

cv_hex_
converts an ASCII representation of a hexadecimal integer to fixed binary (35).

cv_hex_check_
same as cv_hex_ except that a code is returned indicating the possibility of a conversion error.

cv_oct_
converts an ASCII representation of an octal integer to fixed binary (35) of an octal integer.

cv_oct_check_
same as cv_oct_ except that a code is returned indicating the possibility of a conversion error.

cv_ptr_
converts a virtual pointer to a pointer value.

date_time_
converts a clock reading to an ASCII string.

decode_clock_value_
converts a binary time value into an ASCII string.

ebcdic_to_ascii_
performs conversion from EBCDIC to ASCII.

encode_clock_value_
converts a month, day, year, hour, minute, second, microsecond, and time zone into a system clock reading.

find_bit_
performs common bit string search operations.

find_char_
performs the function of the PL/I search and verify builtin functions.

lex_string_
parses ASCII character strings.

mlr_
moves a character string by copying the characters from left to right.

mrl
moves a character string by copying the characters from right to left.

mvt_
provides for translation of character strings using translations which are not known at compile time.

numeric_to_ascii_
formats a real decimal floating-point number.

numeric_to_ascii_base_
formats a real decimal floating-point number based in any number system from 2 to 16.

parse_channel_name_
parses a character string that is intended to be an IOM channel number.

parse_file_
parses ASCII text into symbols and break characters.

print_data_
formats and prints the output of a PL/I put data statement.

set_bit_offset_
    returns pointer to specified bit in segment referenced by input pointer.
set_char_offset_
    returns pointer to specified character in segment referenced by input pointer.
sort_seg_
    provides entry points for sorting segments and character strings.
translate_bytes_to_hex9_
    translates a bit string to a character string containing the hexadecimal representation of the bits.
unique_bits_
    returns a unique bit string.
unique_chars_
    converts a unique bit string to a unique character string.
valid_decimal_
    checks decimal data for validity.

Condition Mechanism

add_epilogue_handler_
    adds to the list of handlers called when a process or run unit is terminated.
condition_
    establishes a handler for a condition in the calling block activation.
condition_interpreter_
    prints formatted error message for most conditions.
continue_to_signal_
    enables on unit that cannot completely handle condition to tell signalling program to search stack for other on units for condition.
exponent_control_
    provides control over system's behavior in event of computational overflow or underflow.
find_condition_frame_
    returns a pointer to the most recent condition frame.
find_condition_info_
    returns information about condition when signal occurs.
hcs_$get_exponent_control
    returns flag settings that control handling of overflow and underflow conditions.
hcs_$set_exponent_control
    changes flag settings that control handling of overflow and underflow conditions.
heap_manager_$push_heap_level
    creates a new heap level, allocates the heap header and chains the previous heap to the current heap.
heap_manager_$pop_heap_level
    resets the heap to the previous level freeing the old heap and any variables allocated therein.
heap_manager_$get_heap_header
    returns a pointer to the heap header for the specified execution level.
heap_manager_$get_heap_level
    returns the current execution level from the current heap header.
heap_manager_$get_heap_area
    returns a pointer to the heap area for the specified level.

prepare_mc_restart_
: checks machine conditions for restartability, and permits modifications to them for user changes to process execution before condition handler returns.

sct_manager_
: manipulates the System Condition Table; can set a static handler, get a pointer to one, and call one.

signal_
: signals occurrence of given condition.

sus_signal_handler_
: is the static condition handler for the sus_ condition.

unwinder_
: performs nonlocal goto on Multics stack.

Object Segment Manipulation

component_info_
: returns information about a component of a bound segment.

create_data_segment_
: creates a standard object segment from PL/I data.

decode_definition_
: returns information about a definition in the object segment.

get_bound_seg_info_
: supplies structural information about a bound segment.

get_definition_
: returns pointer to specified definition within an object segment.

get_entry_arg_descs_
: returns information about the calling sequence of an entry point.

get_entry_point_dcl_
: returns attributes needed to construct a PL/I declare statement.

object_info_
: prints structural and identifying information extracted from object segment.

stu_
: retrieves information from the runtime symbol table section of an object segment.

translator_info_
: supplies source segment information for use by translators building object segments.

tssi_
: simplifies use of storage system by language translators.

Process Synchronization

create_ips_mask_
: returns a bit string that can be used to disable specified ips interrupts.

get_lock_id_
: returns a 36-bit unique identifier to be used in setting locks.

hcs_$get_ips_mask
: returns the value of the current ips mask.

hcs_$reset_ips_mask
: replaces the entire ips mask with a specified ips mask.

hcs_$set_ips_mask
: replaces the entire ips mask with a specified ips mask.

hcs_$validate_processid
: determines whether a 36-bit quantity is the unique identifier of a process which is currently active on the system.

hcs_$wakeup
: sends interprocess communication wakeup to blocked process over specified event channel.

hphcs_$ips_wakeup
: sends a specified IPS signal to a specified process.

ipc_
: user interface to Multics interprocess communication facility.

set_lock_
: allows multiple processes to synchronize their use of shared data.

Resource Control Package (RCP)

cv_rcp_attributes_
: manipulates RCP resource attribute specifications and descriptions.

interpret_resource_desc_
: displays selected contents of RCP resource description.

resource_control_
: provides interface to Multics resource control facility.

resource_info_
: returns selected information about RCP resource types defined on the system.

Run Units

add_epilogue_handler_
: adds to the list of handlers called when a process or run unit is terminated.

execute_epilogue_
: cleans up language I/O buffers in conjunction with run units.

run_
: sets up special environment for executing programs.

run_$environment_info
: returns information about run environment.

Data Management

before_journal_manager_
: provides the means to manipulate and obtain information about before journals.

file_manager_
: interface between the data storage and retrieval services of data management and Multics file access and control mechanisms.

transaction_manager_
: begins and ends transactions on behalf of users, returns information about transactions, and recovers transactions after system failure.

System Metering

meter_gate_
    returns data about specific gate entries to the caller.
spg_util_
    collects metering information from the Multics supervisor and subtracts it from the previous sample taken.
spg_ring_0_info_
    returns information about the virtual CPU time spend in the three main gates into ring zero.

Miscellaneous Procedures

abbrev_
    subroutine interface to the abbrev command.
get_ec_version_
    returns the version number of an exec_com.
hash_
    maintains a hash table; contains entry points that initialize a hash table and insert, delete, and search for entries in the table.
hash_index_
    computes the value of a hash function.
help_
    locates info segs.
qedx_
    provides a subroutine interface to the Multics qedx Editor for use by subsystems wishing to edit arbitrary strings of ASCII text.
random_
    returns random numbers.
rehash_
    reformats a hash table of the form maintained by hash_ into a different size.
send_mail_
    sends a message and an optional wakeup to a user.
send_message_
    sends an interactive message to be received by the message facility.
set_ext_variable_
    allows the caller to look up an external variable by name.
sort_items_
    provides a general sorting facility.
sort_items_indirect_
    provides a facility for sorting a group of data items.
sort_seg_
    provides entry points for sorting segments and character strings.
sweep_disk_
    walks a given subroutine over a subtree of the directory hierarchy.
system_info_
    provides user with information on system parameters.

teco_get_macro_
called by teco to search for an external macro.

ttt_info_
extracts information from the terminal type table (TTT).

user_info_
returns miscellaneous information about the current user.

value_
reads and maintains value segments containing name-value pairs.

SECTION 2

SUBROUTINE DESCRIPTIONS

This section contains descriptions of the Multics subroutines and functions, presented in alphabetic order. The term "subroutine" in this section refers alike to subroutines and functions, where the difference is not important. The individual descriptions specify for each name whether it represents a subroutine or a function. 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

The "Usage" section contains a sample declare statement and a sample call (or asign) statement expressed in PL/I notation. It is to be assumed, unless otherwise specified, that arguments are required.

ARGUMENTS

Arguments described under the "Usage" heading are explained in this section. 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 to introduce specific subject matter. Additional headings used include "Examples" (for sample code fragments) and "Structure" (used to define the structure of an include file).

STATUS CODES

The standard status codes returned by the subroutines are further identified, when appropriate, as either storage system or I/O system. Certain codes have been included in the individual subroutine description if they have a special meaning in the context of that subroutine; no attempt is made to show all of the possible error codes.

A list of system status codes and their meanings appears in the Programmer's Reference Manual. The reader should not assume that the code(s) given in a particular subroutine description are the only ones that can be returned. Since a code of 0 means that the given operation was executed successfully, this value is omitted from the list of possible codes under "code" in the "where" list.

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 zero.

Name: abbrev_

The abbrev_ subroutine provides a means of expanding abbreviations in command lines and changing data in and extracting data from the profile segments used by the abbrev command. All of the features of the command itself are available and a simple expand entry point is provided for returning expanded command lines.

The main entry point is used to expand and execute a command line. The command line can be an abbrev request line, as recognized by the abbrev command documented in the Commands Manual. An abbrev request line can be used to add and delete abbreviations and change the modes of operation of abbrev. The abbrev command need not be invoked in the process before the abbrev_ subroutine can be called.

USAGE

```
declare abbrev_ entry (ptr, fixed bin(21), fixed bin(35));

call abbrev_ (line_ptr, line_len, code);
```

ARGUMENTS

line_ptr
: is a pointer to a character string to be interpreted as a command line or an abbrev request line. (Input)

line_len
: is the number of characters in the input line. (Input)

code
: is a standard status code returned by the command processor. (Output)

Entry: abbrev_$expanded_line

This entry point returns an expanded version of an input string. See the description of the abbrev command for a discussion of abbrev expansion.

USAGE

```
declare abbrev_$expanded_line entry (ptr, fixed bin(21), ptr,
      fixed bin(21), ptr, fixed bin(35));

call abbrev_$expanded_line (in_ptr, in_len, space_ptr, space_len,
      out_ptr, out_len);
```

ARGUMENTS

in_ptr
is a pointer to a character string to be expanded. (Input)

in_len
is the number of characters in the input string. (Input)

space_ptr
is a pointer to a work space where the expanded character string can be placed. (Input)

space_len
is the number of characters available in the work space. (Input)

out_ptr
points to the expanded string. (Output)

out_len
is the number of characters in the expanded string. (Output)

NOTES

If the length of the expanded string exceeds the length of the work space provided, the expanded line is allocated in the system free area (see the get_system_free_area_ subroutine), It is the user's responsibility to free this storage when it is no longer needed.

The space_ptr pointer should not point to the same string as in_ptr since expansion is done directly into the work space.

Entry: abbrev_$set_cp

This entry point sets up a different command processor to be called by the abbrev_ subroutine after a command line is expanded. Its argument is an entry. If the first pointer in the entry is null, the command processor to be called is command_processor_.

USAGE

```
declare abbrev_$set_cp entry (entry);

call abbrev_$set_cp (cp_entry);
```

ARGUMENTS

cp_entry
is a command processor entry point.

EXAMPLES

The code:

```
chars = ".a abl " || char_string;
call abbrev_ (addr (chars), length (chars), code);
```

sets up abl as an abbreviation for the character string stored in chars.

The code:

```
chars = "delete foo; logout";
call abbrev_ (addr (chars), length (chars), code);
```

calls the command processor with the string arrived at by expanding the command line:

```
delete foo; logout
```

That is, if foo is an abbreviation for *.pl1, the command processor is given the line:

```
delete *.pl1; logout
```

to be executed.

The code:

```
chars = some_string;
cp    = addr (chars);
xcp   = addr (xchars);
call abbrev_$expanded_line (cp, length (chars),
   xcp, length (xchars), out_ptr, out_len);
```

copies some_string into chars and leaves the expanded version in xchars, unless the length of the expanded version is greater than length(chars). In that case the expanded version is in allocated storage. In either case, out_ptr points to the expanded version and out_len is its length.

Name: absolute_pathname_

The absolute_pathname_ subroutine is used to convert a relative or absolute pathname into a full absolute pathname. This entry does not accept the syntax for specifying archive component pathnames; if one is supplied, an error code is returned. See the information on naming conventions in the Programmer's Reference Manual for details.

USAGE

```
dcl absolute_pathname_ entry (char(*), char(*), fixed bin (35));

call absolute_pathname_ (pathname, full_pathname, code);
```

ARGUMENTS

pathname
: is the relative or absolute pathname to be expanded. (Input)

full_pathname
: is the full, absolute pathname derived from the input pathname. (Output)

code
: is a standard system error code. (Output) If an error has occurred, it can have one of the following values:

error_table_$lesserr
: too many less-than ("<") characters in pathname.

error_table_$badpath
: invalid syntax in pathname.

error_table_$pathlong
: the expanded pathname is longer than 168 characters.

error_table_$entlong
: the entryname portion of the expanded pathname is longer than 32 characters.

error_table_$archive_pathname
: the input pathname specified an archive component; this feature is only supported by the expand_pathname_$component and expand_pathname_$component_add_suffix entrypoints.

error_table_$no_wdir
: a relative pathname is specified, but no working directory is in force for the process.

Entry: absolute_pathname_$add_suffix

This entrypoint expands a relative or absolute pathname into a full, absolute pathname, adding a suffix to the entryname if that suffix is not already present.

USAGE

```
dcl absolute_pathname_$add_suffix entry (char (*), char (*), char (*),
      fixed bin (35));

call absolute_pathname_$add_suffix (pathname, suffix, full_pathname,
      code);
```

ARGUMENTS

pathname
is the relative or absolute pathname to be expanded. (Input)

suffix
is the suffix to be added to the entryname portion of the pathname. (Input) The period separating the entryname and the suffix must not be included. If a null string is supplied, no suffix is added.

full_pathname
is the full, absolute pathname derived from the input pathname. (Output)

code
is a standard system error code. (Output) It can have the same values described for absolute_pathname_.

Name: active_fnc_err_

The active_fnc_err_ subroutine is called by active functions when they detect unusual status conditions. This subroutine formats an error message and then signals the condition active_function_error. The default handler for this condition prints the error message and then returns the user to command level. See the Programmer's Reference Manual for additional information on default handling.

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

USAGE

```
declare active_fnc_err_ entry options (variable);

call active_fnc_err_ (code, caller, control_string, argl, ..., argN);
```

ARGUMENTS

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

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

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

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 active_fnc_err_ subroutine has the format:

```
caller:  system_message user_message
```

where:

caller
: is the caller argument described above and should be the name of the procedure detecting the error.

system_message
: is a standard message from a standard status table corresponding to the value of code. If code is equal to 0, no system_message is returned.

user_message
: is constructed by the ioa_ subroutine from the control_string and argi arguments described above. If the control_string and argi arguments are not given, user_message is omitted.

Entry: active_fnc_err_$suppress_name

This entry point is functionally the same as active_fnc_err_, but it suppresses the caller name and the colon at the beginning of the error message. The caller name is nevertheless passed to the active_function_error handler.

USAGE

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

call active_fnc_err_$suppress_name (code, caller, control string,
     arg1,...argN);
```

where all arguments are the same as above.

Name: add_bit_offset_

This function returns a pointer to a bit relative to the bit referenced by the input pointer. The displacement to the desired bit may be positive, negative, or zero.

USAGE

```
declare add_bit_offset_ entry (ptr, fixed bin (24)) returns (ptr)
     reducible;

new_pointer_value = add_bit_offset_ (pointer_value, bit_displacement);
```

ARGUMENTS

pointer_value
    is the original pointer to which the bit displacement is applied. (Input)

bit_displacement
    is the displacement in bits to be applied to the above pointer. (Input)

new_pointer_value
    is the result of this operation. (Output)

NOTES

If the result of applying the displacement would cause the pointer to reference outside the legal boundaries of a segment (either a negative offset or an offset beyond 256K words), the result of the call is not defined.

EXAMPLES

The program fragment:

```
current_bit_ptr = add_bit_offset_ (current_bit_ptr, -1);
```

changes the value of current_bit_ptr to locate the previous bit in the segment.

Name: add_char_offset_

This function returns a pointer to a character relative to the character referenced by the input pointer. The displacement to the desired character may be positive, negative, or zero.

USAGE

```
declare add_char_offset_ entry (ptr, fixed bin (21)) returns (ptr)
     reducible;

new_pointer_value = add_char_offset_ (pointer_value, char_displacement);
```

ARGUMENTS

pointer_value
: is the original pointer to which the character displacement is applied. (Input)

char_displacement
: is the displacement in characters to be applied to the above pointer. (Input)

new_pointer_value
: is the result of this operation. (Output)

NOTES

If the pointer supplied to add_char_offset_ does not point to a character boundary, this operation is applied to a pointer value which references the character containing the bit located by the input pointer.

Thus, the program fragment:

```
a_ptr = add_char_offset_ (a_ptr, 0);
```

may be used to insure that "a_ptr" points to a character boundary.

If the result of applying the displacement would cause the pointer to reference outside the legal boundaries of a segment (either a negative offset or an offset beyond 256K words), the result of the call is not defined.

EXAMPLES

The program fragment:

```
current_char_ptr = add_char_offset (current_char_ptr, -1);
```

changes the value of current_char_ptr to locate the previous character in the segment.

Name: add_epilogue_handler_

The add_epilogue_handler_ subroutine is used to add an entry to the list of those handlers called when a process or run unit is terminated. A program established as an epilogue handler during a run unit is called when the run unit is terminated. If the process continues after the run unit is terminated, the handler is discarded from the list of those called when the process is terminated. Hence, epilogue handlers established during a run unit are not retained beyond the life of the run unit.

USAGE

```
declare add_epilogue_handler_ entry (entry, fixed bin (35));

call add_epilogue_handler_ (ev, code);
```

ARGUMENTS

ev
: is an entry value to be placed on the list of such values to be called when the run unit or process is cleaned up. (Input)

code
: is a standard status code. (Output)

NOTE

The add_epilogue_handler_ subroutine effectively manages two lists of epilogue handlers: those for the run unit, if a run unit is active, and those for the process. While a run unit is active, it is not possible to add entries to the list for the process. There is no way to establish a process epilogue handler while a run unit is active. The caller of execute_epilogue_ (logout, new_proc, etc.) must indicate whether all or just the run unit handlers are to be invoked.

Name: adjust_bit_count_

The adjust_bit_count_ subroutine performs the basic work of the adjust_bit_count command. It is called to find the last nonzero word or character of a segment or multisegment file and set the bit count accordingly. In the case of a multisegment file, empty trailing components are deleted and the returned bit count is the sum of the bit counts of the nonzero components. Only the bit count of the last component is altered.

USAGE

```
declare adjust_bit_count_ entry (char(168) aligned, char(32) aligned,
     bit(1) aligned, fixed bin(35), fixed bin(35));

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

ARGUMENTS

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

entryname
is the entryname of the segment. (Input)

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.

bit_count
is the computed bit count for the segment. (Output) 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.

code
is a standard status code. (Output)

Name: aim_check_

The aim_check_ subroutine provides a number of entry points for determining the relationship between two access attributes. An access attribute can be either an authorization or an access class. See also the read_allowed_, read_write_allowed_, and write_allowed_ subroutines in this document.

Entry: aim_check_$equal

This entry point compares two access attributes to determine whether they satisfy the equal relationship of the access isolation mechanism (AIM).

USAGE

```
declare aim_check_$equal entry (bit(72) aligned, bit(72) aligned)
      returns (bit(1) aligned);

returned_bit = aim_check_$equal (acc_att1, acc_att2);
```

ARGUMENTS

acc_atti
    are access attributes. (Input)

returned_bit
    is the result of the comparison. (Output)
    "1"b   acc_att1 equals acc_att2.
    "0"b   acc_att1 does not equal acc_att2.

Entry: aim_check_$greater

This entry point compares two access attributes to determine whether they satisfy the greater-than relationship of the AIM.

USAGE

```
declare aim_check_$greater entry (bit(72) aligned, bit(72) aligned)
      returns (bit(1) aligned);

returned_bit = aim_check_$greater (acc_att1, acc_att2);
```

ARGUMENTS

acc_atti
    are access attributes. (Input)

returned_bit
    is the result of the comparison. (Output)
    "1"b acc_att1 is greater than acc_att2.
    "0"b acc_att1 is not greater than acc_att2.

Entry: aim_check_$greater_or_equal

This entry point compares two access attributes to determine whether they satisfy either the greater-than or the equal relationships of the AIM.

USAGE

```
declare aim_check_$greater_or_equal entry (bit(72) aligned, bit(72)
      aligned) returns (bit(1) aligned);

returned_bit = aim_check_$greater_or_equal (acc_att1, acc_att2);
```

ARGUMENTS

acc_atti
    are access attributes. (Input)

returned_bit
    is the result of the comparison. (Output)
    "1"b acc_att1 is greater than or equal to acc_att2.
    "0"b acc_att1 is not greater than or equal to acc_att2.

Entry: aim_check_$in_range

Returns a flag indicating whether a specified access attribute is within the specified access attribute range.

USAGE

```
declare aim_check_$in_range entry (bit(72) aligned, (2) bit(72) aligned)
      returns (bit(1) aligned);

result = aim_check_$in_range (test_acc_att, acc_att_range);
```

ARGUMENTS

test_acc_att
    is the access attribute to be tested to see if it is within the range. (Input)

acc_att_range
    is an access attribute range. (Input)

in_range
    will be "1"b if and only if acc_att_range (2) >= test_acc_att >= acc_att_range (1). (Output)

Name: aim_util_

The aim_util_ subroutine contains entrypoints that manipulate AIM access classes and authorizations.

Entry: aim_util_$get_access_class

This entry point extracts the access class from an authorization.

USAGE

```
declare aim_util_$get_access_class entry (bit(72) aligned) returns
      (bit(72) aligned);

access_class = aim_util_$get_access_class (authorization);
```

ARGUMENTS

authorization
    is a standard AIM authorization marking. (Input)

access_class
    is a standard AIM access class marking. (Output)

Entry: aim_util_$get_privileges

This entry point extracts the privileges from a standard AIM authorization.

USAGE

```
declare aim_util_$get_privileges entry (bit(72) aligned) returns
      (bit(36) aligned);

privileges = aim_util_$get_privileges (authorization);
```

ARGUMENTS

authorization
: is a standard AIM authorization marking. (Input)

privileges
: is a standard AIM privilege string. (Output) See the include file aim_privileges.incl.pl1 for the interpretation of this string.

Entry: aim_util_$get_level

This entry point extracts the sensitivity level from an access class or authorization.

USAGE

```
declare aim_util_$get_level entry (bit(72) aligned) returns (fixed bin);

level = aim_util_$get_level (access_class);
```

ARGUMENTS

access_class
: is a standard AIM access class or authorization marking. (Input)

level
: is a sensitivity level number. (Output) Levels range from 0 to 7. Level names are available via system_info_$level_names.

Entry: aim_util_$get_categories

This entry point extracts the categories from a standard AIM access class or authorization.

USAGE

```
declare aim_util_$get_categories entry (bit(72) aligned) returns
       (bit(36) aligned);

categories = aim_util_$get_categories (access_class);
```

ARGUMENTS

access_class
: is a standard AIM access class or authorization marking. (Input)

categories
: is a bit string representing the category information contained in the access class. (Output) If the i'th bit of the bit string is a 1, then the i'th category is included in the access class marking. Category names are available from system_info_$category_names.

Entry: aim_util_$make_access_class

This entry point constructs an access class marking from a level and a set of categories.

USAGE

```
declare aim_util_$make_access_class (fixed bin, bit(36) aligned, bit(72)
     aligned);

call aim_util_$make_access_class (level, categories, access_class);
```

ARGUMENTS

level
: is a sensitivity level number, from 0 to 7. (Input)

categories
: is a category bit string. (Input) See aim_util_$get_categories for the construction of this string.

access_class
: is a standard AIM access class marking. (Output)

Name: archive_

The archive_ subroutine is used to access individual components in archives, list the components of an archive, and obtain information about archive components.

Entry: archive_$get_component

This entry, given a pointer to an archive and its bitcount, and the name of the desired component in the archive, returns a pointer to the component and the bitcount of the component. It is used when there is a specific component in the archive which is to be referenced. For applications that wish to serially access all the components in an archive, archive_$next_component is more appropriate. This entry only returns a pointer and length for the component; if more information is desired, the archive_$get_component_info entrypoint should be used.

USAGE

```
declare archive_$get_component entry (pointer, fixed bin(24), char(*),
      pointer, fixed bin(24), fixed bin(35));

call archive_$get_component (archive_ptr, archive_bc, component_name,
      component_ptr, component_bc, code);
```

ARGUMENTS

archive_ptr
: is a pointer to the archive segment to be searched. (Input) It need not point to the base of a segment; it is converted to a segment base pointer by archive_, so a pointer to anywhere in the segment may be given here.

archive_bc
: is the bitcount of the archive segment. (Input)

component_name
: is the name of the component to be searched for. (Input) It can be up to 32 characters long.

component_ptr
: is a pointer to the first word of the archive component if the specified component was found, or null otherwise. (Output) It is a pointer into the segment pointed to by archive_ptr.

component_bc
: is the bitcount of the archive component pointed to by component_ptr. (Output) It describes a region of the archive segment which contains the specified component; if an attempt is made to reference past the end of this area, invalid data may be referenced.

code
is a standard system status code, one of the following: (Output)
error_table_$no_component
indicates that the specified component was not found in the archive.
error_table_$not_archive
indicates that archive_ptr points to a segment which does not appear to be a properly formatted archive.
error_table_$archive_fmt_err
indicates that, although the segment pointed to by archive_ptr does appear to be a valid archive, it contains an incorrectly formatted archive header. The archive should be repaired before further use either by extracting all the still-accessible components and creating a new archive, or by manipulating it with a text editor to access the apparent components.

Entry: archive_$get_component_info

This entry, given a pointer to an archive and its bitcount, and the name of the desired component in the archive, fills in a caller-supplied structure with information describing the archive component. Also see archive_$get_component and archive_$next_component_info.

USAGE

```
declare archive_$get_component_info entry (pointer, fixed bin(24),
      char(*), pointer, fixed bin(35));

call archive_$get_component_info (archive_ptr, archive_bc,
      component_name, archive_component_info_ptr, code);
```

ARGUMENTS

archive_ptr
is a pointer to the archive segment to be searched. (Input) It need not point to the base of a segment; it is converted to a segment base pointer by archive_, so a pointer to anywhere in the segment can be given here.

archive_bc
is the bitcount of the archive segment. (Input)

component_name
is the name of the component to be searched for. (Input) It can be up to 32 characters long.

archive_component_info_ptr
is a pointer to a user-supplied archive_component_info structure, described below. (Input) The caller must have previously set archive_component_info.version to the appropriate version number, currently ARCHIVE_COMPONENT_INFO_VERSION_1. The structure is filled in with information describing the selected archive component if it can be found.

code
is a standard system status code. (Output) It can have any of the values which can be returned by archive_$get_component, and can also have the following value:
error_table_$unimplemented_version
indicates that the version number in the caller-supplied archive_component_info structure is not correct.

STRUCTURE

The archive_component_info_ptr points to the following structure (described in the archive_component_info.incl.pl1 include file):

```
dcl 1 archive_component_info       aligned based (archive_component_info_ptr),
      2 version                    fixed bin,
      2 comp_bc                    fixed bin (24),
      2 comp_ptr                   ptr,
      2 name                       char (32) unaligned,
      2 time_modified              fixed bin (71),
      2 time_updated               fixed bin (71),
      2 comp_lth                   fixed bin (19),
      2 access                     bit (36) unaligned;
```

STRUCTURE ELEMENTS

version
must be set to ARCHIVE_COMPONENT_INFO_VERSION_1 by the caller. All other structure elements are output.

comp_bc
is the bit_count of the archive component.

comp_ptr
is a pointer to the base of the component.

name
is the name of the component.

time_modified
is a clock reading corresponding to the date/time contents modified of the segment from which this component was most recently updated. This is the value reported in the "modified" column by the "ac tl" command. It may be inaccurate by several hours if the archive was updated in a different time zone than the current time zone.

time_updated
is a clock reading corresponding to the date/time when this component was last updated in the archive. This is the value reported in the "updated" column by the "ac tl" command. It may be inaccurate by several hours if the archive was updated in a different time zone than the current time zone.

comp_lth
is the size, in words, of the component. Both the size in words and the bit_count are provided as a convenience to the caller. The size in words is derived from the bit_count.

access
is the representation of the effective access mode recorded with the archive component. The first bit is "r" access, the second is "e", and the third is "w". Even if "a" access appears in the archive itself, it will be ignored.

Entry: archive_$list_components

This entry, given a pointer to an archive and its bitcount, and a pointer to an area, allocates an array of archive_component_info structures in the area to describe all the components in the archive, and returns a pointer to and the size of this array. This entry is intended to be used in applications where it is more convenient to loop through an array processing archive components than it is to step through the components by using archive_$next_component_info. There is no corresponding list interface which just returns name, pointer and bit_count; the complete archive_component_info structure is always supplied.

USAGE

```
declare archive_$list_components entry (pointer, fixed bin(24), fixed
      bin, pointer, pointer, fixed bin, fixed bin(35));

call archive_$list_components (archive_ptr, archive_bc, info_version,
      area_ptr, archive_component_info_array_ptr, n_components, code);
```

ARGUMENTS

archive_ptr
is a pointer to the archive segment to be searched. (Input) It need not point to the base of a segment; it is converted to a segment base pointer by archive_, so a pointer to anywhere in the segment can be given here.

archive_bc
is the bitcount of the archive segment. (Input)

info_version
is the version number for the archive_component_info structure array which will be allocated and returned. (Input) The only supported version is ARCHIVE_COMPONENT_INFO_VERSION_1.

area_ptr
is a pointer to a caller-supplied area in which the returned array of archive_component_infos will be allocated. (Input) If area_ptr is null, no list will be allocated, but n_components will still be set; this can be used when it is desired to merely count the components in the archive.

archive_component_info_array_ptr
is a pointer returned which points to an array of archive_component_info structures describing all the components in the archive. (Output) It should be declared as follows:

```
dcl 1 archive_component_info_array (n_components) aligned
      like archive_component_info based
      (archive_component_info_array_ptr);
```

The version number in all the elements of this array will be the same as was passed in the info_version argument. The archive_component_info_array_ptr will be null if there are no components in the archive; n_components will be returned as zero, and the code will be zero as well. It will also be null if a null area_ptr was supplied.

n_components
is the number of components in the archive. (Output) This can be zero if the archive is empty, and is still valid.

code
is a standard system status code, one of the following: (Output)
error_table_$not_archive
indicates that archive_ptr points to a segment which does not appear to be a properly formatted archive.
error_table_$archive_fmt_err
indicates that, although the segment pointed to by archive_ptr does appear to be a valid archive, it contains an incorrectly formatted archive header. The archive should be repaired before further use either by extracting all the still-accessible components and creating a new archive, or by manipulating it with a text editor to access the apparent components.

Entry: archive_$next_component

This entry, given a pointer to an archive and its bitcount, and a pointer to the base of a component (or null), returns a pointer to the next component in the archive, its name, and its bitcount. If there are no components remaining in the archive, the pointer is returned null on output. The first time this is called for a particular archive, the component pointer should be supplied as null. This entry is intended to be used to step through all the components of an archive, one at a time. The archive should not be modified while this is being done, or the results will be unpredictable. See also archive_$get_component and archive_$next_component_info.

USAGE

```
declare archive_$next_component entry (pointer, fixed bin(24), pointer,
      fixed bin(24), char(*), fixed bin(35));

call archive_$next_component (archive_ptr, archive_bc, component_ptr,
      component_bc, component_name, code);
```

ARGUMENTS

archive_ptr
is a pointer to the archive segment to be searched. (Input) It need not point to the base of a segment; it is converted to a segment base pointer by archive_, so a pointer to anywhere in the segment can be given here.

archive_bc
is the bitcount of the archive segment. (Input)

component_ptr
on input, this is a pointer to the previous component in the archive, or null to indicate that the next component should be the first component in the archive. (Input/Output) On output, this is a pointer to the next component in the archive, or null if there are no components remaining after the one it pointed to on input.

component_bc
is the bitcount of the selected component. (Output)

component_name
is the name of the selected component. (Output)

code
is a standard system status code, one of the following: (Output)
error_table_$not_archive
indicates that archive_ptr points to a segment which does not appear to be a properly formatted archive.
error_table_$archive_fmt_err
indicates that, although the segment pointed to by archive_ptr does appear to be a valid archive, it contains an incorrectly formatted archive header. The archive should be repaired before further use either by extracting all the still-accessible components and creating a new archive, or by manipulating it with a text editor to access the apparent components.

Entry: archive_$next_component_info

This entry, given a pointer to an archive, the bitcount of the archive, and a pointer to the base of a component (or null), returns a pointer to the next component in the archive and fills in an archive_component_info structure to describe it. If there are no components remaining in the archive, the pointer is returned null on output. The first time this is called for a particular archive, the component pointer should be supplied as null. See also archive_$get_component_info and archive_$next_component.

USAGE

```
declare archive_$next_component_info entry (pointer, fixed bin(24),
     pointer, pointer, fixed bin(35));

call archive_$next_component_info (archive_ptr, archive_bc,
     component_ptr, archive_component_info_ptr, code);
```

ARGUMENTS

archive_ptr
: is a pointer to the archive segment to be searched. (Input) It need not point to the base of a segment; it is converted to a segment base pointer by archive_, so a pointer to anywhere in the segment may be given here.

archive_bc
: is the bitcount of the archive segment. (Input)

component_ptr
: on input, this is a pointer to the previous component in the archive, or null to indicate that the next component should be the first component in the archive. (Input/Output) On output, this is a pointer to the next component in the archive, or null if there are no components remaining after the one it pointed to on input.

archive_component_info_ptr
: is a pointer to a user-supplied archive_component_info structure, described in the description of the archive_$get_component_info entrypoint. (Input) The caller must have previously set archive_component_info.version to the appropriate version number. currently ARCHIVE_COMPONENT_INFO_VERSION_1. The structure is filled in with information describing the selected archive component if component_ptr is returned non-null.

code
: is a standard system status code. (Output) It may have any of the values which can be returned by archive_$next_component, and may also have the following value:
: error_table_$unimplemented_version
: : indicates that the version number in the caller-supplied archive_component_info structure is not correct.

This page intentionally blank.

This page intentionally blank.

This page intentionally blank.

Name: area_info_

The area_info_ subroutine returns information about an area.

USAGE

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

call area_info_ (info_ptr, code);
```

ARGUMENTS

info_ptr
points to the structure described in "Notes" below. (Input)

code
is a system status code. (Output)

NOTES

The structure pointed to by info_ptr is described by the following PL/I declaration (defined by the system include file, area_info.incl.pl1):

```
dcl 1 area_info                aligned based,
      2 version                fixed bin,
      2 control,
        3 extend               bit(1) unaligned,
        3 zero_on_alloc        bit(1) unaligned,
        3 zero_on_free         bit(1) unaligned,
        3 dont_free            bit(1) unaligned,
        3 no_freeing           bit(1) unaligned,
        3 system               bit(1) unaligned,
        3 mbz                  bit(30) unaligned,
      2 owner                  char(32) unaligned,
      2 n_components           fixed bin,
      2 size                   fixed bin(30),
      2 version_of_area        fixed bin,
      2 areap                  ptr,
      2 allocated_blocks       fixed bin,
      2 free_blocks            fixed bin,
      2 allocated_words        fixed bin(30),
      2 free_words             fixed bin(30);
```

STRUCTURE ELEMENTS

version
is set by the caller and should be 1.

control
are control bits describing the format and type of the area.

extend
indicates whether the area is extensible.
"1"b yes
"0"b no

zero_on_alloc
indicates whether blocks are cleared (set to all zeros) at allocation time.
"1"b yes
"0"b no

zero_on_free
indicates whether blocks are cleared (set to all zeros) at free time.
"1"b yes
"0"b no

dont_free
indicates whether free requests are disabled (for debugging).
"1"b yes
"0"b no

no_freeing
indicates whether the allocation method assumes no freeing will be done.
"1"b yes
"0"b no

system
causes the use of hcs_$make_seg instead of get_temp_segments to create the first component. It assumes that the original area is all zeroes, rather than explicitly zeroing it.
"1"b yes
"0"b no

mbz
is not used and must be zeros.

owner
is the name of the program that created the area if the area is extensible.

n_components
is the number of components in the area.

size
is the total number of words in the area.

version_of_area
is 0 for (old) buddy system areas and 1 for standard areas.

areap
is filled in by the caller and can point to any component of the area.

allocated_blocks
is the number of allocated blocks in the area.

free_blocks
is the number of free blocks in the area (not including virgin storage within components, i.e., storage after the last allocated block).

allocated_words
is the number of allocated words in the area.

free_words
is the number of free words in the area not counting virgin storage.

No information is returned about version 0 areas except the version number.

If the no_freeing bit is on ("1"b), the counts of free and allocated blocks are returned as 0.

Entry: area_info_$get_block_data_info

This entrypoint returns a pointer and length for the first block or next block in an area, and whether or not it is free. This allows a program to step through an area looking at each block in turn. Extensible areas are handled correctly.

USAGE

```
declare area_info_$get_block_data_info entry (ptr, bit (1), ptr, ptr,
      ptr, fixed bin (18), bit (1), fixed bin (35));

call area_info_$get_block_data_info (area_ptr, next_ptr_flag,
      block_data_ptr, output_area_ptr, next_data_ptr, data_size,
      block_allocated_flag, code);
```

ARGUMENTS

area_ptr
is a pointer to the area in which the data block will be found. (Input)

next_ptr_flag
if "1"b, then return information about the block after the one pointed to by a block_data_ptr. (Input)

block_data_ptr
pointer to a data block in the area. If it is null then it will be internally initialized to the first block in the area. (Input)

output_area_ptr
is a pointer to the area which actually contains the block about which information is returned. it will be equal to area_ptr unless the area is extensible and the returned block information required going to the next segment in the area. When stepping through the blocks in an area, this pointer should be used as input (i.e. area_ptr) for the next call. (Output)

next_data_ptr
is a pointer to the block in which information is returned. It will be equal to block_data_ptr unless next_ptr_flag was set, in which case it will point to the block after the one pointed to by block_data_ptr. (Output)

data_size
is the size, in words, of the returned data block. (Output)

block_allocated_flag
If "1"b, then the block is allocated. If "0"b, then the block is free. (Output)

code
is a standard system status code. it is returned as error_table_$end_of_info if the block about which information is requested is in virgin storage in the area (i.e. the end of the area has been reached). (Output)

Name: arithmetic_to_ascii_

The arithmetic_to_ascii_ subroutine formats any arithmetic value into a compact ascii form. An integer, fractional, or exponential format can be used, depending on the number to be converted. Fixed-point numbers are truncated during the formatting process; floating-point numbers are rounded.

USAGE

```
declare arithmetic_to_ascii_ entry (ptr, fixed bin, bit(1) aligned,
     fixed bin, fixed bin, char(132) varying);

call arithmetic_to_ascii_ (v_ptr, type, packed, precision, scale,
     result);
```

ARGUMENTS

v_ptr
    is a pointer to the value to be converted. (Input) It can be any arithmetic data type (real or complex, fixed or float, binary or decimal, single or double precision).

type
    is a standard Multics descriptor type. (Input) See the Programmer's Reference Manual for a list of standard Multics data types.

packed
    indicates whether the value is packed or unpacked. (Input)
    "0"b value is unpacked.
    "1"b value is packed.

precision
    is the precision of the value to be converted. (Input)

scale
    is the scale factor of the value to be converted. (Input)

result
    is the character-string representation of the value to be converted; it contains no blanks. (Output)

NOTES

If the value is complex, the real and imaginary parts are formatted by correcting them to float decimal(59) and converting each part separately. The result returned by the arithmetic_to_ascii_ subroutine is the concatenation of the real and imaginary converted parts, with a leading sign and trailing "i" supplied for the imaginary part.

Name: ascii_to_bcd_

The ascii_to_bcd subroutine performs isomorphic (one-to-one reversible) conversion from ASCII to BCD.

USAGE

```
dcl ascii_to_bcd_ entry (char(*), bit(*));

call ascii_to_bcd_ (ascii_in, bcd_out);
```

ARGUMENTS

ascii_in
: is the ascii input characters tro convert to BCD. (Input)

bcd_out
: is the BCD equivalent of the input string. (Output) Note that both upper and lower case ASCII characters are converted to the single case BCD characters. ASCII characters that do not have a match in BCD will be converted to a question mark (?). For more information see "Notes" below.

NOTES

The ASCII question mark (?) and any ASCII characters (other than lowercase letters) will be mapped into a BCD question mark (?). The valid BCD characters are as follows:

0123456789 [@:?ABCDEFGHI&.] (<\^JKLMNOPQR-$*);';/STUVWXYZ_,%= #<space>

BCD must be aligned on a 6-bit BCD character boundary.

Name: ascii_to_ebcdic_

The ascii_to_ebcdic_ subroutine performs isomorphic (one-to-one reversible) conversion from ASCII to EBCDIC. The input data is a string of valid ASCII characters. A valid ASCII character is defined as a 9-bit byte with an octal value in the range 0 <= octal_value <= 177.

This entry point accepts an ASCII character string and generates an EBCDIC character string of equal length.

USAGE

```
declare ascii_to_ebcdic_ entry (char(*), char(*));

call ascii_to_ebcdic_ (ascii_in, ebcdic_out);
```

ARGUMENTS

ascii_in
: is a string of ASCII characters to be converted. (Input)

ebcdic_out
: is the EBCDIC equivalent of the input string. (Output)

Entry: ascii_to_ebcdic_$ae_table

This entry point defines the 128-character translation table used to perform conversion from ASCII to EBCDIC. The mappings implemented by the ascii_to_ebcdic_ and ebcdic_to_ascii_ subroutines are isomorphic; i.e., every valid character has a unique mapping, and mappings are reversible. (See the ebcdic_to_ascii_ subroutine.) The result of an attempt to convert a character that is not in the ASCII character set is undefined.

USAGE

```
declare ascii_to_ebcdic_$ae_table char(128) external static;
```

ISOMORPHIC ASCII/EBCDIC CONVERSION TABLE

ASCII GRAPHIC	ASCII OCTAL	EBCDIC HEXADECIMAL	EBCDIC GRAPHIC
NUL	000	00	NUL
SOH	001	01	SOH
STX	002	02	STX
ETX	003	03	ETX
EOT	004	37	EOT
ENQ	005	2D	ENQ
ACK	006	2E	ACK
BEL	007	2F	BEL
BS	010	16	BS
HT	011	05	HT
LF	012	25	NL
VT	013	0B	VT
FF	014	0C	NP
CR	015	0D	CR
SO	016	0E	SO
SI	017	0F	SI
DLE	020	10	DLE
DC1	021	11	DC1
DC2	022	12	DC2
DC3	023	13	TM
DC4	024	3C	DC4
NAK	025	3D	NAK
SYN	026	32	SYN
ETB	027	26	ETB
CAN	030	18	CAN
EM	031	19	EM
SUB	032	3F	SUB
ESC	033	27	ESC
FS	034	1C	IFS
GS	035	1D	IGS
RS	036	1E	IRS
US	037	1F	IUS
space	040	40	space
!	041	5A	!
"	042	7F	"
#	043	7B	#
$	044	5B	$
%	045	6C	%
&	046	50	&
'	047	7D	'
(	050	4D	(
)	051	5D	)
*	052	5C	*
+	053	4E	+

ASCII		EBCDIC	
GRAPHIC	OCTAL	HEXADECIMAL	GRAPHIC
,	054	6B	,
-	055	60	-
.	056	4B	.
/	057	61	/
0	060	F0	0
1	061	F1	1
2	062	F2	2
3	063	F3	3
4	064	F4	4
5	065	F5	5
6	066	F6	6
7	067	F7	7
8	070	F8	8
9	071	F9	9
:	072	7A	:
;	073	5E	;
<	074	4C	<
=	075	7E	=
>	076	6E	>
?	077	6F	?
@	100	7C	@
A	101	C1	A
B	102	C2	B
C	103	C3	C
D	104	C4	D
E	105	C5	E
F	106	C6	F
G	107	C7	G
H	110	C8	H
I	111	C9	I
J	112	D1	J
K	113	D2	K
L	114	D3	L
M	115	D4	M
N	116	D5	N
O	117	D6	O
P	120	D7	P
Q	121	D8	Q
R	122	D9	R
S	123	E2	S
T	124	E3	T
U	125	E4	U
V	126	E5	V
W	127	E6	W
X	130	E7	X

ASCII		EBCDIC	
GRAPHIC	OCTAL	HEXADECIMAL	GRAPHIC
Y	131	E8	Y
Z	132	E9	Z
[	133	AD	[(see "Notes")
\	134	E0	\
]	135	BD	] (see "Notes")
^	136	5F	logical NOT
_	137	6D	_
grave accent	140	79	grave accent
a	141	81	a
b	142	82	b
c	143	83	c
d	144	84	d
e	145	85	e
f	146	86	f
g	147	87	g
h	150	88	h
i	151	89	i
j	152	91	j
k	153	92	k
l	154	93	l
m	155	94	m
n	156	95	n
o	157	96	o
p	160	97	p
q	161	98	q
r	162	99	r
s	163	A2	s
t	164	A3	t
u	165	A4	u
v	166	A5	v
w	167	A6	w
x	170	A7	x
y	171	A8	y
z	172	A9	z
{	173	C0	{
|	174	4F	solid bar
}	175	D0	}
tilde	176	A1	tilde
DEL	177	07	DEL

NOTES

The graphics ([and]) do not appear in (or map into any graphics that appear in) the standard EBCDIC character set. They have been assigned to otherwise "illegal" EBCDIC code values in conformance with the bit patterns used by the TN text printing train.

Calling the ascii_to_ebcdic_ subroutine is as efficient as using the PL/I translate builtin. since conversion is performed by a single MVT instruction and the procedure runs in the stack frame of its caller.

This mapping differs from the ASCII to EBCDIC punched card code mapping as discussed in the Programmer's Reference Manual. The characters that differ when mapped are: [ ] \ and NL (newline).

Name: ask_

The ask_ subroutine provides a flexible terminal input facility for whole lines, strings delimited by blanks, or fixed-point and floating-point numbers. Special attention is given to prompting the terminal user.

The main entry point returns the next string of characters delimited by blanks or tabs from the line typed by the user. If the line buffer is empty, the ask_ subroutine formats and types out a prompting message and reads a line from the user_input I/O switch.

USAGE

```
declare ask_ entry options (variable);

call ask_ (ctl, ans, ioa_args);
```

ARGUMENTS

ctl
: is an ioa_ control string (char(*)) in the same format as that used by the ioa_ subroutine. (Input)

ans
: is the return value (char(*)). (Output)

ioa_args
: are any number of arguments to be converted according to ctl. (Input)

Entry: ask_$ask_c

This entry point tests to determine if there is anything left on the line. If so, it returns the next symbol, as in the ask_$ask_ entry point, and sets a flag to 1. Otherwise, it sets the flag to 0 and returns.

USAGE

```
declare ask_$ask_c entry (char(*), fixed bin);

call ask_$ask_c (ans, flag);
```

ARGUMENTS

ans
: is the next symbol, if any. (Output)

flag
: is the symbol flag. (Output). Its value can be:
  1 if the symbol is returned.
  0 if there is no symbol.

Entry: ask_$ask_cint

This entry point is a conditional entry for integers. If an integer is available on the line, it is returned and the flag is set to 1. If the line is empty, the flag is set to 0. If there is a symbol on the line, but it is not a number, it is left on the line and the flag is set to −1.

USAGE

```
declare ask_$ask_cint entry (fixed bin, fixed bin);

call ask_$ask_cint (int, flag);
```

ARGUMENTS

int
: is the returned value, if any. (Output)

flag
: is the int flag. (Output). Its value can be:
  1 if int is returned.
  0 if the line is empty.
  −1 if there is no number.

Entry: ask_$ask_cflo

This entry point works like the ask_$ask_cint entry point but returns a floating value, if an integer is available.

USAGE

```
declare ask_$ask_cflo entry (float bin, fixed bin);

call ask_$ask_cflo (flo, flag);
```

ARGUMENTS

flo
    the returned value, if any. (Output)

flag
    is the flow flag. (Output). Its value can be:
    0 if the line is empty.
    1 if the value is returned.
    -1 if it is not a number.

Entry: ask_$ask_cline

This entry point returns any part of the line that remains. A flag is set if the rest of the line is empty.

USAGE

```
declare ask_$ask_cline entry (char(*), fixed bin);

call ask_$ask_cline (line, flag);
```

ARGUMENTS

line
    is the returned line, if any. (Output)

flag
    is the line flag. (Output). Its value can be:
    1 if the line is returned.
    0 if the line is empty.

Entry: ask_$ask_clr

This entry point clears the internal line buffer. Because the buffer is internal static, the input of one program can accidentally be passed to another unless the second begins with a call to this entry point. If a value typed by the user is incorrect and if the program wishes to ask for the line to be retyped, the ask_$ask_clr entry point can also be called.

USAGE

```
declare ask_$ask_clr entry;

call ask_$ask_clr;
```

Entry: ask_$ask_cnf

This entry point works like the ask_$ask_cint entry point except that it returns a value of "on" or "off" if an integer is available.

USAGE

```
declare ask_$ask_cnf entry (char(*), fixed bin);

call ask_$ask_cnf (ans, flag);
```

ARGUMENTS

ans
: is a value of "on" or "off" if such a value is present. (Output)

flag
: is the yn flag. (Output). Its value can be:
  1 if a "on" or "off" value is returned.
  0 if the line is empty.
  -1 if the next value on the line is not "on" or "off"

Entry: ask_$ask_cyn

This entry point works like the ask_$ask_cint entry point except that it returns a value of yes (or y) or no (or n) if an integer is available.

USAGE

```
declare ask_$ask_cyn (char(*), fixed bin);

  ans
     call ask_$ask_cyn (ans, flag);
```

ARGUMENTS

ans
: is a value of yes (or y) or no (or n) if such a value is present. (Output)

flag
: is the yn flag. (Output). Its value can be:
  1 if a yes (or y) or no (or n) value is returned.
  0 if the line is empty.
  -1 if the next value on the line is not yes (or y) or no (or n).

Entry: ask_$ask_int

This entry point works the same as the ask_$ask_ entry point except that the next item on the line must be a number. An integer value is returned. Numbers can be fixed point or floating point, positive or negative. A leading dollar sign or a comma is ignored. If the value typed is not a number, the program types:

"string" nonnumeric. Please retype:

and waits for the user to retype the line.

USAGE

```
declare ask_$ask_int entry options (variable);

call ask_$ask_int (ctl, int, ioa_args);
```

ARGUMENTS

ctl
: is an ioa_ control string (char(*)) in the same format as that used by the ioa_ subroutine. (Input). If a period is typed, zero is returned.

int
: is the return value (fixed bin). (Output)

ioa_args
: are any number of arguments to be converted according to ctl. (Input)

Entry: ask_$ask_flo

This entry point works like the ask_$ask_int entry point except that it returns a floating value.

USAGE

```
declare ask_$ask_flo entry options (variable);

call ask_$ask_flo (ctl, flo, ioa_args);
```

ARGUMENTS

ctl
: is an ioa_ control string (char(*)) in the same format as that used by the ioa_ subroutine. (Input). If a period is typed, zero is returned.

flo
: is the return value (float bin). (Output)

ioa_args
: are any number of arguments to be converted according to ctl. (Input)

Entry: ask_$ask_line

This entry returns the remainder of the line typed by the user. Leading blanks are removed. If there is nothing left on the line, the program prompts and reads a new line.

USAGE

```
declare ask_$ask_line entry options (variable);

call ask_$ask_line (ctl, line, ioa_args);
```

ARGUMENTS

ctl
: is an ioa_ control string (char(*)) in the same format as that used by the ioa_ subroutine. (Input). If a period is typed, zero is returned.

line
: is the return value (char(*)). (Output)

ioa_args
: are any number of arguments to be converted according to ctl. (Input)

Entry: ask_$ask_n

This entry point scans the line and returns the next symbol without changing the line pointer. A call to the ask_ entry point later returns the same value.

USAGE

```
declare ask_$ask_n entry (char (*), fixed bin);

call ask_$ask_n (ans, flag);
```

ARGUMENTS

ans
: is the returned symbol, if any. (Output)

flag
: is the ans flag. (Output). Its value can be:
  0 if the line is empty.
  1 if the symbol is returned.

Entry: ask_$ask_nf

This entry point works like ask_$ask_yn except that it returns a value of "on" or "off".

USAGE

```
declare ask_$ask_nf entry options (variable);

call ask_$ask_nf (ctl, line, ioa_args);
```

ARGUMENTS

ctl
: is an ioa_ control string (char(*)) in the same format as that used by the ioa_ subroutine. (Input) If a period is typed, zero is returned.

line
: is the return value (char(*)). (Output)

ioa_args
: are any number of arguments to be converted according to ctl. (Input)

Entry: ask_$ask_nflo

This entry point scans the line for floating point numbers.

USAGE

```
declare ask_$ask_nflo entry (float bin, fixed bin);

call ask_$ask_nflo (flo, flag);
```

ARGUMENTS

flo
: is the returned value, if any. (Output)

flag
: is the flow flag. (Output). Its value can be:
  0 if the line is empty.
  1 if the value is returned.
  -1 if it is not a number.

Entry: ask_$ask_nint

This entry point scans the line for integers. The second argument is returned as -1 if there is a symbol on the line but it is not a number, 1 if successful, and 0 if the line is empty.

USAGE

```
declare ask_$ask_nint entry (fixed bin, fixed bin);

call ask_$ask_nint (int, flag);
```

ARGUMENTS

int
: is the returned value, if any. (Output)

flag
: is the int flag. (Output). Its value can be:
  1 if int is returned.
  0 if the line is empty.
  -1 if there is no number.

Entry: ask_$ask_nline

This entry point initiates a scan of the rest of the line.

USAGE

```
declare ask_$ask_nline entry (char(*), fixed bin);

call ask_$ask_nline (line, flag);
```

ARGUMENTS

line
: is the returned line, if any. (Output)

flag
: is the line flag. (Output). Its value can be:
  1 if the line is returned.
  0 if the line is empty.

Entry: ask_$ask_nnf

This entry point returns the next symbol, if it is an "on" or "off" value, without changing the line pointer.

USAGE

```
declare ask_$ask_nnf entry (char(*), fixed bin);

call ask_$ask_nnf (ans, flag);
```

ARGUMENTS

ans
: is a value of "on" or "off" if such a value is present. (Output)

flag
: is the yn flag. (Output). Its value can be:
  1 if a "on" or "off" value is returned.
  0 if the line is empty.
  -1 if the next value on the line is not "on" or "off."

Entry: ask_$ask_nyn

This entry point returns the next symbol, if it is a yes (or y) or no (n) value, without changing the line pointer. The arguments are the same as those used with the ask_$ask_cint entry point.

USAGE

```
declare ask_$ask_nyn entry (char(*), fixed bin);

call ask_$ask_nyn (ans, flag);
```

ARGUMENTS

ans
: is a value of yes (or y) or no (or n) if such a value is present. (Output)

flag
: is the yn flag. (Output). Its value can be:
  - 1 if a yes (or y) or no (or n) value is returned.
  - 0 if the line is empty.
  - -1 if the next value on the line is not yes (or y) or no (or n)

Entry: ask_$ask_prompt

This entry point deletes the current contents of the internal line buffer and prompts for a new line. The line is read in and the entry returns.

USAGE

```
declare ask_$ask_prompt entry options (variable);

call ask_$ask_prompt (ctl, ioa_args);
```

ARGUMENTS

ctl
: is a control string (char(*)) similar to that typed by the ioa_ subroutine. (Input)

ioa_args
: are any number of arguments to be converted according to ctl. (Input)

Entry: ask_$ask_setline

This entry point sets the internal static buffer for the ask_ subroutine to the given input line so that the line can be scanned.

USAGE

```
declare ask_$ask_setline entry (char(*));

call ask_$ask_setline (line);
```

ARGUMENTS

line
: is the line to be placed in the ask_ buffer. (Input). Trailing blanks are removed from line. A carriage return is optional at the end of line.

Entry: ask_$ask_yn

This entry point works like the ask_$ask_int entry point except that it returns a value of yes (or y) or no (or n). Its arguments are the same as those used with the ask_$ask_int entry point.

USAGE

```
declare ask_$ask_yn entry options (variable);

call ask_$ask_yn (ctl, ans, ioa_args);
```

ARGUMENTS

ctl
: is an ioa_ control string (char(*)) in the same format as that used by the ioa_ subroutine. (Input). If a period is typed, zero is returned.

ans
: is a value of yes (or y) or no (or n) if such a value was present. (Input)

ioa_args
: are any number of arguments to be converted according to ctl. (Input)

Name: assign_

The assign_ subroutine assigns a specified source value to a specified target. This subroutine handles the following data types: 1-12, 19-22, 33, 34, 41-46. Any other type will produce an error. This subroutine uses rounding in the conversion when the target is floating point or when the source is floating and the target is character, and uses truncation in all other cases.

USAGE

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

call assign_ (target_ptr, target_type, target_length, source_ptr,
     source_type, source_length);
```

ARGUMENTS

target_ptr
: points to the target of the assignment; it can contain a bit offset. (Input)

target_type
: specifies the type of the target; its value is 2*M+P where M is the Multics standard data type code (see the Programmer's Reference Manual) and P is 0 if the target is unpacked and 1 if the target is packed. (Input)

target_length
: is the string length or arithmetic scale and precision of the target. If the target is arithmetic, the target_length word consists of two adjacent unaligned halfwords. The left halfword is a fixed bin(17) representing the signed scale and the right halfword is a fixed bin(18) unsigned integer representing the precision. (Input) The include file encoded_precision.incl.pl1 declares this as:

```
dcl 1 encoded_precision  based aligned,
      2 scale            fixed bin(17) unaligned,
      2 prec             fixed bin(18) unsigned unaligned;
```

source_ptr
: points at the source of the assignment; it can contain a bit offset. (Input)

source_type
: specifies the source type using the same format as target_type. (Input)

source_length
: is the string length or arithmetic scale and precision of the source using the same format as target_length. (Input)

Entry: assign_$computational_

The assign_$computational_ entry assigns a specified source value to a specified target. It can handle any computational Multics data type. This includes all PL/I computational data and all COBOL and FORTRAN data types. This entry uses the same rules for rounding and truncation as assign_.

USAGE

```
declare assign_$computational_ entry (ptr, ptr, fixed bin(35));

call assign_$computational_ (tar_str_ptr, src_str_ptr, code);
```

ARGUMENTS

tar_str_ptr
: is a pointer to a structure which defines the address and attributes of the target. The format of this structure is defined below. (Input)

src_str_ptr
: is a pointer to a structure giving the attributes of the source. This structure has the same format as the one used for the target. (Input)

code
: is a standard system code. It will be zero if the conversion was sucessful, or error_table_$bad_conversion if either data type was not computational. It is also possible that the conversion condition will be signalled, if the source data can not be converted to the requested target type. (Output)

NOTES

The format of the structures used to describe the source and target data is given by computational_data.incl.pl1. It is:

```
dcl 1 computational_data          aligned based,
      2 address                   ptr aligned,
      2 data_type                 fixed bin(17),
      2 flags                     aligned,
        3 packed                  bit(1) unal,
        3 pad                     bit(35) unal,
      2 prec_or_length            fixed bin(24),
      2 scale                     fixed bin(35),
      2 picture_image_ptr         ptr aligned;
```

STRUCTURE ELEMENTS

address
: is a pointer to the data where the data is (source) or where it is to go (target). It is the responsibility of the caller to ensure that there is sufficient room for the target.

data_type
is a standard Multics data type. A list of all Multics data types appears in the Programmer's Reference Manual. The include file std_descriptor_types.incl.pl1 defines symbolic names for these types.

packed
is "1"b if the data is packed.

pad
is reserved for expansion and must be all "0"b.

prec_or_length
is the arithmetic precision or string length of the data, as appropriate.

scale
is the arithmetic scale factor of the data, or zero if the data is not arithmetic.

picture_image_ptr
for picture data, is a pointer to the picture image block for the picture, otherwise it is ignored. A picture image block is a structure in the runtime symbol table. Only PL/I and the Multics debuggers know how to access it, so user programs should not try to convert to or from pictures using this entry.

Entry: assign_$assign_round_

This entry assigns a source value to a target value, but always rounds. Otherwise it is identical to assign_.

Entry: assign_$assign_truncate_

This entry is identical to assign_ except that it always truncates.

Name: bcd_to_ascii_

The bcd_to_ascii_ subroutine performs isomorphic (one-to-one reversible) conversion from BCD to ASCII.

USAGE

```
dcl bcd_to_ascii_ entry (bit(*), char(*));

call bcd_to_ascii_ (bcd_in, ascii_out);
```

ARGUMENTS

bcd_in
: is a bit string that represents the BCD characters to convert. (Input)

ascii_out
: is the lower case ASCII equivalent of the input string. (Output)

NOTES

BCD must be aligned on a 6-bit BCD character boundary.

Name: before_journal_manager_

The before_journal_manager_ subroutine provides the means to manipulate, and obtain information about, before journals. Before journals are used to store before images of protected data management (DM) files, for the purpose of rolling back modifications to these files in the event of failure.

See the section entitled "Multics Data Management" in the *Programmer's Reference Manual,* Order No. AG91, for a complete description of before journals and their use.

Entry: before_journal_manager_$close_bj

This entry point closes the specified before journal, making it unavailable to the current process. A journal can be opened more than once in a process, in which case the same opening id is returned for each open request. In that case, the close operation merely decreases by one the number of journal openings in the process. If a close_bj request is issued by a process on a journal while the process still has an active transaction in that journal, the journal cannot be closed and an error code is returned to the caller. If the journal to be closed was the default before journal for the process, the before journal which was last opened in the process (if any) becomes the default before journal (see "Notes" under the set_default_bj entry).

USAGE

```
declare before_journal_manager_$close_bj entry (bit(36) aligned, fixed
      bin(35));

call before_journal_manager_$close_bj (bj_opening_id, code);
```

ARGUMENTS

bj_opening_id
: is the opening identifier of the before journal. (Input)

code
is a standard system error code. (Output)

Entry: before_journal_manager_$create_bj

This entry point creates a before journal file as specified by the input arguments.

USAGE

```
declare before_journal_manager_$create_bj entry (char(*), char(*), fixed
      bin, fixed bin, fixed bin(35));

call before_journal_manager_$create_bj (dir_name, entry_name,
      n_control_intervals, control_interval_size, code);
```

ARGUMENTS

dir_name
is the pathname of the directory in which the before journal is to be created. (Input)

entry_name
is the entry name of the before journal to be created. The .bj suffix must be included. (Input)

n_control_intervals
is the size of the journal expressed in the number of control intervals. (Input) A before journal is a circular file; when information is no longer useful (i.e., before images for committed or aborted transactions), it will be overwritten, allowing the space to be reused. In estimating the size of a journal, you should consider the number of transactions to be using the journal simultaneously, as well as their profiles, i.e., their length in time and the rate at which they modify data, to optimize performance.

control_interval_size
is the size of the before journal control interval in number of bytes. (Input) The size is currently fixed at 4096.

code
: is a standard system error code. (Output)

Entry: before_journal_manager_$get_bj_path_from_oid

This entry point returns the directory pathname and the entry name of the specified before journal. For this operation to be successful, the before journal must be open in the current process.

If a zero code is returned, the operation is successful and the dir_name and entry_name arguments are set to the proper values. If a nonzero code is returned, the operation did not succeed and the values of dir_name and entry_name are left unchanged.

USAGE

```
declare before_journal_manager_$get_bj_path_from_oid entry (bit(36)
      aligned, char(*), char(*), fixed bin(35));

call before_journal_manager_$get_bj_path (bj_oid, dir_name, entry_name,
      code);
```

ARGUMENTS

bj_oid
: is the opening identifier of the before journal for which the pathname is requested. (Input)

dir_name
: is the pathname of the directory in which the before journal resides. (Output)

entry_name
: is the entry name of the before journal. (Output)

code
: is a standard system error code. (Output)

Entry: before_journal_manager_$get_default_bj

This entry point returns the opening identifier of the before journal to be used as the default in those cases where a before journal specification is expected but not supplied. The rules for determining this default before journal are described in "Notes" under the set_default_bj entry point. If the journal which is to serve as the default before journal is not open at the time of this call, it is opened automatically.

USAGE

```
declare before_journal_manager_$get_default_bj entry (bit(36) aligned,
     fixed bin(35));

call before_journal_manager_$get_default_bj (bj_oid, code);
```

ARGUMENTS

bj_oid
: is the opening identifier of the current default before journal. (Output)

code
: is a standard system error code. (Output)

Entry: before_journal_manager_$open_bj

This entry point makes the before journal specified by the pathname, ready for use by any transaction of the current process. A process may have several before journals open at the same time, and may also have the same journal opened more than one time. When a transaction is started, one of the open journals must be associated with the transaction, if the transaction needs a before journal. One can expect that in most cases, a process will open only one before journal, which will be used by all its transactions.

This entry may also change the default before journal for the process to the newly opened journal (see "Notes" under set_default_bj).

USAGE

```
declare before_journal_manager_$open_bj entry (char(*), char(*), bit(36)
     aligned, fixed bin(35));

call before_journal_manager_$open_bj (dir_name, entry_name,
     bj_opening_id, code);
```

ARGUMENTS

dir_name
: is the pathname of the directory in which the before journal to be opened resides. (Input)

entry_name
: is the entry name of the before journal to be opened. The .bj suffix must be included (Input)

bj_opening_id
: is the opening identifier of the journal. (Output) This specifier must be used subsequently by the current process to identify this journal.

code
    is a standard system error code. (Output)

NOTES

When a before journal is opened, it is remembered in a per system table containing the pathnames and unique identifiers of all before journals opened in the system. This table is used after a system crash to determine which journals must be reopened and examined in order to perform a rollback operation. To preserve the integrity of this table, it is written out to disk automatically each time it is updated with a newly opened journal.

If a process opens the same before journal more than one time, the opening identifier received from the open_bj will be the same for each call. The process must close a before journal the same number of times it opens it, to render the journal inaccessible through the same opening identifier.

Entry: before_journal_manager_$set_default_bj

This entry point causes the specified before journal to become the default before journal. When no before journal is explicitly specified by the user at the beginning of a transaction, the default before journal for the process will be assigned to the transaction. The default before journal must be one of the before journals open in the process.

USAGE

```
declare before_journal_manager_$set_default_bj entry (bit(36) aligned,
     fixed bin(35));

call before_journal_manager_$set_default_bj (bj_opening_id, code);
```

ARGUMENTS

bj_opening_id
    is the opening identifier of the before journal. (Input)

code
    is a standard system error code. (Output)

NOTES

Several before_journal_manager_ entries expect an opening id to specify which before journal to use. If bj_opening_id is null, the following default assignments are attempted, in the order in which they are mentioned below, until one of them succeeds:

- The current default before journal in this process, if there is one; otherwise,
- The most recently open before journal among those that are still open, if there is one; otherwise,
- The system before journal. If the system before journal has not been opened yet in the current process, it is automatically opened.

Entry: before_journal_manager_$set_transaction_storage_limit

This entry point sets the maximum number of bytes a single transaction may use.

USAGE

```
declare before_journal_manager_$set_transaction_storage_limit entry
      (char (*), char (*), fixed bin (35), fixed bin (35));

call before_journal_manager_$set_transaction_storage_limit (dir_name,
      entryname, storage_limit, code);
```

ARGUMENTS

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

entryname
: is the entryname of the before journal. (Input)

storage_limit
: is the maximum number of bytes a single transaction may use in the before journal. (Input)

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

Name: bit_offset_

The bit_offset_ subroutine returns the bit offset (relative to the base of the segment) of the bit located by the supplied pointer value.

USAGE

```
declare bit_offset_ entry (ptr) returns (fixed bin (24)) reducible;

bit_offset = bit_offset_ (pointer_value);
```

ARGUMENTS

pointer_value
  is a pointer whose bit offset is to be determined. (Input)

bit_offset
  is the bit offset of the supplied pointer. (Output)

NOTES

The first bit in a segment has a bit offset of zero.

Name: cb_menu_

The cb_menu_ subroutine allows a COBOL program to use the Multics menu facility (menu_). Through cb_menu_ a COBOL program may create a menu object, display the menu, and get a user-entered selection from a menu. Once a menu object has been created, the COBOL program can use this menu object by referencing it via a menu-id returned to the caller when the menu object was created or when a stored menu object was retrieved.

The functionality available is provided through the various entry points described below.

Entry: cb_menu_$create

Utilized to create a menu-object. Returns a menu-id which may be subsequently used by other entry points.

USAGE

declarations:

```
01  choices-table.
    02 choices      PIC X(n1) OCCURS (m1) TIMES.
01  headers-table.
    02 headers      PIC X(n2) OCCURS (m2) TIMES.
01  trailers-table.
    02 trailers     PIC X(n3) OCCURS (m3) TIMES.
01  keys-table.
    02 keys         PIC X(1)  OCCURS (m4) TIMES.

01  menu-format.
    02 menu_version USAGE IS COMP-6
    02 constraints USAGE IS COMP-6
       03  max-width.
       03  max-height.
    02  no-of-columns USAGE IS COMP-6.
    02  flags.
       03  center-headers  PIC 9(1).
       03  center-trailers PIC 9(1).
    02  pad-char  PIC X(1).

01  menu-needs  USAGE IS COMP-6.
    02  lines-needed.
    02  width-needed.
    02  no-of-options.

77  menu-id   USAGE IS COMP-6.
77  ret-code  USAGE IS COMP-6.


call "cb_menu_$create" USING choices-table, headers-table,
      trailers-table, menu-format, keys-table, menu-needs,
      menu-id, ret-code.
```

STRUCTURE ELEMENTS

choices-table
: is a table of elementary data items which are the text of the options that the user wishes to display in the menu. n1 is the length, in characters, of the longest character string comprising the text of an option. m1 is the extent of the table, i.e., the number of options in the menu being described. This table must be at least of extent 1.

headers-table
is a table of elementary data items to be displayed at the top of the menu. (Input) n2 is the length, in characters, of the longest header specified. m2 is the extent of the table, i.e., the number of headers (lines) desired. At least one header must be specified (if the first header is set to space(s), no headers will be used).

trailers-table
is an table of trailers (displayed immediately below the menu). (Input) n3, m3, are analogous to n2, m2 respectively.

menu-format
is a group item defining the format of the menu being created. (Input)

In the COBOL program the caller is responsible for setting the following elementary data items:

```
menu-version       the version number of the menu facility.
                   (only version 1 is currently defined)
max-width          maximum width of the window on which the
                   menu is to be displayed.
max-height         maximum height of window on which the
                   menu is to be displayed.
no-of-columns      number of columns to be used to display
                   the options.
center-headers     0 or 1; 0 = no, 1 = yes.
center-trailers    0 or 1 (same as center-headers)
```

keys-table
is a table (maximum value of m4 is 61) that identifies the keystroke to be associated with each choice. (Input) This table must be at least as long as the number of choices in the menu. Each element in the table must be unique.

menu-needs
a group item that contains menu related information on successful execution of call. (Output)

```
Returned information:

lines-needed       the number of lines required
                   to display the menu.
width-needed       the number of columns needed
                   to display the menu.
no-of-options      the number of options defined
                   in the menu.
```

menu-id
the menu-object identifier (i.e., it is the menu object "pointer".) (Output) It must not be altered in any way by the application program.

ret-code
return code. (Output) (See Appendix B.)

Entry: cb_menu_$delete

Deletes a menu object from a given value segment.

USAGE

declarations:

```
     77   dir-name       PIC X(168).
     77   entry-name     PIC X(32).
     77   name-of-menu   PIC X(32).
     77   ret-code       USAGE IS COMP-6.

     call "cb_menu_$delete" USING dir-name, entry-name, name-of-menu,
            ret-code.
```

STRUCTURE ELEMENTS

dir-name
pathname of the directory containing the menu object. (Input)

entry-name
entry name of value segment containing the menu object. (Input) The suffix "value" need not be specified.

name-of-menu
name used to identify the menu object when the menu object was stored. (Input)

ret-code
return code. (Output) (See Appendix B.)

Entry: cb_menu_$describe

Returns information about a menu object. It returns the number of options in the menu, the number of lines and number of columns required to display the menu. It is primarily used to determine if the menu can be displayed in a given window.

USAGE

declarations:

```
01  menu-needs  USAGE IS COMP-6.
    02  lines-needed.
    02  width-needed.
    02  no-of-options.

77  menu-id     USAGE IS COMP-6.
77  ret-code    USAGE IS COMP-6.

call "cb_menu_$describe" USING menu-id, menu-needs, ret-code.
```

STRUCTURE ELEMENTS

menu-id
the menu identifier returned by cb_menu_$create (or cb_menu_$retrieve in cases where the menu object has been stored). (Input)

menu-needs
a group item that contains menu related information on successful execution of call. (Output)

Returned information:

lines-needed	the number of lines needed to display the menu.
width-needed	the number of columns needed to display the menu.
no-of-option	the number of options defined in the menu.

ret-code
return code. (Output) (See Appendix B.)

Entry: cb_menu_$destroy

Used to free storage of a menu (not to be confused with cb_menu_$delete, which is used to delete the menu object from a value segment). Destroying the menu has no effect on the screen contents.

USAGE

declarations:

```
77  menu-id   USAGE IS COMP-6.
77  ret-code  USAGE IS COMP-6.

call "cb_menu_$destroy" USING  menu-id, ret-code.
```

STRUCTURE ELEMENTS

menu-id
menu identifier returned by cb_menu_$create or cb_menu_$retrieve. (Input/Output) (If usage-mode is 0 (see cb_menu_$init2) this operand will be ignored.) Set to an invalid value on return to prevent the old menu-id from being accidentally used.

ret-code
return code. (Output) (See Appendix B.)

Entry: cb_menu_$display

Invoked to display a menu in a given window.

USAGE

declarations:

```
77  window-id    USAGE IS COMP-6.
77  menu-id      USAGE IS COMP-6.
77  ret-code     USAGE IS COMP-6.

call "cb_menu_$display" USING  window-id, menu-id, ret-code.
```

STRUCTURE ELEMENTS

window-id
a window identifier returned by cb_window_$create entry point. (Input) If usage-mode = 0 this operand will be ignored (see cb_menu_$init2).

menu-id
    menu identifier returned when the menu object was created or retrieved. (Input)

ret-code
    return code. (Output) (See Appendix B.)

Entry: cb_menu_$get_choice

Returns the choice made by the user, i.e., a number representing either the menu item chosen or the function key (or its equivalent escape sequence) entered.

USAGE

```
declarations:

     77    function-key-info  PIC X(nl).
     77    window-id          USAGE IS COMP-6.
     77    menu-id            USAGE IS COMP-6.
     77    fkeys              USAGE IS COMP-6.
     77    selection          USAGE IS COMP-6.
     77    ret-code           USAGE IS COMP-6.

     call "cb_menu_$get_choice" USING window-id, menu-id,
          function-key-info, fkeys, selection, ret-code.
```

STRUCTURE ELEMENTS

window-id
    a window identifier returned by the cb_window_$create entry point. (Input) If usage-mode = 0 this operand will be ignored (see cb_menu_$init2).

menu-id
    menu identifier returned by cb_menu_$create or cb_menu_$retrive. (Input)

function-key-info
a character elementary data item (n1 as required) used to specify the role of function keys (if they exist for the terminal being used) or an equivalent set of escape sequences if the terminal does not have function keys or not the function keys required by the application. (Input) The objective is to let the application use the terminal's function keys if possible, else specify key sequences to be used to simulate function keys. Each character in the string corresponds to one function key. If the character is a space, then it is not relevant if the corresponding function key exists or not. If the character is not a space, that character will be used to simulate a function key if the terminal does not have function keys. If the terminal does not have a function key for every non-space character in the string, then function keys will be simulated. Thus, the string " ?p q" means that the caller does not care whether the terminal has function key 0 or 3, but the caller does wish to use function keys 1,2, and 4. If any of these 3 function keys is not present on the terminal, then esc-? will substitute for F1, esc-p will substitute for F2, and esc-q will substitute for F4.

fkeys
fkeys = 1 user entered a function key or escape sequence fkeys = 0 user selected an option (Output)

selection
is a number representing the choice made by the user. (Output) If the user has chosen an option, it is a number between 1 and the highest defined option. If the user has entered a function key, or escape sequence simulating a function key, it is the number associated with the function key.

ret-code
return code. (Output) (See Appendix B.)

Entries: cb_menu_$init1, cb_menu_$init2

These must be the first calls made to the menu manager. They set up the necessary environment for the menu application and return information concerning the user I/O window.

USAGE

```
declarations:

     inter    code
     integer  usage-mode

     call cb_menu_$init1

     call cb_menu_$init2 (usage-mode, user-window-lines,
          user-window-columns, user-window-id, ret-code)
```

STRUCTURE ELEMENTS

usage-mode
usage-mode = 0 means that the caller does not wish to do any explicit window management. (Input) When he/she wishes to display a menu, the window required will be automatically created. This means that the application will operate in a two window mode, the window containing the menu, and the user_io window. Both windows will be managed automatically for the user. If the user specifies this mode, all calls to the cb_window_ subroutine will be ignored and will return an appropriate error code. See Error Code Handling, below. All calls to the cb_menu_ subroutine that require a window identifier will ignore the user provided window-id.

usage-mode = 1 means that the user wishes to define the number and characteristics of the windows to be used in the application. Thus, calls to cb_window_ will be supported and, for the entry points of cb_menu_ that require a window identifier, the caller must use a legal window-id (returned by cb_window_$create).

user-window-lines
the number of physical lines (rows) of the user i/o window when cb_menu_$init is called (which must be the first cb_menu_ call in the application.) Undefined if usage-mode = 0. (Output)

user-window-columns
the number of columns of the user i/o window at time that cb_menu_$init is called (see immediately above). (Output) Undefined if usage-mode = 0.

user-window-id
window identifier of the user i/o window. (Output) Undefined if usage-mode = 0.

ret-code
return code. (Output) (See Appendix B.)

Entry: cb_menu_$list

Used to list the menu object(s), stored in value segment. The menu objects selected are those that match the string input by the caller.

USAGE

declarations:

```
     01  matched-names.
         02  no-of-matches     USAGE IS COMP-6.
         02  menu-names        PIC X(32)  OCCURS (ml) TIMES.

     77  dir-name        PIC X(168).
     77  entry-name      PIC X(32).
     77  match-string    PIC X(32).
     77  ret-code        USAGE IS COMP-6.

     call "cb_menu_$list" USING dir-name, entry-name, match-string,
           matched-names, ret-code.
```

STRUCTURE ELEMENTS

dir-name
pathname of directory containing the menu object. (Input)

entry-name
entry name of value segment containing the menu object. (Input) The suffix "value" need not be specified.

match-string
a character elementary data item that is to be used as the selection criteria for determining what menu object, if any, is contained in the specified value segment that match (or contain) this string. (Input)

no-of-matches
the number of matches found. (Output) If none, then it is 0.

menu-names
On return, contains the names of all menu objects, in the specified value segment, that match the character string match-string. (Output) Note, if ml is not large enough to contain all the names, only ml names will be returned.

ret-code
return code. (Output) (See Appendix B.)

Entry: cb_menu_$retrieve

Used to retrieve a menu object previously stored via the cb_menu_$store subroutine.

USAGE

declarations:

```
    77  dir-name        PIC X(168).
    77  entry-name      PIC X(32).
    77  name-of-menu    PIC X(32).
    77  menu-id         USAGE IS COMP-6.
    77  ret-code        USAGE IS COMP-6.

    call "cb_menu_$retrieve" USING dir-name, entry-name, name-of-menu,
          menu-id, ret-code.
```

STRUCTURE ELEMENTS

dir-name
pathname of the directory containing the menu object. (Input)

entry-name
entry name of value segment containing menu object. (Input) The suffix "value" need not be specified.

name-of-menu
name of the menu object used when the object was stored. (Input)

menu-id
is the menu id returned by the call. (Output)

ret-code
return code. (Output) (See Appendix B.)

Entry: cb_menu_$store

Used to store a menu object in a specified value segment.

USAGE

declarations:

```
     77  dir-name       PIC X(168).
     77  entry-name     PIC X(32).
     77  name-of-menu   PIC X(32).
     77  create-seg     USAGE IS COMP-6.
     77  menu-id        USAGE IS COMP-6.
     77  ret-code       USAGE IS COMP-6.

     call "cb_menu_$store" USING dir-name, entry-name, name-of-menu,
           create-seg, menu-id, ret-code.
```

STRUCTURE ELEMENTS

dir-name
pathname of directory into which the menu object is to be placed. (Input)

entry-name
entry name of value segment into which menu object is to be placed. (Input) The suffix "value" need not be specified.

name-of-menu
is the name to be assigned to the stored menu object. (Input)

create-seg
create-seg = 0 means do not store if value segment identified by entry-name does not already exist. (Input) create-seg = 1 means create value segment, if it does not already exist, and store menu object in it.

menu-id
is the menu object identifier returned by cb_menu_$create or cb_menu_$retrieve. (Input)

ret-code
return code. (Output) (See Appendix B.)

Entry: cb_menu_$terminate

Must be the last call to the menu manager in the menu application.

USAGE

declarations: none

```
call "cb_menu_$terminate".
```

STRUCTURE ELEMENTS

There are no arguments.

Name: cb_window_

This is the basic video interface subroutine to be used by COBOL to create/destroy/change windows. (If usage-mode = 0 (see cb_menu_$init2) this subroutine should not be called.)

Its facilities are available through the following entry points.

Entry: cb_window_$change

This entry points provides a facility for changing the size of an existing window. The size of a window can always be "shrunk", however it can be increased only it does not overlap with another defined window. (If usage-mode = 0 (see cb_menu_$init2) this entry point should not be called.)

USAGE

declarations:

```
77  window-id   USAGE IS COMP-6.
77  first-line  USAGE IS COMP-6.
77  height      USAGE IS COMP-6.
77  ret-code    USAGE IS COMP-6.

call "cb_window_$change" USING window-id, first-line, height,
     ret-code.
```

STRUCTURE ELEMENTS

window-id
    window identifier returned by cb_window_$create. (Input)

first-line
    new first line number for the window being changed. (Input) A positive value.

height
    new height for the window being changed. (Input) A positive value.

ret-code
    return code. (Output) (See Appendix B.)

Entry: cb_window_$clear_window

Used to clear a specified window.

USAGE

```
declarations:

     77  window-id        USAGE IS COMP-6.
     77  ret-code         USAGE IS COMP-6.

     call "cb_window_$clear_window" USING  window-id, ret-code.
```

STRUCTURE ELEMENTS

window-id
    the window identifier (returned by cb_window_$create) of the window to be cleared. (Input)

ret-code
    return code. (Output) (See Appendix B.)

Entry: cb_window_$create

This entry is used to create a new window on the terminal screen. (If usage-mode = 0 (see cb_menu_$init2) this entry point should not be called.)

USAGE

declarations:

```
     77  switch-name     PIC X(32).
     77  first-line      USAGE IS COMP-6.
     77  height          USAGE IS COMP-6.
     77  window-id       USAGE IS COMP-6.
     77  ret-code        USAGE IS COMP-6.

     call "cb_window_$create" USING first-line, height, switch-name,
           window-id, ret-code.
```

STRUCTURE ELEMENTS

first-line
: is the line number where the window is to start. (Input)

height
: the number of lines used by the window, i.e., its height. (Input)

switch-name
: the name that the caller wishes to associate with the switch. (Input)

window-id
: the returned id of the window just created. (Output) It must not be altered in any way by the application program.

ret-code
: return code. (Output) (See Appendix B.)

Entry: cb_window_$destroy

Used to destroy a previously created window. (If usage-mode = 0 (see cb_menu_$init2) this entry point should not be called.)

USAGE

declarations:

```
77  window-id   USAGE IS COMP-6.
77  ret-code    USAGE IS COMP-6.

call "cb_window_$destroy"  USING  window-id, ret-code.
```

STRUCTURE ELEMENTS

window-id
: window identifier (returned by the cb_window_$create). (Input/Output) It is reset to an illegal value by this call.

ret-code
: return code. (Output) (See Appendix B.)

COBOL MENU APPLICATION EXAMPLES

In the following two COBOL examples, a "Message" menu application is created that allows you to display, print, discard, or forward messages. Example 1 is a simple COBOL program that interfaces with the Multics menu manager via the cb_menu_ routine. Note in example 1 that window management functions are called automatically through arguments in the ft_menu_$init2 subroutine.

Example 2 is a COBOL program that interfaces with the Multics menu manager through the cb_menu_routine; in example 2, however, window management functions are performed by the cb_window_ routine.

EXAMPLE 1:

In this example, all window management is done automatically.

```
   /*********************************************************************
   *        A simple COBOL program interfacing with the Multics        *
   *        menu manager via the cb_menu_ routine.                     *
   *********************************************************************

    CONTROL DIVISION.
    DEFAULT GENERATE AGGREGATE DESCRIPTORS.
    IDENTIFICATION DIVISION.

    PROGRAM-ID.
 cbtest1.

    AUTHOR.
R. I.
```

```
     ENVIRONMENT DIVISION.
     CONFIGURATION SECTION.
     SOURCE-COMPUTER.
 Multics.

     OBJECT-COMPUTER.
 Multics.

    /*****************************************************************

     DATA DIVISION.

     WORKING-STORAGE SECTION.

     01 choices-table.
        02 choices  PIC X(15) OCCURS 6 TIMES.
     01 headers-table.
        02 headers  PIC X(14) OCCURS 1 TIMES.
     01 trailers-table.
02 trailers PIC X(32) OCCURS 1 TIMES.
     01 keys-table.
02 keys     PIC X(1)  OCCURS 6 TIMES.

     01 menu-format.
02 menu-version  USAGE IS COMP-6 VALUE 1.
02 constraints   USAGE IS COMP-6.
   03 max-width  VALUE 79.
   03 max-height VALUE 10.
02 no-of-columns USAGE IS COMP-6 VALUE 2.
02 flags.
   03 center-headers    PIC 9(1) VALUE 1.
   03 center-trailer    PIC 9(1) VALUE 1.
        02 padder   PIC X(1) VALUE "-".

     01 menu-needs     USAGE IS COMP-6.
02 lines-needed.
02 width-needed.
02 no-of-options.

     77 dir-name             PIC X(168).
     77 entry-name           PIC X(32).
     77 menu-name            PIC X(32).
     77 function-key-info    PIC X(1) VALUE "q".
     77 me                   PIC X(7) VALUE "cbtest1".

     77 menu-id
              USAGE IS COMP-6.
     77 ret-code             USAGE IS COMP-6.
     77 window-id            USAGE IS COMP-6.
```

```
    77 fkeys                  USAGE IS COMP-6.
    77 option                 USAGE IS COMP-6.
    77 easy-mode              USAGE IS COMP-6 VALUE zero.
    77 user-window-lines      USAGE IS COMP-6.
    77 user-window-columns    USAGE IS COMP-6.
    77 user-window-id         USAGE IS COMP-6.
    77 create-seg             USAGE IS COMP-6.

    77 keys-not-unique        USAGE IS COMP-6.
    77 too-few-keys           USAGE IS COMP-6.
    77 bad-arg                USAGE IS COMP-6.

   /*****************************************************************
    PROCEDURE DIVISION.
   *  The call to the cv_error_$name are used to collect the code for
   *  certain error messages that are of interest this application.
   *  Once these codes are retrieved the occurrence of that error can
   *  be easily tested for.

    START-IT.
          CALL "cb_menu_$init1".
CALL "cb_menu_$init2" USING easy-mode, user-window-lines,
   -    user-window-columns, user-window-id, ret-code.

   *  The calls to cb_menu_$init1 & 2 MUST be the first calls to cb_menu_.
   *  They set up the appropriate environment for the menu application.

        IF ret-code EQUAL TO zero GO TO NEXT-ERR-CODE.
CALL "com_err_" USING ret-code, me, "Internal error.
             Could not set up appropriate environment.".
GO TO STOP-IT.

CALL "cv_error_$name" USING "menu_et_$keys_not_unique",
   -    keys-not-unique, ret-code.
call "ioa_" USING "Error code for keys-not-unique = ^d", keys-not-unique.
IF ret-code EQUAL TO zero GO TO NEXT-ERR-CODE.
CALL "com_err_" USING ret-code, me, " (calling cv_error_$name)".
GO TO STOP-IT.
    NEXT-ERR-CODE.
CALL "cv_error_$name" USING "error_table_$bad_arg", bad-arg, ret-code.
IF ret-code EQUAL TO zero GO TO LAST-ERR-CODE.
CALL "com_err_" USING ret-code, me , " (calling cv_error_$name)".
GO TO STOP-IT.
    LAST-ERR-CODE.
CALL "cv_error_$name" USING "menu_et_$too_few_keys", too-few-keys,
   -    ret-code.
IF ret-code EQUAL TO zero GO TO SET-UP.
CALL "com_err_" USING ret-code, me, " (calling cv_error_$name)".
GO TO STOP-IT.
    SET-UP.
```

```
        MOVE 1 TO menu-version.
MOVE "Display Message" TO choices(1).
MOVE "Print Message" TO choices(2).
MOVE "Discard Message" TO choices(3).
        MOVE "Forward Message" TO choices(4).
MOVE "Reply Message" TO choices(5).

MOVE "List Messages" TO choices(6).
MOVE " MULTICS MAIL " TO headers(1).
MOVE "Press F1 or enter esc-q to quit" TO trailers(1).
MOVE "1" TO keys(1).
        MOVE "2" TO keys(2).
MOVE "3" TO keys(3).
        MOVE "4" TO keys(4).
MOVE "5" TO keys(5).
MOVE "6" TO keys(6).


    MENU-CREATE.
        DISPLAY  choices-table.
DISPLAY menu-version.
CALL "cb_menu_$create" USING choices-table, headers-table,
   -    trailers-table, menu-format, keys-table, menu-needs,
   -    menu-id, ret-code.

   *  This call creates a menu object and return the menu object
   *  identifier. This menu object is referenced as "menu-id".
IF ret-code EQUAL TO zero GO TO STORE-MENU.
CALL "com_err_" USING ret-code, me, " (calling cb_menu_$create)".
GO TO STOP-IT.
   STORE-MENU.
MOVE ">udd>m>ri" TO dir-name.
MOVE "menus_seg" TO entry-name.
MOVE "cb_read_mail_menu" TO menu-name.
        MOVE 1 TO create-seg.
CALL "cb_menu_$store" USING dir-name, entry-name, menu-name,
   -    create-seg, menu-id, ret-code.
IF ret-code EQUAL TO zero GO TO DISPLAY-MENU.
CALL "com_err_" USING ret-code, me, "(calling cb_menu_$store)".
GO TO STOP-IT.
    DISPLAY-MENU.
CALL "cb_menu_$display" USING window-id, menu-id, ret-code.

   *   This call displays the menu in its own window at top of screen.
   *   Since the usage-mode was set to 0, the program does not have to
   *   create the window before calling cb_menu_$display.
   *   The window-id argument is ignored.

IF ret-code EQUAL TO zero GO TO GET-CHOICE.
CALL "com_err_" USING ret-code, me, "Internal error.
```

```
        Menu could not be displayed."
 GO TO STOP-IT.
     GET-CHOICE.

   *    Defines the function key requirements, i.e.,
   *      if the terminal has function key 1 (F1) then F1 will be used
   *    to "quit", otherwise "esc q" will be used to "quit".

 CALL "cb_menu_$get_choice" USING window-id, menu-id,
   -    function-key-info, fkeys, option, ret-code.
 IF ret-code EQUAL TO zero GO TO TEST-FKEY.
 CALL "com_err_" USING ret-code, me, "Internal error. While getting
        user's choice.".
 GO TO STOP-IT.
     TEST-FKEY.
 IF fkeys EQUAL TO 1
    CALL "ioa_" USING "Exiting at your request."
    GO TO STOP-IT
 ELSE
    CALL "ioa_" USING "You chose option ^d.", option
    GO TO GET-CHOICE.

     STOP-IT.
 CALL "cb_menu_$terminate".
   *   cb_menu_$terminate MUST be the last call to cb_menu_ in the
   *   application. It terminates the environment set up cb_menu_$init.

       EXIT PROGRAM.
```

EXAMPLE 2:

In this example, COBOL interfaces with the Multics menu manager and the Multics window manager via the cb_menu_ and cb_window_ subroutines. fif

```
/******************************************************************
* A simple COBOL program interfacing with the Multics *
* menu manager and window manager via the cb_menu_ and *
* cb_window_ routines, respectively. *
******************************************************************

 CONTROL DIVISION.
 DEFAULT GENERATE AGGREGATE DESCRIPTORS.
 IDENTIFICATION DIVISION.

 PROGRAM-ID.
cbtest2.

 AUTHOR.
R. !.
```

```
     ENVIRONMENT DIVISION.
     CONFIGURATION SECTION.
     SOURCE-COMPUTER.
 Multics.
     OBJECT-COMPUTER.
 Multics.

    /*****************************************************************

     DATA DIVISION.

     WORKING-STORAGE SECTION.

     01 choices-table1.
        02 choices1 PIC X(9) OCCURS 2 TIMES.
     01 choices-table2.
        02 choices2 PIC X(15) OCCURS 6 TIMES.
     01 choices-table3.
        02 choices3 PIC X(21) OCCURS 4 TIMES.
     01 headers-table.
        02 headers PIC X(23) OCCURS 1 TIMES.
     01 trailers-table.  02 trailers PIC X(52) OCCURS 1 TIMES.
     01 keys-table.  02 keys PIC X(1) OCCURS 6 TIMES.

     01  menu-format.   02  menu-version  USAGE  IS  COMP-6  VALUE  1.   02
constraints USAGE IS COMP-6.
   03 max-width VALUE 80.
   03 max-height VALUE 10.  02 no-of-columns USAGE IS COMP-6 VALUE 2.
02 flags.
   03 center-headers PIC 9(1) VALUE 1.
   03 center-trailer PIC 9(1) VALUE 1.
        02 padder PIC X(1) VALUE "-".

     01   menu-needs1   USAGE   IS   COMP-6.   02   lines-needed1.   02
width-needed1.  02 no-of-options1.

     01   menu-needs2   USAGE   IS   COMP-6.   02   lines-needed2.   02
width-needed2.  02 no-of-options2.

     01 menu-needs3 USAGE IS COMP-6.  02 lines-needed3.
```

```
02 width-needed3.
02 no-of-options3.

    77 dir-name PIC X(168).
    77 entry-name PIC X(32).
    77 menu-name PIC X(32).
    77 function-key-info PIC X(2) VALUE "qf".
    77 me PIC X(7) VALUE "cbtest2".
    77 switch-name PIC X(32).

    77 lines-needed USAGE IS COMP-6.
    77 first-line USAGE IS COMP-6.

    77 height USAGE IS COMP-6.
    77 menu-id USAGE IS COMP-6.
    77 menu-id1 USAGE IS COMP-6.
    77 menu-id2 USAGE IS COMP-6.
    77 menu-id3 USAGE IS COMP-6.
    77 ret-code USAGE IS COMP-6.
    77 curr-window-id USAGE IS COMP-6.
    77 window-id USAGE IS COMP-6.
    77 window-id1 USAGE IS COMP-6.
    77 window-id2 USAGE IS COMP-6.
    77 fkeys USAGE IS COMP-6.
    77 option USAGE IS COMP-6.
    77 do-it-yourself USAGE IS COMP-6 VALUE 1.
    77 user-window-lines USAGE IS COMP-6.
    77 user-window-columns USAGE IS COMP-6.
    77 user-window-id USAGE IS COMP-6.
    77 create-seg USAGE IS COMP-6.

    77 bad-window-id USAGE IS COMP-6.
    77 nonexistent-window USAGE IS COMP-6.
    77 insuff-room-for-window USAGE IS COMP-6.

   /****************************************************************

    PROCEDURE DIVISION.

   * The call to the cv_error_$name are used to collect the code for
   * certain error messages that are of interest this application.
   * Once these codes are retrieved the occurrence of that error can
   * be easily tested for.

    START-IT.
 CALL "cv_error_$name" USING "video_et_$bad_window_id",
   - bad-window-id, ret-code.
 IF ret-code EQUAL TO zero GO TO NEXT-ERR-CODE.
 CALL "com_err_" USING ret-code, me, " (calling cv_error_$name)".
 GO TO STOP-IT.
```

```
    NEXT-ERR-CODE.
CALL "cv_error_$name" USING "video_et_$nonexistent_window",
   - nonexistent-window, ret-code.
IF ret-code EQUAL TO zero GO TO LAST-ERR-CODE.
CALL "com_err_" USING ret-code, me , " (calling cv_error_$name)".
GO TO STOP-IT.
    LAST-ERR-CODE.
CALL "cv_error_$name" USING "video_et_$insuff_room_for_window",
   - insuff-room-for-window, ret-code.
IF ret-code EQUAL TO zero GO TO SET-UP.
CALL "com_err_" USING ret-code, me, " (calling cv_error_$name)".
GO TO STOP-IT.
    SET-UP.
MOVE "Read Mail" TO choices1(1).
MOVE "Send Mail" TO choices1(2).

MOVE "Display Message" TO choices2(1).
        MOVE "Print Message" TO choices2(2).
MOVE "Discard Message" TO choices2(3).
MOVE "Forward Message" TO choices2(4).
        MOVE "Reply Message" TO choices2(5).
        MOVE "List Messages" TO choices2(6).

MOVE "Send New Message" TO choices3(1).
MOVE "Send Deferred Message" TO choices3(2).
        MOVE "Print Sent Message" TO choices3(3).
        MOVE "Save Sent Message" TO choices3(4).


MOVE "1" TO keys(1).
        MOVE "2" TO keys(2).
MOVE "3" TO keys(3).
        MOVE "4" TO keys(4).
MOVE "5" TO keys(5).
MOVE "6" TO keys(6).


CALL "cb_menu_$init1".
CALL "cb_menu_$init2" USING do-it-yourself, user-window-lines,
   - user-window-columns, user-window-id, ret-code.

   * The call to cb_menu_$init1 & 2 MUST be the first call to cb_menu_.
   * It sets up the appropriate environment for the menu application.
   * The application must do the window management, since
   * "do-it-youself" is set to 1.

        IF ret-code EQUAL TO zero GO TO CREATE-FIRST-MENU.
CALL "com_err_" USING ret-code, me, "Internal error.  Could not set up
         appropriate environment.".
GO TO STOP-IT.
```

```
    CREATE-FIRST-MENU.

   * Create first menu object.

MOVE "F1 (or esc-q) = quit" TO trailers(1).
MOVE "MULTICS MAIL" TO headers(1).
CALL "cb_menu_$create" USING choices-table1, headers-table,
   - trailers-table, menu-format, keys-table, menu-needs1,
   - menu-id1, ret-code.

IF ret-code EQUAL TO zero GO TO CREATE-SECOND-MENU.
CALL "com_err_" USING ret-code, me, " (calling cb_menu_$create)".
GO TO STOP-IT.
    CREATE-SECOND-MENU.

   * Create second menu object.

MOVE "F1 (or esc-q) = quit; F2 (or esc-f) = first menu" TO trailers(1).
MOVE "READ MAIL" TO headers(1).
CALL "cb_menu_$create" USING choices-table2, headers-table,
   - trailers-table, menu-format, keys-table, menu-needs2,
   - menu-id2, ret-code.
IF ret-code EQUAL TO zero GO TO CREATE-THIRD-MENU.
CALL "com_err_" USING ret-code, me, " (calling cb_menu_$create)".
GO TO STOP-IT.
    CREATE-THIRD-MENU.

   * Create third menu object.

MOVE "SEND MAIL" TO headers(1).
CALL "cb_menu_$create" USING choices-table3, headers-table,
   - trailers-table, menu-format, keys-table, menu-needs3,
   - menu-id3, ret-code.
IF ret-code EQUAL TO zero GO TO STORE-MENU.
CALL "com_err_" USING ret-code, me, " (calling cb_menu_$create)".
GO TO STOP-IT.

   STORE-MENU.
MOVE ">udd>m>ri" TO dir-name.
MOVE "menu_seg" TO entry-name.
MOVE "cb_test_menu_" TO menu-name.
        MOVE 1 TO create-seg.
CALL "cb_menu_$store" USING dir-name, entry-name, menu-name,
   - create-seg, menu-id1, ret-code.
IF ret-code EQUAL TO zero GO TO DISPLAY-IT.
CALL "com_err_" USING ret-code, me, "(calling cb_menu_$store)".
GO TO STOP-IT.
    DISPLAY-IT.
MOVE -1 TO curr-window-id.
```

```
    * Setting curr-wind-id to "-1" means that there is no current window
    * defined.
 MOVE menu-id1 TO menu-id.
 MOVE lines-needed1 TO lines-needed.

     DISPLAY-FIRST-MENU.

 PERFORM CHANGE-MENU THRU GOBACK.
    * The user i/o window has been "shrunk", the window for the first
menu
    * has been created, and the first menu has been displayed.
         MOVE window-id TO window-id1.
 IF ret-code EQUAL TO zero GO TO GET-IT.
 CALL "com_err_" USING ret-code, me, "Internal error.
       Menu could not be displayed."
 GO TO STOP-IT.
     GET-IT.
 PERFORM GET-CHOICE.
    * Get the user input.  Two values are returned.  (1) fkey.  If fkey
= 1,
    * then the user entered a function key (or its equivalent escape
    * sequence).  If fkey = 0 then the user has selected an option.  (2)
option.
    * If fkey = 1 then option is the function key number entered.  (F1 =
1,
    * F2 = 2, etc.).  If fkey = 0, then option is the option number
selected,
    * option = 1 means option 1 selected, etc.

 IF ret-code EQUAL TO zero GO TO TEST-FKEY.
 CALL "com_err_" USING ret-code, me, "Internal error.
       While getting user's choice.".
 GO TO STOP-IT.
     TEST-FKEY.
 IF fkeys EQUAL TO 1
    IF option EQUAL TO 1
      CALL "ioa_" USING "Exiting at your request."
      GO TO STOP-IT
    ELSE
             GO TO GET-IT
 ELSE
    IF option EQUAL TO 1
      MOVE menu-id2 TO menu-id
             MOVE lines-needed2 TO lines-needed
      PERFORM CHANGE-MENU THRU GOBACK
    ELSE
      MOVE menu-id3 TO menu-id
      MOVE lines-needed3 TO lines-needed
      PERFORM CHANGE-MENU THRU GOBACK.
 IF ret-code NOT EQUAL TO zero
```

```
  CALL "com_err_" USING ret-code, me, "Internal error.
       While trying to display menu."
  GO TO STOP-IT
ELSE
            MOVE window-id TO window-id2.
    NEXT-GET-IT.
PERFORM GET-CHOICE.
IF fkeys EQUAL TO zero GO TO CHOSE-OPTION.
IF option EQUAL TO 1
  CALL "ioa_" USING "Exiting at your request."
  GO TO STOP-IT
ELSE
  IF option GREATER 2
            GO TO NEXT-GET-IT
  ELSE
    MOVE menu-id1 TO menu-id
    MOVE lines-needed1 TO lines-needed
    GO TO DISPLAY-FIRST-MENU.
    CHOSE-OPTION.
CALL "ioa_" USING "You chose option ^d.", option.
GO TO NEXT-GET-IT.
    GET-CHOICE.
CALL "cb_menu_$get_choice" USING window-id, menu-id,
   - function-key-info, fkeys, option, ret-code.

    CHANGE-MENU.

   * Destroy the current menu window.
IF (curr-window-id ) EQUAL TO -1 GO TO CHANGE-USER-WIND.
CALL "cb_window_$destroy" USING curr-window-id, ret-code.
          IF ret-code EQUAL TO zero GO TO CHANGE-USER-WIND.
GO TO GOBACK.
    CHANGE-USER-WIND.
COMPUTE first-line = lines-needed + 1.
COMPUTE height = user-window-lines - lines-needed.
CALL "cb_window_$change" USING user-window-id, first-line, height,
   - ret-code.
IF ret-code EQUAL TO zero GO TO CREATE-NEW-WIND
        ELSE GO TO GOBACK.
    CREATE-NEW-WIND.
MOVE "menu-window" TO switch-name.
        MOVE 1 TO first-line.
CALL "cb_window_$create" USING first-line, lines-needed,
   - switch-name, window-id, ret-code.
IF ret-code EQUAL TO zero GO TO DISPLAY-MENU
ELSE GO TO GOBACK.
    DISPLAY-MENU.
        MOVE window-id TO curr-window-id.
CALL "cb_menu_$display" USING window-id, menu-id, ret-code.
        CALL "cb_window_$clear_window" USING user-window-id, ret-code.
```

```
    GOBACK.
EXIT.

    STOP-IT.
CALL "cb_menu_$terminate".
   * cb_menu_$terminate MUST be the last call to cb_menu_ in the
   * application.  It terminates the environment set up cb_menu_$init.

        EXIT PROGRAM.
```

Name: change_default_wdir_

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

USAGE

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

call change_default_wdir_ (path, code);
```

ARGUMENTS

path
    is the pathname of the directory that is to become the default working directory. (Input)

code
    is a storage system status code. (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), fixed bin(35));

call change_wdir_ (path, code);
```

ARGUMENTS

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

code
    is a storage system status code. (Output)

Name: char_offset_

This function returns the character offset (relative to the base of the segment) of the character located by the supplied pointer value.

USAGE

```
dcl char_offset_ entry (ptr) returns (fixed bin (21)) reducible;

character_offset = char_offset_ (pointer_value);
```

ARGUMENTS

pointer_value
    is a pointer whose character offset is to be determined. (Input)

character_offset
    is the character offset of the supplied pointer. (Output)

NOTES

The first character in a segment has a character offset of zero.

If the pointer supplied to char_offset_ does not point to a character boundary, the offset returned is that of the character containing the bit located by the pointer.

Name: char_to_numeric_

The char_to_numeric_ subroutine converts a user-supplied string to a numeric type, or signals the conversion condition if it cannot be converted. The attributes of the numeric data created are returned.

USAGE

```
declare char_to_numeric_ entry (ptr, fixed bin(35), fixed bin(35), ptr,
      fixed bin(21));

call char_to_numeric_ (target_ptr, enc_type, enc_prec, source_ptr,
      source_len);
```

ARGUMENTS

target_ptr
: points to a buffer where the numeric data may be written. No check is made that the buffer is large enough to hold the data. (Input)

enc_type
: is the encoded type of the data created. Its value is 2*M+P, where M is a standard Multics type code, and P is 1 if the data is packed, or 0 if it is not. (P should always be 0.) The value of Multics type codes are defined in the Programmer's Reference Manual. (Output)

enc_prec
: is the encoded precision of the data created. The format of an encoded precision is given by encoded_precision.incl.pl1. See the description of the assign_ subroutine. (Output)

source_ptr
: points to the character string to convert to numeric. (Input)

source_len
: is the number of characters in the input string. (Input)

Name: check_gate_access_

This subroutine will allow a caller to determine whether a user has access to a gate before trying to call it. It will differentiate between not finding the gate and not having access.

USAGE

```
dcl check_gate_access_ entry (char(*), ptr, fixed bin (35));

call check_gate_access_ (gate_name, ref_ptr, code);
```

ARGUMENTS

gate_name
: is the name of the gate. (e.g., "phcs_")

ref_ptr
: is a pointer used to determine the desired referencing directory. (Input) It can be null (), in which case the referencing_dir search rule is not used, or can be a pointer to a procedure, usually the caller of check_gate_access_, whose containing directory will be used as the referencing directory.

code
: is a standard system status code. (Output) It's value will be zero if the gate is located using the search rules of the current ring and if the access to the gate includes execute access. If the gate cannot be located, the error code returned is error_table_$noentry. If the gate is located, but execute access is lacking, then error_table_$moderr is returned.

NOTES

Programs which can take alternate paths based on the access of lack of access to a gate should use this subroutine rather than trying to reference the gate explicitly and generating an access violation audit message in the process.

Name: check_star_name_

The check_star_name_ subroutine analyzes a character string to be sure that it has been formed according to the rules of the star convention, and optionally checks that it also conforms to the rules for forming entrynames. It returns a starname type code that indicates whether the string is a starname, and whether the starname matches every possible name.

Entry: check_star_name_$check_star_name_

This entrypoint accepts a character string and a bit mask as its inputs, and analyzes the character string according to the tests selected by the bit mask.

USAGE

```
declare check_star_name_ entry (char(*), bit(36) aligned, fixed bin(2),
      fixed bin(35));

call check_star_name_ (starname, control_mask, type, code);
```

ARGUMENTS

starname
: is the character string to be analyzed. Trailing spaces in the character string are ignored. (Input)

control_mask
: is a bit string constructed from constants listed below. (Input)

type
: is one of the starname type codes listed below. (Output)

code
: is one of the standard status codes listed below. (Output)

LIST OF CONTROL_MASK CONSTANTS

These constants are defined in check_star_name.incl.pl1, and can be combined in most cases using a PL/I boolean or operator (|).

CHECK_STAR_IGNORE_ARCHIVE
: permit the archive component pathname delimiter, double colon ("::") in the starname, and treat it as a pair of nonspecial characters. By default, this would be rejected.

CHECK_STAR_IGNORE_ENTRYPOINT
: permit the entrypoint convention delimiters, dollar sign ("$") and vertical bar ("|") in the starname, and treat them as nonspecial characters. By default, they would be rejected.

CHECK_STAR_IGNORE_EQUAL
permit the equal convention characters, equal sign ("=") and percent sign ("%") in the starname, and treat them as nonspecial characters. By default, they would be rejected.

CHECK_STAR_IGNORE_LENGTH
permit an entryname starname or a component name starname to be more than 32 characters long. By default, this is not permitted. The containing dir and entrypoints of path are not checked for length.

CHECK_STAR_IGNORE_NONASCII
permit nonASCII characters in an entryname starname or a component name starname, and treat them as nonspecial characters. By default, they would be rejected.

CHECK_STAR_IGNORE_NULL
permit null components in the starname. By default, they would be rejected.

CHECK_STAR_IGNORE_PATH
permit the pathname delimiters, less than ("<") and greater than (">") in the starname, and treat them as nonspecial characters. By default, they would be rejected.

CHECK_STAR_PROCESS_ARCHIVE
if the archive component pathname delimiter is present, analyze the substring preceding it and the substring following it separately. If either name is a starname, indicate that the match procedure must be used. A second archive delimiter will be rejected. If this is combined with CHECK_STAR_PROCESS_ENTRYPOINT, an archive delimiter following the entrypoint delimiter will be rejected.

CHECK_STAR_PROCESS_ENTRYPOINT
if one of the entrypoint convention delimiters is present, check the substring preceding it and the substring following it separately. If either name is a starname, indicate that the match procedure must be used. A second entrypoint delimiter will be rejected. If it is combined with CHECK_STAR_PROCESS_ARCHIVE, an entrypoint delimiter preceding the archive delimiter will be rejected.

CHECK_STAR_PROCESS_PATH
if pathname delimiters are present, analyze only the substring following the rightmost pathname delimiter. If this string is of zero length, report that PL/I comparison can be used, because the expanded pathname will end in the name of a directory, and valid directory names can't contain star convention characters. (This is intended for names like "<". Names like ">udd>" may be rejected by expand_pathname_, but are acceptable to check_star_name_.)

CHECK_STAR_REJECT_WILD
return error_table_$nostars if any star convention characters are present.

error_table_$badequal
equal convention characters were found and the control_mask did not permit them.

error_table_$badpath
the directory name contains a nonASCII character and CHECK_STAR_PROCESS_PATH was specified but CHECK_STAR_IGNORE_NONASCII was not.

error_table_$badstar
the string violates the rules for forming starnames.

error_table_$entlong
The string was more than 32 characters long and the control_mask did not permit it. If CHECK_STAR_PROCESS_PATH was specified, the entryname part of the string was more than 32 characters long. If CHECK_STAR_PROCESS_ARCHIVE was specified, either the entryname or the component name was more than 32 characters long.

error_table_$inconsistent
the control_mask was in error, specifying both CHECK_STAR_PROCESS and CHECK_STAR_IGNORE the same test.

error_table_$invalid_ascii
the entryname contains a nonASCII character and CHECK_STAR_IGNORE_NONASCII was not specified.

error_table_$nostars
stars or question marks were found and CHECK_STAR_REJECT_WILD was specified in the control_mask. Note that star_type will correctly reflect the starname type for this case.

error_table_$null_name_component
the string contains null components and the control_mask did not permit them.

NOTES

See the description of the hcs_$star_ entrypoint in hcs_ to find how to list the directory entries that match a given starname. See match_star_name_ to find how to match a starname with an entryname. See starname.gi.info for the rules governing the formation and interpretation of starnames. See entryname.gi.info for the rules governing the formation of entrynames.

Entry: check_star_name_$entry

This entrypoint accepts the entryname to be analyzed as input.

USAGE

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

call check_star_name_$entry (entryname, code);
```

ARGUMENTS

entryname
: is the entryname to be validated. Trailing spaces in the entryname character string are ignored. (Input)

code
: is one of the nonstandard status codes listed below. (Output)

LIST OF STATUS CODES

0
: the entryname is valid and is not a starname (does not contain asterisks or question marks).

1
: the entryname is valid and is a starname (does contain asterisks or question marks).

2
: the entryname is valid and is a starname that matches every entryname.

error_table_$badstar
: the entryname is invalid. It violates the rules for forming starnames, or it violates the rules for constructing entrynames.

NOTES

This entrypoint is obsolete. Use the check_star_name_ entrypoint for new applications. The new entrypoint returns a variety of different standard error codes explaining a rejection whereas this entrypoint can only return a single standard error code value for compatability.

See the description of the hcs_$star_ entrypoint in hcs_.info to find how to list the directory entries that match a given starname. See match_star_name_.info to find how to match a starname with an entryname. See starname.gi.info for the rules governing the formation and interpretation of starnames. See entryname.gi.info for the rules governing the formation of entrynames.

Entry: check_star_name_$path

This entrypoint accepts a pathname as its input and analyzes the final entryname in that pathname.

USAGE

```
declare check_star_name_$path entry (char(*), fixed bin(35));

call check_star_name_$path (path, code);
```

ARGUMENTS

path
is the pathname whose final entryname is to be validated. Trailing spaces in the pathname character string are ignored. (Input)

code
is one of the nonstandard status codes listed below. (Output)

LIST OF STATUS CODES

0
the entryname is valid and is not a starname (does not contain asterisks or question marks).

1
the entryname is valid and is a starname (does contain asterisks or question marks).

2
the entryname is valid and is a starname that matches every entryname.

error_table_$badstar
the entryname is invalid. It violates the rules for forming starnames, or it violates the rules for forming pathnames.

NOTES

This entrypoint is obsolete. Use the check_star_name_ entrypoint for new applications. The new entrypoint returns a variety of different standard error codes explaining a rejection whereas this entrypoint can only return a single standard error code value for compatibility.

See the description of the hcs_$star_ entrypoint in hcs_ to find how to list the directory entries that match a given starname. See match_star_name_ to find how to match a starname with an entryname. See starname.gi.info for the rules governing the formation and interpretation of starnames. See pathname.gi.info for the rules governing the formation of pathnames.

Name: clock_

The clock_ function 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, or decompose it into its component parts, respectively.

USAGE

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

date_time = clock_ ();
```

ARGUMENTS

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

NOTES

The clock PL/I builtin function should be used in PL/I programs instead of this subroutine, because it is more efficient.

This page intentionally left blank.

CHECK_STAR_IGNORE_ALL
combines all the CHECK_STAR_IGNORE flags. This can be used to analyze a starname to determine its type without applying any of the tests intended for validating entrynames.

CHECK_STAR_ENTRY_DEFAULT
combines exactly the same tests used by the obsolete check_star_name_$entry entrypoint, which is equivalent to combining CHECK_STAR_IGNORE_ENTRYPOINT and CHECK_STAR_IGNORE_EQUAL.

CHECK_STAR_PATH_DEFAULT
combines exactly the same tests used by the obsolete check_star_name_$path entrypoint, which is equivalent to combining CHECK_STAR_PROCESS_ARCHIVE, CHECK_STAR_IGNORE_ENTRYPOINT, CHECK_STAR_IGNORE_EQUAL, and CHECK_STAR_PROCESS_PATH.

LIST OF STARNAME TYPE CODES

These type constants are defined in check_star_name.incl.pl1.

STAR_TYPE_MATCHES_EVERYTHING (2)
no comparison is necessary, since the starname matches all possible strings.

STAR_TYPE_USE_MATCH_PROCEDURE (1)
the procedure match_star_name_ must be used to compare the string to possible matching strings, because it is a starname containing stars (asterisks) and question marks.

STAR_TYPE_USE_PL1_COMPARE (0)
the string is not a starname and can be compared using PL/I comparison rules.

LIST OF STATUS CODES

0
the string passes all of the selected tests, and the starname type output indicates whether the string is a starname.

error_table_$archive_pathname
the archive component pathname delimiter was found, and the control_mask did not permit it.

error_table_$bad_arg
the control mask specified an unimplented test.

error_table_$bad_file_name
the string violates the rules for forming entrynames and the control_mask did not permit it.

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 can also be called with a code of 0 to report an error which does not have a status code associated with it. Status codes are described in the Programmer's Reference Manual.

See also the active_fnc_err_ subroutine which should be used by active functions for printing error messages.

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

USAGE

```
declare com_err_ entry options (variable);

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

ARGUMENTS

code
: is a standard status code, which normally is fixed binary (35), but can be any computational data type. (Input) If it is not already fixed binary (35), it will be converted to fixed binary (35).

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

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

argI
: are ioa_ subroutine arguments to be substituted into the control_string argument. (Input) These arguments are optional. They can only be used, however, if the control_string argument is given first (see "Notes" below).

NOTES

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

```
caller:  system_message user_message
```

where:

caller
    is the name of the program detecting the error.

system_message
    is a standard message from the error table corresponding to the value of code. If code is equal to 0, no system_message is printed.

user_message
    is constructed by the ioa_ subroutine from the control_string and argI arguments. If the control_string and argI arguments are omitted, no user_message is printed.

If code is error_table_$active_function, com_err_ will print a slightly different message and signal the active_function_error condition to prevent the command from being restarted. The message printed will be:

```
caller:  This command cannot be invoked as an active function.
         user_message
error:   Attempt to invoke command caller as an active function.
```

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

This 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, argl, ...,
      argN);
```

ARGUMENTS
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. System conditions are described in the Programmer's Reference Manual. The default handler for this condition simply returns control to the command_query_ subroutine, which writes the question on a specified I/O switch. It then reads from another I/O switch 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_ (info_ptr, answer, caller, control_string,
     arg1, ..., argN);
```

ARGUMENTS

info_ptr
: is a pointer to the query_info structure described in "Info Structure" below. (Input)

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

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

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

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

INFO STRUCTURE

The following is the query_info structure (found in the include file query_info.incl.pl1):

```
dcl 1 query_info                aligned,
      2 version                 fixed bin,
      2 switches                aligned,
        3 yes_or_no_sw          bit(1) unaligned,
        3 suppress_name_sw      bit(1) unaligned,
        3 cp_escape_control     bit(2) unaligned,
        3 suppress_spacing      bit(1) unaligned,
        3 literal_sw            bit(1) unaligned,
        3 prompt_after explanation
                                bit (1) unaligned,
        3 padding               bit(29) unaligned,
      2 status_code             fixed bin(35),
      2 query_code              fixed bin(35),
      2 question_iocbp          ptr,
      2 answer_iocbp            ptr,
      2 repeat_time             fixed bin(71),
      2 explanation_ptr         ptr,
      2 explanation_len         fixed bin (21);
```

STRUCTURE ELEMENTS

version
: is the version number of this structure. (Input) The version number must be set by the caller and identifies the format of the structure. The current version is a static variable named query_info_version_6 in query_info.incl.pl1.

yes_or_no_sw
: indicates whether an answer of a particular form is expected. (Input)
"0"b accepts any answer.
"1"b accepts only a yes or no answer.

suppress_name_sw
: controls whether the name of the calling procedure appears in the question. (Input)
"0"b includes name and following colon.
"1"b omits name and colon.

cp_escape_control
: controls whether the command_processor_ escape mechanism is enabled for this call. (Input)
"00"b obeys the static default.
"01"b allows lines to begin with ".." but does not interpret them as command_processor_ escapes.
"10"b disallows escape, ignores default.
"11"b allows escape, ignores default.

suppress_spacing
controls the insertion of a newline before the question and two spaces after it. (Input)
"0"b inserts extra space.
"1"b omits extra space.

literal_sw
is "1"b to suppress any special interpretation of characters (for example, "..") and suppress stripping of leading whitespace.

prompt_after_explanation
is "1"b to repeat the original question after printing any explanation indicated by a non-null explanation argument.

padding
is unused space. (Input)

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

query_code
is additional arbitrary qualifying information passed by the caller of command_query_. (Input) It is intended for use by specialized handlers for command_question.

question_iocbp
is an iocb pointer for the I/O switch over which the caller wants the question to be written. (Input) A null pointer indicates that the of the user_i/o switch is to be used by default.

answer_iocbp
is an iocb pointer for the I/O switch from which the caller wants the answer to be read. (Input) A null pointer indicates that the user_input switch is to be used by default.

repeat_time
is the number of seconds to wait for an answer before repeating the question on the switch pointed to by question_iocbp. (Input) A value less than 30 indicates that the question is not to be repeated.

explanation_ptr
is a pointer to a string to be printed if the user answers "?". (Input)

explanation_len
is the length of the explanation string. (Input)

NOTES

The question prepared by the command_query_ subroutine has the format:

```
caller: message
```

where the message is constructed by the ioa_ subroutine from the control_string and argN arguments. If the control_string and argN arguments are not given, the message portion of the question is omitted.

If the user answers with a single question mark (?), the explanation_ptr field is examined. If it is non-null and explanation_len is greater than 0, the explanation string pointed to is printed and the user is expected to answer again. Otherwise, the string "Answer: " is printed and the user is expected to answer again.

Case insensitive "yes", "y", and "n" are acceptable responses to a yes or no question.

In an absentee process with the yes_or_no_sw on, an answer other case insensitive "yes", "y", "no", "n", or "?" causes the absentee process to signal command_query_error.

If the answer to a question begins with "..", and the cp_escape feature is enabled for the question, the rest of the answer following the ".." is passed to the command processor. Control then returns to command_query_, which prompts with "Answer: " after the command has been executed. The cp_escape feature is normally enabled in the standard Multics environment; a subsystem, however, can elect to turn it off, either globally or for a particular question. The prompt of "Answer: " is used rather than repeating the question because the question may be quite long and take significant time to print. If it is necessary to see the question again, answering "..repeat_query" repeats it.

Entry: command_query_$set_cp_escape_enable

This entry sets the static default switch that allows or disallows the command processor escape feature. It also returns the previous value for the switch. Since escapes are disabled initially, it is necessary to call this entry to enable the feature. This entry is called by process_overseer_, which sets it so that the escape is permitted in a normal Multics environment.

USAGE

```
declare command_query_$set_cp_escape_enable entry (bit(1) aligned,
      bit(1) aligned);

call command_query_$set_cp_escape_enable (new_value, old_value);
```

ARGUMENTS

new_value
is the new value for the default. (Input)
"0"b feature is disabled by default.
"1"b feature is enabled by default.

old_value
is the old value of the default. (Output) If it has never been set, it is "0"b.

Entry: command_query_$yes_no

This entry asks the user for a yes or no answer.

USAGE

```
dcl command_query_$yes_no entry options (variable);

call command_query_$yes_no (yes_sw, query_code, caller, explanation,
      question, arg1, ..., argN);
```

ARGUMENTS

yes_sw
is a bit (1) return value, ON for "yes" or "y" and OFF for "no" or "n", case insensitive. (Output) Other answers are not accepted from the user.

query_code
is a standard status code. (Input) If it is nonzero, the question is preceded by the corresponding error message.

caller
is the character string name of the calling program. (Input)

explanation
is an explanation of the question, printed when the user answers "?". (Input) The explanation is an ioa_ control string, in which parameters are replaced by the values of the argN arguments. For a description of control strings, see the ioa_ subroutine.

question
is the question, also in the form of an ioa_ control string. (Input) Parameters are replaced by the same argN arguments as for the explanation.

argN
are character string arguments to the ioa_ control strings specified by explanation and question. (Input)

NOTES

The same arguments are substituted in both explanation and question control strings. Each control string can use ^s to skip particular arguments.

EXAMPLES

The following shows a use of the explanation argument:

```
call command_query_$yes_no (yes_sw, 0, "delete_notifications",
     "Do you want to delete all messages that have been printed?",
     "Delete?");
```

This call produces the following behavior when the user answers "?":

```
delete_notifications:  Delete?  ?
Do you want to delete all messages that have been printed?  yes
```

The following explanation and question use parameter substitution and a nonzero query_code argument:

```
call command_query_$yes_no
     (yes_sw, error_table_$namedup, "create_tff",
     "Do you want to delete the old ^a ^a before creating a new one?"
     "Delete old ^s^a?", "segment", pathname);
```

producing the following:

```
create_tff: Name duplication.  Delete old >udd>d>c>h.tff?  ?
Do you want to delete the old segment >udd>d>c>h.tff before
     creating a new one?  yes
```

Name: component_info_

This subroutine returns information about a component of a bound segment similar to that returned by object_info_. The component may be specified either by name or by offset.

Entry: component_info_$name

This entry point specifies the component by name.

USAGE

```
declare component_info_$name entry (ptr, char(32) aligned, ptr,
      fixed bin(35));

call component_info_$name (seg_ptr, comp_name, arg_ptr, code);
```

ARGUMENTS

seg_ptr
: is a pointer to the bound segment.

comp_name
: is the name of the component.

arg_ptr
: is a pointer to a structure to be filled in (see "Notes" below).

code
: is a standard status code. (Output)

Entry: component_info_$offset

This entry point specifies the component by its offset.

USAGE

```
declare component_info_$offset entry (ptr, fixed bin(18), ptr,
      fixed bin(35));

call component_info_$offset (seg_ptr, offset, arg_ptr, code);
```

ARGUMENTS

seg_ptr
: is a pointer to the bound segment. (Input)

offset
: is an offset into the bound segment corresponding to the text, internal static or symbol section of some component. (Input)

arg_ptr
: is a pointer to a structure to be filled in (see "Notes" below).

code
: is a standard status code. (Output)

NOTES

The structure to be filled in (a declaration of which is found in component_info.incl.pl1) is declared as follows:

```
dcl 1 ci                   aligned,
      2 dcl_version        fixed bin,
      2 name               char(32) aligned,
      2 text_start         ptr,
      2 stat_start         ptr,
      2 symb_start         ptr,
      2 defblock_ptr       ptr,
      2 text_lng           fixed bin,
      2 stat_lng           fixed bin,
      2 symb_lng           fixed bin,
      2 n_blocks           fixed bin,
      2 standard           bit(1) aligned,
      2 compiler           char(8) aligned,
      2 compile_time       fixed bin(71),
      2 user_id            char(32) aligned,
      2 cvers              aligned,
        3 offset           bit(18) unaligned,
        3 length           bit(18) unaligned,
      2 comment            aligned,
        3 offset           bit(18) unaligned,
        3 length           bit(18) unaligned,
      2 source_map         fixed bin;
```

STRUCTURE ELEMENTS

dcl_version
: is the version number of this structure. It is set by the caller and must be 1.

name
: is the name of the component, i.e., the name specified in a bindfile objectname statement; also, the name of the component as archived.

text_start
: is a pointer to the base of the component's text section.

stat_start
: is a pointer to the base of the component's internal static.

symb_start
: is a pointer to the base of the component's symbol section.

defblock_ptr
: is a pointer to the component's definition block.

text_lng
: is the length, in words, of the component's text section.

stat_lng
: is the length, in words, of the component's internal static.

symb_lng
: is the length, in words, of the component's symbol section.

n_blocks
: is the number of blocks in the component's symbol section.

standard
: is on if the component is in standard object format.

compiler
: is the name of the component's compiler.

compile_time
: is a clock reading of the date/time the component was compiled.

user_id
: is the standard Multics User_id of the component's creator.

cvers.offset
: is the offset of the printable version description of the component's compiler, in words, relative to symb_start.

cvers.length
: is the length, in characters, of the component's compiler version.

comment.offset
: is the offset of the component's compiler comment, in words, relative to symb_start.

comment.length
: is the length, in characters, of the component's comment.

source_map
: is the offset of the component's source map structure, in words, relative to symb_start.

Name: compute_common_aim_ceiling_

This subroutine computes the maximum authorization or access class which is in common between two Multics systems given the definitions of their AIM attributes.

USAGE

```
declare compute_common_aim_ceiling_ entry (ptr, bit(72) aligned, ptr,
      bit(72) aligned, fixed bin(35));

call compute_common_aim_ceiling_ (aim_attributes_1_ptr,
      common_ceiling_1, aim_attributes_2_ptr, common_ceiling_2, code);
```

ARGUMENTS

aim_attributes_1_ptr
: is a pointer to the aim_attributes structure defining the AIM attributes of the first system. (Input) This structure is declared in aim_attributes.incl.pl1.

common_ceiling_1
: is set to the maximum authorization or access class in common between the two systems in terms of the AIM attributes of the first system. (Output)

aim_attributes_2_ptr
: is a pointer to the aim_attributes structure defining the AIM attributes of the second system. (Input)

common_ceiling_2
: is set to the maximum authorization or access class in common between the two systems in terms of the AIM attributes of the second system. (Output)

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

  0
  : the common access ceiling was successfully computed.

  error_table_$unimplemented_version
  : one of the aim_attributes structures supplied by the caller was of a version not supported by this procedure.

  error_table_$ai_no_common_max
  : there is no set of AIM attributes in common between the two systems.

NOTES

See the description of the get_system_aim_attributes_ subroutine for a definition of the aim_attributes structure.

See the Programmers' Reference Manual for a definition of common access ceiling.

Name: condition_

This subroutine establishes a handler for a condition in the calling block activation if a handler for the specified condition is currently established in the calling block.

A description of the condition mechanism is given in the *Multics Programmer's Reference* manual in the section entitled "The Multics Condition Mechanism".

USAGE

```
declare condition_ entry (char(*), entry);

call condition_ (name, handler);
```

ARGUMENTS

name
is the name of the condition for which the handler is to be established. (Input)

handler
is the handler to be invoked when the condition is raised. (Input)

NOTES

The condition name unclaimed_signal is an obsolete special condition name and should not be used.

The PL/I on statement and the condition_ subroutine must not be invoked during the same block activation in order to establish a handler for the same condition.

In PL/I Version 2, when a call to condition_ appears within the scope of a begin block or internal procedure of a procedure, the no_quick_blocks option must be specified in the procedure statement of that procedure. The no_quick_blocks option is a nonstandard feature of the Multics PL/I language and, therefore, programs using it may not be transferable to other systems.

This page intentionally left blank.

Name: condition_interpreter_

The condition_interpreter_ subroutine can be used by subsystem condition handlers to obtain a formatted error message for all conditions except quit, alrm, and cput. Some conditions do not have messages and others cause special actions to be taken. These are described in "Notes" below. (For more information on conditions, see the Programmer's Reference Manual.)

USAGE

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

call condition_interpreter_ (area_ptr, m_ptr, mlng, mode, mc_ptr,
      cond_name, wc_ptr, info_ptr);
```

ARGUMENTS

area_ptr
: is a pointer to the area in which the message is to be allocated, if the message is to be returned. The area size should be at least 300 words. If null, the message is printed on the error_output I/O switch. (Input)

m_ptr
: points to the allocated message if area_ptr is not null; otherwise it is not set. (Output)

mlng
: is the length (in characters) of the allocated message if area_ptr is not null. If area_ptr is null, the length is not set. Certain conditions (see "Notes" below) have no messages; in these cases, mlng is equal to 0. (Output)

mode
: is the desired mode of the message to be printed or returned. (Input) It can have the following values:
1 normal mode
2 brief mode
3 long mode

mc_ptr
: if not null, points to machine conditions describing the state of the processor at the time the condition was raised. (Input)

cond_name
: is the name of the condition being raised. (Input)

wc_ptr
: is usually null; but when mc_ptr points to machine conditions from ring 0, wc_ptr points to alternate machine conditions. (Input)

info_ptr
if not null, points to the information structure described in the Programmer's Reference Manual. (Input)

NOTES

The following conditions cause a return with no message:

command_error
command_question
finish
stringsize

Name: continue_to_signal_

The continue_to_signal_ subroutine enables an on unit that cannot completely handle a condition to tell the signalling program, upon its return, to search the stack for other on units for the condition. The search continues with the stack frame immediately preceding the frame for the block containing the on unit. However, if a separate on unit for the any_other condition is established in the same block activation as the caller of the continue_to_signal_ subroutine, that on unit is invoked before the stack is searched further.

USAGE

```
declare continue_to_signal_ entry (fixed bin(35));

call continue_to_signal_ (code);
```

ARGUMENTS

code
is a standard status code and is nonzero if continue_to_signal_ was called when no condition was signalled. (Output)

Name: convert_access_audit_flags_

This subroutine is provided to convert a security audit flag back and forth between its character string representation and the internal binary representation.

Entry: convert_access_audit_flags_$from_string

This entry point converts the textual representation to internal representation.

USAGE

```
dcl convert_access_audit_flags_$from_string entry (char (*), bit (36)
      aligned, fixed bin (35));

call convert_access_audit_flags_$from_string (flags_str, audit_flags,
      code);
```

ARGUMENTS

flags_str
: is the textual representation of the security audit flags. (Input)

audit_flags
: is the bit string internal representation of the flags. (Output)

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

Entry: convert_access_audit_flags_$to_string

This entry point converts from internal representation to textual representation.

USAGE

```
dcl convert_access_audit_flags_$to_string (entry (char (*), bit (36),
      aligned, fixed bin (35));

call convert_access_audit_flags_$to_string (flags_str, audit_flags,
      code);
```

ARGUMENTS
flags_str is the textual representation of the security audit flags. (Output)

audit_flags
: is the bit string internal representation of the flags. (Output)

code
: is a standard system status code. (Output) It can have one of the following values:

  error_table_$badarg
  : audit_flags is illegally constructed

  error_table_$smallarg
  : flags_str is too small for result

NOTES

The format of the flags string is as follows:

```
flag-string := flag-item [, flag-item]
flag-item := object-type-keyword "grant=" audit-level-keyword
flag-item := object-type-keyword "-deny=" audit-level-keyword
flag-item := even-type-keyword
object-type-keyword := "fsobj" | "fsattr" | "rcp" | "admin" |
                                 "special" | "other"
audit-level-keyword := "none" | "modify_access" | "modify" | "read"
event-type-keyword := "admin_op" | "priv_op" | "fault" |
                                 "cc_1_10" | "cc_10_100"
```

example:

fsobj-grant=modify,rcp-deny=modify_access,other-grant=none, fault

---

Name: convert_access_class_

The convert_access_class_ subroutine is provided to convert an access attribute 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 access attribute as a short character string for use in entrynames.

Entry: convert_access_class_$decode

This entry point takes the character string produced by the convert_access_class_$encode entry point and returns the original access attribute. The null string and the string "system_low" are both converted to return the system_low access attribute.

USAGE

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

call convert_access_class_$decode (acc_att, decoded_string);
```

ARGUMENTS

acc_att
is the the decoded authorization. (Output)

decoded_string
is a short string (maximum of 15 characters) that uniquely represents the input access attribute. (Input)

Entry: convert_access_class_$encode

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

USAGE

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

call convert_access_class $encode (acc_att, encoded_string);
```

ARGUMENTS

acc_att
is the input access attribute (Input)

encoded_string
is a short string (maximum of 15 characters) that uniquely represents the input access attribute. (Output)

Entry: convert_access_class_$from_string

This entry point converts the character string representation of an access attribute to an encoded binary form suitable for storage in system tables and as input to the various modules that accept the binary form.

USAGE

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

call convert_access_class_$from_string (acc_att, string, code);
```

ARGUMENTS

acc_att
is the binary representation of string. (Output)

string
: is the character string to be converted (see "Notes" below). (Input)

code
: is a standard status code. (Output) It can be one of the following:
  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 access attribute is greater than the system_high access attribute.

NOTES

The string argument must be of the form:

```
name1,name2,...,nameN
```

where namei represents the mnemonic for a sensitivity level or access category. The print_auth_names command can 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 access attribute allowed).

Entry: convert_access_class_$from_string_range

This entry point converts a character string to the form of a binary access attribute range.

USAGE

```
declare convert_access_class_$from_string_range entry (bit(72) aligned
      dimension(2), char(*), fixed bin(35));

call convert_access_class_$from_string_range (acc_att_range, string,
      code)
```

ARGUMENTS

acc_att_range
: is the binary representation of string. (Output)

string
: is the character string to be converted (see "Notes" below). (Input)

code
: is a standard status code. (Output) It can be one of the following:
  error_table_$ai_invalid_string
    one or more namei are misspelled (see "Notes" below).
  error_table_$ai_above_allowed_max
    no error in conversion; but the resulting access attribute is greater than the system_high access attribute.
  error_table_$ai_invalid_range
    no error in conversion; but acc_att_range (2) does not represent an access attribute greater than or equal to acc_att_range (1).

NOTES

The string must be one of the two forms:

```
name1,name2,...,nameN

name1a,name2a,...,nameNa:name1b,name2b,...nameNb
```

where namei represents the mnemonic for a sensitivity level or access category. If the string is in the first form, both elements of acc_att_range will be set to equal values (similar to the operation of convert_access_class_$from_string). If string is in the second form, acc_att_range (1) will be returned as the binary representation of the part of string left of the colon, and acc_att_range (2) will be returned as the binary representation of the part of the string right of the colon.

Entry: convert_access_class_$minimum

This entry point accepts an array of access attributes and a binary number indicating how many elements to process from the array. It returns an access attribute 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_access_class_$minimum entry (dim(*) bit(72) aligned,
     fixed bin, bit(72) aligned);

call convert_access_class_$minimum (acc_att_array, n_elements,
     minimum_acc_att)
```

ARGUMENTS

acc_att_array
    are the input access attributes(Input)

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

minimum_acc_att
    is the result. (Output)

Entry: convert_access_class_$to_string

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

USAGE

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

call convert_access_class_$to_string (acc_att, string, code);
```

ARGUMENTS

acc_att
    is the access attribute to be converted. (Input)

string
    is the resultant character string (see "Notes" below). (Output)

code
    is a standard status code. (Output) It can be one of the following:
    error_table_$smallarg
        string is too short to hold the converted result (see "Notes" below).
    error table_$ai_invalid binary
        either the level number or category set is invalid; the resulting output is also invalid.

NOTES

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

If the access attribute is equal to the site access ceiling as defined by the installation_parms and returned by system_info_$access_ceiling, then "system_high" is returned in the string.

Entry: convert_access_class_$to_string_range

This entry point accepts a binary access attribute range pair and returns it as a printable string. This output string is suitable for input to the convert_access_class_$from_string_range entry point. Each level/category name has a maximum length of 32 characters.

USAGE

```
declare convert_access_class_$to_string_range entry (bit (72) aligned
      dimension (2), char (*), fixed bin (35));

call convert_access_class_$to_string_range (acc_att_range, string,
      code);
```

ARGUMENTS

acc_att_range
: is the binary representation of an access attribute range to be converted. (Input)

string
: is the resultant character string (see "Notes" below). (Output)

code
: is a standard status code. (Output) It can be one of the following:
  error_table_$smallarg
    string is too short to hold the converted result (see "Notes" below).
  error_table_$ai_invalid_binary
    either the level number or category set is invalid; the resulting output is also invalid.
  error_table_$ai_invalid_range
    no error in conversion; but acc_att_range (2) does not represent an access attribute greater than or equal to acc_att_range (1).

NOTES

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

If either of the access attributes is equal to the site access ceiling as defined by the installation_parms and returned by system_info_$access_ceiling, then "system_high" is returned in the string for that attribute.

Entry: convert_access_class_$to_string_range_short

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

USAGE

```
declare convert_access_class_$to_string_range_short entry (bit(72)
      aligned dimension(2),char(*), fixed bin(35);

call convert_access_class_$to_string_range_short (acc_att_range, string,
      code);
```

ARGUMENTS

acc_att_range
: is the binary representation of an access attribute range range to be converted. (Input)

string
: is the resultant character string (see "Notes" below). (Output)

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

  error_table_$smallarg
  : string is too short to hold the converted result (see "Notes" below).

  error_table_$ai_invalid_binary
  : either the level number or category set is invalid: the resulting output is also invalid.

  error_table_$ai_invalid_range
  : no error in conversion; but acc_att_range (2) does not represent an access attribute greater than or equal to acc_att_range (1).

NOTES

If either of the access attributes is equal to the site access ceiling as defined by the installation_parms and returned by system_info_$access_ceiling, then "system_high" is returned in the string for that attribute.

Entry: convert_access_class_$to_string_short

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

USAGE

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

call convert_access_class_$to_string_short (acc_att, string, code);
```

ARGUMENTS

acc_att
: is the binary representation of an access attribute to be converted. (Input)

string
: is the resultant character string. (Output)

code
: is a standard status code. (Output) It can be one of the following:
  error_table_$smallarg
    string is too short to hold the converted result (see "Notes" below).
  error_table_$ai_invalid_binary
    either the level number or category set is invalid; the resulting output is also invalid.

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. 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_ (time_string, clock, code);
```

ARGUMENTS

time_string
: the string to be converted. (Input) See Time String below for a description of valid string values.

clock
: the resulting clock value. (Output) Unchanged if an error occurs.

code
: is a standard status code. (Output) It can have one of the following values:

error_table_$bad_conversion
: a conversion condition occurred while trying to convert a value.

error_table_$dt_ambiguous_time
: there is no language common to all words in the time string.

error_table_$dt_bad_fw
: fiscal_week < 1 or fiscal_week > year_max (which is 52 or 53).

error_table_$dt_hour_gt_twelve
: the hour given exceeds 12.

error_table_$dt_multiple_date_spec
: more than one instance of a date has been given.

error_table_$dt_multiple_diw_spec
: day of the week specified more than once.

error_table_$dt_multiple_meaning
: the time string does not have the same meaning in all potential languages, these being the intersection of all the languages possible for all words present.

error_table_$dt_multiple_time_spec
: more than one instance of a time has been given.

error_table_$dt_multiple_zone_spec
: the zone may only be specified once.

error_table_$dt_time_conversion_error
: For any of the following reasons:

a. General syntax error
b. Month without a day number.
c. Midnight or noon preceded by an hour other than 12.
d. Improper use of comma or period.
e. Improper use of offset.

error_table_$dt_size_error
: the size condition occurred while converting the time string.

error_table_$too_many_tokens
: the time string contains more tokens than the routine is prepared to handle.

error_table_$dt_unknown_word
: a word in a time string is not found in the time_info_ token list.

TIME STRING

The time string can have up to six parts -- adverbial offset, date, time, day of week, signed offset, and time zone. Adverbial offsets, if present, must appear leftmost in the string. Beyond that, all of the parts are optional and may be in any order. The parts may 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 (usually made up of the first three letters of the word). No distinction is made between uppercase and lowercase characters. Although this description gives examples in English, each of the words is available in several languages. Any of these languages may be used in time strings, but all words within a given string must be in the same language. To see the languages defined on your site, type

```
display_time_info -lang
```

A numeric field consists of an optionally signed integer (or fraction) of one or more decimal digits. The special characters that may be used in either alphabetic or numeric fields are: the slash (/), the period (.), the colon (:), the plus (+), the minus (-), and the comma (,). Blanks are not required between alphabetic and numeric fields in the time strings; however, they are required between two numeric fields unless the second field begins with a plus (+) or minus (-) sign. For example:

```
2days4hours10minutes
1245.17+7hours
10/17/79Wednesday
```

Underscores may be used in place of blanks in the time string. For example:

```
09/25/79__1442.6_+5_hours
```

Usually when a user enters a time string, the time zone is omitted. Although the zone is seldom seen, it is very important. The time zone determines the interpretation of items given in the time string. Also, the zone is involved in supplying defaults for missing items. All defaults are taken from the current absolute time, adjusted by a working time zone. If a zone is given in the string, that becomes the working zone. Otherwise, the process default time zone is used.

This means that whether you convert a string with an explicit zone, such as "XXXX_ast" or set the process default to "ast" and then convert the string "XXXX", you get the same absolute time. (Note that setting the process default will also influence output conversion, while giving an explicit zone does not.) To display your default zone, type:

```
print_time_defaults zone
```

Multics accepts dates from the year 0001 through 9999. The Julian calendar is used for dates from 0001-01-01 through 1582-10-04. The Gregorian calendar is used for dates from 1582-10-15 through 9999-12-31. (The dates from October 5, 1582 through October 14, 1582 do not exist. They were dropped when the Gregorian calendar was adopted. The leap day is always February 29. The lower limit on dates of January 1, 0001 AD was picked since it begins the era. The upper limit on dates of December 31, 9999 was chosen to limit year numbers to four digits. The time zones as now defined are used regardless of the year. The Multics date/time system does not account for "leap seconds" and, therefore, the difference between any two binary clock values that are precisely an integral number of days (hours, minutes, seconds, etc.) apart is guaranteed to be evenly divisible by the number of microseconds in a day (hour, minute, second, etc.).

The six parts of the time string are described below. In these descriptions, whenever an assumed value is mentioned, it refers to the date/time adjusted to the working zone.

1. date
   is the day of the year and may be specified only once. Dates may be specified using normal date format, calendar date format, day of the week, date keywords, fiscal week, request-id, or may be omitted entirely. If no date is present, it is assumed to be the next occurrence of the time specified. For example, "10A" gives the date on which 10:00 am next occurs. If no date and no time are specified, the current date is used.

In normal date format, dates are specified as month (or month abbreviation), day of month, and year; or as day of month, month, and 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, 1978, then March 20 is equivalent to March 20, 1978; while March 12 is the same as March 12, 1979. There are three forms of normal date, illustrated by the examples below:

```
16 March    16 March 1978
March 16    March 16 1978    March 16, 1978 (The comma is optional.)
3/16     3/16/78     3/16/1978
```

Calendar date format permits dates to be specified as a year, month, and day of month, separated by minus signs. This is the International Standards Organization (ISO) standard format. The year is required, and may be given as a year of the century. The calendar date format is illustrated by the examples below:

```
79-12-31  or  1979-12-31
  (represents December 31, 1979)
```

The day of the week is a date specifier if present with no other form of date. It then selects the first occurrence of the named day AFTER today.

The date keywords are "yesterday", "today", and "tomorrow". For example,

```
6:35A today
yesterday +120days
```

The fiscal week is of the form FWyyyyww. "FW" is the fiscal indicator (in English), "yyyy" is the year number, and "ww" is the week number. The fiscal week begins on Monday and ends on Sunday. This form converts to the date of the Monday, but another day within the week may be selected by adding a day name. For example, "FW198413 m" gives "03/26/84 0000. Mon", while "FW198413 m Wed" gives "03/28/84 0000. Wed". The fiscal indicator may be separated from the number but the ordering must remain, i.e. "FW185425" or "FW 185425" but not "185425 FW".

A request-id is a 19-character string used by several programs in the system, such as list_output_request. It contains a complete date from year, in century down thru microseconds in this form:

```
yymmddHHMMSS.SSSSSS
```

If no zone is specified, it is interpreted in GMT, not the process default. A request-id specifies a time as well as a date, so no other time specification may be given.

2. day of week
is the day of the week (e.g., Monday) and may be present only once. When the day of the week is present along with one of the other forms of date specification, that date must fall on the indicated day of the week.

3. time
is the time of day and may only be present once. If omitted, it is assumed to be the current time. Time may be given as 24-hour format, 12-hour format, or the time keyword "now". The 24-hour time format consists of a four-digit number followed by a period. (hhmm., where hh represents hours, and mm represents minutes). This number may be followed by an optional decimal fraction-of-a-minute field (e.g., hhmm.m). Also acceptable are hours and minutes fields separated by colons (hh:mm). This may be optionally followed by either a fraction-of-a-minute field (hh:mm.m), or a seconds field (hh:mm:ss). The seconds, in turn, may be include a fraction-of-second field (e.g., hh:mm:ss.s). Examples of 24-hour time are:

```
1545.
1545.715
15:45
15:45.715
15:45:42
15:45:42.08
```

The 12-hour time format must end with a meridiem designator (i.e., A, P, am, pm, noon, (or n), midnight (or m)). Midnight and noon can be indicated by simply giving the meridiem designator. The designator may be preceded by time expressed as hours, hours:minutes, or hours:minutes:seconds (including an optional fraction of a second or fraction of a minute, as mentioned above). Examples of 12-hour time are:

```
midnight
5 am
5:45A
3:59:59.000001pm
11:07:30.5pm
12 n
```

There is a set of illegal times, 24:00-24:59, which are handled anyway. These are taken to mean 0:00-0:59 of the following day. Note that midnight is the beginning of a day (00:00) not the end.

4. signed offset
is an adjustment to be made to the clock value specified by the other fields. Offsets may be specified in any and all of the following units (i.e. singular, plural, or abbreviation):

```
year         years         yr
month        months        mo
week         weeks         wk
day          days          da
hour         hours         hr
minute       minutes       min
second       seconds       sec
microsecond  microseconds  usec
```

Each unit may be present one or more times, each preceded by an optionally signed fixed point number. If offset fields are the only thing present, the offsets are added to the default values of date and time, as described above.

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).

Examples of offset fields are:

```
3 weeks -60 hours
  (60 hours before 3 weeks after now)
1.5 hr 5min
  (an hour and 35 minutes from now)
1 hour 5 minutes
  (an hour and five minutes from now)
```

The order in which offset values are applied to the clock value can affect the resultant clock value. Offset values are applied in the following order:

```
year, month, week, day, hour,
  minute, second, microsecond
```

Assuming that today is September 25, 1979, then:

```
10/1 -1 day +1 month
```

results in a clock value for 10/31/79, rather than for 10/30/79.

"Monday 6 am 2 weeks" means "two weeks after the next occurrence of Monday, at 6:00 am on that day".

NOTE: There is also a non-offset use of these words, available in combination with the word "this", i.e. "this month". Some of these combinations can be used in building date and time parts. For example, "this_month_1,_this_year" or "this_hour:23" is valid, while just "this_day" is not. The exact form of this combination will vary according to language. In some languages, the word for "this" changes according to which unit it is applied to. In other languages, there may be a single word which does the job. To list the word used as "this" for each unit, type:

```
display_time_info_$offset -language LANGUAGE_NM
```

5. adverbial offset
is a before/after kind of adjustment and may be used any number of times. This offset is recognized by the presence of "before", "on", or "after" in the time string. If present, adverbial offsets must appear first. These are the forms available:

```
DAY-NAME before
DAY-NAME on or before
DAY-NAME before or on
DAY-NAME after
DAY-NAME on or after
DAY-NAME after or on
SIGNED-OFFSETs before
SIGNED-OFFSETs after
```

When adverbial offsets are present, they partition a string into a series of adjustments followed by a base time. These sections are processed in a right to left manner. Referring to the first example below, there are 3 sections. First "6:00 am 400sec" is handled, supplying all necessary defaults and making the ordinary (400sec) offset adjustment. Then "Monday after" is applied to give a new value. And finally "2 wk -5min after" is applied to this new value to give the final value.

```
2 wk -5min after Monday after 6:00 am 400sec
20 minutes before now
2 days after today
2500 weeks after 1776-7-4
Tue after Mon on or after 11/1
```

This last item describes election day in the USA, i.e. the first Tuesday after the first Monday in November.

6. zone
   is the time zone to be used in making the conversion to Greenwich mean time, which is the internal form of all clock readings. It may be either a zone differential, or any of the zone abbreviations known at the site. A zone differential is a 5-character string, "sHHMM" ("s" is a sign, "HH" is a 2-digit hour, and "MM" is a 2-digit minute). This may only be used immediately following a time specification. "12:15-0330" says that 12:15 is the local time and -0330 specifies that the local time was generated by subtracting 3.5 hours from GMT. To list the zone abbreviations known at a site, type:

   ```
   display_time_info -zones
   ```

   If any defaults are needed, the current instant in time is broken down into years, months, days, etc. with respect to a "working zone". This working zone can make a great deal of difference, because, for example, at a given instant it can be Tuesday in New York and Wednesday in Bankok, or it can be 22:07 in London and 3:37 in Singapore. Thus, the zone is as important in applying defaults to week days and years as it is to hours and minutes.

   Many of the date/time commands allow a "-zone X" argument to specified. In this case, X may be any of the zones known at the site. It may NOT be a time differential.

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)
```

ARGUMENTS

string
: is the character representation of the clock reading desired. (Input)

clock
: is the computed clock value relative to the clock_in argument. (Output)

clock_in
is the clock time used to compute the clock value. (Input)

code
is a standard status code. (Output)

Name: convert_dial_message_

The convert_dial_message_ subroutine is used in conjunction with the dial_manager_ subroutine to control dialed terminals. It converts an event message received from the answering service over a dial control event channel into status information more easily used by the user.

Entry: convert_dial_message_$return_io_module

This entry point is used to process event messages from the answering service regarding the status of a dialed terminal or an auto call line. In addition to returning line status, this entry point also returns the device name and I/O module name for use in attaching the line through the iox_ subroutine. See the MPM Subroutines for further description of the iox_ subroutine.

USAGE

```
declare convert_dial_message_$return_io_module entry (fixed bin(71),
      char(*), char(*), fixed bin, 1 aligned, 2 bit(1) unal, 2 bit(1)
      unal, 2 bit(1) unal, 2 bit(33) unal, fixed bin(35));

call convert_dial_message_$return_io_module (message, channel_name,
      io_module, n_dialed, flags, code);
```

ARGUMENTS

message
is the event message to be decoded. (Input)

channel_name
is the name of the channel that has dialed up or hung up. (Output)

io_module
is the name of the iox_ I/O module to be used with the assigned device. (Output)

n_dialed
is the number of terminals currently dialed to the process or -1. (Output)

flags
is a bit string of the following structure: (Output)

```
dcl 1 flags        aligned,
      2 dialed_up  bit(1) unal,
      2 hung_up    bit(1) unal,
      2 control    bit(1) unal,
      2 pad        bit(33) unal;
```

Only the first three bits have meaning, and only one can be on at a time. See "Notes" below for complete details.

code
is a standard status code. (Output) See "Notes" below.

NOTES

The message may be either a control message or an informative message. Informative messages have flags.control off ("0"b), n_dialed is set to −1, channel is set to the name of the channel involved, io_module is set to the name of an I/O module, and either flags.dialed_up or flags.hung_up is on, indicating that the named channel has either just dialed up or just hung up. The io_module name is provided as a convenience; the caller is not required to use the name returned by this subroutine.

Control messages have flags.control on ("1"b), and n_dialed is set to the number of dialed terminals or −1. The code is either 0 (request accepted) or one of the following values:

error_table_$action_not_performed
the requested action was not performed; typically, this indicates an attempt to manipulate a channel that the requesting process can not control.

error_table_$ai_out_range
access to the requested channel is prohibited by AIM.

error_table_$bad_name
the channel_name does not conform to required syntax.

error_table_$badcall
the dial message was −1. The dial_manager_ subroutine will set dial_manager_arg.dial_message to −1 when an error occurs and there is no answering service dial_message to return.

error_table_$bigarg
the dial_out_destination is too long.

error_table_$dial_active
the process is already serving a dial qualifier.

error_table_$dial_id_busy
the dial_qualifier is already being used by another process.

error_table_$insufficient_access
the running process does not have the access permission required to perform the requested operation.

error_table_$invalid_resource_state
the channel is not configured to allow the requested operation.

error_table_$name_not_found
the dial_qualifier is not registered.

error_table_$no_connection
it was not possible to complete the connection, e.g., dial-out failure.

error_table_$no_dialok
the requesting process does not have the dialok attribute.

error_table_$order_error
an error occurred while processing an order on this channel.

error_table_$request_not_recognized
indicates a software error.

error_table_$resource_not_free
the requested channel is already in use.

error_table_$resource_unavailable
no channel could be found that satisfied required characteristics.

error_table_$resource_unknown
the channel specified does not exist.

error_table_$unable_to_check_access
typically indicates that the process does not have required access, but may indicate an administrative error.

error_table_$unimplemented_version
the version of the dial_manager_arg structure supplied is not supported by dial_manager_. This error code may also indicate an internal software error.

Name: convert_status_code_

The convert_status_code_ subroutine returns the short and long status messages from the standard status table containing the given status code. Status codes are described in the Programmer's Reference Manual.

USAGE

```
declare convert_status_code_ entry (fixed bin(35), char(8) aligned,
     char(100) aligned);

call convert_status_code_ (code, shortinfo, longinfo);
```

ARGUMENTS

code
: is a standard status code. (Input)

shortinfo
: is a short status message corresponding to code. (Output)

longinfo
: is a long status message corresponding to code; the message is padded on the right with blanks. (Output)

NOTES

If code does not correspond to a valid status code, shortinfo is "XXXXXXXX", and longinfo is "Code ddd", where ddd is the decimal representation of code.

Name: copy_

This subroutine produces a copy of a Multics non-directory branch. Name duplication is handled by nd_handler_.

USAGE

```
dcl copy_ external entry (ptr); call

copy_ (copy_options_ptr);
```

ARGUMENTS

copy_options_ptr
: is the pointer to copy_options structure (Input)

NOTES

All errors are handled via sub_err_. An attempt to copy a segment into itself is refused.

STRUCTURE

The copy_options structure is defined as follows:

```
1 copy_options              aligned based (copy_options_ptr),
  2 version                 char (8),
  2 caller_name             char (32) unal,
  2 source_dir              char (168) unal,
  2 source_name             char (32) unal,
  2 target_dir              char (168) unal,
  2 target_name             char (32) unal,
  2 flags,
    3 no_name_dup           bit (1) unaligned,
    3 raw                   bit (1) unaligned,
    3 force                 bit (1) unaligned,
    3 delete                bit (1) unaligned,
    3 target_err_switch     bit (1) unaligned,
    3 mbz                   bit (31) unaligned,
  2 copy_items              like copy_flags;
```

STRUCTURE ELEMENTS

version
: is the current version of this structure and has the value of the named constant COPY_OPTIONS_VERSION_1.

caller_name
: is the name of the program calling copy_, required when querying the user about duplicate names. See no_name_dup below.

source_dir
: is the absolute pathname of the directory containing the entry to be copied.

source_name
: is the name of the entry to be copied.

target_dir
: is the absolute pathname of the directory into which a copy of the entry is to be placed.

target_name
: is the name of the entry created to hold the copy of the original entry.

no_name_dup
: is set to "0"b if the user is to be queried in case of a duplication of the target_name and "1"b if there is to be no query, in which case sub_err_ is signalled.

raw
    is set to "0"b if copy_ is to honor the extended type of the entry, and "1"b if it is to regard it as a standard type entry.

force
    is set to "1"b if access to the target is to be forced.

delete
    is set to "1"b if the original is to be deleted after it is copied.

target_err_switch
    is set if an error occurred referencing the target.

mbz
    is reserved for future use and must be set to zero.

copy_items
    is structured like the copy_flags structure, which is defined in the include file copy_flags.incl.pl1. The structure is defined as follows:

```
1 copy_flags aligned based,
  2 names bit (1) unaligned,
  2 acl bit (1) unaligned,
  2 ring_brackets bit (1) unaligned,
  2 max_length bit (1) unaligned,
  2 copy_switch bit (1) unaligned,
  2 safety_switch bit (1) unaligned,
  2 dumper_switches bit (1) unaligned,
  2 entry_bound bit (1) unaligned,
  2 extend bit (1) unaligned,
  2 update bit (1) unaligned,
  2 mbz bit (26) unaligned;
```

When variables in the copy_flags structure have a value of "1"b, the designated attribute are copied to the new entry (as long as the attribute is supported for the type of entry). In the case of extend, the contents of the original entry may be appended to the end of the target entry. In the case of update, the contents of the original entry may replace the contents of the target entry.

Name: copy_acl_

The copy_acl_ subroutine copies the access control list (ACL) from one file, segment, multisegment file, or directory to another, replacing the current ACL if necessary.

USAGE

```
declare copy_acl_ entry(char(*), char(*), char(*), char(*), bit(1)
      aligned, fixed bin(35));

call copy_acl_ (source_dir, source_ent, target_dir, target_ent,
      target_error_sw, code);
```

ARGUMENTS

source_dir
is the pathname of the directory containing the source file or source directory whose ACL is to be copied. (Input)

source_ent
is the entryname of the source file or source directory. (Input)

target_dir
is the pathname of the directory containing the target file or target directory whose ACL is replaced. (Input)

target_ent
is the entryname of the target file or target directory. (Input)

target_error_sw
is "0"b if the status code reflects an error in listing the ACL of the source file or directory, and is "1"b if the code reflects an error in replacing the ACL of the target file or directory. (Output)

code
is a standard status code. (Output)

NOTES

An attempt to copy the ACL from a source file to a target directory, or from a source directory to a target file causes an error. Source and target must both be a file, or both a directory.

Links are chased in the processing of the source and target pathnames.

Name: copy_dir_

Copies a subtree from one point in the hierarchy to another, and optionally deletes the source subtree.

USAGE

```
dcl copy_dir_ entry(char(*), char(*), char(*), char(*), ptr, fixed
      bin(35));

call copy_dir_ (caller, source_dir, source_ename, target_dir,
      target_ename, pcopy_dir_options, code);
```

ARGUMENTS

caller
: is the name of the calling procedure. (Input)

source_dir
: is the pathname of the source directory. (Input)

source_ename
: is the source entry name. (Input)

target_dir
: is the pathname of the target directory. (Input)

target_ename
: is the target entry name. (Input)

pcopy_dir_options
: is a pointer to the copy_dir_options structure shown below under "Info Structure". (Input)

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

INFO STRUCTURE

The following structure is declared in copy_dir_options.incl.pl1:

```
dcl 1 copy_dir_options        aligned based(pcopy_dir_options),
      2 version               fixed bin,
      2 entry_control         aligned,
        3 link                bit(1) unal,
        3 seg                 bit(1) unal,
        3 dir                 bit(1) unal,
        3 msf                 bit(1) unal,
        3 nnlk                bit(1) unal,
        3 pad1                bit(31) unal,
      2 operation_control     aligned,
        3 delete              bit(1) unal,
        3 brief               bit(1) unal,
        3 force               bit(1) unal,
        3 replace             bit(1) unal,
        3 update              bit(1) unal,
        3 acl                 bit(1) unal,
        3 primary             bit(1) unal,
        3 link_translation    bit(1) unal,
        3 chase               bit(1) unal,
        3 parent_ac_sw        bit(1) unal,
        3 pad2                bit(26) unal;

dcl copy_dir_options_version_0 fixed bin init(0) int static options(constant);
dcl pcopy_dir_options         ptr;
```

STRUCTURE ELEMENTS

version
    is the version number of this structure, currently copy_dir_options_version_0.

link
    if set to "1"b then links are copied.

seg
    if set to "1"b then segments are copied.

dir
    if set to "1"b then inferior directories are copied. If this is not set then the subtree is not walked.

msf
    if set to "1"b then multisegment-files are copied.

nnlk
    if set to "1"b then non-null links are copied.

pad1
is unused and must be zero.

delete
if set to "1"b then the source_dir is deleted after the copying is complete.

brief
if set to "1"b suppresses the printing of warning messages such as "Bit count is inconsistent with current length" and "Current length is not the same as records used".

force
if set to "1"b executes, when target_dir already exists, without asking the user. If force is not set, the user is queried.

replace
if set to "1"b deletes the existing contents of target_dir before the copying begins. If target_dir is non-existent or empty, this control argument has no effect. The default is to append the contents of source_dir to the existing contents of target_dir. Setting of replace conflicts with the setting of update, and error_table_$inconsistent is returned.

update
if set to "1"b causes copying of only those entries in source_dir that have comparable entries in target_dir. Setting of update conflicts with the setting of replace, and error_table_$inconsistent is returned.

acl
if set to "1"b gives the ACL on the source_dir entry to its copy in target_dir. Although initial ACLs are still copied, they are not used in setting the ACL of the new entries when not set.

primary
if set to "1"b only primary names are copied. If not set, all the names of the selected entries are copied.

link_translation
if set to "1"b then links will be translated. If there are references to the source directory in the link pathname of a link being copied, the link pathname is changed to refer to the target directory.

chase
if set to "1"b copies the target of a link. Chasing links eliminates link translation.

parent_ac_sw
if set to "1"b when target directories need creating. The access class of the target_dir is obtained from the target's parent directory. Otherwise, the access class is determined from the source_dir. This switch may be used by privileged applications to make a downgraded copy of an upgraded hierarchy. The caller must have previously set the seg and dir AIM privileges in order to read the contents to the upgraded hierarchy.

pad2
is unused and must be zero.

ACCESS REQUIRED

Status permission is required for source_dir and all of the directories in its tree. Status permission is required for the directory containing source_dir. Read access is required on all files under source_dir. Append and modify permission are required for the directory containing target_dir if target_dir does not exist. Modify and append permission are required on target_dir if it already exists.

If acl is not specified, the system default ACLs are added, then the initial ACL for the containing directory is applied (which may change the system supplied ACL). Initial ACLs are always copied for the current ring of execution.

NOTES

If target_dir already exists and force is not specified, the user is so informed and asked if processing should continue. If target_dir is contained in source_dir, an appropriate error message is printed and the subroutine returns.

If name duplication occurs while appending the source_dir to the target_dir and the name duplication is between directories; the user is queried whether processing should continue. If the user answers yes, the contents of the directory are copied (appended) but none of the attributes of that directory are copied. If the answer is no, the directory and its subtree is skipped. If name duplication should occur between segments, the user is asked whether to delete the existing one in target_dir.

If replace is specified or target_dir does not exist, name duplication does not occur.

If part of the tree is not copied (by specifying a storage system entry key), problems with link translation may occur. If the link target in the source_dir tree was in the part of the tree not copied, there may be no corresponding entry in the target_dir tree. Hence, translation of the link causes the link to become null.

Name: cpu_time_and_paging_

The cpu_time_and_paging_ subroutine returns the virtual CPU time used by the calling process since it was created as well as a measure 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);
```

ARGUMENTS

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

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

pd_faults
: was previously the total number of page faults from the paging device for the calling process. This value is always returned as zero. (Output)

Name: create_data_segment_

The create_data_segment_ subroutine is used in conjunction with the create_data_segment command to create a standard object segment from PL/I data structures passed to it as parameters. The create_data_segment_ subroutine is called from a PL/I program that has defined in it either one or two specific PL/I structures, whose contents are to be placed in the text and/or static sections of the object segment to be created. The level-2 structure component names become entry point names for the object segment, i.e., names that can be found by links so that other programs may reference the data by name.

USAGE

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

call create_data_segment_ (cds_arg_ptr, code);
```

ARGUMENTS

cds_arg_ptr
is a pointer to a structure (see "Structure" below) containing information to be passed to the create_data_segment_ subroutine, specifying the structures to be used to create the object segment. (Input)

code
is a standard status code. (Output) It can be error_table_$translation_failed if no object segment is created.

STRUCTURE

The structure that passes information to the create_data_segment_ subroutine can be found in the library include file cds_args.incl.pl1. It is declared as follows:

```
dcl 1 cds_args                based aligned,
      2 sections (2),
        3 p                   ptr,
        3 len                 fixed bin (18),
        3 struct_name         character (32),
      2 seg_name              character (32),
      2 num_exclude_names     fixed bin,
      2 exclude_array_ptr     ptr,
      2 switches,
        3 defs_in_link        bit(1) unal,
        3 separate_static     bit(1) unal,
        3 have_text           bit(1) unal,
        3 have_static         bit(1) unal,
        3 pad                 bit(32) unal;
```

STRUCTURE ELEMENTS

sections
describe the PL/I structures in the calling program that are used to define the text and static sections of the object segment; section (1) describes the structure to be used for the text section, (if cds_args.switches.have_text is on), and section (2) describes the structure to be used for the static section (if cds_args.switches.have_static is on).

p
is a pointer to a region of data, described by the appropriate structure, whose contents are to be copied into the appropriate section of the object segment.

len
is the length, in words, of the region pointed to by p. It must be the same as the word size of the appropriate structure.

struct_name
is the level-1 name of the structure in the calling process that is used to define the entry point (segdef) names of the corresponding section of the object segment.

The structure must be known throughout the PL/I language scoping rules to the block that contains the call to create_data_segment_.

This structure must not be an array at its outermost level. It can be of any storage class and can contain arbitrary "like" attributes.

All level-2 names in this structure will become entry point (segdef) names in the corresponding section of the object segment, unless excluded by the exclude array (see below). The location of the entry point (segdef) will be at an offset in the corresponding section of the object segment equal to the offset of the given component in the supplied structure. Hence, only a name defining a field that begins on a word boundary may be validly used.

seg_name
is the entryname of the object segment to be created in the working directory. The seg_name must be the same as the entry name of their source segment without the suffix ".cds".

num_exclude_names
is the number of names in the exclude array. It should be 0 if there is no exclude array. (See below.)

exclude_array_ptr
is a pointer to the exclude array, if one is provided. It may be null.

The exclude array is an array of character(32) star names (see the match_star_names_ subroutine) that select those level-2 names in the supplied structures that should not be made into entry point names. For instance, the names "pad*" and "mbz*" would eliminate all names beginning with either mbz or pad.

If no exclude array is supplied, all level-2 names are made into entry point names.

switches
control the options of the create_data_segment_ subroutine.

defs_in_link
controls placement of the definition section.
"1"b places definition section of the object segment in its linkage section; this option creates a nonstandard object segment, and should not be used.
"0"b places definitions contiguous to the text section.

separate_static
: controls whether the object segment has a separate static section.
"1"b separate static section
"0"b static resides in the linkage section

have_text
: indicates whether or not there is a text section.
"1"b cds_args.sections(1) describes a structure to be used for defining the text section of the object segment
"0"b there is no text section (zero length)

have_static
: indicates whether or not there is a static section.
"1"b cds_args.sections(2) describes a structure to be used for defining the static section of the object segment
"0"b there is no static section (zero length)

pad
: is reserved, and must be all zeros.

NOTES

The brief translator name placed in object segments produced by the create_data_segment_ subroutine is cds.

If the defs_in_link switch is supplied as on ("1"b), then a nonrelocatable, nonstandard object segment is produced.

All text and static-resident information created is supplied with absolute relocation. Hence, one must be wary of threads and pointers in one's structures, as they are not relocated if the object segment is bound.

The program that calls the create_data_segment_ subroutine must be in the PL/I language. It must be compiled with the -table control argument. The create_data_segment command provides for this.

It is essential that structures specified by cds_args.sections be at least referenced in the calling program, or they are not described in the runtime symbol table.

The create_data_segment_ program, in its capacity as a translator, issues diagnostic messages on the terminal, as opposed to returning detailed status codes.

All regions of the text and/or static sections not explicitly set by the calling program, whether via "init" attributes or explicit code, may not be assumed to contain zero or any other quantity.

Name: create_ips_mask_

The create_ips_mask_ subroutine returns a bit string that can be used to disable specified ips (interprocess signal) interrupts (also known as ips signals).

USAGE

```
declare create_ips_mask_ entry (ptr, fixed bin, bit(36) aligned);

call create_ips_mask_ (array_ptr, lng, mask);
```

ARGUMENTS

array_ptr
is a pointer to an array of ips names that are declared as char(32) aligned. (Input)

lng
is the number of elements in the array pointed to by array_ptr. (Input)

mask
is a mask that disables all of the ips signals named in the array. (See "Notes" below.)

NOTES

If any of the names are not valid ips signal names, the condition create_ips_mask_err is signalled. Currently, the allowed ips names are:

quit
cput
alrm
neti
sus_
trm_
wkp_
pgt_
system_shutdown_scheduled_
dm_shutdown_scheduled_

If the first name in the array is -all, then a mask is returned that masks all interrupts.

The returned mask contains a "0"b in the bit position corresponding to each ips name in the array and a "1"b in all other bit positions. The bit positions are ordered as in the above list. It should be noted that it is necessary to complement this mask (using a statement of the form "mask = ^mask") in cases where the requirement is for a mask with "1" bits corresponding to specified interrupts. An ips mask is used as an argument to the hcs_$reset_ips_mask and hcs_$set_ips_mask entry points.

Name: cross_ring_io_

Entry: cross_ring_io_$allow_cross

The cross_ring_io_$allow_cross entry point must be called to allow use of an I/O switch via cross-ring attachments from an outer ring. The call must be made in the inner ring before the outer ring attempts to attach.

USAGE

```
declare cross_ring_io_$allow_cross entry (char(*), fixed bin,
      fixed bin(35));

call cross_ring_io_$allow_cross (switch_name, ring, code);
```

ARGUMENTS

switch_name
: is the inner ring switch name. (Input)

ring
: is the highest validation level from which switch_name may be used. (Input)

code
: is a standard status code. (Output)

NOTES

This entry may be called more than once with the same switch_name argument. Subsequent calls are ignored.

Name: cu_

The cu_ subroutine contains a number of useful command utility programs that provide functions not directly available in the PL/I language. Although the various cu_ entry points are designed primarily for the use of command writers, many may prove useful to Multics users and subsystem developers. The entry points can be divided into four functional categories: argument processing, ready states, stack utility, and miscellaneous.

The following is a list of all the entry points in the cu_ subroutine, divided into the four categories. A brief explanation of each category follows the list. The entry points themselves are then described, in alphabetical order.

Argument Processing

cu_$af_arg_count
cu_$af_arg_count_rel
cu_$af_arg_ptr
cu_$af_arg_ptr_rel
cu_$af_return_arg
cu_$af_return_arg_rel
cu_$arg_count
cu_$arg_count_rel
cu_$arg_list_ptr
cu_$arg_ptr
cu_$arg_ptr_rel
cu_$generate_call

Ready States

cu_$get_ready_mode
cu_$get_ready_procedure
cu_$ready_proc
cu_$reset_ready_procedure
cu_$set_ready_mode
cu_$set_ready_procedure

Command Processor Escape

cu_$cp
cu_$get_command_name
cu_$get_command_name_rel
cu_$get_command_processor
cu_$reset_command_processor
cu_$set_command_processor

Stack Utility

cu_$grow_stack_frame
cu_$shrink_stack_frame
cu_$stack_frame_ptr
cu_$stack_frame_size

Active String Evaluation

cu_$evaluate_active_string
cu_$get_evaluate_active_string
cu_$reset_evaluate_active_string
cu_$set_evaluate_active_string

Command Error Handlers

cu_$cl
cu_$get_cl_intermediary
cu_$reset_cl_intermediary
cu_$set_cl_intermediary

Ring Validation Level

cu_$level_get
cu_$level_set

Miscellaneous

cu_$caller_ptr
cu_$decode_entry_value
cu_$make_entry_value

AIDS IN ARGUMENT PROCESSING

These entry points are designed to be used by such programs as commands and active functions, which in turn may be invoked with a variable number of arguments. The entry points are tools to be used in obtaining the number of arguments, or a pointer to an argument or argument list, or to reference the return argument of an active function. Knowledge of the details of implementation is not necessary.

READY STATES

These entry points enable the user to invoke a ready procedure, to determine the state of a ready procedure or ready mode, or, if need be, to change it.

STACK UTILITY

These entry points enable the user to perform operations on his stack frame; in general, they are for advanced applications.

ACTIVE STRING EVALUATION

These entry points enable the user to evalueate active strings within a closed subsystem environment which are a sequence of one or more active function invocations with their arguments.

COMMAND ERROR HANDLERS

These entry points enable the user to handle any error conditions that can be signalled within a closed subsystem environment by passing control to the procedure entry point currently defined as the standard error handler. A diagnostic message is printed and a procedure is called to reenter command level.

COMMAND PROCESSOR ESCAPE

These entry points permit the user to escape from the closed subsystem environment to execute other commands by passing the execute request to the current procedure entry point defined as the command processor.

RING VALIDATION LEVEL

These entry points enable the user to change the current protection ring validation level for procedures that must distinguish the periods of time when it is acting in behalf of itself (i.e., in its own ring) and when it is acting in behalf of another procedure that can be in an outer (i.e., less privileged) protection ring.

MISCELLANEOUS

These entry points enable the user to perform a variety of tasks that do not fit any of the above categories.

Entry: cu_$af_arg_count

This entry point should be called by an active function. It returns to its caller the number of arguments passed to the caller by its caller, not including the active function return argument. If the caller has not been invoked as an active function, a standard status code is returned, and, if the code is error_table_$not_act_fnc, nargs is the number of arguments in the call (similar to the cu_$arg_count entry point described below).

USAGE

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

call cu_$af_arg_count (nargs, code);
```

ARGUMENTS

nargs
is the number of input arguments passed to the caller. (Output)

code
is a standard status code. (Output)
error_table_$nodescr
no argument descriptors were passed to the caller or an incorrect argument list header was encountered.
error_table_$not_act_fnc
the caller was not invoked as an active function.

NOTES

This entry point and the five following entry points beginning with $af_ have been provided so that active functions need not have knowledge of the mechanism for returning arguments.

The entry points cu_$af_arg_count and cu_$af_arg_count_rel are retained for historical reasons; active function procedures should call cu_$af_return_arg and cu_$af_return_arg_rel instead to obtain the location and maximum length of the return argument as well as the arg_count. This information will be needed for the active function to return a value. When the procedure is invoked as an active function, the value of arg_count returned by cu_$af_arg_count will be one less than the value returned by cu_$arg_count, otherwise they will be the same.

Entry: cu_$af_arg_count_rel

This entry point is similar to cu_$af_arg_count, but instead of looking in the argument list of its caller, it is given a pointer to the argument list.

USAGE

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

call cu_$af_arg_count_rel (nargs, arg_list_ptr, code);
```

ARGUMENTS

nargs
is the number of input arguments passed to the caller. (Output)

arg_list_ptr
is a pointer to an argument list. (Input)

code
is a standard status code. (Output)
error_table_$nodescr
no argument descriptors were passed to the caller or an incorrect argument list header was encountered
error_table_$not_act_fnc
the caller was not invoked as an active function

Entry: cu_$af_arg_ptr

This entry point assumes it has been called by an active function. It operates in the same fashion as cu_$arg_ptr (described below), except that it verifies that the caller was invoked as an active function, and does not allow the return argument to be accessed. If the (i+1)st argument does not exist, the code error_table_$noarg is returned. The return argument is always the last one; thus, use of this entry point and cu_$af_return_arg allows the active function to be independent of the position of the return argument in the argument list (see "Notes" under cu_$af_arg_count above).

USAGE

```
declare cu_$af_arg_ptr entry (fixed bin, ptr, fixed bin(21),
      fixed bin(35));

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

ARGUMENTS

arg_no
is the number of the desired argument. (Input)

arg_ptr
is a pointer to the unaligned character-string argument specified by arg_no. (Output) It is set to the null value if any error is encountered.

arg_len
is the length (in characters) of the argument specified by arg_no. (Output) It is set to 0 if any error is encountered.

code
is a standard status code. (Output)
error_table_$nodescr
the argument list does not contain descriptors. In this case, arg_len is set to zero.
error_table_$not_act_fnc
the caller was not invoked as an active function.
error_table_$noarg
the program does not have an arg_no'th argument. In this case, arg_ptr is set to null and arg_len is set to zero.

Entry: cu_$af_arg_ptr_rel

This entry point is similar to cu_$af_arg_ptr but instead of looking in the argument list of its caller, it is given a pointer to the argument list.

USAGE

```
declare cu_$af_arg_ptr_rel entry (fixed bin, ptr, fixed bin(21),
      fixed bin(35), ptr);

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

ARGUMENTS

arg_no
is the number of the desired argument. (Input)

arg_ptr
is a pointer to the unaligned character-string argument specified by arg_no. (Output) It is set to the null value if any error is encountered.

arg_len
is the length (in characters) of the argument specified by arg_no. (Output) It is set to 0 if any error is encountered.

arg_list_ptr
is a pointer to an argument list. (Input)

code
is a standard status code. (Output)
error_table_$nodescr
the argument list does not contain descriptors. In this case, arg_len is set to zero.
error_table_$not_act_fnc
the caller was not invoked as an active function.
error_table_$noarg
the program does not have an arg_no'th argument. In this case, arg_ptr is set to null and arg_len is set to zero.

Entry: cu_$af_return_arg

This entry point assumes it has been called by an active function. It makes the active function's return argument available as described in "Notes" below. It is provided to permit writing of active functions that accept an arbitrary number of arguments (see "Notes" under cu_$af_arg_count above).

USAGE

```
declare cu_$af_return_arg entry (fixed bin, ptr, fixed bin(21),
      fixed bin(35));

declare return_string char (max_length) varying based (rtn_string_ptr);

call cu_$af_return_arg (nargs, rtn_string_ptr, max_length, code);
```

ARGUMENTS

nargs
is the number of input arguments passed to the caller. (Output)

rtn_string_ptr
is a pointer to the varying return argument of the active function. (Output)

max_length
is the maximum length of the varying string pointed to by rtn_string_ptr. (Output)

code
is a standard status code. (Output)
error_table_$nodescr
no argument descriptors were passed to the caller or an incorrect argument list header was encountered.
error_table_$not_act_fnc
the caller was not invoked as an active function.

NOTES

An active function that takes an arbitrary number of arguments uses this entry point to return a value. It calls the entry point to get a pointer to the return argument and to get its maximum length. It declares the based varying string, return_string, as described above. It then assigns its return value to return_string. Even if error_table_$not_act_fnc is returned, nargs will be set to the proper value.

Entry: cu_$af_return_arg_rel

This entry point is similar to cu_$af_return_arg, but instead of looking in the argument list of its caller, it is given a pointer to the argument list.

USAGE

```
declare cu_$af_return_arg_rel entry (fixed bin, ptr, fixed bin(21),
      fixed bin(35), ptr);

call cu_$af_return_arg_rel (nargs, rtn_string_ptr, max_length, code,
      arg_list_ptr);
```

ARGUMENTS

nargs
: is the number of input arguments passed to the caller. (Output)

arg_list_ptr
: is a pointer to an argument list. (Input)

rtn_string_ptr
: is a pointer to the varying return argument of the active function. (Output)

max_len
: is the maximum length of the varying string pointed to by rtn_string_ptr. (Output)

code
: is a standard status code. (Output)
  error_table_$nodescr
    no argument descriptors were passed to the caller or an incorrect argument list header was encountered.
  error_table_$not_act_fnc
    the caller was not invoked as an active function.

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, fixed bin (35));

call cu_$arg_count (arg_count, code);
```

ARGUMENTS

arg_count
    is the number of arguments. (Output)

code
    is a standard status code. (Output)
    error_table_$nodescr
        no argument descriptors were passed to the caller or an incorrect argument list header was encountered.
    error_table_$active_function
        the caller was invoked as an active function.

NOTES

Even if the code is nonzero, arg_count may still be valid. If error_table_$active_function is returned, the arg_count will be the total number of arguments, including the active function return argument. This number may differ from that returned by cu_$af_return_arg, described below. This entry point is intended for use with command procedures that may not be used as active functions.

For compatibility with old programs, the code argument may be omitted.

Entry: cu_$arg_count_rel

This entry point returns the number of arguments in any specified argument list.

USAGE

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

call cu_$arg_count_rel (arg_count, arg_list_ptr, code);
```

ARGUMENTS

arg_count
    is the number of arguments. (Output)

arg_list_ptr
    is a pointer to an argument list. (Input) This pointer can be obtained by calling cu_$arg_list_ptr, described below.

code
    is a standard status code. (Output)
    error_table_$nodescr
        no argument descriptors were passed to the owner of the argument list or an incorrect argument list header was encountered.
    error_table_$active_function
        the owner of the argument list was invoked as an active function.

Entry: cu_$arg_list_ptr

It is sometimes desirable to design a PL/I procedure to accept a variable number of arguments of varying data types (e.g., the ioa_ subroutine). In these cases, the PL/I procedure must be able to interrogate its argument list directly to determine the number, type, and location of each argument. The cu_$arg_list_ptr entry point is designed for use in such cases and returns a pointer to the argument list of its caller.

USAGE

```
declare cu_$arg_list_ptr entry (ptr);

call cu_$arg_list_ptr (arg_list_ptr);
```

ARGUMENTS

arg_list_ptr
is a pointer to the argument list of the caller. (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 a variable-length unaligned character string (i.e., declared char(*)). This entry point returns a pointer to the character-string argument specified by the argument number and also returns the length of the argument.

USAGE

```
declare cu_$arg_ptr entry (fixed bin, ptr, fixed bin(21), fixed
      bin(35));

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

ARGUMENTS

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

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

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

dcl argument char(arg_len) based(arg_ptr);

code
: is a standard status code. (Output)
error_table_$nodescr
: the argument list does not contain descriptors. In this case, argl_len set is to zero.
error_table_$noarg
: the program does not have an arg_no'th argument. In this case, arg_ptr is set to null and arg_len is set to zero.

NOTES

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_$arg_ptr_rel

Some PL/I procedures may need to reference arguments passed to other procedures. This entry point permits a procedure to reference arguments in any specified argument list.

USAGE

```
declare cu_$arg_ptr_rel entry (fixed bin, ptr, fixed bin(21),
      fixed bin(35), ptr);

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

ARGUMENTS

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

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

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

code
is a standard status code. (Output)
error_table_$nodescr
the argument list does not contain descriptors. In this case, argl_len is set to zero.
error_table_$noarg
the program does not have an arg_no'th argument. In this case, arg_ptr is set to null and arg_len is set to zero.

arg_list_ptr
is a pointer to the argument list from which this argument is being extracted. (Input) This pointer can be determined by calling cu_$arg_list_ptr in the program whose argument list is to be processed and then passing it to the program requesting reference to the argument list.

Entry: cu_$caller_ptr

This entry point allows a routine to obtain a pointer to its caller. The pointer that is returned points to the instruction within the text section after the instruction that called out.

USAGE

```
declare cu_$caller_ptr entry (ptr);

call cu_$caller_ptr (caller_ptr);
```

ARGUMENTS

caller_ptr
is a pointer into the text section of the caller. (Output) If null, the invoker of the cu_ subroutine has no caller.

Entry: cu_$cl

The cu_$cl entry point is called by all standard error handlers after printing a diagnostic message. This entry point passes control to the procedure specified by the last call to cu_$set_cl_intermediary. It takes an optional argument which is passed directly to that procedure. If no such procedure has been specified (the norm), control is passed to the standard procedure, which establishes a new command level (see Notes below).

USAGE

```
declare cu_$cl entry (1 aligned, 2 bit(1) unaligned, 2 bit(35)
      unaligned);

dcl 1 flags          aligned,
       2 reset_sw  bit (1) unaligned,
       2 mbz       bit (35) unaligned;

call cu_$cl (flags);
```

ARGUMENTS

flags.reset_sw
: specifies whether the intermediary procedure should perform a "resetread" control order on the standard "user_i/o" I/O switch. (Input)
"1"b do a "resetread" operation,
"0"b do not perform a "resetread" operation.

flags.mbz
: is reserved for future use and must be "0"b. (Input)

NOTES

If no argument is given, cu_$cl passes a static argument with flags.reset_sw set to "1"b.

Establishing a new command level consists of saving the attachments of the standard I/O switches (user_input, user_output, and error_output), restoring these attachments to their default state, and entering a new loop of reading and executing command lines. If the "start" command is issued, the attachments of the standard I/O switches are restored to the state saved above and control is returned to the caller of cu_$cl to continue from the interrupted exection. If the "release" command is issued, the interrupted execution is aborted, the I/O switches are not restored, and control is returned to the previous command level.

Entry: cu_$cp

The cu_$cp entry point, called when a Multics command line is recognized, passes the command line to the currently defined command processor for processing. Some standard Multics commands (e.g., qedx) permit the user to escape from them to execute other commands. In this case, the escapable command passes the line to be executed to the command processor. The cu_$cp entry point is called by any standard command that recognizes other Multics command lines.

USAGE

```
declare cu_$cp entry (ptr, fixed bin(21), fixed bin(35));

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

ARGUMENTS

line_ptr
is a pointer to the beginning of a character string containing a command line to be processed. (Input)

line_len
is the length of the command line in characters. (Input)

code
is a standard status code or the nonstandard code 100. (Output) If an error has been detected, 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. A returned code of 100 indicates that the command line is blank and no ready message should be printed.

Entry: cu_$decode_entry_value

This entry point extracts the pointer components of a PL/I entry value.

USAGE

```
declare cu_$decode_entry_value entry (entry, ptr, ptr);

call cu_$decode_entry_value (entry_value, ep_ptr, env_ptr);
```

ARGUMENTS

entry_value
is the entry value to be decoded. (Input)

ep_ptr
is the entry point pointer, i.e., a pointer to the actual executable code. (Output)

env_ptr
is the environment pointer. (Output)

NOTES

Using the codeptr and environmentptr PL/I built-in functions are preferable to using the cu_$decode_entry_value subroutine.

Entry: cu_$evaluate_active_string

This entry point evaluates an active string. An active string consists of one or more active function invocations and their arguments. Other entries are provided for subsystem writers to specify the procedure to be called by this entry.

USAGE

```
declare cu_$evaluate_active_string entry (ptr, char(*), fixed bin,
     char(*) varying, fixed bin (35));

call cu_$evaluate_active_string (info_ptr, active_string, string_type,
     return_value, code);
```

ARGUMENTS

info_ptr
: is reserved for future expansion and must be null. (Input)

active_string
: is the active string to be evaluated. (Input) It should not include the outermost brackets.

string_type
: specifies the type of active string to be evaluated. (Input) Its possible values are:

NORMAL_ACTIVE_STRING
: the active string return value should be rescanned for all command processor constructs. ([...])

TOKENS_ONLY_ACTIVE_STRING
: the active string return value should be rescanned only for whitespace and quotes. (|[...])

ATOMIC_ACTIVE_STRING
: the active string return value should not be rescanned. (||[...])

return_value
: is the result of the evaluation. (Output)

code
: is a standard system status code. (Output) If its value is error_table_$command_line_overflow, the maximum length of the return_value argument was not large enough to hold the result of the evaluation. In this case, the result will be truncated.

NOTES

The constants used above for string_type are defined in the cp_active_string_types.incl.pl1 include file. The active string should not be enclosed in brackets.

Entry: cu_$generate_call

The cu_$generate_call entry point is used to generate a standard call to a specified procedure with a specified argument list. This call is designed for cases in which a PL/I procedure has explicitly built an argument list from its input data. The principal use of this entry is by command processors that call a command with an argument list built from a command line input from a terminal.

USAGE

```
declare cu_$generate_call entry (entry, ptr);

call cu_$generate_call (proc_entry, a_ptr);
```

ARGUMENTS

proc_entry
: is the procedure entry point to be called. (Input)

a_ptr
: is a pointer to the argument list to be passed to the called procedure. (Input)

Entry: cu_$get_cl_intermediary

This entry point returns to the caller the procedure entry currently being invoked by a call to cu_$cl.

USAGE

```
declare cu_$get_cl_intermediary entry (entry);

call cu_$get_cl_intermediary (proc_entry);
```

ARGUMENTS

proc_entry
: is the procedure entry being called by the standard error handlers after printing a diagnostic message. (Output)

Entry: cu_$get_command_name

This entrypoint allows a routine called via the command processor to obtain the name used on the command line to invoke the procedure. The values returned are as follows:

Name used on command line	Returned Value
name	name
path>name	path>name
name$entrypoint	name$entrypoint
path>name$entrypoint	path>name$entrypoint

USAGE

```
declare cu_$get_command_name entry (ptr, fixed bin (21), fixed bin(35));

call cu_$get_command_name (command_name_ptr, command_name_length,
     error_code);
```

ARGUMENTS

command_name_ptr
Is a pointer to the command name of length command_name_length. If null, the command name is unavailable for the current routine. (Output)

command_name_length
Is the length of the returned command name. If zero the command name is unavailable for the current routine. (Output)

error_code
Is a standard status code. If the command name is unavailable its value is equal to error_table_$no_command_name_available. (Output)

Entry: cu_$get_command_name_rel

This entrypoint allows a routine called via the command processor to obtain the name used on the command line to invoke the procedure. The values returned are as follows:

Name used on command line	Returned Value
name	name
path>name	path>name
name$entrypoint	name$entrypoint
path>name$entrypoint	path>name$entrypoint

USAGE

```
declare cu_$get_command_name_rel entry (ptr, fixed bin (21),
      fixed bin(35), ptr);

call cu_$get_command_name_rel (command_name_ptr, command_name_length,
      error_code, arglist_ptr);
```

ARGUMENTS

command_name_ptr
: Is a pointer to the command name of length command_name_length. If null, the command name is unavailable for the current routine. (Output)

command_name_length
: Is the length of the returned command name. If zero the command name is unavailable for the current routine. (Output)

error_code
: Is a standard status code. If the command name is unavailable its value is equal to error_table_$no_command_name_available. (Output)

arglist_ptr
: Is a pointer to the argument list from which this argument is being extracted. This pointer can be determined by calling cu_$arg_list_ptr in the program whose argument list is to be processed and then passing it to the program requesting reference to the argument list. (Input)

Entry: cu_$get_command_processor

This entry point returns to the caller the entry value of the procedure currently being invoked by a call to cu_$cp.

USAGE

```
declare cu_$get_command_processor entry (entry);

call cu_$get_command_processor (proc_entry);
```

ARGUMENTS

proc_entry
is the procedure entry point to which control is passed upon receiving a call to cu_$cp. (Output)

Entry: cu_$get_evaluate_active_string

This entry point returns to the caller the entry value of the procedure currently being invoked by a call to cu_$evaluate_active_string.

USAGE

```
declare cu_$get_evaluate_active_string entry (entry);

call cu_$get_evaluate_active_string (active_string_procedure);
```

ARGUMENTS

active_string_procedure
is the procedure entry point to which control is passed upon receiving a call to cu_$evaluate_active_string. (Output)

Entry: cu_$get_ready_mode

This entry point returns the value of the internal static ready flags.

USAGE

```
declare cu_$get_ready_mode entry (1 aligned, 2 bit(1) unaligned,
      2 bit(35) unaligned);

dcl 1 mode aligned,
       2 ready_sw bit(1) unaligned,
       2 mbz bit(35) unaligned;

call cu_$get_ready_mode (mode)
```

ARGUMENTS

mode.ready_sw
is the current value of the static ready switch. (Output)
"1"b print ready message.
"0"b do not print ready message.

mode.mbz
is reserved for future use and must be "0"b. (Output)

Entry: cu_$get_ready_procedure

This entry point returns the entry value of the current ready procedure of the process.

USAGE

```
declare cu_$get_ready_procedure entry (entry);

call cu_$get_ready_procedure (ready_entry);
```

ARGUMENTS

ready_entry
: is the current ready procedure. (Output)

Entry: cu_$grow_stack_frame

This entry point allows its caller to allocate temporary storage by extending the caller's current stack frame.

USAGE

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

call cu_$grow_stack_frame (len, data_ptr, code);
```

ARGUMENTS

len
: is the length (in words) by which the caller's stack frame is to be extended. (Input) The standard Multics call, push, and return discipline requires that stack frames begin on mod 16 word boundaries. Therefore, if len is not a mod 16 number, the stack frame is grown by the next mod 16 quantity greater than len.

data_ptr
: is a pointer to the first location of len words allocated in the caller's stack frame. (Output)

code
: is a standard status code. (Output)

NOTES

The cu_$grow_stack_frame and cu_$shrink_stack_frame entry points are for advanced subsystems writers only and should be used only when absolutely necessary. Most PL/I programs can be written to use begin blocks to allocate extra storage in the current stack frame. The entry points rely on internal workings of the PL/I compiler that are not guaranteed to continue working forever.

Entry: cu_$level_get

The cu_$level_get entry point is used to obtain the current ring validation level. This entry point is normally used prior to a call to cu_$level_set to save the current validation level.

USAGE

```
declare cu_$level_get entry (fixed bin);

call cu_$level_get (level);
```

ARGUMENTS

level
    is the current ring validation level. (Output)

Entry: cu_$level_set

The cu_$level_set entry point is used to change the current protection ring validation level. This entry point is useful for procedures that must distinguish the periods of time when the procedure is acting in behalf of itself (i.e., its own ring) and when it is acting in behalf of another procedure that can be in an outer (i.e., less privileged) protection ring.

USAGE

```
declare cu_$level_set entry (fixed bin);

call cu_$level_set (level);
```

ARGUMENTS

level
    specifies the new protection validation level and must be greater than or equal to the current ring number. (Input) The current ring number can be determined by the get_ring_ subroutine.

Entry: cu_$make_entry_value

The cu_$make_entry_value entry point constructs a PL/I entry value from a pointer to an entry point. The environment pointer of the entry value will be null, so the entry point pointer must point to an external procedure.

USAGE

```
declare cu_$make_entry_value entry (pointer, entry);

call cu_$make_entry_value (ep_ptr, entry_value);
```

ARGUMENTS

ep_ptr
is the entry point pointer. (Input)

entry_value
is the entry value. (Output)

Entry: cu_$ready_proc

The ready_proc entry point is used to call the current ready procedure of the process. It takes an optional argument, which it passes to the ready procedure. The ready procedure is automatically invoked by the listener after each command line is processed. The ready procedure of the standard command environment prints the ready message. The cu_$set_ready_procedure subroutine can be called to change the ready procedure.

USAGE

```
declare cu_$ready_proc entry;

call cu_$ready_proc ();

or:

dcl cu_$ready_proc entry (1 aligned, 2 bit(1) unaligned,
      2 bit(35) unaligned);

dcl 1 mode       aligned,
       2 ready_sw bit(1) unaligned,
       2 mbz      bit(35) unaligned;

call cu_$ready_proc (mode);
```

ARGUMENTS

mode.ready_sw
specifies whether the ready procedure should print a ready message. (Input)
"1"b print ready message
"0"b do not print ready message

mode.mbz
is reserved for future use and must be "0"b. (Input)

NOTES

If no argument is given, a static ready switch is passed to the ready procedure. The default value of the static ready switch is "1"b. The value of the static ready switch can be obtained using the cu_$get_ready_mode entry point and changed using the cu_$set_ready_mode entry point. The listener invokes the cu_$ready_proc entry point without an argument. The ready_off command turns off the static ready switch, the ready_on command turns it on, and the ready command calls the cu_$ready_proc entry point with an argument whose ready_sw component is "1"b. Thus, if a user-written ready procedure honors the ready switch, its printing of the ready message can be controlled by the standard ready, ready_on, and ready_off commands.

Entry: cu_$reset_cl_intermediary

This entry point resets the procedure invoked by calls to cu_$cl to the standard system supplied procedure.

USAGE

```
declare cu_$reset_cl_intermediary entry ();

call cu_$reset_cl_intermediary ();
```

Entry: cu_$reset_command_processor

This entry point resets the procedure invoked by calls to cu_$cp to the standard system supplied procedure.

USAGE

```
declare cu_$reset_command_processor entry ();

call cu_$reset_command_processor ();
```

Entry: cu_$reset_evaluate_active_string

This entry point resets the procedure invoked by calls to cu_$evaluate_active_string to the standard system supplied procedure.

USAGE

```
declare cu_$reset_evaluate_active_string entry ();

call cu_$reset_evaluate_active_string ();
```

Entry: cu_$reset_ready_procedure

This entry point resets the procedure invoked by calls to cu_$ready_proc to the standard system supplied procedure.

USAGE

```
declare cu_$reset_ready_procedure entry ();

call cu_$reset_ready_procedure ();
```

Entry: cu_$set_cl_intermediary

The Multics system provides a set of procedures to handle any error conditions that can be signalled within a process (see the description of the signal_ subroutine). The standard error handlers attempt to print an understandable diagnostic and call a procedure to reenter command level. However, in order to allow use of the standard error handling procedures in a closed subsystem environment, the error handlers do not call the standard error handlers directly but call the cu_$cl entry point. This entry point passes control to the procedure entry point currently defined by the last call to cu_$set_cl_intermediary. If cu_$set_cl_intermediary has never been called in the process, control is passed to the standard error handlers on a call to cu_$cl.

USAGE

```
declare cu_$set_cl_intermediary entry (entry);

call cu_$set_cl_intermediary (proc_entry);
```

ARGUMENTS

proc_entry
: is the procedure entry to be called by the standard error handlers after printing a diagnostic message. (Input)

Entry: cu_$set_command_processor

Some standard Multics commands permit the user to escape from them to execute other commands. In this case, the escapable command passes the line to be executed to the command processor. To allow use of these escapable standard commands in a closed subsystem environment, instead of calling the command processor directly, the cu_$cp entry point is called. The latter passes control to the procedure entry point defined as the current command processor. The cu_$set_command_processor entry point allows a subsystem developer to replace the standard command processor with a different procedure. This mechanism can be used to ensure that the subsystem remains in full control and still allow subsystem users the use of many standard commands.

USAGE

```
declare cu_$set_command_processor entry (entry);

call cu_$set_command_processor (proc_entry);
```

ARGUMENTS

proc_entry
: is the procedure entry point to which control is passed upon receiving a call to cu_$cp. (Input)

Entry: cu_$set_evaluate_active_string

Some standard Multics commands (e.g., compose and exec_com) permit the user to evaluate active strings which are a sequence of one or more active function invocations with their arguments. To allow the use of these commands in a closed subsystem, instead of calling the command processor directly to evaluate the active string, the cu_$evaluate_active_string entry is called. The latter passes control to the procedure entry point defined as the current active string evaluator. The cu_$set_evaluate_active_string entry point allows a subsystem developer to replace the standard active string evaluator with a different procedure. This mechanism can be used to insure that the subsystem remains in full control and still allow subsystem users the use of many standard commands.

USAGE

```
declare cu_$set_evaluate_active_string entry (entry);

call cu_$set_evaluate_active_string (active_string_procedure);
```

ARGUMENTS

active_string_procedure
: is the procedure entry point to which control is passed upon receiving a call to cu_$evaluate_active_string. (Input)

Entry: cu_$set_ready_mode

This entry point allows the user to change the value of the static ready mode.

USAGE

```
declare cu_$set_ready_mode entry (1 aligned, 2 bit(1) unaligned,
      2 bit(35) unaligned);

dcl 1 mode       aligned,
       2 ready_sw bit(1) unaligned,
       2 mbz      bit(35) unaligned;

call cu_$set_ready_mode (mode);
```

ARGUMENTS

mode.ready_sw
    is the new value of the static ready switch. (Input)
    "1"b print ready message
    "0"b do not print ready message

mode.mbz
    is reserved for future use and must be "0"b. (Input)

Entry: cu_$set_ready_procedure

This entry point allows the user to change the ready procedure invoked by cu_$ready_proc.

USAGE

```
declare cu_$set_ready_procedure entry (entry);

call cu_$set_ready_procedure (ready_entry);
```

ARGUMENTS

ready_entry
    is the procedure entry point that is to become the new ready procedure of the process. (Input)

Entry: cu_$shrink_stack_frame

This entry point allows its caller to deallocate temporary storage by reducing the caller's current stack frame. Such storage must have been allocated via a call to cu_$grow_stack_frame.

USAGE

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

call cu_$shrink_stack_frame (ptr, code);
```

ARGUMENTS

ptr
: is a pointer to the first word of the storage to be deallocated. (Input) It must point to a mod 16 word boundary. The stack frame from the word indicated by ptr to the end of the frame is deallocated.

code
: is a standard status code. (Output)

Entry: cu_$stack_frame_ptr

The cu_$stack_frame_ptr entry point returns a pointer to the stack frame of its caller. The stackframeptr builtin function should be used to get this information in PL/I programs, since it is more efficient.

USAGE

```
declare cu_$stack_frame_ptr entry (ptr);

call cu_$stack_frame_ptr (stack_ptr);
```

ARGUMENTS

stack_ptr
: is a pointer to the stack frame of its caller. (Output)

Entry: cu_$stack_frame_size

The cu_$stack_frame_size entry point returns the size (in words) of the stack frame of its caller.

USAGE

```
declare cu_$stack_frame_size entry (fixed bin);

call cu_$stack_frame_size (size);
```

ARGUMENTS

size
: is the size (in words) of the caller's stack frame. (Output)

Name: cv_bin_

The cv_bin_ subroutine converts the binary representation of an integer (of any base) to a 12-character ASCII string.

USAGE

```
declare cv_bin_ entry (fixed bin, char(12) aligned, fixed bin);

call cv_bin_ (n, string, base);
```

ARGUMENTS

n
: is the binary integer to be converted. (Input)

string
: is the ASCII equivalent of n. (Output)

base
: is the base to use in converting the binary integer (e.g., base is 10 for decimal integers). (Input)

Entry: cv_bin_$dec

This entry point converts the binary representation of an integer of base 10 to a 12-character ASCII string.

USAGE

```
declare cv_bin_$dec entry (fixed bin, char(12) aligned);

call cv_bin_$dec (n, string);
```

ARGUMENTS

n
is the binary integer to be converted. (Input)

string
is the ASCII equivalent of n. (Output)

NOTES

This function can be performed more efficiently in PL/I by:

```
string = ltrim (char (n));
```

Entry: cv_bin_$oct

This entry point converts the binary representation of an octal integer to a 12-character ASCII string.

USAGE

```
declare cv_bin_$oct entry (fixed bin, char(12) aligned);

call cv_bin_$oct (n, string);
```

ARGUMENTS

n
is the binary integer to be converted. (Input)

string
is the ASCII equivalent of n. (Output)

NOTES

If the character-string representation of the number exceeds 12 characters, then only the low-order 12 digits are returned.

Name: cv_dec_

The cv_dec_ function accepts an ASCII representation of a decimal integer and returns the fixed binary(35) representation of that number. (See also cv_dec_check_.)

USAGE

```
declare cv_dec_ entry (char(*)) returns (fixed bin(35));

a = cv_dec_ (string);
```

ARGUMENTS

string
    is the string to be converted. (Input)

a
    is the result of the conversion. (Output)

NOTES

If string is not a proper character representation of a decimal number, a will contain the converted value of the string up to, but not including, the incorrect character within the string.

This function can be performed more efficiently in PL/I by:

```
a = convert (a, string);
```

Name: cv_dec_check_

This function differs from cv_dec_ only in that a code is returned indicating the possibility of a conversion error. (See also cv_dec_.)

USAGE

```
declare cv_dec_check_ entry (char(*), fixed bin(35))
      returns (fixed bin(35));

a = cv_dec_check_ (string, code);
```

ARGUMENTS

string
: is the string to be converted. (Input)

code
: is a code that equals 0 if no error has occurred; otherwise, it is the index of the character of the input string that terminated the conversion. See "Notes" below. (Output)

a
: is the result of the conversion. (Output)

NOTES

Code is not a standard status code and, therefore, cannot be passed to com_err_ and other subroutines that accept only standard status codes.

This function can be performed more efficiently in PL/I by:

```
on conversion,size goto badnumber;
a = convert (a, string);
revert conversion,size;
    .
    .
    .
badnumber:
call com_err_ (error_table_$bad_conversion, proc, string);
return;
```

Name: cv_dir_mode_

The cv_dir_mode_ subroutine converts a character string containing access modes for directories into a bit string used by the ACL entries.

USAGE

```
declare cv_dir_mode_ entry (char(*), bit(*), fixed bin(35));

call cv_dir_mode_ (char_modes, bit_modes, code);
```

ARGUMENTS

char_modes
    are the character string access modes. (Input)

bit_modes
    are the bit string access modes. (Output)

code
    is a standard status code. (Output) It can be:
    error_table_$bad_acl_mode
        if char_modes contains an invalid directory access mode character

NOTES

If char_modes is "null" or "n", bit_modes is set to "0"b. The mode characters in char_modes can occur in any order. Spaces are ignored. The following table indicates which bit in bit_modes is turned on when the access mode character is found.

Access Mode	Bit in bit_modes
s	1
m	2
a	3

These values are declared in access_mode_values.incl.pl1.

Name: cv_entry_

The cv_entry_ function converts a virtual entry to an entry value. A virtual entry is a character-string representation of an entry value. The types of virtual entries accepted are described under "Virtual Entries" below.

USAGE

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

entry_value = cv_entry_ (ventry, referencing_ptr, code);
```

ARGUMENTS

ventry
: is the virtual entry to be converted. See "Virtual Entries" below for more information. (Input)

referencing_ptr
: is a pointer to a segment in the referencing directory. This directory is searched according to the referencing_dir search rule to find the entry. A null pointer may be given if the referencing_dir search rule is not to be used. (Input)

code
: is a standard status code. (Output)

entry_value
: is the entry value that results from the conversion. (Output)

VIRTUAL ENTRIES

The cv_entry_ function converts virtual entries that contain one or two components — a segment identifier and an optional offset into the segment. Altogether, eleven forms are accepted. They are shown in the table below.

In the table that follows, W is an octal word offset from the beginning of the segment. It may have a value from 0 to 777777 inclusive.

path|W
entry at octal word W of segment identified by absolute or relative pathname path.

path|
same as path|0.

path|entry_pt
entry at word identified by entry point entry_pt in the object file (segment or MSF) identified by path.

dir>entry$entry_pt
entry at word identified by entry point entry_pt in the the object file identified by pathname dir>entry.

<dir>entry$entry_pt
entry at word identified by entry point entry_pt in the object file identified by pathname <dir>entry.

<entry$entry_pt
entry at word identified by entry point entry_pt in the object file identified by pathname <entry.

path
same as path|[entry path].

ref_name$entry_pt
entry at word identified by entry point entry_pt in the file found via search rules whose reference name is ref_name.

ref_name$W
entry at octal word W of segment found via search rules whose reference name is ref_name.

ref_name$
same as ref_name$0.

ref_name
same as ref_name$ref_name but like "path" if it contains ">" or "<" characters.

NOTES

Use of a pathname in a virtual entry causes the referenced segment to be initiated with a reference name equal to its final entryname. Name duplication errors occurring during the initiation are resolved by terminating the previously known name.

The referencing_ptr is used in a call to the hcs_$make_entry entry point.

The cv_entry_ function returns an entry value that may be used in a call to cu_$generate_call. If an entry pointer is required, rather than an entry variable, use the cv_ptr_ subroutine.

A virtual entry not containing the $ or | characters is interpreted as a pathname if it contains a > or < character, otherwise, it is a reference name.

Name: cv_error_ subroutine

Entry: cv_error_ subroutine$name

This entry point converts an error name (e.g., error_table_$badarg) to an error code.

USAGE

```
dcl cv_error_$name entry (char(*), fixed bin (35), fixed bin (35));

call cv_error_$name (error_name, converted_code, code);
```

ARGUMENTS

error_name
    is the name of the error to be converted. (Input)

converted_code
    is the result of converting error_name. (Output)

code
    is a standard system status code. If code is non-zero, converted_code is undefined. (Output)

Name: cv_float_

The cv_float_ subroutine converts an ASCII representation of a floating point number and returns a single precision floating point representation. If an illegal character is encountered, its index in the string is returned and the number is set to 0.0e0. (See also cv_float_double_.)

USAGE

```
declare cv_float_ entry (char(*), fixed bin(35)) returns (float bin);

a = cv_float_ (string, code);
```

ARGUMENTS

string
: is the string to be converted. (Input)

code
: is the index in string of the first illegal character, if found; otherwise it is zero.

This page intentionally left blank.

a
    is the result of the conversion. (Output)

NOTES

Code is not a standard status code. Therefore, it can not be passed to com_err_ and other subroutines that accept only standard status codes.

This function can be performed more efficiently in PL/I by:

```
on conversion,size goto badnumber;
a = convert (a, string);
revert conversion,size;
    .
    .
    .
badnumber:
call com_err_ (error_table_$bad_conversion, proc, string);
return;
```

Name: cv_float_double_

The cv_float_double_ subroutine converts an ASCII representation of a floating point number and returns a double precision floating point representation. If an illegal character is encountered, its index in the string is returned and the number is set to 0.0e0. (See also cv_float_.)

USAGE

```
declare cv_float_double_ entry (char (*), fixed bin(35)) returns
      (float bin(63));

a = cv_float_double_ (string, code);
```

ARGUMENTS

string
    is the string to be converted. (Input)

code
    is the index in string of the first illegal character, if found; otherwise it is zero.

a
    is the result of the conversion. (Output)

NOTES

Code is not a standard status code. Therefore, it can not be passed to com_err_ and other subroutines that accept only standard status codes.

This function can be performed more efficiently in PL/I by:

```
on conversion,size goto badnumber;
a = convert (a, string);
revert conversion,size;
   .
   .
   .
badnumber:
call com_err_ (error_table_$bad_conversion, proc, string);
return;
```

Name: cv_fstime_

Given a file system time value, this function returns a Multics clock value.

USAGE

```
declare cv_fstime_ entry (bit(36) aligned)
      returns (fixed bin(71));

clock_value = cv_fstime_ (fstime);
```

ARGUMENTS

fstime
: the file system time to be converted. (Input) Such values are returned by such entry points as hcs_$status_, hcs_$status_long, hcs_$star_list, hcs_$star_dir_list.

clock_value
: the Multics clock value which corresponds to fstime. (Output)

Name: cv_hex_

The cv_hex_ function takes an ASCII representation of a hexadecimal integer and returns the fixed binary(35) representation of that number. The ASCII representation may contain either uppercase or lowercase characters. (See also cv_hex_check_.)

USAGE

```
declare cv_hex_ entry (char(*)) returns (fixed bin(35));

a = cv_hex_ (string);
```

ARGUMENTS

string
: is the string to be converted. It must be nonvarying. (Input)

a
: is the result of the conversion. (Output)

Name: cv_hex_check_

This function differs from the cv_hex_ function only in that a code is returned indicating the possibility of a conversion error. (See also cv_hex_.)

USAGE

```
declare cv_hex_check_ entry (char(*), fixed bin(35)),
     returns (fixed bin(35));

a = cv_hex_check_ (string, code);
```

ARGUMENTS

string
: is the string to be converted. It must be nonvarying. (Input)

code
: is a code that equals 0 if no error occurred; otherwise, it is the index of the character that terminated the conversion. See "Note" below. (Output)

a
: is the result of the conversion. (Output)

NOTES

Code is not a standard status code and, therefore, cannot be passed to com_err_ and other subroutines that accept only standard status codes.

Name: cv_mode_

The cv_mode_ subroutine converts a character string containing access modes for segments into a bit string used by the ACL entries.

USAGE

```
declare cv_mode_ entry (char (*), bit (*), fixed bin(35));

call cv_mode_ (char_modes, bit_modes, code);
```

ARGUMENTS

char_modes
  are the character string access modes. (Input)

bit_modes
  are the bit string access modes. (Output)

code
  is a standard status code. (Output) It can be:
  error_table_$bad_acl_mode
    if char_mode contains an invalid segment access mode character.

NOTES

If char_modes is "null" or "n", bit_modes is set to "0"b. The mode characters in char_modes may occur in any order. Spaces are ignored. The following table indicates what bit in bit_modes is turned on when the access mode character is found.

Access Mode	Bit in bit_modes
r	1
e	2
w	3

These values are declared in access_mode_values.incl.pl1.

Name: cv_oct_

The cv_oct_ function takes an ASCII representation of an octal integer and returns the fixed binary(35) representation of that number. (See also cv_oct_check_.)

USAGE

```
declare cv_oct_ entry (char(*)) returns (fixed bin(35));

a = cv_oct_ (string);
```

ARGUMENTS

string
: is the string to be converted. (Input)

a
: is the result of the conversion. (Output)

Name: cv_oct_check_

This function differs from the cv_oct_ function only in that a code is returned indicating the possibility of a conversion error. (See also cv_oct_.)

USAGE

```
declare cv_oct_check_ entry (char(*), fixed bin(35)) returns
      (fixed bin(35));

a = cv_oct_check_ (string, code);
```

ARGUMENTS

string
: is the string to be converted. It must be nonvarying. (Input)

code
: is a code that equals 0 if no error occurred; otherwise it is the index of the character that terminated the conversion. See "Notes" below. (Output)

a
: is the result of the conversion. (Output)

NOTES

Code is not a standard status code and, therefore, cannot be passed to com_err_ and other subroutines that accept only standard status codes.

Name: cv_ptr_

The cv_ptr_ function converts a virtual pointer to a pointer value. A virtual pointer is a character-string representation of a pointer value. The types of virtual pointers accepted are described under "Virtual Pointers" below.

USAGE

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

ptr_value = cv_ptr_ (vptr, code);
```

ARGUMENTS

vptr
: is the virtual pointer to be converted. See "Virtual Pointers" below for more information. (Input)

code
: is a standard status code. (Output)

ptr_value
: is the pointer that results from the conversion. (Output)

Entry: cv_ptr_$terminate

This entry point is called to terminate the segment that has been initiated by a previous call to cv_ptr_.

USAGE

```
declare cv_ptr_$terminate (ptr);

call cv_ptr_$terminate (ptr_value);
```

ARGUMENTS

ptr_value
: is the pointer returned by the previous call to cv_ptr_. (Input)

NOTES

Pointers returned by the cv_ptr_ function cannot be used as entry pointers. The cv_ptr_ function constructs the returned pointer to a segment in a way that avoids copying of the segment's linkage and internal static data into the combined linkage area. The cv_entry_ function is used to convert virtual entries to an entry value.

The segment pointed to by the returned ptr_value is initiated with a null reference name. The cv_ptr_$terminate entry point should be called to terminate this null reference name.

VIRTUAL POINTERS

The cv_ptr_ function converts virtual pointers that contain one or two components — a segment identifier and an optional offset into the segment. Altogether, seventeen forms are accepted. They are shown below.

In the list that follows, W is an octal word offset from the beginning of the segment; it can have a value from 0 to 777777 inclusive. B is a decimal bit offset within the word; it can have a value from 0 to 35 inclusive. The possible forms are:

path|W(B)
: points to octal word W, decimal bit B, of the segment or multisegment file (MSF) identified by absolute or relative pathname path. If the path you give identifies a MSF, the offset given is in component 0 of the MSF.

path|W
: same as path|W(0).

path|
: same as path|0(0).

path
: same as path|0(0).

path|entry_pt
: points to the word identified by entry point entry_pt in the object file (segment or MSF) identified by path.

dir>entry$entry_pt
: points to the word identified by entry point entry_pt in the object file identified by pathname dir>entry.

<dir>entry$entry_pt
: points to the word identified by entry point entry_pt in the object file identified by pathname <dir>entry.

<entry$entry_pt
: points to the word identified by entry point entry_pt in the object file identified by pathname <entry.

ref_name$entry_pt
: points to the word identified by entry point entry_pt in the file whose reference name is ref_name.

ref_name$W(B)
points to the octal word W, decimal bit B, of the segment or MSF whose reference name is ref_name. If ref_name is a reference name on an MSF (i.e., on component 0 of the MSF), the word and bit offsets are applied within component 0.

ref_name$W
same as ref_name$W(0).

ref_name$
same as ref_name$0(0).

segno|W(B)
points to the octal word W, decimal bit B, of the segment whose octal segment number is segno.

segno|W
same as segno|W(0).

segno|
same as segno|0(0).

segno
same as segno|0(0).

segno|entry_pt
points to the word identified by entry point entry_pt in the segment whose octal segment number is segno. If segno identifies component 0 of an object MSF, the pointer returned may not point within the segment identified, since the target of a definition in component 0 of an object MSF will be in another component of the object MSF.

A null pointer is represented by the virtual pointer 77777|1, by -1|1, or by -1.

A virtual entry not containing the $ or | characters is interpreted as a pathname if it contains a > or < character, otherwise, it is a reference name.

Archive component pathnames are permitted.

Name: cv_rcp_attributes_

The cv_rcp_attributes_ subroutine contains several entry points that are useful in manipulating RCP resource attribute specifications and descriptions. RCP resource attribute descriptions are printable strings that describe the attributes of resources (devices and volumes).

See the Programmer's Reference Manual for a description of the Resource Control Facility.

RCP resource attribute specifications are encoded representations of attribute descriptions. They can be absolute, relative, or multiple. An absolute attribute specification represents a complete and consistent state of all the attributes of a resource. A relative attribute description represents a desired modification to the state of all the attributes of a resource, and must be applied to an absolute attribute specification to produce the desired change in that absolute specification. A multiple attribute specification does not represent a consistent state of all the attributes of a resource at any given time, but is useful for representing the union of all such consistent states, i.e., potential attributes.

Entry: cv_rcp_attributes_$from_string

This entry point accepts a printable RCP attribute description and produces an RCP attribute specification.

USAGE

```
declare cv_rcp_attributes_$from_string entry (char (*),
      (2) bit (72), char (*) varying, fixed bin (35));

call cv_rcp_attributes_$from_string (type, attributes, string, code);
```

This page intentionally left blank.

ARGUMENTS

type
: specifies the type of resource to which attributes apply. (Input)

attributes
: is an RCP attribute specification (see "Notes" below). (Output)

string
: is a printable RCP attribute description. (Input)

code
: is a standard status code. (Output)

Entry: cv_rcp_attributes_$from_string_rel

This entry point generates a relative attribute specification that can later be applied to attribute specifications of specific resources via the cv_rcp_attributes_$modify_rel entry point.

USAGE

```
declare cv_rcp_attributes_$from_string_rel entry (char (*),
      char (*) varying, bit (72) dimension (4), fixed bin (35));

call cv_rcp_attributes_$from_string_rel (type, string, rel_attributes,
      code);
```

ARGUMENTS

type
: specifies the type of resource to which string applies. (Input)

string
: is a printable RCP attribute description. (Input)

rel_attributes
: is the relative RCP attribute specification. (Output)

code
: is a standard status code. (Output)

Entry: cv_rcp_attributes_$modify

This entry point applies a printable RCP resource attribute description (representing a relative attribute specification) to a given resource specification and returns a new attribute specification. The resulting attribute specification consists of the original attribute specification modified by the attributes specified in the printable description.

USAGE

```
declare cv_rcp_attributes_$modify entry (char(*), bit(72) dimension(2),
      char(*) varying, bit(72) dimension(2), fixed bin(35));

call cv_rcp_attributes_$modify (type, attributes, string,
      new_attributes, code);
```

ARGUMENTS

type
: specifies the type of resource to which attributes and string apply. (Input)

attributes
: is an absolute RCP attribute specification. (Input)

string
: is a printable RCP attribute description that modifies attributes. (Input)

new_attributes
: is the new absolute RCP attribute specification. (Output)

code
: is a standard status code. (Output)

Entry: cv_rcp_attributes_$modify_rel

This entry point applies a relative attribute specification produced by the cv_rcp_attributes_$from_string_rel entry point to an absolute attribute specification of a specific resource.

USAGE

```
declare cv_rcp_attributes_$modify_rel entry (bit (72) dimension (2),
      bit (72) dimension (4), bit (72) dimension (2));

call cv_rcp_attributes_$modify_rel (attributes, rel_attributes,
      new_attributes);
```

ARGUMENTS

attributes
is an absolute attribute specification. (Input)

rel_attributes
is a relative attribute specification to be applied to attributes. (Input)

new_attributes
is the resulting absolute attribute specification. (Output)

NOTES

The caller must ensure that attributes and rel_attributes refer to the same resource type, i.e., that they are generated by previous calls to cv_rcp_attributes_ where the type arguments are identical.

Entry: cv_rcp_attributes_$protected_change

This entry point accepts an absolute attribute specification for a resource and a relative attribute specification that is to modify it. It returns a value expressing whether or not this modification affects protected attributes of the resource. No modification is actually attempted by this entry.

USAGE

```
declare cv_rcp_attributes_$protected_change entry (bit (72)
      dimension(2), bit (72) dimension(4)) returns (bit (1) aligned);

protected_change = cv_rcp_attributes_$protected_change (attributes,
      rel_attributes);
```

ARGUMENTS

attributes
is an RCP attribute specification. (Input)

rel_attributes
is a relative attribute specification to be applied to attributes. (Input)

protected_change
is "1"b if this operation modifies protected attributes of the resource; otherwise, it is "0"b. (Output)

Entry: cv_rcp_attributes_$reduce_implications

This entry point accepts an attribute specification for a volume and returns the necessary minimal attribute specification that a device must possess to be able to accept the volume.

USAGE

```
declare cv_rcp_attributes_$reduce_implications entry (char (*), bit (72)
      dimension(2), char (*), bit (72) dimension (4), fixed bin (35));

call cv_rcp_attributes_$reduce_implications (vol_type, vol_attributes,
      dev_type, dev_attributes, code);
```

ARGUMENTS

vol_type
: specifies the type of volume from which vol_attributes is obtained. (Input)

vol_attributes
: is an absolute attribute specification for the volume type specified. (Input)

dev_type
: is the resource type of the device that accepts the given volume type. (Output)

dev_attributes
: is a minimal relative attribute specification for a device capable of accepting a volume with the given attributes. (Output)

code
: is a standard status code. (Output)

Entry: cv_rcp_attributes_$test_valid

This entry point determines whether a given attribute specification is absolute, relative, multiple, or invalid.

USAGE

```
declare cv_rcp_attributes_$test_valid entry (char(*), bit 72
      dimension (2), fixed bin, fixed bin (35));

call cv_rcp_attributes_$test_valid (type, attributes, validity, code);
```

ARGUMENTS

type
: specifies the type of resource to which attributes apply. (Input)

attributes
: is an RCP attribute specification. (Input)

validity
: shows whether the attribute specification is absolute, relative, or multiple. (Output)
0 is an absolute attribute specification
1 is a relative attribute specification
2 is a multiple attribute specification

code
: is a standard status code. (Output)

Entry: cv_rcp_attributes_$to_string

This entry point takes an RCP resource attribute specification and produces a printable RCP attribute description.

USAGE

```
declare cv_rcp_attributes_$to_string entry (char (*), bit (72)
     dimension (2), char (*) varying, fixed bin (35));

call cv_rcp_attributes_$to_string (type, attributes, string, code);
```

ARGUMENTS

type
: specifies the type of resource from which attributes are obtained, e.g., disk_drive (see "Notes" below). (Input)

attributes
: is an RCP attribute specification (see "Notes" below). (Input)

string
: is a printable RCP attribute description. (Output)

code
: is a standard status code. (Output)

NOTES

A list of defined resource types can be obtained via the list_resource_types command.

Name: cv_userid_

The cv_userid_ subroutine converts a character string containing an abbreviated User_id into one containing all three components, i.e. Person_id.Project_id.tag.

USAGE

```
declare cv_userid_ entry (char(*)) returns (char(32));

user_id = cv_userid_ (string);
```

ARGUMENTS

string
is the abbreviated User_id. (Input)

user_id
is a User_id containing all three components. (Output)

NOTES

The Person_id, Project_id and tag components are truncated to 20, 9 and 1 characters, respectively. An asterisk ("*") is supplied for missing components.

EXAMPLES

```
Abbreviated User_id    Full User_id
-------------------    ------------
Smith.Project.a        Smith.Project.a
Smith.Project          Smith.Project.*
Smith                  Smith.*.*
.Project               *.Project.*
```

Name: date_time_

The date_time_ system is a utility which encodes, decodes, adjusts, or formats a Multics standard calendar clock value. The clock reading is assumed to be in microseconds relative to 1901-01-01 0:00 gmt. The ASCII times involved may be one of several languages and in a choice of time zones around the world.

Entry: date_time_$date_time_

The date_time_ subroutine converts a system clock value to ASCII representation. It will be in terms of the process default language and zone.

USAGE

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

call date_time_ (clock, str);
```

ARGUMENTS

clock
is the clock value to be formatted. (Input)

str
is the resultant character string. (Output)

NOTES

The format string which produces the resultant string is:

```
"^my/^dm/^yc  ^Hd^99v.9MH ^xxxxza^xxxxda"
```

which produces strings like this:

```
mm/dd/yy  HHMM.M zzzddd
07/14/83  1435.4 mst Thu
```

See date_time_$format for a description of time format strings.

The ASCII representation of time, which date_time_ attempts to return in string, is 24 characters long. If string is declared by the caller with a length of N and N is less than 24, then only the first N characters are returned. If N is greater than 24, then the result is returned padded on the right with spaces.

If clock is not a valid date, "01/01/01 0000.0 gmt Tue" is returned.

Entry: date_time_$format

This entry does a generalized formatting of a Multics standard calendar clock value. A format string is supplied which describes the layout and content of the desired result. The zone and/or language in which the result is to be displayed may be specified.

USAGE

```
declare date_time_$format entry (char (*), fixed bin(71), char (*),
      char (*)) returns (char (250) var);

result = date_time_$format (format, clock, zone, lang);
```

ARGUMENTS

format
: either a keyword, or an ioa-like control string describing the desired result in terms of literal characters and date/time selectors. (Input) See Notes on Time Format, below.

clock
: a clock value to be displayed. (Output)

zone
: the short name of the zone in which output time value is expressed. (Input) "system_zone" means use the system default zone. "" means use the per-process default zone. (Input)

lang
: the language in which month names, day names and time zones are expressed. (Input) "system_lang" means use the system default time language. "" means use per-process default time language.

result
: is the string which is the result of the conversion. (Output)

ERROR HANDLING

There are many errors which may occur while trying to format a string. These will seldom occur in a thoroughly debugged environment, so there is no error code used to report the (usual) success. Instead, the sub_err_ mechanism is used when an error occurs. The information in the sub_error_info structure is generally quite explicit as to the type of error and usually quite detailed in its explanation. Within a sub_error_ handler it is quite easy to display this data. First, sub_error_info.name will contain "date_time_$format". Then a nice-looking message will be produced by:

```
call com_err_ (sub_error_info.status_code, "my_name", "^a",
   sub_error_info.info_string);
```

A detailed message which could result from this is:

```
my_name: The picture contains a syntax error.
Format is: "^yc-^98my-^99dm"
error at:        ^
```

This is the set of values which may be present in the status_code field:

error_table_$badcall
    the environment was not set up properly before calling this procedure.

error_table_$bad_conversion
    a conversion condition occurred while trying to convert a value.

error_table_$dt_bad_format_selector
    unrecognized selector in format string.

error_table_$dt_date_too_big
    the date given is after 9999-12-31_23:59:59.999999_GMT.

error_table_$dt_date_too_small
    the date given is before 0001-01-01_00:00:00.000000_GMT.

error_table_$dt_no_format_selector
    the format string contains no selectors and is not a known keyword.

error_table_$dt_unknown_time_language
    the language specified by the caller was not found in time_info_.

error_table_$dt_year_too_big
    the clock value given, when converted to the specified time zone, is after the year 9999.

error_table_$dt_year_too_small
    the clock value given, when converted to the specified time zone, is before the year 0001.

error_table_$picture_bad
    the picture supplied is in error.

error_table_$picture_scale
    the picture scale factor not in the range -128:+127.

error_table_$picture_too_big
the normalized picture exceeds 64 characters.

error_table_$size_error
the size condition occurred during processing.

error_table_$unimplemented_version
a structure is not a version this procedure can handle.

error_table_$unknown_zone
the time zone specified by the caller was not found in time_info_.

NOTES ON TIME FORMAT

By means of convert_date_to_binary_, an input time string is converted to internal form. This is the usual form for storing dates in data bases. To convert one of these into a readable form, a programmer may call upon date_time_ to get a 24-character form like this:

```
03/14/79  0000.0 cet Fri
```

But when other formats are needed, date_time_$format is available. It takes a clock value and a control string describing the format wanted and returns a string ready for printing.

Most date/time outputs from the system software are usable as date/time inputs to system software. But, the time format mechanism is highly flexible and may easily generate formats which are not recognizable. Worse yet are the strings which are apparently recognized but which are ambiguous. A point in case is the commonly used string "7/1/82". In the United States, this means the 7th month, first day; but many places in Europe, this would be taken to mean the 7th day of the first month. Multics follows the United States interpretation.

TIME FORMAT

The control string to date_time_$format is either a keyword or a character string consisting of text and/or selectors. The selectors are always identified by a leading circumflex character (^). There are 2 types of selectors: ^<keyword>, which allows a keyword to be imbedded within a format; and the general form ^XX. XX is a 2 letter code which specifies what information is wanted. An optional PL/I picture specification may be placed between the ^ and XX if the default form is not adequate. If the control string does not contain any circumflex characters, it must then be one of the known set of keywords. Each keyword identifies a control string for a predetermined format named by that keyword.

LIST OF FORMAT KEYWORDS

all
```
^9999yc-^my-^dm__^Hd:^MH:^99.(6)9UM^zd_^za_^da ^fi
^(6)9fw ^ma dy^dy dc^dc Uc^Uc
```

calendar_clock
```
^9999yc-^my-^dm__^Hd:^MH:^99.(6)9UM_^za_^da
```

clock
```
^9999yc-^my-^dm ^Hd:^MH:^99.(6)9UM ^za ^da
```

date
    the process default value for date

date_time
    the process default value for date and time

iso_date
```
^9999yc-^my-^dm
```

iso_date_time
```
^9999yc-^my-^dm ^Hd:^MH:^SM ^za
```

iso_long_date
```
^9999yc-^my-^dm ^da
```

iso_long_date_time
```
^9999yc-^my-^dm ^Hd:^MH:^99.(6)9UM ^za
```

iso_long_time
```
^Hd:^MH:^99.(6)9UM
```

iso_time
```
^Hd:^MH:^SM
```

multics_date
```
^my/^dm/^yc
```

multics_date_time
```
^my/^dm/^yc ^Hd^99v.9MH ^xxxxza^xxxda
```

multics_time
```
^Hd:^MH
```

request_id
```
^yc^my^dm^Hd^MH^99.(6)9UM
```

system_date_time
    the system default value for date and time

system_date
    the system default value for date

system_time
    the system default value for time

time
    the process default value for time

A site may change the "system" keywords. To list the "system" keywords for your site, type:

```
print_time_defaults -system
```

For an application which depends upon the historic date/time formats, the 3 builtin "multics" keywords are available.

Processing of a control string proceeds 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 copied to the output string. The selector is then interpreted and executed. This causes a datum from the input clock value to be edited into the output string. Processing continues in this way until the control string is exhausted.

Dates and times placed in the output string may be expressed in units of years, months, weeks, days, hours, minutes, seconds and microseconds. The total calendar value can be expressed as a single unit. For example, the calendar value representing 79-09-08 9:42A GMT could be expressed as 1979 years, as 722702 days, or as 722702.112499 days. This is the set of "total" selectors:

^yc
    total number of Years in the Calendar value.

^mc
    total number of Months in the Calendar value.

^dc
    total number of Days in the Calendar value.

^Hc
    total number of Hours in the Calendar value.

^Mc
    total number of Minutes in the Calendar value.

^Sc
total number of Seconds in the Calendar value.

^Uc
total number of Microseconds in the Calendar value.

Dates and times may also be expressed as the number of units remaining after a larger unit has been removed from the calendar value. For example, 09/08/79 09:42 includes units for 9th month of the year, 8th day of the month, 9th hour of the day, and 42nd minute of the hour. The following are the most common:

^my
Month in the Year

^dm
Day of the Month.

^dw
Day of the Week.

^Hd
Hour of the Day (24-hour format).

^Hh
Hour in Half-day (12-hour format).

^MH
Minute of the Hour.

^SM
Second of the Minute.

^US
Microsecond of the Second

There are several items of date/time data which are nonnumeric, such as day of week, day of month, and time zone used for conversion.

^mn
Month Name

^ma
Month name, Abbreviated

^dn
Day Name

^da
Day name, Abbreviated

^zn
time Zone Name

^za
time Zone name, Abbreviated

^zd
Zone Differential (char(5))

^mi
Meridiem Indicator ("A" or "P")

^fi
Fiscal Indicator ("FW" in english)

The selectors of numeric data are, in general, made up of 2 letters taken from this sequence: c y m w d H M S U

The letters stand for calendar, year, month, week, day, Hour, Minute, Second, and microsecond. All 81 combinations are not, however, valid. The form can generally be read as "unit of unit", i.e. "seconds of week". The first unit is always smaller than the second one. In trying to keep the specifiers reasonably mnemonic (in English) there is a problem. Both month minute and microsecond begin with an "m". To that end, all date values are used as lower case letters while all time values are in upper case. Microsecond is expressed as MU, which resembles the Greek letter M, the scientific notation for the word micro.

It proves difficult to try to handle all the forms needed in a general manner. "Hd" is Hour in Day and is thus 24-hour time. This is not always what is wanted. "Hh" is chosen as Hour in Half-day to get the 12-hour form of time. To go along with this there is "mi" for Meridiem Indicator. This gives "A" or "P" to make up AM or PM. This does not give "AM" or "PM" because ANSI and ISO standards specify that time be given as "3P", not "3PM". The users who want the M will just put the literal in, i.e. "^miM".

One other way of looking at a calendar value is in terms of fiscal week. This is selected with the "^fw" code. Its value is 4 digits of year followed by 2 digits of week number of the year, i.e. "yyyyww". The default format for the ^fw code has been chosen to give a value of "yww"

This table shows the complete set of selectors. The row specifies what unit is wanted, the column specifies within what other unit, i.e. ^Sy is "Seconds of Year".

```
                            DATE/TIME SELECTORS
        | of   | of   | of   | of   | of   | of   | of   | of   |
        |calen-| year |month | week | day  | hour |minute|second|
 -------| dar  |      |      |      |      |      |      |      |
 micro- +------+------+------+------+------+------+------+------+
 second | ^Uc  | ^Uy  | ^Um  | ^Uw  | ^Ud  | ^UH  | ^UM  | ^US  |
        +------+------+------+------+------+------+------+------+
 second | ^Sc  | ^Sy  | ^Sm  | ^Sw  | ^Sd  | ^SH  | ^SM  |
        +------+------+------+------+------+------+------+
 minute | ^Mc  | ^My  | ^Mm  | ^Mw  | ^Md  | ^MH  |
        +------+------+------+------+------+------+
   hour | ^Hc  | ^Hy  | ^Hm  | ^Hw  | ^Hd  |
        +------+------+------+------+------+
    day | ^dc  | ^dy  | ^dm  | ^dw  |        month  day    zone
        +------+------+------+------+      +------+------+------+
  month |      | ^my  |             name   | ^mn  | ^dn  | ^zn  |
        +------+------+                    +------+------+------+
   year | ^yc  |                    abbrev | ^ma  | ^da  | ^za  |
        +------+                           +------+------+------+
        | ^Hh  | <-hour of half-day    differential  | ^zd  |
        +------+    (12 hour form)                   +------+
        | ^mi  | <-meridiem indicator ("A" or "P")
        +------+
        | ^fw  | <-fiscal week (form: yyyyww)
        +------+
        | ^fi  | <-fiscal indicator ("FW" in english)
        +------+
```

The formatting of date and time values can be controlled by an optional PL/I picture specification included in the selector. For example, a code of "^OO99yc" formats the total years in the calendar value into a 2-digit year of the 20th century. ^9999yc provides a full, 4-digit year. The following is a brief description of the most frequently-used picture characters. For a more complete discussion of PL/I pictures, refer to the *Multics PL/I Language Specification Manual*, Order No. AG94, and the *Multics PL/I Reference Manual*, Order No. AM83.

9

represents a mandatory decimal digit in the displayed value.

z

represents a decimal digit in the displayed value. Nonsignificant zeros on the left are replaced by a space when they occupy a "z" digit position.

.

produces a period in the displayed value. This has no relation to the location of the decimal point in the value actually being displayed. If zero suppression is in effect, this is replaced with a space.

,

produces a comma in the displayed value. It has all the characteristics of the period.

v

locates the value's decimal point in the result. This determines how the value digits are oriented with respect to the picture specification. If no "v" is given, it is assumed to appear after the rightmost picture character.

The picture characters above are sufficient for displaying most numeric values. For example, the control string ^99Hd^99.v9MH represents the time in hours, minutes and tenth of minutes. The control string ^zz9.999vUS represents the number of milliseconds of the second, using the decimal point and "v" to scale the microsecond unit. Scaling can also be performed by a picture scale factor.

f(N)

scales the value by multiplying or dividing by a power of 10, thus shifting the location of the decimal point in the value. For example, f(2) shifts the decimal 2 places left, effectively dividing the value by 100. f(-3) shifts 3 places right, effectively multiplying by 1000.

Using a picture scale factor, the milliseconds in excess of a second can be displayed to the nearest tenth using the control string

```
^zz9.9f(3)US
```

A value of 48634 microseconds would be displayed as " 48.6" milliseconds.

There are 2 extensions to numeric picture handling which can be used in time format selectors.

Z

represents a decimal digit in the displayed value. Nonsignificant zeros to the left of the decimal point are omitted from the displayed value when they occupy a "Z" digit position. Nonsignificant zeros to the right of the decimal point are omitted from the displayed value when they occupy a "Z" digit position.

"Z" characters must appear as the leftmost or rightmost digit positions in the picture specification, since these are the positions which nonsignificant zeros can occupy. "Z" performs a selective ltrim or rtrim (of zero) operation on the displayed value. For example, our millisecond specification given above could be specified as ^ZZ9.9ZZUS without using a picture scale factor. With this specification, 48630 microseconds would be displayed as "48.63" milliseconds (without the leading space or trailing zero).

O

represents a decimal digit in the displayed value that should be omitted. Specifying ^99yc for a year like 1941 will result in a size condition, since it takes 4 digits to handle that number. To get the year in century, you may use ^OO99yc. This gives 4 digits into which the value is placed and then the first 2 digits are discarded. Note that a picture like OOz9 with a value of 1502 will give "02" because the zero-suppression applies to the 1502 and then the first 2 digits are dropped.

Character date/time values such as day of the week, month name, and time zone can be formatted using a character picture specification with the "x" picture character.

x

represents a position which may contain any character. Since national characters occur in some of the time names, the use of the "a" character should be avoided. Values are left-justified in the picture specification, with truncation of the rightmost characters if the value is longer than the picture, or padding with spaces on the right if the value is shorter than the picture.

For example, ^xxxxxxxxxdn displays Wednesday as "Wednesday" and Monday as "Monday ". A picture repetition factor can be used to shorten the control string to "^(9)xdw". With ^(5)xmn, January is displayed as "Janua" and May is displayed as "May ". Remember that in some languages, the abbreviation of a time name is not the first three letters of it.

The selector picture specification allows an extension of the "x" picture specification.

X

represents an optional character position in the displayed value. The character position is omitted if there is no corresponding character in the value being displayed.

"X" characters must appear as the rightmost character positions in the picture specification, since this is the position in which nonsignificant spaces can occupy. "X" performs a selective rtrim operation on the displayed value.

The code ^(9)Xdw displays Wednesday and Monday both without trailing spaces.

This table shows the default picture specifications for all selectors. The row specifies what unit is wanted, the column specifies within what other unit, i.e. ^Sy is "Seconds of Year".

```
                          DEFAULT PICTURE VALUES
        | of    | of    | of    | of    | of    | of    | of    | of    |
        |calen- |year   |month  |week   | day   |hour   |minute |second |
________| dar   |       |       |       |       |       |       |       |
micro-  +-------+-------+-------+-------+-------+-------+-------+-------+
second  |(18)Z9 |(14)Z9 |(13)Z9 |(12)Z9 |(11)Z9 |(10)Z9 |(8)Z9  |(5)Z9  |
        +-------+-------+-------+-------+-------+-------+-------+-------+
second  |(12)Z9 |(12)Z9 |(8)Z9  |(6)Z9  |(5)Z9  |(4)Z9  |  99   |
        +-------+-------+-------+-------+-------+-------+-------+
minute  |(10)Z9 |(6)Z9  |(5)Z9  |(5)Z9  |(4)Z9  |  99   |
        +-------+-------+-------+-------+-------+-------+
  hour  |(8)Z9  |(4)Z9  |(3)Z9  |(3)Z9  |  99   |
        +-------+-------+-------+-------+-------+
   day  |(7)Z9  | 999   |  99   |   9   |        month   day     zone
        +-------+-------+-------+-------+       +-------+-------+-------+
 month  |       |  99   |               name    |(32)X  |(32)X  |(64)X  |
        +-------+-------+                       +-------+-------+-------+
  year  | 0099  |                       abbrev  | (8)X  | (8)X  | (8)X  |
        +-------+                               +-------+-------+-------+
        |  99   | <-hour of half-day      differential          |s9999  |
        +-------+    (12 hour form)                             +-------+
        |  x    | <-meridiem indicator
        +-------+
        |000999 | <-fiscal week (form: yyyyww)
        +-------+
        |  xx   | <-fiscal indicator
        +-------+
```

Examples: The following table shows how date and times strings are displayed by a variety of control strings.

```
CONTROL STRING
   "DISPLAYED VALUE"
^mn ^Z9dm, ^9999yc
   "September 8, 1979"
^mn ^z9dm, ^9999yc
   "September  8, 1979"
^dm ^ma ^9999yc ^zn
   "08 Sep 1979 Mountain Standard Time"
^my/^dm/^yc  ^Hd^99v.9MH ^za ^da
   "09/08/79  0242.4 mst Sat"
^Hd:^MH:^SM^zd
   "02:42:25-0700"
^9999yc-^my-^dm__^Hd:^MH:^99.(6)9UM_^za_^da
   "1979-09-08__02:42:25.048634_mst_Sat"
<-^<multics_time>xyz^<multics_date>->
   "<-02:42xyz09/08/79->"
```

Entry: date_time_$format_max_length

This entry returns the length of the longest result which a format can generate. The zone and/or language in which to work may be specified.

USAGE

```
declare date time_$format_max_length entry (char (*), char (*), char (*))
      returns (fixed bin);

max_len = date_time_$format_max_length (format, zone, lang);
```

ARGUMENTS

format
: a control string intended to be given to date_time_$format. (Input) b..argx zone the short name of the zone in which conversion will be done. (Input) "system_zone" means use the system default zone. "" means use the per-process default zone. (input)

lang
: the language in which month names and time zones are expressed. "system_lang" means use the system default time language. "" means use per-process default time language.

max_len
: is the length of the longest string which can result from processing the given format. (Output)

ERROR HANDLING

The sub_err_ mechanism is used when an error occurs. The inofrmation in the sub_error_info structure is explicit as to the type of error and detailed in its explanation. Within a sub_error_handler it is easy to display this data. Then a message will be produced by:

```
call com_err_ (sub_error_info.status_code, "my_name", "^a",
   sub_error_info.info_string);
```

An example of a detailed message that could result from this:

```
   my_name:  The picture contains a syntax error.
 Format is:  "^yc-^98my-^99dm"
  error at:        ^
```

See the list under date_time_$format for the values that can be present in the status_code field.

Entry: date_time_$from_clock

Given a Multics standard calendar clock value and an output time zone name, return the month, day of the month, the year, the hour of the day, the minute of the hour, the second of the minute, the number of microseconds, the day in week, the day in year, and the day in clock. The caller may specify one of the time zones in the time_info_ in which the decoded clock value is to be expressed, or may request that the value be expressed in one of the default time zones.

USAGE

```
declare date_time_$from_clock entry (fixed bin(71), char(*), ptr, fixed
      bin(35));

call date_time_$from_clock_ (clock, zone, addr(time_value), code);
```

ARGUMENTS

clock
: is the binary clock value to be decoded. (Input)

zone
: the short name of the zone in which output time value is expressed. (Input) "system_zone" means use the system default zone. "" means use the per-process default zone.

time_value
: is the structure containing time parts. (Output) See "Structure" below.

code
: is a standard status code. (Output) It can have one of the following values--

error_table_$dt_date_too_big
: the date given is after 9999-12-31_23:59:59.999999_GMT.

error_table_$dt_date_too_small
: the date given is before 0001-01-01_00:00:00.000000_GMT.

error_table_$dt_year_too_big
: the clock value given, when converted to the specified time zone, is after the year 9999.

error_table_$dt_year_too_small
: the clock value given, when converted to the specified time zone, is before the year 0001.

error_table_$unimplemented_version
: a structure is not a version this procedure can handle.

error_table_$unknown_zone
the time zone specified by the caller was not found in time_info_.

STRUCTURE

This is the structure used by date_time_$from_clock to return the parts of a clock value. It is also used by date_time_$to_clock to hold the input values which are to be combined to make a clock value. This structure is declared in time_value.incl.pl1.

```
dcl 1 time_value   aligned based(Ptime_value),
      2 version    char (8),
      2 yc         fixed bin,
      2 my         fixed bin,
      2 dm         fixed bin,
      2 Hd         fixed bin,
      2 MH         fixed bin,
      2 SM         fixed bin,
      2 US         fixed bin(20),
      2 fw         fixed bin(20),
      2 dw         fixed bin,
      2 dy         fixed bin,
      2 dc         fixed bin(22),
      2 Uc         fixed bin(71),
      2 za         char (5),
      2 zone_index fixed bin,
      2 leap_year  fixed bin;
```

STRUCTURE ELEMENTS

version
Version of this structure (Vtime_value_3).

yc
Year part of date (e.g. 1978). All values in this structure are time zone adjusted.

my
Month part of date (e.g. 7= July)

dm
Day of month part of date (e.g. 4)

Hd
Hour of the day (e.g. 18)

MH
Minute of the hour (e.g. 35)

SM
Second of the minute (e.g. 59)

US
Microseconds in excess of second

fw
Fiscal week (a number representing yyyyww)

dw
Day of the week (1=Mon, 7=Sun).

dy
Day of the year (e.g. 12/31 = 365 or 366).

dc
Number of days in calendar value (e.g. Jan 1, 0001 => 1).

Uc
Number of microseconds in calendar value (e.g. Jan 1, 0001 midnight => 0).

za
The name of the zone in which the data is expressed.

zone_index
The index in time_info_$zone_names of the zone.

leap_year
This is a 1 if it is a leap year, otherwise it is a 0.

"date" For date_time_$to_clock fields of the structure are only valid in certain combinations. This table shows with the *'s which fields may be present together. All others must be zero.

```
                              CASE

                       +-1-+-2-+-3-+-4-+
     time_value.yc     | * | * |   |   |   In cases 1, 2, & 4, if dw is
     time_value.my     | * |   |   |   |   present,  it is used to verify
     time_value.dm     | * |   |   |   |   the value converted.
     time_value.fw     |   |   | * |   |
     time_value.dw     |   |   |(*)|   |   In case 3 it actually defines
     time_value.dy     |   | * |   |   |   a day.  If not present, Monday
     time_value.dc     |   |   |   | * |   is assumed.
                       +-v-+-v-+-v-+-v-+
                         |   |   |   +-clock_date = converted (dc)
                         |   |   +-----clock_date = converted (fw,dw)
                         |   +---------clock_date = converted (yc,dy)
                         +-------------clock_date = converted (yc,my,dm)
```

Entry: date_time_$from_clock_interval

Given two clock values, return the number of years, months, weeks, days, hours, minutes, seconds, and microseconds between them. The set of units to use is specified, as well as whether any are to include the fractional remainder.

USAGE

```
declare date_time_$from_clock_interval entry (fixed bin(71), fixed
      bin(71), ptr, fixed bin(35));

call date_time_$from_clock_interval (clock1, clock2, addr
      (time_offsets), code);
```

ARGUMENTS

clock1
: is the base time value. (Input) The output is expressed relative to this binary clock value.

clock2
: is the offset time value. (Input) clock1 is in essence subtracted from this value. If this value is later than clock1, all results will be positive. If this value is earlier, all results will be negative.

time_offsets
: is the structure containing resulting time values. (Output) See "Notes" below.

code
: is a standard status code. (Output) It can have one of the following values--

error_table_$dt_bad_day_of_week
: the returned clock reading does not fall on the given day of the week.

error_table_$dt_bad_dm
: day_in_month<1 or day_in_month>month_size.

error_table_$dt_bad_dy
: day_in_year < 0 or day_in_year > year_size (which is 355 for 1582).

error_table_$dt_bad_my
: month_in_year<1 or month_in_year>12.

error_table_$dt_date_not_exist
: the date given is in the nonexistent range of 1582-10-05 through 1582-10-14

error_table_$dt_date_too_big
: the date given is after 9999-12-31_23:59:59.999999_GMT.

error_table_$dt_date_too_small
the date given is before 0001-01-01_00:00:00.000000_GMT.

error_table_$dt_no_interval_units
no units given in which to express the interval.

error_table_$dt_offset_too_big_negative
an offset is so big that when it is applied, it yields a date before 0001-01-01_00:00:00.000000_GMT.

error_table_$dt_offset_too_big_positive
a negative offset is so big that when it is applied, it yields a date after 9999-12-31_23:59:59.999999_GMT.

error_table_$dt_year_too_big
the clock value given, when converted to the specified time zone, is after the year 9999.

error_table_$dt_year_too_small
the clock value given, when converted to the specified time zone, is before the year 0001.

error_table_$unimplemented_version
a structure is not a version this procedure can handle.

NOTES

The following structure is used by both date_time_$from_clock_interval and date_time_$offset_to_clock. For from_clock_interval, it contains all of the offset values which define the indicated interval. For offset_to_clock, it contains all the values to be added to clock value. This structure is declared in time_offset.incl.pl1.

```
dcl 1 time_offset   aligned based(Ptime_offset),
      2 version     char (8),
      2 flag,
        3 yr        fixed bin,
        3 mo        fixed bin,
        3 wk        fixed bin,
        3 da        fixed bin,
        3 hr        fixed bin,
        3 min       fixed bin,
        3 sec       fixed bin,
        3 Usec      fixed bin,
      2 val,
        3 yr        float dec (20),
        3 mo        float dec (20),
        3 wk        float dec (20),
        3 da        float dec (20),
        3 hr        float dec (20),
        3 min       float dec (20),
        3 sec       float dec (20),
        3 Usec      float dec (20),
      2 dw,
        3 flag      fixed bin,
        3 val       fixed bin;
```

STRUCTURE ELEMENTS

version
Version of this structure (Vtime_offset_2).

flag
For from_clock_interval, this structure specifies which units are to be used to express the interval. For offset_to_clock, it specifies which fields contain data to be used. These fields may contain one of 3 values. The meaning depends on which operation is being done.

UNUSED (=0)
the corresponding time_offset.val units are not used.

USED (=1)
For offset_to_clock, the corresponding time_offset.val unit is applied as an offset.

INTEGER (=1)
For from_clock_interval, an integer value is returned in the corresponding time_offset.val units field.

FRACTION (=2)
For from_clock_interval, the corresponding time_offset.val units field is returned as a real number of units (including fractional units).

The fields in this sub-structure represent years, months, weeks, days, hours, minutes, seconds, and microseconds.

val
The fields in this sub-structure contain the various values. For offset_to_clock they are the values to be added to the clock. They may be negative if need be. For from_clock_interval they are the values which made up the indicated interval. Each of these fields is in use only if its flag is set. These fields are named just like the flags are.

dw.flag
specifies how a day of week adjustment is to be applied by offset_to_clock. When applying a day of week offset, the day of week given in dw.val will be used as an offset from the given clock_in value. It must have one of the following values, which may be referred to using the named constants:

BEFORE (=2)
select the date before clock_in on which the specified day of week most recently occurred.

ON_OR_BEFORE (=-1)
select the date on or before clock_in on which the specified day of week most recently occurred.

UNUSED (=0)
do not apply a day of week offset.

ON_OR_AFTER (=1)
select the date on or after clock_in on which the specified day of week will next occur.

AFTER (=2)
select the date after clock_on on which the specified day of week will next occur.

dw.value
specifies a day of week to be used as an offset. It may range from +1 to +7, where 1 = Monday, ... and 7 = Sunday.

Entry: date_time_$fstime

This entry performs the same function as date_time_ given a 36-bit storage system date value.

USAGE

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

call date_time_$fstime (ssclock, str);
```

ARGUMENTS

ssclock
is an internal storage system clock value. (Input)

str
is the resultant character string. (Output)

Entry: date_time_$offset_to_clock

This entry point creates a new Multics clock value by adjusting an input clock value to a specified day-of-week and then adding relative date/time offsets. The relative date/time values include a year offset, month offset, week offset, day offset, hour offset, minute offset, second offset, and microsecond offset. Any of these values may be zero (no offset from input clock value) or negative (backwards offset from input clock value). In addition, an input time zone is specified.

USAGE

```
declare date_time_$offset_to_clock entry (ptr, fixed bin(71), char(*),
      fixed bin(71), fixed bin(35));

call date_time_$offset_to_clock (addr(time_offset), clock_in, zone,
      clock, code);
```

ARGUMENTS

time_offset
is the structure containing time offsets to be applied. (Input) Structure is defined in time_offset.incl.pl1.

clock_in
is the clock value to which offsets are applied. (Input)

zone
is the zone in which clock_in is to be interpreted. (Input)

clock
is the resulting clock value. (Output)

code
is a standard status code. (Output) It can have one of the following values--

error_table_$dt_bad_day_of_week
the returned clock reading does not fall on the given day of the week.

error_table_$dt_bad_dm
day_in_month<1 or day_in_month>month_size.

error_table_$dt_bad_dy
day_in_year < 0 or day_in_year > year_size (which is 355 for 1582).

error_table_$dt_bad_my
month_in_year<1 or month_in_year>12.

error_table_$dt_date_not_exist
the date given is in the nonexistent range of 1582-10-05 through 1582-10-14

error_table_$dt_date_too_big
the date given is after 9999-12-31_23:59:59.999999_GMT.

error_table_$dt_date_too_small
the date given is before 0001-01-01_00:00:00.000000_GMT.

error_table_$dt_offset_too_big_negative
an offset is so big that when it is applied, it yields a date before 0001-01-01_00:00:00.000000_GMT.

error_table_$dt_offset_too_big_positive
a negative offset is so big that when it is applied, it yields a date after 9999-12-31_23:59:59.999999_GMT.

error_table_$dt_year_too_big
the clock value given, when converted to the specified time zone, is after the year 9999.

error_table_$dt_year_too_small
the clock value given, when converted to the specified time zone, is before the year 0001.

error_table_$unimplemented_version
a structure is not a version this procedure can handle.

NOTES

See the notes under date_time_$from_clock_interval for the description of the time_offset structure. The order of applying these offsets can affect the resultant clock value. In all cases, the order required by convert_date_to_binary_ has been used. The order is as follows:

1) decode the input clock value into absolute date/time values specified in terms of the input time zone. This zone may affect the day-of-week represented by the input clock value, and hence, may affect any day-of-week offset adjustment.

2) apply any day-of-week offset by adding/subtracting days to/from the absolute date until the day-of-week represented by the decoded clock value equals the specified day-of-week.

3) apply any year offset to the decoded clock value. If applying the year offset results in a nonexistent date, then use the previous existing day, e.g. "1583-10-10 -1yr" would yield 1582-10-04.

4) apply any month offset to the decoded clock value. If applying the month offset results in a nonexistent date, then use the last day of the month (taking leap years into account), e.g. "Jan 31 3 months" would yield April 30. instead.

5) apply the day offset, hour offset, minute offset, second offset, and microsecond offset.

6) encode the resultant absolute date/time specification into the output clock value.

Entry: date_time_$set_lang

This entry sets or resets the user's default time language.

USAGE

```
declare date_time_$set_lang (char(*), fixed bin(35));

call date_time_$set_lang (lang, code);
```

ARGUMENTS

lang
: the language which is to be made current. (Input) "system_lang" means use the system default time language.

code
: is a standard status code. (Output) It can have one of the following values--

error_table_$dt_unknown_time_language
: the language specified by the caller was not found in time_info_.

Entry: date_time_$set_zone

This entry sets or resets the user's default zone.

USAGE

```
declare date_time_$set_zone entry (char(*), fixed bin(35));

call date_time_$set_zone (zone code);
```

ARGUMENTS

zone
the short name of the zone which is to be made current. (Input) "system_zone" means use the system default zone.

code
is a standard status code. (Output) It can have the following value:

error_table_$unknown_zone
the time zone specified by the caller was not found in time_info_.

Entry: date_time_$to_clock

Given any or all of the following - years, months, days, hours, minutes, seconds, microseconds, day in week, day in year, or day in clock - returns a standard clock value which represents the encoding of these values. All the values must be valid, i.e. not greater than 23, etc.

USAGE

```
declare date_time_$to_clock (ptr, fixed bin(71), fixed bin(35));

call date_time_$to_clock (addr (time_value), clock, code);
```

ARGUMENTS

time_value
is the structure containing time parts. (Input) The structure is defined in time_value.incl.pl1.

clock
is the encoded clock value. (Output)

code
is a standard status code. (Output) It can have one of the following values--

error_table_$bad_time
the time represented by hour, minute and second is invalid, e.g. 23:60 or negative time values

error_table_$dt_bad_day_of_week
the returned clock reading does not fall on the given day of the week.

error_table_$dt_bad_dm
day_in_month<1 or day_in_month>month_size.

error_table_$dt_bad_dy
day_in_year < 0 or day_in_year > year_size (which is 355 for 1582).

error_table_$dt_bad_my
month_in_year<1 or month_in_year>12.

error_table_$dt_conflict
there is a conflicting combination of day-in-calendar, day-in-year, month-in-year, day-in-month and fiscal-week.

error_table_$dt_date_not_exist
the date given is in the nonexistent range of 1582-10-05 through 1582-10-14

error_table_$dt_date_too_big
the date given is after 9999-12-31_23:59:59.999999_GMT.

error_table_$dt_date_too_small
the date given is before 0001-01-01_00:00:00.000000_GMT.

error_table_$unimplemented_version
a structure is not a version this procedure can handle.

error_table_$unknown_zone
the time zone specified by the caller was not found in time_info_.

NOTES

See the notes under date_time_$from_clock for the description of the time_value structure.

Entry: date_time_$valid_format

This entry checks the validity of a format string using precisely the same tests as date_time_$format.

USAGE

```
declare date_time_$valid_format (char (*), fixed bin, fixed bin(35));

call date_time_$valid_format (format, errloc, code);
```

ARGUMENTS

format
either a keyword, or an ioa-like control string describing the desired result in terms of literal characters and date/time selectors. (Input) See the date_time_$format entry point for a description of valid format strings.

errloc
is the character index in the format string where the error occurred. (Output) This is meaningful only if it and code are both nonzero.

code
is a standard status code. (Output) It can have one of the following values--

error_table_$dt_bad_format_selector
unrecognized selector in format string.

error_table_$bad_conversion
a conversion condition occurred while trying to convert a value.

error_table_$dt_no_format_selector
the format string contains no selectors and is not a known keyword.

error_table_$picture_bad
the picture supplied is in error.

error_table_$picture_scale
the picture scale factor not in the range -128:+127.

error_table_$picture_too_big
the normalized picture exceeds 64 characters.

error_table_$size_error
the size condition occurred during processing.

error_table_$unimplemented_version
a structure is not a version this procedure can handle.

Name: datebin_

The datebin_ subroutine has several entry points to convert clock readings into binary integers (and vice versa) representing the year, month, day, hour, minute, second, current shift, day of the week, number of days since January 1, 1901, and the number of days since January 1 of the year indicated by the clock. Clock readings are Multics Greenwich mean time (GMT); all other arguments represent local time.

If arguments passed to datebin_ are not in the valid range, the returned arguments are generally 0 (in certain cases, no checking should be done).

Entry: datebin_$clockathr

This entry point returns a clock reading for the next time the given hour occurs.

USAGE

```
declare datebin_$clockathr entry (fixed bin, fixed bin(71));

call datebin_$clockathr (zz, clock);
```

ARGUMENTS

zz
: is the desired hour and minutes expressed as hhmm in decimal (e.g., 1351). (Input)

clock
: is a calendar clock reading with the number of microseconds since 0000 GMT January 1, 1901. (Output)

Entry: datebin_$datebin

This entry point returns the month, day, year, hour, minute, second, weekday, shift, and number of days since January 1, 1901, given a calendar clock reading.

USAGE

```
declare datebin_ entry (fixed bin(71), fixed bin, fixed bin, fixed bin,
     fixed bin, fixed bin, fixed bin, fixed bin, fixed bin, fixed bin);

call datebin_ (clock, absda, mo, da, yr, hr, min, sec, wkday, s);
```

ARGUMENTS

clock
: is a calendar clock reading with the number of microseconds since 0000 GMT January 1, 1901. (Input)

absda
: is the number of the days the clock reading represents (with January 1, 1901 = 1). (Output)

mo
: is the month (1-12). (Output)

da
: is the day of the month (1-31). (Output)

yr
is the year (1901-1999). (Output)

hr
is the hour of the day (0-23). (Output)

min
is the minute of the hour (0-59). (Output)

sec
is the second of the minute (0-59). (Output)

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

s
is the shift, as defined in installation_parms. (Output)

Entry: datebin_$datofirst

This entry point returns the number of days since January 1, 1901, up to but not including January 1 of the year specified.

USAGE

```
declare datebin_$datofirst entry (fixed bin, fixed bin);

call datebin_$datofirst (yr, datofirst);
```

ARGUMENTS

yr
is the year (1901-1999). (Input)

datofirst
is the number of days since January 1, 1901, up to, but not including, January 1 of the year specified. (Output)

Entry: datebin_$dayr_clk

This entry point returns the day of the year (1-366) given a calendar clock reading. If clock is invalid, -1 is returned.

USAGE

```
declare datebin_$dayr_clk entry (fixed bin(71), fixed bin);

call datebin_$dayr_clk (clock, dayr);
```

ARGUMENTS

clock
: is a calendar clock reading with the number of microseconds since 0000 GMT January 1, 1901. (Input)

dayr
: is the day of the year (1-366). (Output)

Entry: datebin_$dayr_mo

This entry point returns the day of the year when given a month, day, and year.

USAGE

```
declare datebin_$dayr_mo entry (fixed bin, fixed bin, fixed bin,
      fixed bin);

call datebin_$dayr_mo (mo, da, yr, dayr);
```

ARGUMENTS

mo
: is the month (1-12). (Input)

da
: is the day of the month (1-31). (Input)

yr
: is the year (1901-1999). (Input)

dayr
: is the day of the year (1-366). (Output)

Entry: datebin_$following_midnight

This entry point, given a clock reading, returns a clock reading for midnight (local time) of that day.

USAGE

```
declare datebin_$following_midnight entry (fixed bin(71),
      fixed bin(71));

call datebin_$following_midnight (oldclock, clock);
```

ARGUMENTS

oldclock
is a calendar clock reading in microseconds since January 1, 1901, 0000 GMT. (Input)

clock
is a calendar clock reading with the number of microseconds since 0000 GMT January 1, 1901. (Output)

Entry: datebin_$last_midnight

This entry point returns a clock reading for the midnight (local time) preceding the current day.

USAGE

```
declare datebin_$last_midnight entry (fixed bin(71));

call datebin_$last_midnight (clock);
```

ARGUMENTS

clock
is a calendar clock reading with the number of microseconds since 0000 GMT January 1, 1901. (Output)

Entry: datebin_$next_shift_change

This entry, given a clock reading, returns the time of the next shift change, the current shift, and the new shift.

USAGE

```
declare datebin_$next_shift_change entry (fixed bin(71), fixed bin(71),
     fixed bin, fixed bin);

call datebin_$next_shift_change (clock, newclock, shift, newshift);
```

ARGUMENTS

clock
is a calendar clock reading with the number of microseconds since 0000 GMT January 1, 1901. (Input)

newclock
is the time the shift changes next after clock. (Output)

shift
is the current shift at time clock. (Output)

newshift
is the shift that begins at time newclock. (Output)

Entry: datebin_$preceding_midnight

This entry point, given a clock reading, returns a clock reading for midnight (local time) of the preceding day.

USAGE

```
declare datebin_$preceding_midnight entry (fixed bin(71),
      fixed bin(71));

call datebin_$preceding_midnight (oldclock, clock);
```

ARGUMENTS

oldclock
is a calendar clock reading in microseconds since January 1, 1901, 0000 GMT. (Input)

clock
is a calendar clock reading with the number of microseconds since 0000 GMT January 1, 1901. (Output)

Entry: datebin_$revert

This entry point returns a calendar clock reading for the month, day, year, hour, minute, and second specified.

USAGE

```
declare datebin_$revert entry (fixed bin, fixed bin, fixed bin,
      fixed bin, fixed bin, fixed bin, fixed bin(71));

call datebin_$revert (mo, da, yr, hr, min, sec, clock);
```

ARGUMENTS

mo
is the month (1-12). (Input)

da
is the day of the month (1-31). (Input)

yr
    is the year (1901-1999). (Input)

hr
    is the hour of the day (0-23). (Input)

min
    is the minute of the hour (0-59). (Input)

sec
    is the second of the minute (0-59). (Input)

clock
    is a calendar clock reading with the number of microseconds since 0000 GMT January 1, 1901. (Output)

Entry: datebin_$revertabs

This entry point returns a calendar clock reading given the number of days since January 1, 1901.

USAGE

```
declare datebin_$revertabs entry (fixed bin, fixed bin(71));

call datebin_$revertabs (absda, clock);
```

ARGUMENTS

absda
    is the number of the days the clock reading represents (with January 1, 1901 = 1). (Input)

clock
    is a calendar clock reading with the number of microseconds since 0000 GMT January 1, 1901. (Output)

Entry: datebin_$shift

This entry point returns the shift given a calendar clock reading. If clock is invalid, -1 is returned.

USAGE

```
declare datebin_$shift (fixed bin(71), fixed bin);

call datebin_$shift (clock, s);
```

ARGUMENTS

clock
is a calendar clock reading with the number of microseconds since 0000 GMT January 1, 1901. (Input)

s
is the shift, as defined in installation_parms. (Output)

Entry: datebin_$this_midnight

This entry point returns a clock reading for midnight (local time) of the current day.

USAGE

```
declare datebin_$this_midnight entry (fixed bin(71));

call datebin_$this_midnight (clock);
```

ARGUMENTS

clock
is a calendar clock reading with the number of microseconds since 0000 GMT January 1, 1901. (Output)

Entry: datebin_$time

This entry point returns the hour, minute and second given a calendar clock reading. If clock is invalid, hr, min, and sec are -1.

USAGE

```
declare datebin_$time entry (fixed bin(71), fixed bin, fixed bin,
      fixed bin);

call datebin_$time (clock, hr, min, sec);
```

ARGUMENTS

clock
is a calendar clock reading with the number of microseconds since 0000 GMT January 1, 1901. (Input)

hr
is the hour of the day (0-23). (Output)

min
is the minute of the hour (0-59). (Output)

sec
is the second of the minute (0-59). (Output)

Entry: datebin_$wkday

This entry point returns the day of the week (Monday = 1 ... Sunday = 7) given a calendar clock reading. If clock is invalid, 0 is returned.

USAGE

```
declare datebin_$wkday entry (fixed bin(71), fixed bin);

call datebin_$wkday (clock, wkday);
```

ARGUMENTS

clock
is a calendar clock reading with the number of microseconds since 0000 GMT January 1, 1901. (Input)

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

Name: decode_definition_

The decode_definition_ subroutine, given a pointer to an object segment definition, returns the decoded information of that definition in a structured, directly accessible format. This subroutine can only be used on one segment at a time because it uses internal static storage.

USAGE

```
declare decode_definition_ entry (ptr, ptr, bit 1 aligned)

call decode_definition_ (def_ptr, structure_ptr, eof)
```

ARGUMENTS

def_ptr
is a pointer to the selected definition. (Input). (The caller extracts this from the previously returned information.) The initial pointer with which decode_definition_ can be called is a pointer to the base of the object segment (i.e., with a zero offset), unless the decode_definition_$init entry point has been called, in which case the initial pointer can be a pointer to the beginning of the definition section (as returned by the object_info_ subroutine).

structure_ptr
is a pointer to the provided structure in which decode_definition_ returns the desired information. (Input). (See "Notes" below.)

eof
is a binary indicator that is "1"b if the current invocation of decode_definition_ causes the search to go beyond the end of the definition list. If that is the case, the returned information in the structure is null. It may also be "1"b if any error occurs. (Output)

NOTES

The structure, contained in the decode_definition_str.incl.pl1 structure, has the following format:

```
dcl  1 decode_definition_common_header based aligned,
       2 next_def      ptr,
       2 prev_def      ptr,
       2 block_ptr     ptr,
       2 section       char (4) aligned,
       2 offset        fixed bin,
       2 entrypoint    fixed bin;

dcl  1 decode_definition_str based aligned,
       2 header        like decode_definition_common_header,
       2 symbol        char (32) aligned;
```

STRUCTURE ELEMENTS

next_def
is a forward pointer to the next definition in the list. It can be used to make a subsequent call to decode_definition_.

prev_def
is a backward pointer to the preceding definition on the list. This pointer can be null if the definition is of the old format.

block_ptr
: is a pointer to the head of the definition block if this is a segname definition and to the head of a segname list if this is not a segname definition. This pointer can be null if the definition is of the old format.

section
: is a symbolic code defining the type of definition. It can assume one of the following values: text, link, stat, symb, or segn (for segname).

offset
: is the offset of the definition within the given section. This is set to 0 if section is segn.

entrypoint
: is nonzero, if this definition is an entry point. The value of this item is the entry point's offset in the text section.

symbol
: is the character string representation of the definition.

Entry: decode_definition_$decode_cref

This entry point, given a pointer to an object segment definition, returns the decoded information of that definition in a structure similar to that returned by decode_definition_, but with a pointer to the symbol name instead the name itself. It is used only by the cross_reference command.

USAGE

```
declare decode_definition_$decode_cref entry (ptr, ptr, bit (1) aligned,
      ptr);

call decode_definition_$decode_cref (def_ptr, decode_def_acc_ptr, eof,
      link_ptr);
```

ARGUMENTS

def_ptr
: must be a pointer to the beginning of the definition section. (Input)

decode_def_acc_ptr
: is a pointer to a structure in which the entry point is to return information. (Input). (See "Notes" below.)

eof
: is a binary indicator that is "1"b if the current invocation of decode_definition_ causes the search to go beyond the end of the definition list. If that is the case, the returned information in the structure is null. It may also be "1"b if any error occurs. (Input)

link_ptr
is a pointer to the base of the linkage section of the object segment the first time this entry is called for a given object segment. It is to be null for subsequent calls. (Input)

NOTES

The structure filled in by this entry point has the following format. It can be found in decode_descriptor_str.incl.pl1.

```
dcl 1 decode_definition_acc based aligned,
      2 header   like decode_definition_common_header,
      2 acc_ptr  ptr;
```

STRUCTURE ELEMENTS

header
all items in this substructure are the same as for the decode_definition_str substructure header.

acc_ptr
is a pointer to the ACC string that is the symbolic name of this definition.

Entry: decode_definition_$full

This entry point, given a pointer to an object segment definition, returns more complete information about that definition. The symbolic name returned by this entry point can contain up to 256 characters. This entry point does not use internal static storage.

USAGE

```
declare decode_definition_$full entry (ptr, ptr, ptr, bit (1) aligned)
      returns (bit(1) aligned);

call decode_definition_$full (def_ptr, structure_ptr, oi_ptr, eof);
```

ARGUMENTS

def_ptr
is a pointer to the selected definition and is extracted from previously returned information. (Input). The initial pointer with which the decode_definition_$full entry point can be called is a pointer to the base of the definition section of the object segment.

structure_ptr
is a pointer to the provided structure into which the decode_definition_$full entry point returns the desired information. (Input). (See "Notes" below.)

oi_ptr
is a pointer to the structure returned by any entry point of the object_info subroutine. (Input)

eof
is a binary indicator that is "1"b if the current invocation of decode_definition_ causes the search to go beyond the end of the definition list. If that is the case, the returned information in the structure is null. It may also be "1"b if any error occurs. (Output)

NOTES

The structure, contained in the decode_definition_str.incl.pl1 structure, has the following format:

```
dcl 1 decode_definition_full based aligned
      2 header            like decode_definition_common_header,
      2 symbol            char (256) aligned,
      2 symbol_lng        fixed bin,
      2 flags,
        3 new_format      bit (1) unaligned,
        3 ignore          bit (1) unaligned,
        3 entrypt_flag    bit (1) unaligned,
        3 retain          bit (1) unaligned,
        3 arg_count       bit (1) unaligned,
        3 desc_sw         bit (1) unaligned,
        3 unused          bit (30) unaligned,
      2 nargs             fixed bin,
      2 desc_ptr          ptr;
```

STRUCTURE ELEMENTS

header
all items in this substructure are the same as for the decode_definition_str substructure header.

symbol
is the character string representation of the definition.

symbol_lng
is the relevant length of the symbol in characters.

new_format
indicates that the definition is in the new format.

ignore
is the linker ignore switch.
"1"b the linker should ignore this definition.
"0"b the linker should not ignore this definition.

entrypt_flag
is the entry point switch.
"1"b the definition is for an entry point
"0"b the definition is for a segdef.

retain
is the retain switch.
"1"b the definition should be retained.
"0"b the definition should not be retained.

arg_count
is the arg_count switch.
"1"b there is an arg_count for this definition.
"0"b there is no arg_count for this definition.

desc_sw
is the descriptor switch.
"1"b there are descriptors for this definition.
"0"b there are no descriptor for this definition.

unused
is padding.

nargs
indicates the number of arguments expected by this entry, if descr_sw equals "1"b.

desc_ptr
points to an array of 18-bit pointers to the descriptors for the entry, if descr_sw equals "1"b.

Entry: decode_definition_$init

This entry point is used for initialization and is especially useful when the object segment does not begin at offset 0 (as for an archive component). This entry point has no effect when the decode_definition_$full entry point is being used.

USAGE

```
declare decode_definition_$init entry (ptr, fixed bin(24));

call decode_definition_$init (seg_ptr, bit_count);
```

ARGUMENTS

seg_ptr
is a pointer to the beginning of an object segment (not necessarily with an offset of 0). (Input)

bit_count
is the bit count of the object segment. (Input)

Name: decode_descriptor_

The decode_descriptor_ subroutine extracts information from argument descriptors. It should be called by any procedure wishing to handle variable length or variable type argument lists. It processes the descriptor format used by PL/I, BASIC, COBOL, FORTRAN, and Pascal.

USAGE

```
declare decode_descriptor_ entry (ptr, fixed bin, fixed bin,
      bit(1) aligned, fixed bin, fixed bin(24), fixed bin);

call decode_descriptor_ (ptr, n, type, packed, ndims, size, scale);
```

ARGUMENTS

ptr
points either directly at the descriptor to be decoded or at the argument list in which the descriptor appears. (Input)

n
controls which descriptor is decoded. If n is 0, ptr points at the descriptor to be decoded; otherwise, ptr points at the argument list header and the nth descriptor is decoded. (Input)

type
is the data type specified by the descriptor. (Output)
-1 is returned if descriptors are not present in the argument list, if the nth descriptor does not exist, or if the descriptor is in an old format

packed
describes how the data is stored. (Output)

"1"b data is packed
"0"b data is not packed

ndims
indicates the number of dimensions of an array. (Output)

N descriptor is an array of N dimensions
0 descriptor is a scalar

size
is the arithmetic precision, string size, or number of structure elements. (Output)

scale
is the scale of an arithmetic value. (Output) *

Name: define_area_

The define_area_ subroutine is used to initialize a region of storage as an area and to enable special area management features as well. The region being initialized may or may not consist of an entire segment or may not even be specified at all, in which case a segment is acquired (from the free pool of temporary segments) for the caller.

See the release_area_ subroutine for a description of how to free up segments acquired via this interface.

USAGE

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

call define_area_ (info_ptr, code);
```

ARGUMENTS

info_ptr
points to the information structure described in "Notes" below. (Input)

code
is a system status code. (Output)

NOTES

The define_area_ subroutine gives the user more control over an area than is defined in the PL/I language. The PL/I empty built-in function cannot empty a define_area_ area; the release_area_ subroutine must be used instead. PL/I offset values and PL/I area assignment cannot be used with extensible areas. In PL/I, an area variable is always initialized. Consequently, if a based area is overlayed upon arbitrary storage instead of being allocated with a PL/I allocate statement, then the define_area_ subroutine must be used to turn the contents of the based area into a PL/I area value.

The structure pointed to by info_ptr is the standard area_info structure used by the various area management routines and is described by the following PL/I declaration defined by the system include file, area_info.incl.pl1:

```
dcl 1 area_info                   aligned based,
      2 version                   fixed bin,
      2 control,
        3 extend                  bit(1) unaligned,
        3 zero_on_alloc           bit(1) unaligned,
        3 zero_on_free            bit(1) unaligned,
        3 dont_free               bit(1) unaligned,
        3 no_freeing              bit(1) unaligned,
        3 system                  bit(1) unaligned,
        3 pad                     bit(30) unaligned,
      2 owner                     char(32) unaligned,
      2 n_components              fixed bin,
      2 size                      fixed bin(30),
      2 version_of_area           fixed bin,
      2 areap                     ptr,
      2 allocated_blocks          fixed bin,
      2 free_blocks               fixed bin,
      2 allocated_words           fixed bin(30),
      2 free_words                fixed bin(30);
```

STRUCTURE ELEMENTS

version
is to be filled in by the caller and should be 1.

control
are control flags for enabling or disabling features of the area management mechanism.

extend
indicates whether the area is extensible. This feature should only be used for per-process, temporary areas.
"1"b yes
"0"b no

zero_on_alloc
indicates whether blocks are cleared (set to all zeros) at allocation time.
"1"b yes
"0"b no

zero_on_free
indicates whether blocks are cleared (set to all zeros) at free time.
"1"b yes
"0"b no

dont_free
indicates whether the free requests are disabled, thereby not allowing reuse of storage within the area.
"1"b yes
"0"b no

no_freeing
indicates whether the allocation method assumes no free requests will ever be made for the area and that, hence, a faster allocation strategy can be used.
"1"b yes
"0"b no

system
causes the use of hcs_$make_seg instead of get_temp_segments to create the first component. It assumes that the original area is all zeroes, rather than explicitly zeroing it.
"1"b yes
"0"b no

pad
is not used and must be all zeros.

owner
is the name of the program requesting that the area be defined. This is needed by the temporary segment manager.

n_components
is the number of components in the area. (This item is not used by the define_area_ subroutine.)

size
is the size, in words, of the area being defined. The minimum size is thirty-two (decimal) words. The maximum size is the maximum number of words in a segment.

version_of_area
is 1 for current areas and 0 for old-style areas. (This item is not used by the define_area_ subroutine.)

areap
is a pointer to the region to be initialized as an area. If this pointer is null, a temporary segment is acquired for the area and areap is set as a returned value. If areap is initially nonnull, it must point to a 0 mod 2 address.

allocated_blocks
is the number of allocated blocks in the entire area. (This item is not used by the define_area_ subroutine.)

free_blocks
is the number of free blocks in the entire area (not counting virgin storage). (This item is not used by the define_area_ subroutine.)

allocated_words
is the number of allocated words in the entire area. (This item is not used by the define_area_ subroutine.)

free_words
is the number of free words in the entire area. (This item is not used by the define_area_ subroutine.)

Name: delete_

The delete_ subroutine deletes segments, directories, multisegment files, and data management files and unlinks links. If the segment, directory, multisegment file or data management file to be deleted is protected (i.e., the safety switch or copy 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 rather than directly addressing entry points in hcs_. If a data management file is subject to a pending transaction, the data management file can not be deleted until the transaction is completed.

Entry: delete_$path

This entry point is called with the pathname of the segment, directory, multisegment file, data management 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);
```

ARGUMENTS

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

entryname
: is the entryname of the segment, directory, multisegment file, data management file, or link. (Input)

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

This page intentionally left blank.

force_sw
"1"b allows the entry to be deleted even if it is protected.
"0"b acts according to question_sw if the entry is protected.

question_sw
"1"b asks the user if a protected entry should be deleted if the force_sw is "0"b; if the user gives a negative response, the subroutine returns the code error_table_$action_not_performed. If question_sw is "1"b and the entryname argument is the name of a directory, the delete_ subroutine prints an error 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.

The following switches allow control by the caller over which storage system entry types can be deleted:

directory_sw
"1"b allow the entryname argument to refer to a directory.
"0"b return the code error_table_$dirseg if the entryname argument refers to a directory.

segment_sw
"1"b allow the entryname argument to refer to a segment, multisegment file, or data management file.
"0"b return the code error_table_$nondirseg if the entryname argument refers to a segment on a multisegment file.

link_sw
"1"b allow the entryname argument to refer to a link (see chase_sw).
"0"b return the code error_table_$not_a_branch if the entryname argument refers to a link.

chase_sw
"1"b allow the target of a link to be deleted, if link_sw = "1"b and the entryname argument refers to a link; the deletion of the segment or directory pointed to by the link is governed by the settings of directory_sw and segment_sw.
"0"b unlink the link if link_sw ="1"b and the entryname argument refers to a link.

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

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, Data Management 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);
```

ARGUMENTS

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

switches
are switches that specify the actions to be taken. (Input) (See the delete_$path entry point).

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

code
is a storage system status code. (Output)

Name: dial_manager_

The dial_manager_ subroutine is the user interface to the answering service dial facility. The dial facility allows a process to communicate with multiple terminals at the same time. This subroutine uses a structure, dial_manager_arg, to receive arguments from its caller. This structure is described below, under "Notes". For more information, see the description of the dial command in the the Commands manual.

The dial_manager_ subroutine uses an event channel to communicate with the answering service. This event channel is specified by dial_manager_arg.dial_channel. The channel must be created by the caller. The answering service sends notices of dial connections and hangups over this channel. The dial_manager_ subroutine goes blocked on the event-wait channel awaiting a response to the request from the answering service. When the user program receives wakeups over this channel, it should call the convert_dial_message_ subroutine to decode the event message.

The dial_manager_$allow_dials and dial_manager_$registered_server entry points establish a dial line. The dial_id specified in dial_manager_arg.dial_qualifier is used as the first argument to the dial command when connecting a terminal to a process. The dial_id may be an alphanumeric string from 1 to 12 characters long. The dial_id "system" and "s" are reserved for the Initializer process. A process can have only one dial line active at a time.

Entry: dial_manager_$allow_dials

This entry point requests that the answering service establish a dial line to allow terminals to dial to the calling process. The caller must set dial_manager_arg.dial_qualifier to the dial_id for the dial line. The caller must also set dial_manager_arg.dial_channel to an event-wait channel in the caller's process. After the dial_manager_$allow_dials entry point has been called, the event channel may be changed to an event-call channel. To connect a terminal to the process, the User_id of the process must be specified as the second argument of the dial command. If the process has already established another dial line, the request is rejected and code is set to error_table_$dial_active.

USAGE

```
declare dial_manager_$allow_dials entry (ptr, fixed bin(35));

call dial_manager_$allow_dials (request_ptr, code);
```

ARGUMENTS

request_ptr
: is a pointer to the dial_manager_arg structure described in "Notes" below. (Input)

code
: is a standard status code. (Output)

Entry: dial_manager_$dial_out

This entry point is used to request that an auto call channel be dialed to a given destination and, if the channel is successfully dialed, that the channel be assigned to the requesting process. The caller must set dial_manager_arg.dial_out_destination to the telephone number to be dialed. The caller must also set dial_manager_arg.dial_channel to an event-wait channel in his process. The answering service sends notice of dial completions and hangups over this channel. After the dial_manager_$dial_out entry point has been called the event channel may be changed to an event-call channel. The user programs receiving the wakeup should call the convert_dial_message_ subroutine to decode the event message. The caller may set dial_manager_arg.channel_name to the name of a specific channel to be used. It is also possible to set dial_manager_arg.channel_name to a starname, in which case the answering service chooses a channel that has a matching name and has all the attributes specified in dial_manager_arg.reservation_string. The name of the chosen channel is not returned by dial_manager_; it must be obtained via a call to convert_dial_message_.

USAGE

```
declare dial_manager_$dial_out entry (ptr, fixed bin(35));

call dial_manager_$dial_out (request_ptr, code);
```

ARGUMENTS

request_ptr
: is a pointer to the dial_manager_arg structure described in "Notes" below. (Input)

code
: is an error status indicator. (Output) It can assume any value documented in the convert_dial_message_ description (earlier in this manual), or one of the following:
  error_table_$bad_conversion
  : a reservation_string value (BAUD_RATE) was not a proper decimal value.
  error_table_$invalid_line_type
  : the value of LINE_TYPE is not acceptable.
  error_table_$bad_arg
  : reservation_string contains an unrecognized attribute.

Entry: dial_manager_$privileged_attach

This entry point allows a privileged process to attach a "slave" channel. The effect is as if that terminal had dialed to the requesting process. The caller must set all variables required by the dial_manager_$allow_dials entry point and then must set dial_manager_arg.channel_name to the name of the channel that is to be attached; dial_manager_arg.dial_qualifier is not used and should be set to the null string. This must be the same name as specified by the channel master file. The slave service type must be specified for this channel in the channel master file. The calling process must have rw access to the access control segment `<channel_name>.acs` in `>sc1>rcp` if this request is to be honored.

USAGE

```
declare dial_manager_$privileged_attach entry (ptr, fixed bin(35));

call dial_manager_$privileged_attach (request_ptr, code);
```

ARGUMENTS

request_ptr
is a pointer to the dial_manager_arg structure described in "Notes" below. (Input)

code
is a standard status code. (Output)

Entry: dial_manager_$registered_server

This entry point is used to request that the answering service establish a dial line to allow terminals to dial to the calling process using only the dial qualifier. The calling process must have rw access to the access control segment `dial.<dial qualifier>.acs` in `>sc1>rcp` if this request is to be honored. If the process has already established a dial line, the request is rejected and code is set to error_table_$dial_active.

USAGE

```
declare dial_manager_$registered_server entry (ptr, fixed bin(35));

call dial_manager_$registered_server (request_ptr, code);
```

ARGUMENTS

request_ptr
is a pointer to the dial_manager_arg structure described in "Notes" below. (Input)

code
is a standard status code. (Output)

Entry: dial_manager_$release_channel

This entry point is used to request the answering service to release the channel specified in channel_name. This channel must be dialed to the caller at the time of this request. The caller must set dial_manager_arg.dial_channel to an event wait channel in the caller's process. The caller also must set dial_manager_arg.channel_name to the name of the channel to be released. The user must make dial_manager_arg.dial_channel an event-wait channel before using this call. If the channel was dialed, the channel is returned to the answering service and another access request may be issued. If the channel is a slave channel, the channel is hung up.

USAGE

```
declare dial_manager_$release_channel entry (ptr, fixed bin(35));

call dial_manager_$release_channel (request_ptr, code);
```

ARGUMENTS

request_ptr
    is a pointer to the dial_manager_arg structure described in "Notes" below. (Input)

code
    is a standard status code. (Output)

Entry: dial_manager_$release_channel_no_hangup

This entry point performs the same function as the dial_manager_$release_channel entry point except that slave channels are not hung up.

Entry: dial_manager_$release_dial_id

This entry point functions as does dial_manager_$shutoff_dials, except that dialed terminals are not hung up. The user can later release dialed terminals by a call to dial_manager_$shutoff_dials or by calls to dial_manager_$release_channel.

USAGE

```
declare dial_manager_$release_dial_id (ptr, fixed bin (35));

call dial_manager_$release_dial_id (request_ptr, code);
```

ARGUMENTS

request_ptr
    is a pointer to the dial_manager_arg structure described in "Notes" below. (Input)

code
    is a standard status code. (Output)

Entry: dial_manager_$release_no_listen

This entry point requests the answering service to release the channel specified in channel_name, which must have been attached by means of the dial_manager_$standd_attach entry point. The channel is left in a hung-up state and is not available for use until an explicit "attach" operator command is issued for the channel. This entry point has the same requirements as the dial_manager_$release_channel entry point.

USAGE

```
declare dial_manager_$release_no_listen entry (ptr, fixed bin (35));

call dial_manager_$release_no_listen (request_ptr, code);
```

ARGUMENTS

request_ptr
    is a pointer to the dial_manager_arg structure described in "Notes" below. (Input)

code
    is a standard status code. (Output)

Entry: dial_manager_$shutoff_dials

This entry point informs the answering service that the user process wishes to prevent further dial connections, and that existing connections should be terminated. The same information should be passed to this entry point as was passed to the dial_manager_$allow_dials or dial_manager_$registered_server entry point. The dial_channel must be an event-wait channel.

USAGE

```
declare dial_manager_$shutoff_dials (ptr, fixed bin(35));

call dial_manager_$shutoff_dials (request_ptr, code);
```

ARGUMENTS

request_ptr
    is a pointer to the dial_manager_arg structure described in "Notes" below. (Input)

code
    is a standard status code. (Output)

Entry: dial_manager_$tandd_attach

This entry point allows a process with appropriate access to attach any communications channel that is in the channel master file and not already in use, for the purpose of performing online testing of the channel. The requesting process acquires the channel in a hung-up, nonlistening state. The channel can be released using either the dial_manager_$release_channel or the dial_manager_$release_no_listen entry point. In the latter case, the channel will be unavailable to users until the operator enters an attach command for the channel. The caller must set all the variables required by the dial_manager_$privileged_attach entry point; dial_manager_arg.dial_qualifier is not used and should be set to the null string.

USAGE

```
declare dial_manager_$tandd_attach entry (ptr, fixed bin (35));

call dial_manager_$tandd_attach (request_ptr, code);
```

ARGUMENTS

request_ptr
is a pointer to the dial_manager_arg structure described in "Notes" below. (Input)

code
is a standard status code. (Output)

ACCESS REQUIRED

The caller must have at least rw access to both `>sc1>rcp>tandd.acs` and `>sc1>rcp>CHAN_NAME.acs`, where CHAN_NAME is the name of the channel to be attached.

Entry: dial_manager_$terminate_dial_out

This entry point is used to request that the answering service hang up an auto call line and unassign it from the requesting process. The caller must set dial_manager_arg.channel_name to the name of the channel being used; channel_name cannot be null. The caller also must set dial_manager_arg.dial_channel to an event-wait channel.

USAGE

```
declare dial_manager_$terminate_dial_out entry (ptr, fixed bin(35));

call dial_manager_$terminate_dial_out (request_ptr, code);
```

ARGUMENTS

request_ptr
is a pointer to the dial_manager_arg structure. (Input) See "Notes" below.

code
is a standard status code. (Output)

NOTES

This structure is used to pass a variety of information to the dial_manager_ subroutine. It is declared in dial_manager_arg.incl.pl1. It has the following declaration:

```
dcl 1 dial_manager_arg            based aligned,
      2 version                   fixed bin,
      2 dial_qualifier            char (22),
      2 dial_channel              fixed bin (71),
      2 channel_name              char (32),
      2 dial_out_destination      char (32),
      2 reservation_string        char (256),
      2 dial_message              fixed bin (71);
      2 access_class              bit (72),
      2 flags                     aligned,
        3 access_class_required   bit (1) unaligned,
        3 privileged_operation    bit (1) unaligned,
        3 mbz                     bit (34) unaligned;
```

STRUCTURE ELEMENTS

version
indicates the version of the structure that is being used. This is set by the caller and must be dial_manager_arg_version_4.

dial_qualifier
is the dial qualifier for calls to the dial_manager_$allow_dials, dial_manager_$registered_server, dial_manager_$shutoff_dials, and dial_manager_$release_dial_id entry points. This field should be set to blanks if it is not used.

dial_channel
is an interprocess communication channel used to receive messages from the answering service. The channel must always be an event-wait channel at the time a call to any dial_manager_ entry is made. If the value of dial_channel is 0, then the answering service will not send any status messages to the requesting process.

channel_name
In calls to the dial_manager_$privileged_attach entry point it indicates which slave channel to attach. In calls to the dial_manager_$dial_out entry point, it indicates which autocall channel should be used for a dial_out attempt. For these two entries, the following convention is observed: the caller can fully specify a channel name or can use the star convention to specify a group of channels from which the answering service is to pick one. This name is matched against both the channel name in the cdt and the generic_destination field for the channel, if one exists.

dial_out_destination
is used for calls to the dial_manager_$dial_out entry point. Interpretation of this value is determined by the multiplexer that controls the channel being dialed out. The standard FNP multiplexer interprets this value as a telephone number and ignores all characters except decimal digits and the exclamation point (!). It recognizes "!" as a dial-tone-wait character and will suspend dialing until the autocall unit receives a dial tone. Any number of "!" characters can exist in a dial_out_destination, and the standard FNP multiplexer will pause at each. This field should be set to blanks if it is not used.

When the destination specifies an X.25 address it may optionally be preceded by "*" or "x29," to indicate that an X.29 (PAD) call should be made. For example, a destination of

```
x.29,3106:mitmul or
*3106:mitmul
```

specifies an X.29-type call on TYMNET.

reservation_string
is used to specify the desired characteristics of a channel in calls to the dial_manager_$dial_out entry. The reservation string (which can be null), consists of reservation attributes separated by commas. The channel used by a dial-out operation must have the characteristics specified in the reservation string. Reservation attributes consist of a keyword and optional argument. Attributes allowed are:

```
baud_rate=BAUD_RATE
line_type=LINE_TYPE
```

The attribute name, such as "baud_rate", must appear literally in the string. BAUD_RATE is a decimal representation of the desired channel line speed and must appear in a baud_rate attribute. LINE_TYPE is a valid line type, chosen from line_types.incl.pl1 and must appear in a line_type attribute. Examples: "baud_rate=300, line_type=ASCII", "line_type=BSC". This field should be set to blanks if it is not used or no particular channel attributes are required.

dial_message
is a copy of the dial_message received from the answering service. The dial_manager_ subroutine makes an answering service request based upon the arguments supplied by its caller; it then waits for a reply from the answering service. This reply is converted using convert_dial_message_, and some of the results of the conversion are immediately available to dial_manager_ callers as output arguments. To obtain other portions of the dial_message absorbed by dial_manager_, the user must call convert_dial_message_ specifying the value of this field. This field is set to -1 if an error occurs in the dial_manager_ or answering service request; convert_dial_message_ rejects attempts to convert such a message with the return code error_table$badcall. (Output)

access_class
is the access class to be associated with the channel to be attached by the dial_manager_$dial_out or dial_manager_$privileged_attach entry. It is only used if access_class_required (below) is "1"b. It must be the same as the requesting process's max authorization, unless the process has the "comm" privilege set, in which case access_class must be equal to or lower than the requesting process's authorization. (Input)

access_class_required
if "1"b, indicates that the channel to be attached by the dial_manager_$dial_out or dial_manager_$privileged_attach entry must have the access class specified by access_class (above) or must be a mult-class channel whose access class can be set to access_class.

privileged_operation
If "1"b, indicates that a call to dial_manager_$accept_dials or dial_manager_$registered_server should establish a privileged dial server. For example, one which accepts channels whose access class is in the range system_low:access_class.

mbz
must be "0"b.

Name: display_access_class_

The display_access_class_ function converts a bit(72) aligned representation of an access authorization or access class into a character string of the form:

```
LL...L:CC...C
```

where `LL...L` is an octal sensitivity level number, and `CC...C` is an octal string representing the access category set.

USAGE

```
declare display_access_class_ entry (bit(72) aligned)
     returns (char(32) aligned);

aim_chars = display_access_class_ (aim_bits);
```

ARGUMENTS

aim_bits
is the binary representation to be converted. (Input)

aim_chars
is the character string representation. (Output)

NOTES

Only significant digits of the level number (usually a single digit from 0 to 7) are printed.

Currently, only 18 access category bits are used, so that only six octal digits are required to represent access categories. Therefore, aim_chars is padded on the right with blanks, which may be used at a later time for additional access information. Trailing zeros are NOT stripped.

If either the level or category field of aim_bits is invalid, the erroneous field is returned as full octal (6 digits for level, 12 digits for category), followed by the string "(undefined)".

Entry: display_access_class_$range

The display_access_class_$range function converts an AIM access class range to a character string when the names of levels and categories are not available.

USAGE

```
declare display_access_class_$range entry ((2) bit(72) aligned)
     returns (char(32) aligned);

string_range = display_access_class_ (AIM_range);
```

ARGUMENTS

AIM_range
is a standard access class range. (Input)

string_range
is a string of the form:

```
l:cccccc-L:CCCCCC
```

where l is the level, from 0 to 7, of the bottom of the range. (Output)
cccccc are the categories of the bottom of the range.

The categories are a bit string (one bit per category) represented in octal.

L is the level, from 0 to 7, of the top of the range.
CCCCCC are categories of the top of the range.

The categories are a bit string (one bit per category) represented in octal.

This page intentionally left blank.

Name: display_file_value_

The display_file_value_ subroutine outputs information about a file on a user-supplied switch.

USAGE

```
dcl display_file_value_ entry (ptr, file, fixed bin (35));

call display_file_value_ (switch, a_file, code);
```

ARGUMENTS

switch
is a pointer to the iocb of the switch on which output is to be written. If it is null, then iox_$user_output is used. (Input)

a_file
is the file, variable, or constant whose value is to be displayed. (Input)

code
is a standard status code. (Output)

NOTES

The output produced is, first, the values of the two pointers that comprise a file. If the file is closed, then a note to that effect is produced, and the values of the file attribute block are given, and that is all.

For all open files, the file name, address of its iocb, and pathname are given. If the file is neither stream nor record type, or if it is both, then a note to the effect that the fsb is inconsistent is given. Attributes relevant to the type of file (stream or record) are given. For stream input files, the current input buffer is printed, with a circumflex above the next character that is to be parsed.

Name: dl_handler_

This subroutine has three entry points that issue queries for each of three situations involving deletion. These situations are:

Deletion of an entry whose safety switch or copy switch is on.
Deletion via a starname that matches all entries, e.g. "**".
Deletion of a directory (delete_dir always queries).

This subroutine returns a status code depending on the user's answer. If the user answers "yes", all three entry points turn off the safety and copy switches, and in the case of a directory, set sma to the user before returning.

The dl_handler_ entry point, called when an entry has its safety switch or copy switch on, issues a query of the form:

```
<caller>:  <path> is protected.  Do you want to delete it?
```

If the user answers yes, dl_handler_ turns off both switches and returns a zero status code.

USAGE

```
dcl dl_handler_ entry (char(*), char(*), char(*), fixed bin(35));

call dl_handler_ (caller, dn, en, code);
```

ARGUMENTS

caller
: is the name of the calling program, used to print the query. (Input)

dn
: is the directory name. (Input)

en
: is the entry name. (Input)

code
: is a standard status code. (Output) It can be:

  0 the user answered yes, switches have been turned off, and the entry can now be deleted.

  error_table_$action_not_performed
  : the user answered no.

  other codes
  : the switches could not be turned off.

The two other entry points have the same calling sequence as dl_handler_.

Entry: dl_handler_$dblstar

This entry point issues the query:

```
Do you want to '<caller> <en>' in <dn>?
```

where caller, the name of the calling program, is assumed to be a suitable verb. This entry point is called, for example, by the delete and unlink commands, which also pass a double starname as the value of en:

```
Do you want to 'delete **' in <dir_path>?
Do you want to 'unlink **' in <dir_path>?
```

Entry: dl_handler_$dirdelete

This entry point assumes it is given a directory pathname, and issues the query:

```
<caller>: Do you want to delete the directory dn>en?
```

This entry point is called, for example, by the delete_dir command.

Name: dprint_

This subroutine contains several entry points used to submit requests to the I/O daemon for printing or punching of segments and multisegment files.

Entry: dprint_

The dprint_ entry point adds a request to print, punch, or plot a segment or multisegment file to the specified queue.

USAGE

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

call dprint_ (dir_name, entryname, dprint_arg_ptr, code);
```

ARGUMENTS

dir_name
: is the absolute pathname of the containing directory. (Input)

entryname
: is the entry name of the segment, multisegment file, or link to the segment or multisegment file to be printed, punched, or plotted. (Input)

dprint_arg_ptr
: is a pointer to the dprint_arg structure (described in "Notes" below) that defines the options for this request. If this pointer is null, the default settings are used for all options. (Input)

code
: is a standard status code. (Output)

NOTES

The dprint_ subroutine uses the following structure, defined in the system include file dprint_arg.incl.pl1, to determine the details of the request. If no structure is supplied, default values are used.

```
dcl 1 dprint_arg            based aligned,
      2 version                  fixed bin,
      2 copies                   fixed bin,
      2 delete                   fixed bin,
      2 queue                    fixed bin,
      2 pt_pch                   fixed bin,
      2 notify                   fixed bin,
      2 heading                  char (64),
      2 output_module            fixed bin,
      2 dest                     char (12),
      2 carriage_control,
        3 nep                    bit(1)  unaligned,
        3 single                 bit(1)  unaligned,
        3 non_edited             bit(1)  unaligned,
        3 truncate               bit(1)  unaligned,
        3 center_top_label       bit(1)  unaligned,
        3 center_bottom_label    bit(1)  unaligned,
        3 esc                    bit(1)  unaligned,
        3 no_separator           bit(1)  unaligned,
        3 padding                bit(28)  unaligned,
      2 pad(30)                  fixed bin(35),
      2 forms                    char (8),
      2 lmargin                  fixed bin,
      2 line_lth                 fixed bin,
      2 class                    char (8),
      2 page_lth                 fixed bin,
      2 top_label                char (136),
      2 bottom_label             char (136),
      2 bit_count                fixed bin(35),
      2 form_name                char (24),
      2 destination              char (24),
      2 chan_stop_path           char (168),
      2 request_type             char (24) unaligned,
      2 defer_until_process_termination fixed bin;
```

STRUCTURE ELEMENTS

version
: is the version number of the structure. This is set by the caller and must be the value of the named constant dprint_arg_version_8 also defined in the include file.

copies
: is the number of copies requested.

delete
: indicates whether the file is to be deleted after printing, punching, or plotting.
  1 deletes the file.
  0 does not delete the file.

queue
: is the priority queue in which the request is placed. If zero is supplied, the default queue of the specified type request will be used.

pt_pch
: indicates whether the request is for printing, punching, or plotting.
  1 print request
  2 punch request
  3 plot request

notify
: indicates whether the requestor is to be notified when the request is completed.
  1 notifies the requestor
  0 does not notify the requestor

heading
: is the string to be used as a heading on the front page of the output. If it is a null string, the requestor's Person_id is used.

output_module
: indicates the I/O module to be used in executing the request.
  1 indicates printing
  2 indicates 7-punching
  3 indicates Multics card code (mcc) punching
  4 indicates "raw" punching
  5 indicates plotting

dest
: is not used. See destination below.

nep
: indicates whether no-endpage mode is used.
  "1"b yes
  "0"b no

single
indicates whether single mode, which causes all vertical tabs and new pages to be converted to new lines, is used.
"1"b yes
* "0"b no

non_edited
indicates whether nonedited mode, which causes all nonprinting control characters and non-ASCII characters to be printed as octal escape sequences, is used.
"1"b yes
* "0"b no

truncate
indicates whether truncate mode is used.
"1"b yes
* "0"b no

center_top_label
indicates whether the top label should be centered.
"1"b yes
"0"b no

center_bottom_label
indicates whether the bottom label should be centered.
"1"b yes
"0"b no

esc
indicates whether escape sequences in the print file should be recognized.
"1"b yes
"0"b no

no_separator
indicates when multiple copies of a request are processed, whether the inner head and tail sheets should be printed.
"1"b no
"0"b yes

padding
is not used.

pad
is not used.

forms
is not used. See form_name below.

lmargin
* indicates the left margin position.

line_lth
indicates the line length. If supplied as -1, the maximum line length for the specified request type will be used.

class
is not used. See request_type below.

page_lth
indicates the page length, i.e., the number of lines per logical page. If supplied as -1, the physical page length will be used.

top_label
is a label to be placed at the top of every page. *

bottom_label
is a label to be placed at the bottom of every page. *

bit_count
is the file's bit count.

form_name
is the name of special forms needed.

destination
is the string to be used to indicate where the output should be delivered. If it is null, the requestor's Project_id is used. *

chan_stop_path
is the path of user channel stops.

request_type
is the request type name to be used to queue the request. If printing is requested, the request type must be of the generic type "printer"; if punching is requested, the request type must be of generic type "punch."; if plotting is requested, the request type must be of generic type "plotter". *

defer_until_process_termination
indicates whether the request should be deferred until the requesting process terminates.
1 defers the request.
0 does not defer the request. *

Entry: dprint_$check_daemon_access

This entry point checks the I/O daemon's access to a given segment or multisegment file. It returns whether the daemon responsible for a given request type has "r" access to the file and "s" access to the containing directory and whether the I/O daemon coordinator can delete the file if requested.

USAGE

```
declare dprint_$check_daemon_access entry (char (*), char (*), char (*),
      bit(1) aligned, bit(1) aligned, bit(1) aligned, char (*),
      fixed bin(35));

call dprint_$check_daemon_access (dirname, entryname, request_type,
      delete_permission, read_permission, status_permission,
      driver_userid, code);
```

ARGUMENTS

dirname
  is the absolute pathname of the containing directory. (Input)

entryname
  is the entry name of the segment, or multisegment file, or a link to the segment or multisegment file for which the daemon's access is to be checked. (Input)

request_type
  is the name of the request type in which a request to print, punch or plot the file will be placed. The access of the driver process for this request type will be returned. (Input)

delete_permission
  indicates whether the I/O coordinator has sufficient access to delete the file if requested. The coordinator requires "m" access to the containing directory to delete the file. (Output)

read_permission
  indicates whether the driver process of the given request type has "r" access to the given segment or multisegment file. (Output)

status_permission
  indicates whether the driver process of the given request type has "s" access to the directory containing the segment or multisegment file. (Output)

driver_userid
  is the name of the process that processes requests for the specified type. This value is in the form "Person_id.Project_id.*". (Output)

code
  is a standard system status code. (Output)

NOTES

The user must have "s" access to the directory containing the segment or multisegment file to determine whether the driver has read access to the file.

The user must have "s" access to the directory containing the directory containing the segment or multisegment file in order to determine whether the I/O coordinator can delete the file and whether the driver process has "s" access to the containing directory.

Entry: dprint_$request_id

This entry point adds a request to print, punch, or plot a segment or multisegment file to the specified queue, and returns the message identifier of the queue entry being made.

USAGE

```
declare dprint_$request_id entry (char(*), char(*), ptr, fixed bin(71),
      fixed bin(35));

call dprint_$request_id (dir_name, entryname, arg_ptr, request_id,
      code);
```

ARGUMENTS

dir_name
: is the absolute pathname of the containing directory. (Input)

entryname
: is the entry name of the segment, multisegment file, or link to the segment or multisegment file to be printed, punched, or plotted. (Input)

dprint_arg_ptr
: is a pointer to the dprint_arg structure (described in system include file dprint_arg.incl.pl1) that defines the options for this request. If this pointer is null, the default settings are used for all options.

request_id
: is the message identifier of the request being enqueued. (Output)

code
: is a standard status code. (Output)

NOTES

The dprint_$request_id entry uses the structure defined in the system include file dprint_arg.incl.pl1 to determine the details fo the request. If no structure is supplied, default values are used.

Entry: dprint_$queue_contents

This entry point returns the number of requests in a specific I/O daemon queue.

USAGE

```
declare dprint_$queue_contents entry (char(*), fixed bin, fixed bin,
      fixed bin(35));

call dprint_$queue_contents (request_type, queue, n_requests, code);
```

ARGUMENTS

request_type
: is the name of the request type whose queue is to be checked. (Input)

queue
: is the number of the queue to be examined. If -1 is specified, the default queue of the given request type is checked and the number of the default queue is returned in this parameter. (Input/Output)

n_requests
: is the number of requests in the specified queue. (Output)

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

Name: dump_segment_

This subroutine prints the dump of a segment formatted in the same way as the dump_segment command would print it. The output format is controlled by a bit string that allows most of the formatting control arguments available to dump_segment.

USAGE

```
declare dump_segment_ entry (ptr, ptr, fixed bin, fixed bin(18),
      fixed bin(18), bit(*));

call dump_segment_ (iocb_ptr, first, block_size, offset, count, format);
```

ARGUMENTS

iocb_ptr
: is a pointer to the I/O control block that specifies where the dump is to be written. (Input)

first
is a pointer to the first word of the data to be dumped. (Input)

block_size
is the number of words in the block if blocked output is desired. If unblocked output is desired, this is zero. (Input)

offset
is an arbitrary offset to be printed in addition to the address of the first word of data to be dumped if the offset option in the format string is specified. (It is reset to this initial value at the start of each block.) (Input)

count
is the number of words to dump, starting with the word pointed to by first. (Input)

format
is a format control bit string with the following definition: (See the dump_segment command in the *Multics Commands and Active Functions Manual,* Order No. AG92, for a full discussion of these arguments.) (Input)

This page intentionally left blank.

Entry: dump_segment_$string

This entry point returns the formatted dump of a segment. The ouput format is controlled by a bit string that allows most of the formatting control arguments available to the dump_segment command.

USAGE

```
dcl dump_segment_$string entry (ptr, fixed bin (21), ptr, fixed bin,
      fixed bin (18), fixed bin (18), bit (*));

call dump_segment_$string (string_ptr, string_length, first, block_size,
      offset, count, format)
```

ARGUMENTS

string_ptr
is the pointer to the varying character string to place the output in. (Input)

string_length
is the maximum length of the varying character string. (Input)

first
is the pointer to the first word of data to be dump. (Input)

block_size
is the output dump in blocks of this number of words. (Input)

offset
is an arbitrary offset to be printed in addition to the address of the first word of data to be dumped if the offset option in the format string is specified. (It is reset to this initial value at the start of each block.) (Input)

count
is the number of words to be dump. (Input)

format
is the bit string controlling the output modes. (Input) Described in dump_segment_format.incl.pl1.

NOTES

The following structure is declared in dump_segment_format.incl.pl1.

```
dcl 1 dump_segment_format_structure  based aligned,
      2 address                      bit (1) unaligned,
      2 offset                       bit (1) unaligned,
      2 short                        bit (1) unaligned,
      2 bcd                          bit (1) unaligned,
      2 ascii                        bit (1) unaligned,
      2 long                         bit (1) unaligned,
      2 ebcdic9                      bit (1) unaligned,
      2 ebcdic8                      bit (1) unaligned,
      2 bit4                         bit (1) unaligned,
      2 hex8                         bit (1) unaligned,
      2 hex9                         bit (1) unaligned,
      2 octal                        bit (1) unaligned,
      2 header                       bit (1) unaligned,
      2 raw_data                     bit (1) unaligned,
      2 interpreted_data             bit (1) unaligned,
      2 suppress_duplicates          bit (1) unaligned,
      2 command_output               bit (1) unaligned,
      2 mbz                          bit (19) unaligned;
```

STRUCTURE ELEMENTS

address
    prints the address (relative to the base of the segment) with the data.

offset
    displays the offset of the first word to be dumped.

short
    compacts a line to have four words per line.

bcd
    interprets data as BCD.

ascii
    interprets data as ASCII.

long
    formats a display line to have 8 words per line.

ebcdic9
    interprets data to be EBCDIC (9-bits).

ebcdic8
    interprets data to be EBCDIC (8-bits).

bit4
translates data to be 4-bit data.

hex8
translates data to be with 8 hexadecimal digits per word.

hex9
translates data to be with 9 hexadecimal digits per word.

octal
raw data is octal.

header
displays a header line containing the pathname of the segment being dumped.

raw_data
displays the raw data.

interpreted_data
displays the interpreted data.

suppress_duplicates
replaces multiple duplicate lines with a single line of equal signs.

command_output
if on, the format of the data returned is identical to the format of the dump_segment command. If off, the format of the data returned is in word format (i.e., if the raw data is being returned, the data is returned in the format of words, with each word separated by a blank; if the interpreted data is being returned, the data is returned in the format of a single string, requoted if necessary).

mbz
must be zero.

Name: ebcdic_to_ascii_

The ebcdic_to_ascii_ subroutine performs isomorphic (one-to-one reversible) conversion from EBCDIC to ASCII. The input data is a string of valid EBCDIC characters. A valid EBCDIC character is defined as a 9-bit byte with a hexadecimal value in the range 00 <= hex_value <= FF (octal value in the range 000 <= oct_value <= 377).

This entry point accepts an EBCDIC character string and generates an ASCII character string of equal length.

USAGE

```
declare ebcdic_to_ascii_ entry (char(*), char(*));

call ebcdic_to_ascii_ (ebcdic_in, ascii_out);
```

ARGUMENTS

ebcdic_in
is the string of EBCDIC characters to be converted. (Input)

ascii_out
is the ASCII equivalent of the input string. (Output)

Entry: ebcdic_to_ascii_$ea_table

This entry point defines the 256-character translation table used to perform conversion from EBCDIC to ASCII. Of the 256 valid EBCDIC characters, only 128 have ASCII equivalents. These latter 128 characters are defined in the Isomorphic ASCII/EBCDIC Conversion Table (in the ascii_to_ebcdic_ subroutine description.) For defined characters, the mappings implemented by the ebcdic_to_ascii_ and ascii_to_ebcdic_ subroutines are isomorphic; i.e., each character has a unique mapping, and mappings are reversible. An undefined (but valid) EBCDIC character is mapped into the ASCII SUB (substitute) character, octal 032; the mapping of such a character is anisomorphic. The result of converting an invalid character is undefined.

USAGE

```
declare ebcdic_to_ascii_$ea_table char(256) external static;
```

NOTES

Calling the ebcdic_to_ascii_ subroutine is extremely efficient, since conversion is performed by a single MVT instruction and the procedure runs in the stack frame of its caller.

Name: enter_abs_request_

This subroutine is used to request the creation of an absentee process.

Entry: enter_abs_request_$enter_abs_request_

This entry point adds a request to create an absentee process.

USAGE

```
dcl enter_abs_request_ entry (ptr, ptr, fixed bin (35));

call enter_abs_request_ (abs_request_info_ptr, abs_return_info_ptr,
      code);
```

ARGUMENTS

enter_abs_request_info_ptr
: is a pointer to the abs_request_info structure (described in system include file abs_request_dcls.incl.pl1) that defines the options for this request. (Input)

abs_return_info_ptr
: is a pointer to the abs_return_info structure (described in system include file) that gives information pertaining to the request's status in the queue.

code
: is a standard status code. (Output)

NOTES

The enter_abs_request_subroutine uses the structure defined in abs_request_dcls.incl.pl1.

```
dcl 1 abs_request_info                structure aligned based
                                        (abs_request_info_ptr),
      2 version                       char (8) aligned,
      2 resource_length               fixed bin,
      2 comment_length                fixed bin,
      2 max_arg_length                fixed bin,
      2 arg_count                     fixed bin,
      2 proxy_personid                char (22) aligned,
      2 proxy_projectid               char (9) aligned,
      2 queue                         char (4) aligned,
      2 deferred_time                 fixed bin (71),
      2 max_cpu_time                  fixed bin (35),
      2 requested_authorization       bit (72),
      2 input_segment_dirname         char (168) unaligned,
      2 input_segment_entryname       char (32) unaligned,
      2 output_segment_dirname        char (168) unaligned,
      2 output_segment_entryname      char (32) unaligned,
```

```
        2 attributes                   aligned,
          3 restartable                bit (1) unaligned,
          3 user_deferred_indefinitely bit (1) unaligned,
          3 secondary_ok               bit (1) unaligned,
          3 truncate_absout            bit (1) unaligned,
          3 notify                     bit (1) unaligned,
          3 attributes_mbz             bit (31) unaligned,
        2 resource                     char (arqi_resource_length refer
                                         (abs_request_info.resource_length)),
          2 sender                     char (32),
          2 comment                    char (arqi_comment_length refer
                                         (abs_request_info.comment_length)),
          2 arguments                  dimension (arqi_arg_count refer
                                         (abs_request_info.arg_count))
                                       char (arqi_max_arg_length refer
                                         (abs_request_info.max_arg_length))
                                           varying;

dcl 1 abs_return_info                  structure aligned based
                                         (abs_return_info_ptr),
        2 version                      char (8) aligned,
        2 request_id                   fixed bin (71),
        2 queue                        char (4) aligned,
        2 queue_requests_count         fixed bin (17);

dcl (abs_request_info_ptr, abs_return_info_ptr)
                                       ptr automatic;

dcl (
    ABSENTEE_REQUEST_INFO_VERSION_2    initial ("arqi_002"),
    ABSENTEE_RETURN_INFO_VERSION_2     initial ("arti_002")
    )                                  char (8) internal static options
                                         (constant);

/**** The following fields should be set before abs_request_info is
  allocated */
dcl arqi_resource_length               fixed bin;
dcl arqi_comment_length                fixed bin;
dcl arqi_max_arg_length                fixed bin;
dcl arqi_arg_count                     fixed bin;

dcl (
    BACKGROUND_QUEUE                   dimension (0:4) init ("0", "1",
                                         "2", "3", "4"),
    FOREGROUND_QUEUE                   init ("fg"),
    DEFAULT_QUEUE                      init ("dft")
    )                                  char (4) aligned
                                         internal static options (constant);
```

ARGUMENTS

version
: should be set to the constant ABSENTEE_REQUEST_DCLS_VERSION_2.

resource_length
: is the length of the resource field.

comment_length
: is the length of the comment field.

max_arg_length
: is the maximum length of any element in the arguments array.

arg_count
: is the number of arguments in the arguments array.

proxy_personid
: enters the request on behalf of the specified user. An absentee process of that User_id is logged in to run the job. The system administrator controls the use of -proxy by an access control segment.

queue
: specifies which absentee queue the request should be placed in. It can be the number of the queue, "0", "1", "2", "3", or "4", or "fg" to specify the foreground queue, or "dft" to specify the default queue.

deferred_time
: delays the creation of the absentee process until the specified time.

max_cpu_time
: is the limit on the CPU time used by the absentee process. The parameter N must be a positive decimal integer specifying the limit in seconds. If N equals 0, the default limit, which is defined by the site for each queue, is used.

requested_authorization
: The authorization that the absentee job is requested to be run.

input_segment_dirname
: is the directory containing the absentee control segment.

input_segment_entryname
: is the absentee control segment name.

output_segment_dirname
: is the directory to contain the output segment. If this argument is a null string, then the output of the absentee process is put in the directory containing the absentee control segment, input_segment_dirname.

output_segment_entryname
: is the name of the output segment. If output_segment_entryname is a null string, the output of the absentee process is directed to a segment whose entryname is the same as the input_segment_entryname, except having the suffix absout instead of absin.

restartable
: indicates that the absentee computation should be started over from the beginning if interrupted.

user_deferred_indefinitely
: indicates that the job is deferred until the operator takes action to run it.

secondary_ok
: indicates that a foreground job can be logged in as a secondary user.

truncate_absout
: indicates that the output_segment should be truncated before the absentee job is run.

resource
: is a resource, such as a tape drive, needed by the job. The resource is reserved for the absentee job before it is logged in. If resource is set to the string "", no resource will be reserved.

sender
: is used by the RJE facility to give the name of the RJE station that is requesting the absentee process to be created. If the absentee process is to be that of the user, sender should be set to the string "".

comment
: is a comment which will be associated with the request. The comment is printed whenever the absentee request is listed.

arguments
: is a sequence of arguments to the absentee control segment.

ari_resource_length
: is the maximum length of abs_request_info.resource. This value should be set before allocating the structure.

ari_comment_length
: is the maximum length of abs_request_info. comment. This value should be set before allocating the structure.

ari_arguments_length
: is the maximum length argument in the arguments array. This value should be set before allocating the structure.

ari_arguments_count
: is the maximum number of arguments in the arguments array. This value should be set before allocating the structure.

This page intentionally left blank.

Name: execute_epilogue_

The execute_epilogue_ subroutine is called during process or run unit termination to call the routines in the list of epilogue handlers. The logout and new_proc commands are the prime callers of execute_epilogue_. It is also called when the run unit terminates to allow programs executing in the run unit to clean up. The add_epilogue_handler_ subroutine is used to add a program to the list that execute_epilogue_ calls.

USAGE

```
declare execute_epilogue_ entry (bit (1) aligned);

call execute_epilogue_ (run_only);
```

ARGUMENTS

run_only
    is set to "1"b if epilogue handlers are to be invoked only for the run unit and not for the entire process. (Input)

Name: expand_pathname_

The expand_pathname_ subroutine is used to convert a relative or absolute pathname into a directory name and entryname.

Entry: expand_pathname_

USAGE

```
dcl expand_pathname_ entry (char(*), char(*), char(*), fixed bin (35));

call expand_pathname_ (pathname, dirname, entryname, code);
```

ARGUMENTS

pathname
    is the relative or absolute pathname to be expanded. (Input)

dirname
    is the directory portion of the expanded pathname. (Output)

entryname
    is the entryname portion of the expanded pathname. (Output)

code
is a standard system error code. (Output) It can have one of the following values:
error_table_$lesserr
too many less-than characters ("<") in pathname.
error_table_$badpath
invalid syntax in pathname.
error_table_$pathlong
the expanded pathname is longer than 168 characters.
error_table_$entlong
the entryname of the expanded pathname is longer than 32 characters.
error_table_$no_wdir
a relative pathname is specified, but no working directory is in force for the process.
error_table_$archive_pathname
the input pathname specified an archive component.

NOTES

This entry does not accept the syntax for specifying archive component pathnames; if one is supplied, an error code is returned. See the information on Constructing and interpreting names in the Programmer's Reference Manual for details.

For compatability with older programs, if pathname is given as a null string, the working directory is used.

Entry: expand_pathname_$add_suffix

This entrypoint expands a relative or absolute pathname into a directory name and entryname portion, adding a suffix to the entryname if that suffix is not already present.

USAGE

```
dcl expand_pathname_$add_suffix entry (char (*), char (*), char (*),
      char (*), fixed bin (35));

call expand_pathname_$add_suffix (pathname, suffix, dirname, entryname,
      code);
```

ARGUMENTS

pathname
is the relative or absolute pathname to be expanded. (Input)

suffix
is the suffix to be added to the entryname. (Input) The period separating the entryname and the suffix should not be included. If a null string is supplied, no suffix is added.

dirname
: is the directory portion of the expanded pathname. (Output)

entryname
: is the entryname portion of the expanded pathname. (Output)

code
: is a standard system error code. (Output) It can have the same values described for expand_pathname_.

Entry: expand_pathname_$component

This entrypoint expands a relative or absolute pathname into a directory name, an archive name, and an archive component portion, or into a directory name and entryname portion if no component name is present.

USAGE

```
dcl expand_pathname_$component entry (char (*), char (*), char (*),
      char (*), fixed bin (35));

call expand_pathname_$component (pathname, dirname, entryname,
      componentname, code);
```

ARGUMENTS

pathname
: is the relative or absolute pathname to be expanded. (Input)

dirname
: is the directory name portion of the expanded pathname. (Output)

entryname
: if the input pathname specifies an archive component, this is the entryname of the archive (with the archive suffix added). (Output) Otherwise, this is the entryname portion of the input pathname.

componentname
: if the input pathname specifies an archive component, this is the component name. (Output) Otherwise, this is the null string.

code
: is a standard system error code. (Output) It can have the same values as for expand_pathname_ except for error_table_$archive_pathname.

Entry: expand_pathname_$component_add_suffix

This entrypoint expands a relative or absolute pathname into a directory name, an entryname, and an archive component name. The specified suffix is added either to the entryname or component name, as appropriate, if it is not already present.

USAGE

```
dcl expand_pathname_$component_add_suffix entry (char (*), char (*),
      char (*), char (*), char (*), fixed bin (35));

call expand_pathname_$component (pathname, suffix, dirname, entryname,
      componentname, code);
```

ARGUMENTS

pathname
: is the relative or absolute pathname to be expanded. (Input)

suffix
: is the suffix to be added to the component name or entryname. (Input) The period separating the entryname and the suffix should not be included. If a null string is supplied, no suffix is added.

dirname
: is the directory name portion of the expanded pathname. (Output)

entryname
: if the input pathname specifies an archive component, this is the entryname of the archive (with the archive suffix added). (Output) Otherwise, this is the entryname portion of the input pathname, with the specified suffix added if it is not already present.

componentname
: if the input pathname specifies an archive component, this is the component name, with the specified suffix added if it is not already present. (Output) Otherwise, this is the null string.

code
: is a standard system error code. (Output) It can have the same values as for the expand_pathname_$component entry.

Name: exponent_control_

The exponent_control_ entry points provide control over the behavior of the system in the event of a computational overflow or underflow. The normal behavior of the system in both cases is to signal a fault condition. (See the Programmer's Reference Manual for more information on conditions and other unusual events). These entry points provide the option of transparently restarting these faults with a known result: zero in the case of an underflow; a user-settable value in the case of an overflow. By default, this value is the largest representable floating point number.

This subroutine affects the system's handling of exponent overflow or underflow only when the overflow or underflow condition is raised. In certain cases, the error condition is raised instead. This subroutine does not affect the system's handling of these cases.

Entries: exponent_control_$fault_overflow, exponent_control_$fault_underflow

These entrypoints instruct the system to signal fault conditions when computations overflow or underflow.

USAGE

```
dcl exponent_control_$fault_overflow entry (fixed bin (35));
dcl exponent_control_$fault_underflow entry (fixed bin (35));

call exponent_control_$fault_overflow (code);
call exponent_control_$fault_underflow (code);
```

ARGUMENTS

code
    is a standard system status code. (Output)

Entries: exponent_control_$restart_overflow,
exponent_control_$restart_underflow

These entrypoints instruct the system to automatically restart overflow and underflow conditions, respectively. In the overflow case, the default value for the result of the computation is used for positive overflows. If the overflow is in a negative direction, the negative of the default value is used.

USAGE

```
dcl exponent_control_$restart_overflow entry (fixed bin (35));
dcl exponent_control_$restart_underflow entry (fixed bin (35));

call exponent_control_$restart_overflow (code);
call exponent_control_$restart_underflow (code);
```

ARGUMENTS

code
is a standard system status code. (Output)

Entry: exponent_control_$restart_overflow_value

This entry point instructs the system to automatically restart overflow conditions, and specifies a value to be returned as the computational result.

USAGE

```
dcl exponent_control_$restart_overflow_value entry (float bin (63),
     fixed bin (35));

call exponent_control_$restart_overflow_value (amax_value, code);
```

ARGUMENTS

amax_value
is the value to be supplied for the result of computations that result in overflows. (Input)

code
is a standard system status code. (Output)

Name: file_manager_

The file_manager_ subroutine is the interface between the data storage and retrieval services of data management and Multics file access and control mechanisms. It also ensures concurrency and recovery protection by invoking data management integrity services when protected data management (DM) files are accessed or modified.

As a direct user interface, the file_manager_ subroutine makes the protection and recovery capabilities of integrity services available to users who write applications using their own data storage and retrieval software.

See the section entitled "Multics Data Management" in the *Multics Programmer's Reference Manual,* Order No. AG91, for a complete description of data storage and retrieval services, integrity services, and DM files.

Entry: file_manager_$close

This entry point closes a DM file in the current process. The file to be closed is designated by its opening identifier.

USAGE

```
declare file_manager_$close entry (bit(36) aligned, fixed bin(35));

call file_manager_$close (oid, code);
```

ARGUMENTS

oid
: is the file opening identifier of the file to be closed. (Input/Output) It is set to zero by this entry point in order to indicate that it is no longer valid.

code
: is a standard status code. (Output)

NOTES

If the file is opened more than once in the process, this operation decreases the number of openings by one. See the description of the open entry for more details.

The user process does not have to be in transaction mode to close a DM file. Aborting a transaction does not cause the file to be reopened, even if it was closed by a delete_close operation and the deletion was rolled back.

Entry: file_manager_$create

This entry point creates a DM file. The caller specifies its pathname and attributes, and must specify whether the file is protected; see "Notes" below.

USAGE

```
declare file_manager_$create (char(*), char(*), ptr, fixed bin(35));

call file_manager_$create (dir_path, entry_name, file_create_info_ptr,
     code);
```

ARGUMENTS

dir_path
: is the absolute pathname of a directory. (Input) The file will be added to this directory.

entry_name
: is the name of the file. (Input)

file_create_info_ptr
points to the file_create_info structure (see below). (Input) If this pointer is null, a protected file is created with the default attribute values.

code
is a standard status code. (Output)

ACCESS REQUIRED

You cannot create a DM file in your home directory unless you have the proper access to the directory above the containing directory, and, in any event, you must have sufficient access to add an entry to the containing directory. The file is created with read and write access for the caller and the DM daemon (Data_Management.Daemon).

NOTES

In the current implementation, file attributes specified in the file_create_info structure such as ring brackets and blocking factor cannot be changed.

STRUCTURE

Following is the structure used to describe the attributes to assign to a file being created. It is declared in dm_file_create_info.incl.pl1.

```
dcl 1 file_create_info   aligned based (file_create_info_ptr),
      2 version           char (8) aligned,
      2 ci_size_in_bytes  fixed bin (35),
      2 blocking_factor   fixed bin,
      2 flags             unal,
        3 protected       bit (1) unal,
        3 no_concurrency  bit (1) unal,
        3 no_rollback     bit (1) unal,
        3 mbz_1           bit (15) unal,
      2 ring_brackets     (2) fixed bin (3) unal,
      2 mbz_3             bit (10) unal,
      2 mbz_2             (30) fixed bin (71);
```

STRUCTURE ELEMENTS

version
must be set by the caller to FILE_CREATE_INFO_VERSION_2. This permits upward compatible changes to this structure.

ci_size_in_bytes
is the control interval size. Currently the only size available is 4096. If this item is zero, a default value is used. Currently the default value is 4096.

blocking_factor
tells the file manager how to allocate disk storage. In the current implementation this is interpreted as the number of control intervals to put in each segment, and only 64 and 255 are allowed. If this item is zero, a default value is used. The current default value is 255.

flags.protected
determines whether the file is protected from transaction failure and from concurrent access by other processes. If the protected bit is on, get, put, and allocate operations are permitted only in transaction mode. Create and delete operations are protected regardless of whether the process is in a transaction. If this bit is off, the file is unprotected, which means that the file and its contents may be damaged by concurrent access or transaction failure. An unprotected file may be accessed within or without a transaction. Accessing a protected file is substantially more expensive than accessing an unprotected file. The default is to provide protection.

flags.no_concurrency
turns off protection against concurrent access by other processes. Concurrency protection is implemented by locking each control interval that is accessed. The get operation locks in share mode. All other operations lock in exclusive mode. Create and delete lock the entire file in exclusive mode. Locking is expensive. This bit turns it off if it is not needed. If protection is off, this bit is ignored. The default is to provide concurrent access protection.

flags.no_rollback
turns off protection against transaction failure. Protection against transaction failure is implemented by journaling a before image of each modification. When a transaction fails, its modifications are undone by restoring these before images. Journaling is expensive. This bit turns it off if it is not needed. If protection is off, this bit is ignored. The default is to provide protection from transaction failure.

ring_brackets
are the extended ring brackets of the file. They specify the range of rings from which the file may be accessed. The first is the write bracket. It's value cannot be less than the data management ring (loosely, the ring of execution of the file_manager_) or the validation level of the creating process. The second is the read bracket. It's value cannot be less than the write bracket. The default for both ring brackets is the validation level of the calling process.

mbz_1, mbz_2, mbz_3
must be initialized to zero by the caller. This is so that upward compatible changes will be able to assume that existing programs put zeros in these areas.

Entry: file_manager_$create_open

Calling this entry point has the effect of calling the create and open entries, but is more efficient. If the file already exists, it is opened and code is dm_error_$file_already_exists. If the file already exists and is already open, the opening identifier is returned and code is dm_error_$file_already_open.

USAGE

```
declare file_manager_$create_open entry (char(*), char(*), ptr, bit(36)
      aligned, fixed bin(35));

call file_manager_$create_open (dir_path, entry_name,
      file_create_info_ptr, oid, code);
```

ARGUMENTS

dir_path
: is the absolute pathname of a directory. (Input) The file is added to this directory.

entry_name
: is the name of the file. (Input)

file_create_info_ptr
: is a pointer to a file_create_info structure into which the file attributes are to be placed. (Input) This structure may in turn be used to create a new DM file into which to copy the old DM file. The structure is defined in dm_fm_create_info.incl.pl1. (See the create entry for a description of this include file).

oid
: is the opening identifier assigned to the file. (Output) If it is not zero, the oid is valid and can be used, regardless of the value of code. If the transaction aborts and the file is deleted, it still needs to be closed, since openings are not undone by rollback.

code
: is a standard status code. (Output) If it is dm_error_$file_already_exists or dm_error_$file_already_open, the operation is considered successful and oid is usable.

Entry: file_manager_$delete_close

Calling this entry point has the same effect as calling the delete and close entries, but is more efficient. It deletes a file that is already open.

USAGE

```
declare file_manager_$delete_close entry (bit(36) aligned, fixed
      bin(35));

call file_manager_$delete_close (oid, code);
```

ARGUMENTS

oid
: is the file opening identifier of the file to be deleted and closed. (Input/Output) It is set to zero by this entry point in order to indicate that it is no longer valid. The file remains closed even if the transaction is aborted, negating the delete operation.

code
: is a standard status code. (Output)

Entry: file_manager_$free

This entry point frees disk space allocated to control intervals. The set of consecutive control intervals is specified by the number of the first control interval and the number of consecutive control intervals starting at the first one. After the disk space for a control interval has been freed, its content is effectively zero. This operation has a high fixed overhead, so it should not be called for one control interval at a time.

If any or all of the control intervals are already free, code is set to dm_error_$ci_already_free. The operation is, nevertheless, successful.

USAGE

```
declare file_manager_$free entry (bit(36) aligned, fixed bin(27), fixed
      bin(27), fixed bin(35));

call file_manager_$free (oid, first_ci, n_ci, code);
```

ARGUMENTS

oid
: is a file opening identifier. (Input)

first_ci
: is the control interval number of the first control interval of the set whose physical space is to be freed. (Input)

n_ci
: is the number of consecutive control intervals whose physical space is to be freed. (Input)

code
: is a standard status code. (Output) If it is dm_error_$ci_already_free, the operation can still be considered a success.

ACCESS REQUIRED

The user must have write access to the file.

NOTES

If the file is protected, the caller must be in transaction mode and the free operation is done under the auspices of the integrity services. If the transaction aborts, the control intervals are reallocated and their contents restored.

Entry: file_manager_$get

This entry point reads data from a control interval. The caller may specify one or several parts. Each part is described by its byte offset relative to the beginning of the addressable portion of the control interval and its length in bytes. Each part has a pointer to a buffer provided by the caller.

If the control interval does not exist, the buffers provided by the caller are filled with zeros and code is set to zero.

USAGE

```
declare file_manager_$get entry (bit(36) aligned, fixed bin(27), ptr,
      fixed bin(35));

call file_manager_$get (oid, ci_num, ci_parts_ptr, code);
```

ARGUMENTS

oid
: is a file opening identifier. (Input)

ci_num
: is a control interval number. (Input)

ci_parts_ptr
: points to the ci_parts structure declared in dm_ci_parts.incl.pl1. (Input) (See below)

code
: is a standard status code. (Output)

NOTES

If the file is protected, the process must be in transaction mode, and unless no_concurrency is specified, get locks the control interval in share mode. It is kept locked until the end of the transaction. This assures that no other transaction can put anything into the control interval, free it, or delete the file during the current transaction. If the control interval is locked in exclusive mode by another transaction, get waits until it finishes. If waiting is pointless because the current transaction is deadlocked with another transaction, the transaction_deadlock condition is signaled.

ACCESS REQUIRED

The calling process must have read access to call the get entry.

STRUCTURE

The ci_parts structure defines the parts of a control interval and is declared in dm_ci_parts.incl.pl1.

```
dcl  1 ci_parts                 aligned based (ci_parts_ptr),
       2 number_of_parts        fixed bin,
       2 part                   (cip_number_of_parts
                                 refer (ci_parts.number_of_parts)),
         3 offset_in_bytes      fixed bin,
         3 length_in_bytes      fixed bin,
         3 local_ptr            ptr;
```

STRUCTURE ELEMENTS

number_of_parts
: is the number of parts. Zero is legal and there is currently no limit on the number of parts.

offset_in_bytes
is the offset of the part within the addressable portion of the control interval. It is the zero relative index of the first byte of the part. It is the number of bytes that are to be skipped, starting at the beginning of the addressable portion.

The addressable portion of a control interval begins with the first byte after the four word header and ends with the last byte before the two word trailer. The only exception is control interval zero. It has a smaller addressable portion, because the file attributes are stored in it.

The length of the addressable portion is 4072 bytes. Control interval zero has 3176 bytes. Constants for these values are declared in dm_ci_lengths.incl.pl1.

length_in_bytes
is the number of bytes in the part. If it is zero, the part is ignored.

local_ptr
is a pointer to the buffer provided by the caller for the part.

Entry: file_manager_$get_ci_header

This entry reads the 4-word control interval header (the structure appears below). The header tells whether a control interval is allocated when it was last modified, and what the unique identifier of the DM file is.

Protection and access are the same as for the get entry.

If the control interval does not exist, this entry returns the header that it would have had, with zero in the time_modified field and code set to zero. It does not create the control interval.

USAGE

```
declare file_manager_$get_ci_header entry (bit(36) aligned, fixed
      bin(27), 1 like ci_header aligned, fixed bin(35));

call file_manager_$get_ci_header (oid, ci_num, ci_header, code);
```

ARGUMENTS

oid
is a file opening identifier. (Input)

ci_num
is a control interval number. (Input)

ci_header
is the ci_header structure, declared in dm_ci_header.incl.pl1. (Input/Output) (See below)

code
is a standard status code. (Output)

STRUCTURE

The ci_header structure is in dm_ci_header.incl.pl1.

```
dcl 1 ci_header          aligned based (ci_header_ptr),
      2 stamp,
        3 version        bit (9) unal,
        3 bj_idx         fixed bin (9) uns unal,
        3 time_modified  fixed bin (53) unal,
      2 id,
        3 uid            bit (36) unal,
        3 size_code
          4 exponent     fixed bin (6) uns,
          4 addon        fixed bin (3) uns,
        3 num            fixed bin (27) uns unal;
```

STRUCTURE ELEMENTS

version
is the version of the structure, currently CI_HEADER_STAMP_VERSION_1.

bj_idx
is used to synchronize control interval writes with before journal writes.

time_modified
is the Multics clock time when this control interval was last modified. If the control interval does not exist, this item is zero.

uid
is the unique identifier of the data management file. It is not the Multics file system uid.

size_code
gives the physical size of the control interval which includes the header and trailer. The size in bytes = (64 + 8 * addon) * 2**exponent.

num
is the control interval number. The number of the first control interval is zero.

Entry: file_manager_$get_ci_ptr

This entry point returns a pointer to the addressable portion of a control interval. Pointers to control intervals should be used only in well defined and contained situations to enhance the performance of accessing data in a control interval for retrieval purposes. This entry is helpful when it is known beforehand that several pieces of data are to be read from the same control interval, but they cannot be read by specifying several parts to the get entry (e.g., the offset of one is dependent on the value of another). Unlike other entries which get data, it is not valid to get a control interval pointer to a control interval that is not allocated.

USAGE

```
declare file_manager_$get_ci_ptr entry (bit(36) aligned, fixed bin(27),
      ptr, fixed bin(35));

call file_manager_$get_ci_ptr (oid, ci_num, ci_ptr, code);
```

ARGUMENTS

oid
: is a file opening identifier. (Input)

ci_num
: is a control interval number. (Input)

ci_ptr
: points to the addressable portion of the control interval. (Input/Output) The addressable portion begins immediately after the control interval header. A null value is returned for this pointer if there is a error and the returned code is not zero.

code
: is a standard status code. (Output) It can be dm_error_$ci_not_allocated if the specified control interval has not been allocated. ci_ptr is set to null.

NOTES

In order to make it possible to look at control intervals via a pointer, the create entry sets the ring brackets on file components to:

```
<DM-ring>,<validation-level>,<validation-level>.
```

If the file is protected, the process must be in transaction mode, and unless no_concurrency is specified, get_ci_ptr locks the control interval in share mode. It is kept locked until the end of the transaction. This assures that no other transaction can put anything into the control interval, free it, or delete the file during the current transaction. If the control interval is locked in exclusive mode by another transaction, get_ci_ptr waits until it finishes. If waiting is pointless because the current transaction is deadlocked with another transaction, the transaction_deadlock condition is signaled.

If the control interval does not exist, an error code of value dm_error_$ci_not_allocated is returned.

ACCESS REQUIRED

The calling process must have read access to call the get_ci_ptr entry.

Entry: file_manager_$get_exclusive

This entry point is the same as get entry except that it locks the control interval exclusively, preventing other transactions from even obtaining the share lock necessary to do a normal get operation.

USAGE

```
declare file_manager_$get_exclusive entry (bit(36) aligned, fixed
      bin(27), ptr, fixed bin(35));

call file_manager_$get_exclusive (oid, ci_num, ci_parts_ptr, code);
```

ARGUMENTS

oid
: is a file opening identifier. (Input)

ci_num
: is a control interval number. (Input)

ci_parts_ptr
: points to the ci_parts structure declared in dm_ci_parts.incl.pl1. (Input) See the get entry.

code
is a standard status code. (Output)

NOTES

This entry is useful for applications that are going to do a put operation into the same control interval.

Obtaining an exclusive lock on a control interval effectively reduces concurrency, so this entry should be used advisedly.

Entry: file_manager_$get_stream

This entry point returns a specified number of bytes from a DM file, given an opening identifier, a file offset, and a buffer in which to place the bytes. This entry treats the DM file as a stream of bytes consisting of the concatenation of the addressable portion of all control intervals in the DM file.

USAGE

```
declare file_manager_$get_stream entry (bit(36) aligned, fixed bin(48),
      ptr, fixed bin(21));

call file_manager_$get_stream (oid, file_offset_in_bytes, buf_ptr,
      buf_length_in_bytes);
```

ARGUMENTS

oid
is the opening identifier of the DM file to be read from. (Input)

file_offset_in_bytes
is the offset given in bytes, from the beginning of the logical address space of the DM file given in bytes with an offset of zero representing the beginning of the file. (Input)

buf_ptr
is a pointer to a buffer where the bytes read from the DM file may be placed. (Input)

buf_length_in_bytes
is the number of bytes that are to be read from the DM file. (Input)

NOTES

If the DM file is protected, the process must be in transaction mode and unless concurrency is specified, get_stream locks in share mode the control intervals in which the specified stream of bytes resides.

ACCESS REQUIRED

The calling process must have read access to the DM file to call the get_stream entry.

Entry: file_manager_$lock_advice

This entry point permits applications to give the file manager advice about locking granularity. For example, if an application is to modify every control interval in a file, it can request the file manager to lock the entire file and save the overhead of locking individual control intervals.

USAGE

```
declare file_manager_$lock_advice entry (bit(36) aligned, fixed bin,
      fixed bin(35));

call file_manager_$lock_advice (oid, lock_mode, code);
```

ARGUMENTS

oid
is a file opening identifier. (Input)

lock_mode
is the finest and weakest lock mode to use on this file for the remainder of the opening. (Input) It must be one of the five following modes: 4 (LOCK_MODE_IS), 5 (LOCK_MODE_IX), 6 (LOCK_MODE_SIX), 2 (LOCK_MODE_S), or 3 (LOCK_MODE_X) which are declared in dm_lock_modes.incl.pl1.

LOCK MODE NAMES

Named constants for the lock modes are provided in the include file dm_lock_modes.incl.pl1

```
dcl  LOCK_MODE_S          fixed bin static options (constant) init (2);
dcl  LOCK_MODE_X          fixed bin static options (constant) init (3);
dcl  LOCK_MODE_IS         fixed bin static options (constant) init (4);
dcl  LOCK_MODE_IX         fixed bin static options (constant) init (5);
dcl  LOCK_MODE_SIX        fixed bin static options (constant) init (6);

dcl  LOCK_ENTIRE_FILE     fixed bin (24) static options (constant) init (-1);

dcl  LOCK_MODE_NAMES (2:6)  char (3) int static options (constant)
                            init ("S", "X", "IS", "IX", "SIX");
```

```
     S      Share
            Let others read it but not modify it.

     X      Exclusive
            Let nobody else read or modify it.

     IS     Intend Share
            I am only using S locks, because I am only reading CIs.

     IX     Intend Exclusive
            I am using S and X locks, because I am reading and modifying CIs.

     SIX    Share with Intend Exclusive
            I am reading control intervals, but only locking the ones I modify.
```

code
    is a standard status code. (Output)

NOTES

Lock advice never abridges protection against concurrent file access by other processes. If no lock advice is given, file manager uses the weakest lock necessary to provide concurrency protection, and the finest granularity available, which is the control interval. Lock advice always causes file manager to use a stronger lock or coarser granularity than absolutely necessary. This reduces concurrency in order to reduce locking overhead.

Lock advice applies to protected files unless the no_concurrency attribute is present. Since it is an attribute of the opening and not of the file, it can be given to any open file, regardless of whether a transaction is in progress. The first time the file is referenced in each transaction, the advice tells the file manager what kind of a global file lock to acquire. If the lock advice is given after the first time the file is referenced, it will not be used until the next transaction. The lock advice is retained until it is changed, or the file is closed.

The advice concerns the type of lock to use at the file level. The only way to control the type of lock used on a control interval is to call get_exclusive instead of get. If no advice is given, IS (intention shared) is presumed. IS is strong enough for get and get_ci_header which lock control intervals in S (share) mode. Put, allocate, and free require that the file lock be upgraded to the stronger IX (intention exclusive) mode, because they lock control intervals in X (exclusive) mode. Create and delete lock the file in X mode, regardless. If advice is given, then all operations lock the file in the advised mode unless it is too weak for the operation. The SIX (shared and intention exclusive) mode means only lock the control intervals that are modified. SIX saves the overhead of locking individual control intervals for get operations because it prevents other transactions from getting anything but an IS lock on the file.

Entry: file_manager_$open

This entry point makes a DM file accessible within a process. The file is specified by its pathname. The file is assigned an opening identifier in the current process, by which it is designated in all subsequent calls to file_manager_.

USAGE

```
declare file_manager_$open entry (char(*), char(*), bit(36) aligned,
     fixed bin(35));

call file_manager_$open (dir_path, entry_name, oid, code);
```

ARGUMENTS

dir_path
: is the absolute pathname of the directory which contains the file. (Input)

entry_name
: is the entry name of the file. (Input)

oid
: is the file opening identifier assigned to the file and returned to the caller. (Output) If it is not zero, it is usable, regardless of code.

code
: is a standard status code. (Output) If it is dm_error_$file_already_open, the operation is considered successful and oid is usable.

NOTES

If the file was already opened in the current process, the open entry does not assign a new opening identifier, but rather returns the opening identifier that was already assigned and sets code to dm_error_$file_already_open. The file manager keeps track of the number of opens and closes. The opening identifier remains valid as long as there are more opens than closes. If all subsystems within a process close a file the same number of times they open it, they will not invalidate each others openings.

There is no requirement for the process to be in transaction mode when opening a file, protected or not. Aborting a transaction has no effect on file openings, even if create_open was called and the create is rolled back. Attempts to use such an opening will result in dm_error_$file_doesnt_exist. The same thing happens if a file is opened and then deleted. Close is the only operation allowed on a file which has been deleted.

Entry: file_manager_$put

This entry point writes data into a control interval. The caller can specify one or several parts of the control interval to be written.

If the control interval does not exist, it is automatically allocated and the content of its addressable portion is initialized to zero.

USAGE

```
declare file_manager_$put entry (bit(36) aligned, fixed bin(27), ptr,
     fixed bin(35));

call file_manager$put (oid, ci_num, ci_parts_ptr, code);
```

ARGUMENTS

oid
: is a file opening identifier. (Input)

ci_num
: is a control interval number. (Input)

ci_parts_ptr
: points to the ci_parts structure declared in dm_ci_parts.incl.pl1. (Input) (See the get entry.)

code
: is a standard status code. (Output)

NOTES

If the file is protected, the process must be in transaction mode, and unless no_concurrency is specified, put locks the control interval in exclusive mode. It is kept locked until the end of the transaction. This assures that no other transaction can put anything into the control interval, get anything from it, free it, or delete the file during the current transaction. If the control interval is locked by another transaction, the put operation must wait until it finishes. If waiting is pointless because the current transaction is deadlocked with another transaction, the transaction_deadlock condition is signaled.

Unless the file is unprotected or has the no_rollback attribute, a put operation causes a before image of data in the control interval to be journalized before actually modifying it. If the transaction should abort, the before journal manager will undo its modifications by restoring the before images.

The modified control interval can not be written to disk until its before image is on disk, because there must be enough information on disk to roll back the transaction even if main memory fails. If the modified control interval were written first and the system failed before the transaction finished and the content of main memory could not be flushed to disk, the modification could not be undone and rollback of the transaction would be incomplete. The data management system holds modified control intervals in main memory until the associated before images are written to disk. The Multics clock value in the control interval header is used for this purpose.

The put request is rejected if either of the following is true:

- The user does not have write permission on the file.
- The file is protected but the process is not in a transaction mode.

Entry: file_manager_$put_stream

This entry point writes a specified number of bytes in a DM file at a given offset in the logical address space. This entry treats the DM file as a stream of bytes made up of the concatenation of the addressable portion of all control intervals in the DM file.

USAGE

```
declare file_manager_$put_stream entry (bit(36) aligned, fixed bin(48),
     ptr, fixed bin(21), fixed bin(35));

call file_manager_$put (oid, file_offset_in_bytes, buf_ptr,
     buf_length_in_bytes, code);
```

ARGUMENTS

oid
is the opening identifier of the DM file. (Input)

file_offset_in_bytes
is the offset in bytes into the logical address space of the DM file where the supplied bytes will be placed. (Input)

buf_ptr
is a pointer to the buffer containing the bytes to be written to the DM file. (Input)

buf_length_in_bytes
is the number of bytes to be written into the DM file from the buffer. (Input)

code
is a standard system status code. (Input)

NOTES

If the DM file is protected, the process must be in transaction mode, and unless concurrency is specified, put_stream locks the control intervals in which the specified stream of bytes resides.

ACCESS REQUIRED

The calling process must have write access to the DM file to call the put_stream entry.

Entry: file_manager_$raw_get

This entry point resembles the get entry, except that it treats the file as if it were unprotected. It does not require that the process be in transaction mode.

USAGE

```
declare file_manager_$raw_get entry (bit(36) aligned, fixed bin(27),
      ptr, fixed bin(35));

call file_manager_$raw_get (oid, ci_num, ci_parts_ptr, code);
```

ARGUMENTS

oid
is a file opening identifier. (Input)

ci_num
    is a control interval number. (Input)

ci_parts_ptr
    points to a ci_parts structure declared in dm_ci_parts.incl.pl1. (Input) (See the get entry.)

code
    is a standard status code. (Output)

Entry: file_manager_$raw_put

This entry point resembles the put entry, except that it treats the file as if it were unprotected. Also, the time_modified stamp in the control interval header is not updated. This operation is intended for applications that need to update protected files in an unprotected manner. It does not require that the process be in transaction mode.

USAGE

```
declare file_manager_$raw_put entry (bit(36) aligned, fixed bin(27),
      ptr, fixed bin(35));

call file_manager_$raw_put (oid, ci_num, ci_parts_ptr, code);
```

ARGUMENTS

oid
    is a file opening identifier. (Input)

ci_num
    is a control interval number. (Input)

ci_parts_ptr
    points to a ci_parts structure declared in dm_ci_parts.incl.pl1. (Input) (See the get entry.)

code
    is a standard status code. (Output)

Entry: file_manager_$simple_get

This entry point is used to get a sequence of bytes from a DM file, given an opening identifier, a control interval number, and control interval offset. The sequence is placed in a caller-supplied buffer. This entry point differs from the get entry in that it can only get bytes from one location within a control interval, and a ci_parts structure does not have to exist to make the call.

USAGE

```
declare file_manager_$simple_get entry (bit(36) aligned, fixed bin(27),
      fixed bin(21), ptr, fixed bin(21));

call file_manager_$simple_get (oid, ci_num, ci_offset_in_bytes, buf_ptr,
      buf_length_in_bytes);
```

ARGUMENTS

oid
: is the opening identifier of the DM file. (Input)

ci_num
: is the control interval in the DM file that contains the data to be fetched. (Input)

ci_offset_in_bytes
: is the offset from the beginning of the control interval to the beginning of the data, expressed in bytes. (Input)

buf_ptr
: is a pointer to a caller supplied buffer where the data is to be placed. (Input)

buf_length_in_bytes
: is the length in bytes of the caller supplied buffer. (Input) This also specifies the number of bytes to be fetched. The sum of ci_offset_in_bytes and buf_length_in_bytes must not exceed the length of a control interval.

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

Entry: file_manager_$simple_put

This entry point places a given sequence of bytes into a DM file, given an open id, a control interval number, and a control interval offset. This entry differs from the put entry point in that it places bytes only at one given location within a control interval, so no ci_parts structure is required.

USAGE

```
declare file_manager_$simple_put entry (bit(36) aligned, fixed bin(27),
      fixed bin(21), ptr, fixed bin(21));

call file_manager_$simple_put (oid, ci_num, ci_offset_in_bytes, buf_ptr,
      buf_length_in_bytes);
```

ARGUMENTS

oid
: is the opening identifier of the DM file. (Input)

ci_num
: is the control interval in the DM file where the data is to be placed. (Input)

ci_offset_in_bytes
: is the offset from the beginning of the control interval to the beginning of where the data is to be placed. (Input)

buf_ptr
: is a pointer to the buffer containing the data. (Input)

buf_length_in_bytes
: is the number of bytes that are to be placed into the DM file. (Input)

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

Entry: file_manager_$status

This entry point returns status information on a DM file.

USAGE

```
declare file_manager_$status entry (bit(36), ptr, fixed bin(35));

call file_manager_$status (oid, file_status_ptr, code);
```

ARGUMENTS

oid
is the opening identifier of the DM file. (Input)

file_status_ptr
is a pointer to a file_status structure to be filled in by this entry. (Input) See the dm_file_status structure described below.

code
is a standard system status code. (Output)

STRUCTURE

This structure lists the information returned by file_manager_$status to describe a DM file. It resides in the include file dm_file_status.incl.pl1.

```
dcl 1 dm_file_status aligned based (dm_file_status_ptr),
      2 version char (8) unaligned,
      2 fm_unique_id bit (36) aligned,
      2 mode bit (36) aligned,
      2 date_time_created fixed bin (71),
      2 ring_brackets (2) fixed bin (3),
      2 switches,
       3 (protected_sw,
          no_concurrency_sw,
          no_rollback_sw) bit (1) unaligned,
       3 pad1 bit (33) unaligned,
      2 highest_ci fixed bin (18),
      2 ci_size fixed bin (18),
      2 pad (5) fixed bin;
```

STRUCTURE ELEMENTS

version
is the current version of the structure, DM_FILE_STATUS_VERSION_1.

fm_unique_id
is the file manager unique identifier (fmuid) of the file, which uniquely identifies it to data management.

mode
is the user's effective access to the file, taking into consideration the extended access (access available via data management operations) and AIM.

date_time_created
is the date-time the file was created.

ring_brackets
are the extended ring brackets of the file as implemented by data Management.

protected_sw
    if ON, data management transactions are required in order to reference the file's data.

no_concurrency_sw
    if ON, only one process can reference the file at a time.

no_rollback_sw
    if ON, the rollback operation is not allowed.

highest_ci
    is the sequential number of the highest control interval allocated in the file.

ci_size
    is the number of bytes per control interval.

Entry: file_manager_$terminate_ci_ptr

This entry point releases a control interval pointer to a specific control interval of a DM file retrieved through the get_ci_ptr entry point. This entry must be called for each call to get_ci_ptr.

USAGE

```
declare file_manager_$terminate_ci_ptr entry (bit(36) aligned,
     fixed bin(27), ptr, fixed bin(35));

call file_manager_$terminate_ci_ptr (oid, ci_num, ci_ptr, code);
```

ARGUMENTS

oid
    is the opening identifier of the DM file. (Input)

ci_num
    is the number of the control interval that ci_ptr points to. (Input)

ci_ptr
    is the control interval pointer to be terminated. (Input)

code
    is a standard system status code. (Output)

Name: find_bit_

This subroutine uses the EIS test character and translate (TCT) instruction to efficiently perform common bit string search operations. Entrypoints are provided to return the bit index of the first or last occurrence of an on bit ("1"b) or off bit ("0"b) in a bit string.

This subroutine operates by dividing the bit string into three search regions: a group of 9-bit bytes aligned on a byte boundary; bits preceding these bytes; and bits following the bytes. Bits preceding or following the bytes are examined bit by bit, using a separate compare bit (CMPB) instruction for each bit. The bytes are examined as a single character string, using one TCT instruction to test all bytes until a byte containing an on or off bit is found. For bit strings longer than 36 bits, this subroutine is significantly faster than the code generated by the PL/I index builtin function, which test all bits on a bit-by-bit basis..

Entry: find_bit_$first_on

This entrypoint returns the index (bit position) of the first (leftmost) bit that is on ("1"b) in a bit string.

USAGE

```
declare find_bit_$first_on entry (bit(*)) returns (fixed bin(24))
      reducible;

index = find_bit_$first_on (bit_string);
```

ARGUMENTS

bit_string
    is the bit string to be examined. (Input)

index
    is the index of the first "1"b bit within the bit string. If no "1"b bits are found, then 0 is returned. (Output)

Entry: find_bit_$first_off

This entrypoint returns the index (bit position) of the first (leftmost) bit that is off ("0"b) in a bit string.

USAGE

```
declare find_bit_$first_off entry (bit(*)) returns (fixed bin(24))
      reducible;

index = find_bit_$first_off (bit_string);
```

ARGUMENTS

bit_string
is the bit string to be examined. (Input)

index
is the index of the first "0"b bit within the bit string. If no "0"b bits are found, then 0 is returned. (Output)

Entry: find_bit_$last_on

This entrypoint returns the index (bit position) of the last (rightmost) bit that is on ("1"b) in a bit string.

USAGE

```
declare find_bit_$last_on entry (bit(*)) returns (fixed bin(24))
     reducible;

index = find_bit_$last_on (bit_string);
```

ARGUMENTS

bit_string
is the bit string to be examined. (Input)

index
is the index of the last "1"b bit within the bit string. If no "1"b bits are found, then 0 is returned. (Output)

Entry: find_bit_$last_off

This entrypoint returns the index (bit position) of the last (rightmost) bit that is off ("0"b) in a bit string.

USAGE

```
declare find_bit_$last_off entry (bit(*)) returns (fixed bin(24))
     reducible;

index = find_bit_$last_off (bit_string);
```

ARGUMENTS

bit_string
is the bit string to be examined. (Input)

index
: is the index of the last "0"b bit within the bit string. If no "0"b bits are found, then 0 is returned. (Output)

Name: find_char_

This subroutine uses the EIS test character and translate (TCT) instruction to perform the function of the PL/I search and verify builtin functions in a highly efficient manner. Search and verify operations can be performed from either the left or the right end of the string.

The search function looks for the first occurrence of any of a set of characters (the search characters) within an input character string. The verify function checks that all characters within an input string are also characters in a verify string; it searches in the input string for the first occurrence of a character not in the verify string.

NOTES

The TCT instruction uses a test-and-translate table to control the searching. Entrypoints are provided to build a table that can be used for several search or verify operations, or to build tables as part of each search or verify operation.

The PL/I compiler generates efficient, in-line TCT or TCTR instructions when the second argument of the search or verify builtin function is a constant (so that the test-and-translate table can be constructed at compile time). When the second argument is a variable, however, PL/I uses a less efficient operator call to perform the search or verify operation. This operator tests each character of the first string to see if it appears in the second string. The rationale for the PL/I operator is that for short first arguments it is more expensive to construct a test-and-translate table at run-time than to do the indexing. Programs that must search lengthy strings with a variable second argument can use find_char_ to avoid using the PL/I operator, thereby regaining the efficiency of the TCT instruction.

The test-and-translate table is an aligned, fixed-length character string, 512 characters long to cover all possible Multics 9-bit byte values. It consists of "\000" characters and non-\000 characters. Searching (or verifying) using a test-and-translate table progresses as follows:

1) Examine the first (or next) character of the input string. If i is the index of the character being examined, then:
   input_char = substr(string, i, 1)

2) For each input_char, examine its corresponding table_char:
   table_char = substr(table,rank(input_char)+1,1)
3) If table_char = "\000", then the test fails and the search continues with step 1.
4) If table_char ^= "\000", then the test succeeds and the search stops. The current value of i is returned as the index value. For the find_char_$translate_first_in_table and $translate_last_in_table entrypoints, table_char is also returned.
5) If the input string is exhausted before the test succeeds, then a value of 0 is returned as the index argument. For the find_char_$translate_first_in_table and $translate_last_in_table entrypoints, "\000" is returned as the table_char.

Entry: find_char_$first_in_list

This entry performs the PL/I search function, returning the character index of the first (leftmost) occurrence of one of the search_chars in the input string. It constructs the test-and-translate table used by the TCT instruction from search_chars string provided by the caller.

USAGE

```
declare find_char_$first_in_list entry (char(*), char(*))
     returns (fixed bin(21)) reducible;

index = find_char_$first_in_list (string, search_chars);
```

ARGUMENTS

string
    is the character string to be searched. (Input)

search_chars
    are characters to be found in the string. (Input)

index
    is the result of the search. It is the PL/I string index (character position) of the first occurrence of one of the search_chars in the input string. 0 is returned if none of the search_chars appear in the input string. (Output)

Entry: find_char_$last_in_list

This entry returns the character index (character position relative to the beginning of the string) of the last (rightmost) occurrence of one of the search_chars in the input string. It performs the PL/I function:

```
index = length(string) - search (reverse(string), chars) + 1
                [when char searched for is found in string]
index = 0       [when char searched for is not found.]
```

It constructs the test-and-translate table used by the TCT instruction from search_chars string provided by the caller.

USAGE

```
declare find_char_$last_in_list entry (char(*), char(*))
      returns (fixed bin(21)) reducible;

index = find_char_$last_in_list (string, search_chars);
```

ARGUMENTS

string
: is the character string to be searched. (Input)

search_chars
: are characters to be found in the string. (Input)

index
: is the result of the search. It is the PL/I string index (character position) of the last (rightmost) occurrence of one of the search_chars in the input string. 0 is returned if none of the search_chars appear in the input string. (Output)

Entry: find_char_$first_not_in_list

This entry performs the PL/I verify function, returning the character index of the first (leftmost) occurrence in the input string of a character which is not one of the verify_chars. It constructs the test-and-translate table from the verify_chars provided by the caller.

USAGE

```
declare find_char_$first_not_in_list entry (char(*), char(*))
      returns (fixed bin(21)) reducible;

index = find_char_$first_not_in_list (string, verify_chars);
```

ARGUMENTS

string
: is the character string to be searched. (Input)

verify_chars
: are characters which are skipped over when searching the string. (Input)

index
: is the result of the verify. It is the PL/I string index (character position) of the first (leftmost) occurrence of a character in the input string which is not one of the verify_chars. 0 is returned if the entire input string contains only the characters in verify_chars. (Output)

Entry: find_char_$last_not_in_list

This entry returns the index (character position relative to the beginning of the string) of the last (rightmost) occurrence of a character in the input string which is not one of the verify_chars. It performs the PL/I function:

```
index = length(string) - verify (reverse(string), chars) + 1;
                  [when character not in chars is found in string]
index = 0;        [when character not in chars is not found in string.]
```

It constructs the test-and-translate table from the verify_chars provided by the caller.

USAGE

```
declare find_char_$last_not_in_list entry (char(*), char(*))
      returns (fixed bin(21)) reducible;

index = find_char_$last_not_in_list (string, verify_chars);
```

ARGUMENTS

string
: is the character string to be searched. (Input)

verify_chars
: are characters to be skipped over when searching the string. (Input)

index
: is the result of the verify. It is the PL/I string index (character position) of the last (rightmost) occurrence of a character in the input string which is not one of the verify_chars. 0 is returned if the entire input string contains only the characters in verify_chars. (Output)

Entry: find_char_$first_in_table

This entry point searches an input string from the left, using a user-defined test-and-translate table. Either a search or a verify operation can be performed, depending upon the contents of the table. find_char_$make_table_from_chars_in_list can be used to construct a search_table from a set of search_chars; find_char_$make_table_from_chars_not_in_list can be used to construct a verify_table from a set of verify_chars; or the user can create a table containing his own values. As described in the Notes, searching continues until an input string character corresponds to a nonzero element of the table. The character index of the input string character is returned.

USAGE

```
declare find_char_$first_in_table entry (char(*), char(512) aligned)
      returns (fixed bin(21)) reducible;

index = find_char_$first_in_table (string, table);
```

ARGUMENTS

string
: is the character string to be searched. (Input)

table
: is the test-and-translate table. (Input)

index
: is the result of the search. It is a PL/I string index (character position) of the first (leftmost) input character corresponding to a nonzero table element. 0 is returned if no input characters correspond to a nonzero table element. (Output)

Entry: find_char_$last_in_table

This entry point searches an input string from the right, using a user-defined test-and-translate table. Either a search or a verify operation can be performed, depending upon the contents of the table. find_char_$make_table_from_chars_in_list can be used to construct a search_table from a set of search_chars; find_char_$make_table_from_chars_not_in_list can be used to construct a verify_table from a set of verify_chars; or the user can create a table containing his own values. As described in the Notes, searching continues until an input string character corresponds to a nonzero element of the table. The character index of the input string character is returned.

USAGE

```
declare find_char_$last_in_table entry (char(*), char(512) aligned)
     returns (fixed bin(21)) reducible;

index = find_char_$last_in_table (string, table);
```

ARGUMENTS

string
: is the character string to be searched. (Input)

table
: is the test-and-translate table. (Input)

index
: is the result of the search. It is a PL/I string index (character position) of the last (rightmost) input character corresponding to a nonzero table element. 0 is returned if no input characters correspond to a nonzero table element. (Output)

Entry: find_char_$translate_first_in_table

This entry point searches an input string from the left, using a user-defined test-and-translate table. Either a search or a verify operation can be performed, depending upon the contents of the table. As described in the Notes, searching continues until an input string character corresponds to a nonzero element of the table. The character index of the input string character is returned, along with the nonzero table element.

USAGE

```
declare find_char_$translate_first_in_table entry (char(*),
     char(512) aligned, fixed bin(21)) returns (char(1));

table_element = find_char_$translate_first_in_table (string, table,
     index);
```

ARGUMENTS

string
: is the character string to be searched. (Input)

table
: is the test-and-translate table. (Input)

index
: is the result of the search. It is a PL/I string index (character position) of the first (leftmost) input character corresponding to a nonzero table element. 0 is returned if no input characters correspond to a nonzero table element. (Output)

table_element
: is the character from the test-and-translate table which corresponds to the input string character selected by index. "\000" is returned when index=0. (Output)

Entry: find_char_$translate_last_in_table

This entry point searches an input string from the right, using a user-defined test-and-translate table. Either a search or a verify operation can be performed, depending upon the contents of the table. As described in the Notes, searching continues until an input string character corresponds to a nonzero element of the table. The character index of the input string character is returned, along with the nonzero table element.

USAGE

```
declare find_char_$translate_last_in_table entry (char(*),
      char(512) aligned, fixed bin(21)) returns (char(1));

table_element = find_char_$translate_last_in_table (string, table,
      index);
```

ARGUMENTS

string
: is the character string to be searched. (Input)

table
: is the test-and-translate table. (Input)

index
: is the result of the search. It is a PL/I string index (character position) of the last (rightmost) input character corresponding to a nonzero table element. 0 is returned if no input characters correspond to a nonzero table element. (Output)

table_element
: is the character from the test-and-translate table which corresponds to the input string character selected by index. "\000" is returned when index=0. (Output)

Entry: find_char_$make_table_of_chars_in_list

This entry constructs a test-and-translate table for use with the find_char_$first_in_table and find_char_$last_in_table entrypoints. Table entries corresponding to characters of search_chars are marked with \777 in the search table. Other table entries are filled with \000.

USAGE

```
declare find_char_$make_table_of_chars_in_list entry (char(*),
      char(512) aligned);

call find_char_$make_table_of_chars_in_list (search_chars,
      search_table);
```

ARGUMENTS

search_chars
is a string of characters whose corresponding entries are to be marked in the resulting translate table. (Input)

search_table
is the test-and-translate table. (Output)

Entry: find_char_$make_table_of_chars_not_in_list

This entry constructs a test-and-translate table for use with the find_char_$first_in_table and find_char_$last_in_table entrypoints. Table entries corresponding to characters of verify_chars remain unmarked (\000 elements) in the table. Other table elements are filled with \777.

USAGE

```
declare find_char_$make_table_of_chars_not_in_list entry (char(*),
      char(512) aligned);

call find_char_$make_table_of_chars_not_in_list (verify_chars,
      verify_table);
```

ARGUMENTS

verify_chars
is a string of characters whose corresponding entries are to remain unmarked in the resulting translate table. (Input)

verify_table
is the test-and-translate table. (Output)

Entry: find_char_$not_ascii_table

This entrypoint is an external variable containing a predefined test-and-translate table which can be used to detect any non-ASCII characters in a character string. Non-ASCII characters are those in which one or both of the 2 leftmost bits of the 9-bit character byte are "1"b (i.e., character > "\177"). The first 128 values in the table are "\000". The next 384 table characters are set to their character offset within the table. This means that:

```
substr(table,n+1,1) = "\000", for n:  000 <= n <= 127
substr(table,n+1,1) = "\n",   for n:  128 <= n <= 511
```

USAGE

```
declare find_char_$not_ascii_table char(512) aligned external static;
```

Name: find_condition_frame_

This subroutine returns a pointer to the most recent condition frame, or the most recent one before a specified frame.

USAGE

```
dcl find_condition_frame_ entry (ptr) returns (ptr);

stack_ptr = find_condition_frame_ (start_ptr);
```

ARGUMENTS

start_ptr
: is a pointer to a stack frame. The most recent condition frame before this stack frame is returned. The start_ptr argument can be obtained by another call to find_condition_frame_. If start_ptr is null, the most recent condition frame is returned. (Input)

stack_ptr
: is a pointer to the desired condition frame. (Output)

NOTES

The condition history can be traced by repeated calls to find_condition_frame_, starting with a null start_ptr argument and repeatedly passing the output stack_ptr as input.

Name: **find_condition_info_**

This subroutine, given a pointer to a stack frame being used when a signal occurred, returns information relevant to that condition.

USAGE

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

call find_condition_info_ (stack_ptr, condition_info_ptr, code);
```

ARGUMENTS

stack_ptr
: is a pointer to a stack frame being used when a condition occurred. It is normally the result of a call to find_condition_frame_; if null, the most recent condition frame is used. (Input)

condition_info_ptr
: is a pointer to the structure (see "Notes" below) in which information is returned. (Input)

code
: is the standard status code. It is nonzero when the stack_ptr argument does not point to a condition frame or, if the stack_ptr argument is null, when no condition frame can be found. (Output)

NOTES

The structure that condition_info_ptr points to is declared in the include file condition_info.incl.pl1. It is declared as:

```
dcl 1 condition_info     aligned based (condition_info_ptr),
      2 mc_ptr           ptr,
      2 version          fixed bin,
      2 condition_name   char(32) varying,
      2 info_ptr         ptr,
      2 wc_ptr           ptr,
      2 loc_ptr          ptr,
      2 flags            unaligned,
        3 crawlout       bit(1),
        3 pad1           bit(35),
      2 pad2             bit(36),
      2 user_loc_ptr     ptr,
      2 pad3             (4) bit(36);
```

STRUCTURE ELEMENTS

mc_ptr
: if not null, points to the machine conditions. Machine conditions are described in the Programmer's Reference Manual.

version
: is the version number of this structure. It should be set to condition_info_version_1. This variable is declared in condition_info.incl.pl1.

condition_name
: is the condition name.

info_ptr
: points to the info structure if there is one; otherwise, it is null. The info structures for various system conditions are described in the Programmer's Reference Manual.

wc_ptr
: is a pointer to machine conditions describing a fault that caused control to leave the current ring. This occurs when the condition described by this structure was signalled from a lower ring and, before the condition occurred, the current ring was left because of a fault. Otherwise, it is null.

loc_ptr
: is a pointer to the location where the condition occurred. If crawlout is "1"b, this points to the last location in the current ring before the condition occurred.

crawlout
: indicates whether the condition occurred in a lower level ring in which it could not be adequately handled.
"0"b no
"1"b yes

pad1
: is currently unused and should be set to "0"b.

pad2
: is currently unused and should be set to "0"b.

user_loc_ptr
: is a pointer to the most recent nonsupport location before the condition occurred. If the condition occurred in a support procedure (e.g., a PL/I support routine), it is possible to locate the user call that preceded the condition.

pad3
: is currently unused and should be set to "0"b.

Name: find_include_file_

The primary entry point of the find_include_file_ subroutine searches for an include file on behalf of a translator. If the include file is found, additional information about the segment found is returned in the parameters. The "translator" search list is used to locate the include file.

Entry: find_include_file_$initiate_count

This entry point is the interface presented to translators. A translator calls this entry point to invoke a search for a single include file segment using the "translator" search list. For more information about search lists, see the search facility commands, and in particular the add_search_paths command in the Commands manual.

USAGE

```
declare find_include_file_$initiate_count entry (char (*), ptr, char (*),
      fixed bin(24), ptr, fixed bin(35));

call find_include_file_$initiate_count (translator, referencing_ptr,
      file_name, bit_count, seg_ptr, code);
```

ARGUMENTS

translator
: is the name of the translator that is calling this procedure (e.g., pl1, alm). (Input)

referencing_ptr
: is a pointer into the segment (normally a pointer to the source line) that caused the invocation of this instance of this procedure. (Input)

file_name
: is the complete entryname of the include file this procedure is to locate (e.g., xxx.incl.pl1). (Input)

bit_count
: is the bit count as obtained from the storage system of the found include file. (Output). If an include file is not found, this parameter is set to 0.

seg_ptr
: is a pointer to the first character of the include file, if found; if not found, this parameter is set to the null pointer value. (Output)

code
is a standard status code. (Output). The code can be:
0
the requested file was found normally. All output parameters have been set normally.
error_table_$zero_length_seg
the requested file was found, but the bit count was zero. All output parameters have been set normally.
error_table_$noentry
the requested file was not found in any of the search directories.
other storage system error codes
the requested file was not found because of some error.

NOTES

If this procedure finds an include file by a link, the seg_ptr parameter correctly designates the actual location of the include file. It is possible, however, that the name of the actual include file is not the same as the file_name argument passed to this procedure. It is the responsibility of the translator to determine if the file_name passed to this procedure is also on the include file actually found. It is also the responsibility of the translator to call the hcs_$terminate_noname entry point on the include file when processing is complete.

Name: find_partition_

The find_partition_ subroutine is used to ascertain information about a disk partition located on some mounted storage system disk. It reads the label and locates the partition, returning information about its size and location, as well as returning the PVID of the volume, for use in a later call to one of the hardcore entries for partition reading and writing. Use of this subroutine requires access to phcs_.

USAGE

```
dcl find_partition_ entry (char (*), char (*), bit (36) aligned,
     fixed bin (35), fixed bin (35), fixed bin (35));

call find_partition_ (pvname, partition_name, pvid, first_record,
     partition_size, code);
```

ARGUMENTS

pvname
is the name of the physical volume on which the partition is located. (Input). The volume must be a presently mounted storage system disk volume.

partition_name
is the name of the disk partition to be located. (Input). It must be four characters long or shorter.

pvid
is the physical volume ID of the volume the partition is located on. (Output). This is returned as a convenience, for use in a later call to one of the hardcore entries for partition I/O.

first_record
is the number (zero origin, from the beginning of the volume) of the first record in the partition. (Output)

partition_size
is the number of words in the partition. (Output)

code
is a nonstandard status code. (Output). It can be one of the following:
0
indicates that the partition exists and that the returned parameters are all correct.
error_table_$pvid_not_found
indicates that the specified physical volume is not presently mounted.
error_table_$entry_not_found
indicates that the specified partition could not be found.
an integer between 1 and 10
indicates that a physical disk error occurred while trying to read the label. Error messages for physical disk errors are declared in the include file fsdisk_errors.incl.pl1, in the array fsdisk_error_message.

Name: find_source_file_

Finds a file given a pathname and an optional suffix. Translators use this to find source programs.

USAGE

```
declare find_source_file_ entry (char(*), char(*), char(*), ptr,
      fixed bin(24), fixed bin(35));

call find_source_file_ (pathname, suffix, entryname, source_ptr,
      bit_count, code);
```

ARGUMENTS

pathname
is the pathname of the source segment. (Input)

suffix
is the suffix to be added to the pathname (if one does not already exist). (Input)

entryname
is the entryname of the source segment. (Output)

source_ptr
is a pointer to the base of the source segment. It is null if the source could not be found. (Output)

bit_count
is the bit count of the source segment. (Output)

code
is a standard system status code. (Output)

Entry: find_source_file_$search_path

Finds a file given a pathname and an optional suffix. Translators use this to find source programs. A search list is used to locate the source file (e.g., the probe search list).

USAGE

```
dcl find_source_file_$search_path entry (char (*), char (*), char (*),
      char (*), ptr, fixed bin(24), fixed bin(35));

call find_source_file_$search_path (pathname, suffix, search_list_name,
      entry_name, source_ptr, bit_count, code);
```

ARGUMENTS

pathname
is the pathname of the source segment. (Input)

suffix
is the suffix to be added to the pathname (if one does not already exit). (Input)

search_list_name
is the search list to use to locate the source file specified by the pathname and suffix input arguments. (See "Notes" below.) (Input)

entryname
is the entry name of the source segment. (Output)

source_ptr
is the pointer to the base of the source segment. It is null if the source could not be found. (Output)

bit_count
    is the bitcount of the source segment. (Output)

code
    is a standard system status code. (Output)

NOTES

If the $search_path entry is used, the "referencing_dir" keyword is interpreted to be the directory portion of the pathname input argument.

Name: format_document_

The format_document_ subroutine is used to fill and adjust text. The subroutine arguments control the formatting, some of which can be overridden by optional control lines in the text. See the "Notes" section below for a discussion of those control lines.

The format_document_ entry point uses directory names and entrynames. The entrynames can reference segments, links, or multisegment files.

USAGE

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

call format_document_ (dir_name_in, entry_name_in, dir_name_out,
      entry_name_out, options_ptr, code);
```

ARGUMENTS

dir_name_in
    is the pathname of the containing directory of the input. (Input)

entry_name_in
    is the entryname of the input segment, link or multisegment file. (Input)

dir_name_out
    is the pathname of the containing directory of the output. (Input)

entry_name_out
    is the entryname of the output segment, link or multisegment file. (Input) If the entry does not exist, it will be created.

options_ptr
is a pointer to the structure in which options are specified. (Input) See "Notes" below.

code
is a standard status code. (Output) It may be one of the following:
error_table_$fatal_error
if the input file cannot be processed.
error_table_$recoverable_error
if an error occurs but processing is completed.

Entry: format_document_$string

This entry point uses char (*) variables as input and output.

USAGE

```
declare format_document_$string entry (char (*), char(*), fixed bin(21),
      ptr, fixed bin(35));

call format_document_$string (instring, outstring, outlen, options_ptr,
      code);
```

ARGUMENTS

instring
is the input string. (Input)

outstring
is the output string. (Input)

outlen
is the length in bytes of the output string. (Output)

options_ptr
is a pointer to the structure in which options are specified. (Input)

code
is a standard status code. (Output) It can be one of the following:
error_table_$smallarg
if the size of the output exceeds that of the output string.
error_table_$fatal_error
if the input file cannot be processed.
error_table_$recoverable_error
if an error occurs but processing is completed.

Entry: format_document_$switch

This entry point uses a directory name and entryname for input and writes its output to an I/O switch. The entryname can reference a segment, link or multisegment file.

USAGE

```
declare format_document_$switch entry (char(*), char(*), ptr, ptr,
      fixed bin(35));
call format_document_$switch (dir_name_in, entry_name_in, iocbptr,
      options_ptr, code);
```

ARGUMENTS

dir_name_in
: is the pathname of the containing directory of the input. (Input)

entry_name_in
: is the entryname of the input segment, link or multisegment file. (Input)

iocbptr
: is a pointer to the control block for the output I/O switch. (Input)

options_ptr
: is a pointer to the structure in which options are specified. (Input) See "Notes" below.

code
: is a standard status code. (Output) It can one of the following:
error_table_$fatal_error
  if the input file cannot be processed.
error_table_$recoverable_error
  if an error occurs but processing is completed.

NOTES

The argument options_ptr points to the following structure (defined in include file format_document_options.incl.pl1):

```
dcl 1 format_document_options aligned based (format_document_options_ptr),
      2 version_number        fixed bin,
      2 indentation           fixed bin,
      2 line_length           fixed bin,
      2 switches,
        3 pgno_sw             bit (1) unaligned,
        3 adj_sw              bit (1) unaligned,
        3 galley_sw           bit (1) unaligned,
        3 error_sw            bit (1) unaligned,
        3 literal_sw          bit (1) unaligned,
        3 file_sw             bit (1) unaligned,
        3 dont_compress_sw    bit (1) unaligned,
        3 break_word_sw       bit (1) unaligned,
        3 max_line_length_sw  bit (1) unaligned,
        3 dont_break_indented_lines_sw
                              bit (1) unaligned,
        3 sub_err_sw          bit (1) unaligned,
        3 dont_fill_sw        bit (1) unaligned,
        3 hyphenation_sw      bit (1) unaligned,
        3 mbz                 bit (23) unaligned
      2 syllable_size         fixed bin;
```

STRUCTURE ELEMENTS

version_number
: is the version number of this structure. (Input) It must have the value 2.

indentation
: indentation value, causing indentation from the left margin. (Input) This space is in addition to any indentation established by the usage of the indent control in the text.

line_length
: is the initial line length value. (Input) It is the equivalent of the ".pdw" control in the text, and can be overridden in the text.

pgno_sw
: (Input)
: "1"b enables page numbering. If galley_sw is off, then each page is to end with a centered page number.
: "0"b indicates that no page numbering is requested.

adj_sw
(Input)
"1"b causes adjust mode to be on initially. This is the equivalent of a ".alb" in the text. It can be overridden in the text.
"0"b causes adjust mode to be off initially. This is the equivalent of a ".all" in the text. It is only meaningful if dont_fill_sw = "0"b.

galley_sw
(Input)
"1"b suppresses vertical margins and page breaks.
"0"b enables vertical margins and page breaks.

error_sw
(Input)
"1"b causes diagnostic error messages to be written to error_output.
"0"b suppresses diagnostic error messages.

literal_sw
(Input)
"1"b causes all input to be treated as text.
"0"b enables format_document_ controls. A line that begins with a period is treated as a control line.

file_sw
(Output)
"1"b indicates that a non-zero storage system status code refers to the output file.
"0"b indicates that a non-zero storage system status code refers to the input file.

dont_compress_sw
(Input)
"1"b causes no compression of adjacent spaces and tab characters, nor enforcement of the convention of ending a sentence with 2 spaces.
"0"b causes adjacent spaces and tab characters that do not begin a line to be converted to a single space, and enforces the convention that a sentence ends with 2 spaces. A sentence is any string that terminates with one of the following character sequences: ". "; "? "; " "; ": "; ".) "; "?) "; or ") ". This switch may be overridden by dont_break_indented_lines for certain input lines.

break_word_sw
(Input)
"1"b causes the line to be broken in the middle of a word if the line exceeds the line length and there are no spaces or tab characters available at which to do so.
"0"b causes an overlength line to be returned and error_table_$recoverable_error to be returned if the line exceeds the linelength and there are no spaces or tab characters at which to break it.

max_line_length_sw
(Input)
"1"b causes the line to be re-adjusted to the line_length given in this structure and error_table_$recoverable_error to be returned if controls in the text cause the calculated line length to be greater than the line_length given in this structure.
"0"b allows the controls in the text to adjust the calculated line length to a value greater than the line_length given in this structure.

dont_break_indented_lines_sw
(Input)
"1"b indicates that if an input line exceeds the specified line length and begins with a blank or horizontal tab, it is not to be broken if a) it is the last line of input, b) it is followed a line that begins with a blank or horizontal tab, or c) it is followed by a blank line. In addition, such lines will not have space compression or sentence formatting performed on them, no matter what value dont_compress_sw has.
"0"b indicates that overlength lines are broken regardless of indentation, and that dont_compress will always control space compression and sentence formatting.

sub_err_sw
(Input)
"1"b indicates that diagnostic errors are to be communicated via calls to sub_err_. The info structure used to communicate information about the error is format_document_error.incl.pl1.
"0"b indicates that no calls to sub_err_ are to be made.

dont_fill_sw
(Input)
"1"b causes fill mode to be off initially. This is the equivalent of a ".fif" in the text. It can be overridden in the text.
"0"b causes fill mode to be on initially. This is the equivalent of a ".fin" in the text.

hyphenation_sw
(Input)
"1"b causes hyphenation mode to be on initially. Hyphenation can be turned on in the text via the "hyn" control and off via the ".hyf" control. The ".hy" control returns hyphenation mode to the initial value, i.e. the state of this switch.
"0"b causes hyphenation mode to be off initially.

mbz
must be zero.

syllable_size
(Input)
indicates the smallest number of characters to be contained in a syllable that is to be separated from the rest of a word by hyphenation. This value can be adjusted in the text via the ".hyn" control. The ".hy" control returns the syllable size to the value of this argument.

CONTROL LINES

The following is a discussion of each of the control lines.

.alb
align the text at both the left and right margins according to the current value of the left indentation and undentation. Text is padded by insertion of uniformly distributed white space. The fill mode must be on for this mode to operate. If the fill mode is off, this control is mapped into the align-left (.all) control. This is the default alignment mode.

.all
align the text on the left margin according to the current values of left indentation and undentation leaving the right margin ragged.

.brf
finish the current output line by formatting any pending texts as a short line.

.brf finish the current page, formatting any pending texts as a short line.

.fif
set the fill mode off. See the discussion of .fin for details.

.fin
set the fill mode on. In fill mode, text words are moved from line to line in such a way that the last word does not extend past the right margin. The default for this mode is on.

.hy
set hyphenation mode and syllable size to the defaults: format_document_options.hyphenation_sw format_document_options.syllable_size.

.hyf
set hyphenation mode off. See the discussion of .hyn for details.

.hyn {N}
set hyphenation mode on and set the syllable size according to the given parameter. the parameter is given as an unsigned integer and specifies the number of characters in the smallest allowed hyphenated syllable. If the parameter is not given, then the default syllable size is used. The default syllable size is the value given in format_document_options.syllable_size.

.in, .inl {+/-N}
if N is given without the optional sign, plus (+) or minus (–), then set the left indentation point to N columns to the right of the left margin. If N is given with the optional sign, then change the current left indentation point by N columns. Positive values for N cause movement to the right. The default value for N is 0. Any value that results in a zero or negative effective line length will produce an error diagnostic message. The left indentation point is never set to the left of the left margin. The left margin is determined by the –indent control argument.

.pdl {+/-N}
if N is given without the optional sign, (+ or –), then set the page length to N lines. If N is given with the optional sign, then change the current page length by N. If the resulting page length is zero or negative, an error diagnostic message is produced. The default value for N is 66.

.pdw {+/-N}
if N is given without the optional sign, then set the page width to N columns. If N is given with the optional sign, then change the current page width by N. If the resulting page width is zero or negative, an error diagnostic message is produced. The default value for N is 65.

.spf {N}
finish the current output line and then add N blank lines. If N is not given, then add 1 blank line.

.un, .unl {+/-N}
adjust the indentation point for only the next output line. If N is unsigned or has the + sign the indentation point is moved n columns to the left. If N has the - sign, the indentation point is moved n columns to the right. The default value for N is the value of the indentation value.

SPECIALIZED ERROR HANDLING

If format_document_options.sub_err_sw is set and errors occur in formatting, the sub_err_ subroutine will be called to signal them rather than aborting the program. The caller of format_document_ can set up a handler for the sub_error_ condition and use the information in the format_document_error structure to make decisions about how to proceed. See the sub_err_ subroutine for an example of how to establish a handler for the sub_error__ condition.

```
dcl 1 format_document_error aligned based (format_document_error_ptr),
      2 version_number          fixed bin,
      2 error_code              fixed bin (35),
      2 line_number             fixed bin,
      2 text_line               char (128) varying;

dcl  format_document_error_ptr ptr;
dcl  format_document_error_version_1 fixed bin int
       static options (constant) init (1);
```

STRUCTURE ELEMENTS

version_number
: is a number representing the version of the format_document_error structure being used. The structure above is version 1.

error_code
: is a standard error code from the fdoc_et_ error table describing the problem encountered. It can be used in a call to com_err_ or convert_status_code_ to announce the error.

line_number
: is the line number on which the error appears in the input.

text_line
: is the offending line (or first 128 chars).

The following values can occur in the error_code element of the format_document_error structure:

LIST OF ERROR CODES

fdoc_et_$indent_too_far_left, fdoc_et_$indtflft
: Attempted to indent past left margin; resets to left margin.

fdoc_et_$indent_too_far_right, fdoc_et_$indtfrgt
: Attempted to indent past right margin; resets to right margin.

fdoc_et_$line_length_too_small, fdoc_et_$lnlntsml
: Effective line length is less than 1.

fdoc_et_$line_too_long, fdoc_et_$lntoolng
: Located a string of more than 256 characters without blank or newline.

fdoc_et_$no_parameter_allowed, fdoc_et_$noparalw
: This control supports no parameters.

fdoc_et_$nonnumeric_parameter, fdoc_et_$nonumpar
: A non-numeric parameter.

fdoc_et_$page_length_lt_13, fdoc_et_$pglnlt13
: Given page length is too small; resets to the current minimum of 13.

fdoc_et_$page_length_lt_14, fdoc_et_$pglnlt14
: Given page length is too small; resets to the current minimum of 14.

fdoc_et_$page_width_exceeds_max, fdoc_et_$pgwdxmax
: The computed page width is too large; resets to the specified maximum.

fdoc_et_$text_too_long_for_line, fdoc_et_$txttulng
Text is too long for output line.

fdoc_et_$undent_too_far_left, fdoc_et_$undtflft
Attempted to undent past left margin; resets to left margin.

fdoc_et_$undent_too_far_right, fdoc_et_$undtfrgt
Attempted to undent past right margin; resets to right margin.

fdoc_et_$unsupported_control, fdoc_et_$unsupctl
The given control is unsupported.

Name: fs_util_

The fs_util_ subroutine provides for uniform handling of file system entries. Supported operations are validate, copy, delete, chname, get_switch, set_switch, get_max_length, set_max_length, set_bit_count, and ACL manipulation. When invoked, the subroutine checks to see if the entry name provided is that of an extended entry and if it is, calls the corresponding entry point of the appropriate suffix_XXX_ subroutine. If the name is not that of an extended entry, then fs_util_ calls the appropriate standard entry entry point handler for the entry.

Entry: fs_util_$add_acl_entries

This entry point is used to add to the Access Control List of an entry. If an access name already appears on the ACL of the entry, its mode is changed to the one specified by the call.

USAGE

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

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

ARGUMENTS

dir_name
is the absolute pathname of the directory containing the entry. (Input)

entryname
is the name of the entry. (Input)

acl_ptr
    is a pointer to the general_acl structure. (Input)

code
    is a standard system status code. (Output)

NOTES

The general_acl structure and the named constant GENERAL_ACL_VERSION_1 are defined in the include file acl_structures.incl.pl1.

The general_acl structure is defined as follows:

```
1 general_acl                  aligned based (acl_ptr),
  2 version                    char (8) aligned,
  2 count                      fixed bin,
  2 entries                    (acl_count refer (general_acl.count))
       aligned like general_acl_entry;

1 general_acl_entry            based,
  2 access_name                character (32) unaligned,
  2 mode                       bit (36) aligned,
  2 status_code                fixed bin (35);
```

STRUCTURE ELEMENTS

version
    is the current version of this structure and has the value of the named constant GENERAL_ACL_VERSION_1.

count
    is the size of the entries array in general_acl.

access_name
    is the name of a user in the form of Person_id.Project_id.instance_tag.

mode
    is a bit string where each bit represents a possible access mode which, when true, indicates an allowed access for the file. access_mode_values.incl.pl1 defines named constants for the mode values of standard entries. Modes for extended entries are defined by the subsystem owning the extended entry type. Use the describe_entry_type command to display extended mode values.

status_code
    is a standard system status code indicating success or the reason for failure to set the ACL entry.

Entry: fs_util_$add_extended_acl_entries

This entry point is used to add to the Extended Access Control List of a standard entry.

USAGE

```
declare fs_util_$add_extended_acl_entries entry (char(*), char(*), ptr,
      fixed bin(35));

call fs_util_$add_extended_acl_entries (dir_name, entryname, acl_ptr,
      code);
```

ARGUMENTS

dir_name
    is the absolute pathname of the directory containing the entry. (Input)

entryname
    is the name of the entry. (Input)

acl_ptr
    is a pointer to the structure general_extended_acl. (Input)

code
    is a standard system status code. (Output)

NOTES

This interface is intended to be used only by extended entry type support routines, (also referenced as suffix_XXX_), in order to map an ACL mode provided to fs_util_ into a standard mode/extended mode pair to be placed on the underlying standard entry or entries which are being used to implement the extended entry type.

The general_extended_acl structure and the named constant GENERAL_EXTENDED_ACL_VERSION_1 are defined in the include file acl_structures.incl.pl1.

The general_extended_acl structure is defined as follows:

```
1 general_extended_acl        aligned based (acl_ptr),
  2 version                   char (8) aligned,
  2 count                     fixed bin,
  2 entries                   (acl_count refer
       (general_extended_acl.count))
       aligned like general_extended_acl_entry;

1 general_extended_acl_entry aligned based,
       2 access_name          character (32) unaligned,
       2 mode                 bit (36) aligned,
       2 extended_mode        bit (36) aligned,
       2 status_code          fixed bin (35);
```

STRUCTURE ELEMENTS

version
: is the current version of this structure and has the value of the named constant GENERAL_EXTENDED_ACL_VERSION_1.

count
: is the size of the entries array in general_extended_acl.

access_name
: is the name of a user in the form of Person_id.Project_id.instance_tag.

mode
: is a bit string where each bit represents a possible access mode which, when true, indicates an allowed access for the file.

extended_mode
: is a bit string where each bit represents a possible extended access mode which, when true, indicates an allowed access for the file.

status
: is a standard system status code indicating success or the reason for failure to set the extended ACL entry.

Entry: fs_util_$chname_file

This entry point is used to change the name of an entry. If only an old_name is given, the effect is to delete. If only a new_name is given, the effect is to add the name. If both are specified, the effect is to rename the entry.

USAGE

```
declare fs_util_$chname_file entry (char(*), char(*), char(*), char(*),
      fixed bin (35));

call fs_util_$chname_file (dir_name, entryname, old_name, new_name,
      code);
```

ARGUMENTS

dir_name
: is the absolute pathname of the directory containing the entry. (Input)

entryname
: is the name of the entry. (Input)

old_name
: is the name to be deleted from the entry. (Input) It can be a null character string ("") in which case no name is deleted. If old_name is null, the new_name must not be null.

new_name
: is the new name to be added to the entry. (Input) It must not already exist in the directory on this or another entry. It can be a null string ("") in which case no name is added to the entry. It it is null, then old_name must not be the only name on the entry.

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

Entry: fs_util_$copy

This entry point is used to copy an entry.

USAGE

```
declare fs_util_$copy entry (ptr, fixed bin (35));

call fs_util_$copy (copy_options_ptr, code);
```

ARGUMENTS

copy_options_ptr
    is a pointer to the copy_options structure. (Input)

code
    is a standard system status code. (Output)

NOTES

The copy_options structure and the named constant COPY_OPTIONS_VERSION_1 are defined in the include file copy_options.incl.pl1.

The copy_options structure is defined as follows:

```
1 copy_options              aligned based (copy_options_ptr),
2 version                   char (8),
2 caller_name               char (32) unal,
2 source_dir                char (168) unal,
2 source_name               char (32) unal,
2 target_dir                char (168) unal,
2 target_name               char (32) unal,
2 flags,
  3 no_name_dup             bit (1) unaligned,
  3 raw                     bit (1) unaligned,
  3 force                   bit (1) unaligned,
  3 delete                  bit (1) unaligned,
  3 target_err_switch       bit (1) unaligned,
  3 mbz                     bit (31) unaligned,
2 copy_items                like copy_flags;
```

STRUCTURE ELEMENTS

version
    is the current version of this structure and has the value of the named constant COPY_OPTIONS_VERSION_1.

caller_name
    is the name of the program calling fs_util_, required when querying the user about duplicate names. See no_name_dup below.

source_dir
    is the absolute pathname of the directory containing the entry to be copied.

source_name
    is the name of the entry to be copied.

target_dir
is the absolute pathname of the directory into which a copy of the entry is to be placed.

target_name
is the name of the entry created to hold the copy of the original entry.

no_name_dup
is set to "0"b if the user is to be queried in case of a duplication of the target_name and "1"b if there is to be no query, in which case an error code will be returned.

raw
is set to "0"b if fs_util_ is to honor the extended type of the entry, and "1"b if it is to bypass this by calling hcs_.

force
is set to "1"b if access to the target is to be forced.

delete
is set to "1"b if the original is to be deleted after it is copied.

target_err_switch
is set if an error occurred referencing the target.

mbz
is reserved for future use and must be set to zero.

copy_items
is structured like the copy_flags structure, which is defined in the include file copy_flags.incl.pl1. The structure is defined as follows:

```
1 copy_flags                aligned based,
  2 names                     bit (1) unaligned,
  2 acl                       bit (1) unaligned,
  2 ring_brackets             bit (1) unaligned,
  2 max_length                bit (1) unaligned,
  2 copy_switch               bit (1) unaligned,
  2 safety_switch             bit (1) unaligned,
  2 dumper_switches           bit (1) unaligned,
  2 entry_bound               bit (1) unaligned,
  2 extend                    bit (1) unaligned,
  2 update                    bit (1) unaligned,
  2 mbz                       bit (26) unaligned;
```

When variables in the copy_flags structure have a value of "1"b, the designated attribute is copied to the new entry. Before the copy is performed, the copy_items members are ANDed with copy_flags members as defined by the suffix_info entry point. Only those attributes specified by both structures are copied. In the case of extend, the contents of the original entry may be appended to the end of the target entry. In the case of update, the contents of the original entry may replace the contents of the target entry.

Entry: fs_util_$delete_acl_entries

This entry point deletes a member of an entry's Access Control List.

USAGE

```
declare fs_util_$delete_acl_entries entry (char(*), char(*), ptr, fixed
      bin(35));

call fs_util_$delete_acl_entries (dir_name, entryname, acl_ptr, code);
```

ARGUMENTS

dir_name
: is the absolute pathname of the directory containing the entry. (Input)

entryname
: is the name of the entry. (Input)

acl_ptr
: is a pointer to the structure general_delete_acl. (Input)

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

NOTES

The general_delete_acl structure and the named constant GENERAL_DELETE_ACL_VERSION_1 are defined in the include file acl_structures.incl.pl1.

The general_delete_acl structure is defined as follows:

```
   1 general_delete_acl            aligned based (acl_ptr),
     2 version                     char (8) aligned,
     2 count                       fixed bin,
     2 entries                     (acl_count refer
          (general_delete_acl.count))
          aligned like delete_acl_entry;

declare 1 general_delete_acl_entry      aligned based,
          2 access_name                 character (32) unaligned,
          2 status_code                 fixed bin (35);
```

STRUCTURE ELEMENTS

version
: is the current version of this structure and has the value of the named constant GENERAL_DELETE_ACL_VERSION_1.

count
: is the size of the entries array in general_delete_acl.

access_name
: is the name of a user in the form of Person_id.Project_id.instance_tag

status_code
: is a standard system status code indicating success or the reason for failure to set the extended ACL entry.

Entry: fs_util_$delentry_file

This entry point deletes a file system entry.

USAGE

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

call fs_util_$delentry_file (dir_name, entryname, code)
```

ARGUMENTS

dir_name
is the absolute pathname of the directory containing the entry. (Input)

entryname
is the name of the entry. (Input)

code
is a standard system status code. (Output)

Entry: fs_util_$get_bit_count

This entry point returns the number of useful bits in an entry.

USAGE

```
declare fs_util_$get_bit_count entry (char(*), char(*), fixed bin(41),
     fixed bin (35));

call fs_util_$get_bit_count (dir_name, entryname, bit_count, code);
```

ARGUMENTS

dir_name
is the absolute pathname of the directory containing the entry. (Input)

entryname
is the name of the entry. (Input)

bit_count
is the number of bits considered useful in the entry. (Output)

code
is a standard system status code. (Output)

Entry: fs_util_$get_max_length

This entry point returns the maximum length setting for an entry.

USAGE

```
declare fs_util_$get_max_length entry (char(*), char(*), fixed bin (35),
     fixed bin (35));

call fs_util_$get_max_length (dir_name, entryname, max_length, code);
```

ARGUMENTS

dir_name
: is the absolute pathname of the directory containing the entry. (Input)

entryname
: is the name of the entry. (Input)

max_length
: is the maximum length of the entry in words. (Output)

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

Entry: fs_util_$get_ring_brackets

This entry point returns the ring brackets of an entry.

USAGE

```
declare fs_util_$get_ring_brackets entry (char(*), char(*), (*)fixed
      bin(3), fixed bin(35));

call fs_util_$get_ring_brackets (dir_name, entryname, ring_brackets,
      code);
```

ARGUMENTS

dir_name
: is the absolute pathname of the directory containing the entry. (Input)

entryname
: is the name of the entry. (Input)

ring_brackets
: are the ring numbers which define the upper bounds of the ring brackets which control the various modes of access to the entry. (Output) Ring brackets are discussed in "Intra Process Access Control" in the *Multics Programmer's Reference Manual,* Order No. AG91.

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

Entry: fs_util_$get_switch

This entry point returns the value of a storage system switch for an entry.

USAGE

```
declare fs_util_$get_switch entry (char (*), char (*), char (*), bit(1)
      aligned, fixed bin(35));

call fs_util_$get_switch (dir_name, entryname, switch_name, value,
      code);
```

ARGUMENTS

dir_name
is the absolute pathname of the directory containing the entry. (Input)

entryname
is the name of the entry. (Input)

switch_name
is the name of the switch whose value is sought. For example, it may be one of the following: "copy," "complete_volume_dump," "damaged," "incremental_volume_dump," "safety," "synchronized," or any additional switch supported by the appropriate extended entry type. (Input)

value
is the value of the requested switch. (Output)

"1"b means the switch is on
"0" means that it is off.

code
is a standard system status code. It should be set to error_table_$argerr if switch_name is invalid. (Output)

Entry: fs_util_$get_type

This entry point returns the type of a specified entry.

USAGE

```
declare fs_util_$get_type entry (char (*), char (*), char (*),
      fixed_bin(35));

call fs_util_$get_type (dir_name, entryname, type, code);
```

ARGUMENTS

dir_name
: is the absolute pathname of the directory containing the entry. (Input)

entryname
: is the name of the entry. (Input)

type
: is either the suffix if extended, or one of the name constants for standard entry types found in the include file suffix_info.incl.pl1. (Output)

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

Entry: fs_util_$get_user_access_modes

This entry point returns a user's effective access mode and extended access mode for an entry. For a description of modes, see "Effective Access" in the *Multics Programmer's Reference Manual,* Order No. AG91.

USAGE

```
declare fs_util_$get_user_access_modes entry (char (*), char (*), char (*),
      fixed bin, bit(36) aligned, bit(36) aligned, fixed bin(35));

call fs_util_$get_user_access_modes (dir_name, entryname, user_name,
      ring, modes, exmodes, code);
```

ARGUMENTS

dir_name
: is the absolute pathname of the directory containing the entry. (Input)

entryname
: is the name of the entry. (Input)

user_name
: is the access name of the user in the form Person_id.Project_id.tag. (Input) This is limited to 32 characters. If null, the access name of the calling process is used.

ring
: is the validation level that is to be used in computing effective access. (Input) It must be a value between 0 and 7 inclusive, or −1. If the ring value is −1, the default value of the validation level of the calling process is used. This default should be used in all cases except those in which a different ring's access is explicitly required.

modes
    are the standard ACL modes of an entry. (Output)

xmodes
    are the extended ACL modes of an entry. (Output)

code
    is a standard system status code. (Output)

NOTES

The include file access_modes_values.incl.pl1 defines named constants for the different values of modes. Extended access modes are defined by the subsystem owning the extended entry.

Entry: fs_util_$list_acl

This entry point lists the components of an entry's Access Control List.

USAGE

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

call fs_util_$list_acl (dir_name, entryname, version, area_ptr, acl_ptr,
      fixed bin(35));
```

ARGUMENTS

dir_name
    is the absolute pathname of the directory containing the entry. (Input)

entryname
    is the name of the entry. (Input)

version
    is the version of the acl structure. (Input)

area_ptr
    is a pointer to an area where fs_util_ can allocate the general_acl structure. 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. (Input)

acl_ptr
is a pointer to the general_acl structure. (Input or Output)

Input: if area_ptr is null, then acl_ptr points to a general_acl structure filled with access names and into which modes will be placed.

Output: if area_ptr is non null, then acl_ptr will point to the start of a newly allocated general_acl structure.

code
is a standard system status code. (Output)

NOTES

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

The general_acl structure and the named constant GENERAL_ACL_VERSION_1 are defined in the include file acl_structures.incl.pl1. For a description of the general_acl structure, see the add_acl_entries entrypoint above.

Entry: fs_util_$list_extended_acl

This entry point returns the contents of the Extended Access Control List of a standard entry.

USAGE

```
declare fs_util_$list_extended_acl entry (char(*), char(*), char(*),
      ptr, ptr, fixed bin(35));

call fs_util_$list_extended_acl (dir_name, entryname, version, area_ptr,
      acl_ptr, code);
```

ARGUMENTS

dir_name
is the absolute pathname of the directory containing the entry. (Input)

entryname
is the name of the entry. (Input)

version
is the version of the acl structure. (Output)

area_ptr
is a pointer to an area where fs_util_ can allocate the general_extended_acl structure. If area_ptr is null, then the user wants access modes for certain extended ACL entries; these will be specified by the structure pointed to acl_ptr. (Input)

acl_ptr
is a pointer to the general_acl structure. (Input or Output)

Input: if area_ptr is null, then acl_ptr points to a general_extended_acl structure filled with access names and into which modes will be placed.

Output: if area_ptr is non null, then acl_ptr will point to the start of a newly allocated general_extended_acl structure.

code
is a standard system status code. (Output)

NOTES

This interface is intended to be used only by extended entry type support routines, (also referenced as suffix_XXX_), in order to map an ACL mode provided to fs_util_ into a standard mode/extended mode pair to be placed on the underlying standard entry or entries which are being used to implement the extended entry type.

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

The general_extended_acl structure and the named constant GENERAL_EXTENDED_ACL_VERSION_1 are defined in the include file acl_structures.incl.pl1. The general_extended_acl structure is described in the add_extended_acl_entries entrypoint described above.

Entry: fs_util_$list_switches

This entry point returns a list of switches supported by the entry type.

USAGE

```
declare fs_util_$list_switches entry (char(*), char(*), char(*), ptr,
      ptr, fixed bin(35));

call fs_util_$list_switches (dir_name, entryname, version, area_ptr,
      switch_list_ptr, code);
```

ARGUMENTS

dir_name
    is the absolute pathname of the directory containing the entry. (Input)

entryname
    is the name of the entry. (Input)

version
    is the version of the switch list structure. (Input)

area_ptr
    is a pointer to an area where fs_util_ can allocate the structure switch_list. (Input)

switch_list_ptr
    is a pointer to the switch_list structure. (Output)

code
    is a standard system status code. (Input)

NOTES

The list_switches structure and the named constant SWITCH_LIST_VERSION_1 are defined in the include file suffix_info.incl.pl1.

The list_switches structure is defined as follows:

```
1 switch_list                aligned based (switch_list_ptr),
  2 version                  char (8),
  2 switch_count             fixed bin,
  2 switch_name_count        fixed bin,
  2 switches                 (alloc_switch_count
      refer (switch_list.switch_count)),
    3 name_index             fixed bin,
    3 name_count             fixed bin,
    3 default_value          bit (1) aligned,
    3 mbz1                   bit (36) aligned,
  2 names                    (alloc_switch_name_count refer
      (switch_list.switch_name_count)) char (32);
```

STRUCTURE ELEMENTS

version
is the current version of this structure and has the value of the named constant SWITCH_LIST_VERSION_1.

switch_count
is the number of switches defined for this entry type.

switch_name_count
is the total number of names of the switches; a switch can have multiple names.

name_index
is the index into suffix_list.names aray of the first name for this switch.

name_count
is the number of names for this switch. The names for this switch are located in switch_list.names(name_index) through switch_list.names(name_index + name_count - 1).

default_value
is the default setting for this switch when the entry is created. is the array of switch names.

Entry: fs_util_$list_switches_for_type

This entry point returns a list of switches for a particular type of entry.

USAGE

```
declare fs_util_$list_switches_for_type entry (char(*), char(*), ptr,
      ptr, fixed bin(35));

call fs_util_$list_switches_for_type (type, version, area_ptr,
      switch_list_ptr, code);
```

ARGUMENTS

type
is either the suffix if extended, or one of the name constants for standard entry types found in the include file suffix_info.incl.pl1. (Input)

version
is the version of the switch_list structure. (Input)

area_ptr
is a pointer to an area where fs_util_ can allocate the structure switch_list. (Input)

switch_list_ptr
is a pointer to the switch_list structure. (Output)

code
is a standard system status code. (Input)

NOTES

The list_switches structure and the named constant SWITCH_LIST_VERSION_1 are defined in the include file suffix_info.incl.pl1. The list_switches structure is described in the list_switches entrypoint above.

Entry: fs_util_$make_entry

This entry point constructs a variable to a specified suffix support subroutine for a specified extended entry.

USAGE

```
declare fs_util_$make_entry entry (char (*), char (*), char (*), entry,
      fixed bin(35));

call fs_util_$make_entry (dir_name, entryname, entrypoint,
      entry_to_call, code);
```

ARGUMENTS

dir_name
is the absolute pathname of the directory containing the entry. (Input)

entryname
is the name of the entry. (Input)

entrypoint
is the name of the entrypoint that is is to be constructed. (Input)

entry_to_call
is the entry variable constructed. (Output)

code
is a standard system status code. (Output)

Entry: fs_util_$make_entry_for_type

This entry point constructs a variable to a specified suffix support subroutine for a specified type of extended entry.

USAGE

```
declare fs_util_$make_entry_for_type entry (char(*), char(*), entry,
      fixed bin(35));

call fs_util_$make_entry_for_type (type, entrypoint, entry_to_call,
      code);
```

ARGUMENTS

type
    is either the suffix if extended, or one of the name constants for standard entry types found in the include file suffix_info.incl.pl1. (Input)

entrypoint
    is the name of the entrypoint that is is to be constructed. (Input)

entry_to_call
    is the entry variable constructed. (Output)

code
    is a standard system status code. (Output)

Entry: fs_util_$replace_acl

This entry point is used to replace Access Control List components for an entry.

USAGE

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

call fs_util_$replace_acl (dir_name, entryname, acl_ptr, no_sysdaemon,
      code);
```

ARGUMENTS

dir_name
    is the absolute pathname of the directory containing the entry. (Input)

entryname
    is the name of the entry. (Input)

acl_ptr
is a pointer to the structure general_extended_acl. (Input)

no_sysdaemon
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 general_acl entries are added. (Input)
"0"b adds rw *.SysDaemon.* entry
"1"b replaces the existing ACL with only the user-supplied general_acl.

code
is a standard system status code. (Output)

NOTES

The general_acl structure and the named constant GENERAL_ACL_VERSION_1 are defined in the include file acl_structure.incl.pl1. The general_acl structure is described above in the entrypoint add_acl_entries.

Entry: fs_util_$replace_extended_acl

This entry point is used to replace Extended Access Control List components for a standard entry.

USAGE

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

call fs_util_$replace_extended_acl (dir_name, entryname, acl_ptr,
      no_sysdaemon, code);
```

ARGUMENTS

dir_name
is the absolute pathname of the directory containing the entry. (Input)

entryname
is the name of the entry. (Input)

acl_ptr
is a pointer to the structure general_extended_acl. (Input)

no_sysdaemon
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 general_acl entries are added. (Input)
"0"b adds rw *.SysDaemon.* entry
"1"b replaces the existing ACL with only the user-supplied general_acl.

code
is a standard system status code. (Output)

NOTES

This interface is intended to be used only by extended entry type support routines, (also referenced as suffix_XXX_), in order to map an ACL mode provided to fs_util_ into a standard mode/extended mode pair to be placed on the underlying standard entry or entries which are being used to implement the extended entry type.

The structure general_extended_acl and the named constant GENERAL_EXTENDED_ACL_VERSION_1 are defined in the include file acl_structures.incl.pl1. The structure general_extended_acl is described in the add_extended_acl_entries entry point above.

Entry: fs_util_$set_bit_count

This entry point sets the number of bits considered useful for an entry.

USAGE

```
declare fs_util_$set_bit_count entry (char (*), char (*), fixed bin
      (41), fixed bin (35);

call fs_util_$set_bit_count (dir_name, entryname, bit_count, code);
```

ARGUMENTS

dir_name
is the absolute pathname of the directory containing the entry. (Input)

entryname
is the name of the entry. (Input)

bit_count
is the number of bits to be considered useful in the entry. (Input)

code
is a standard system status code. (Output)

Entry: fs_util_$set_max_length

This entry point sets the maximum length a particular entry grow to.

USAGE

```
declare fs_util_$set_max_length entry (dir_name, entryname, max_length,
      code);

call fs_util_$set_max_length (char(*), char(*), fixed bin(35), fixed
      bin(35));
```

ARGUMENTS

dir_name
: is the absolute pathname of the directory containing the entry. (Input)

entryname
: is the name of the entry. (Input)

max_length
: is the maximum length in words to be placed on the entry. (Input)

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

Entry: fs_util_$set_ring_brackets

This entry point sets the ring brackets of an entry.

USAGE

```
declare fs_util_$set_ring_brackets entry (char(*), char(*), (*)fixed
      bin(3), fixed bin(35));

call fs_util_$set_ring_brackets (dir_name, entryname, ring_brackets,
      code);
```

ARGUMENTS

dir_name
: is the absolute pathname of the directory containing the entry. (Input)

entryname
: is the name of the entry. (Input)

ring_brackets
: are the bounds of the rings from which an entry is accessible. (Input)

code
is a standard system status code. (Output)

Entry: fs_util_$set_switch

This entry sets the value of a storage system switch for an entry.

USAGE

```
declare fs_util_$set_switch entry (char (*), char (*), char (*), bit(1)
      aligned, fixed bin(35));

call fs_util_$set_switch (dir_name, entryname, switch_name, value,
      code);
```

ARGUMENTS

dir_name
is the absolute pathname of the directory containing the entry. (Input)

entryname
is the name of the entry. (Input)

switch_name
is the name of the switch whose value is to be set. This may be either "copy," "complete_volume_dump," "damaged," "incremental_volume_dump," p"safety," "synchronized," or any switch on an extend entry type. (Input)

value
is the value to which the switch is to be set. (Input)

"1"b means the switch is on
"0" means that it is off.

code
is a standard system status code. It should be set to error_table_$argerr if switch_name is invalid. (Output)

Entry: fs_util_$suffix_info

This entry point returns information about an entry's type.

USAGE

```
declare fs_util_$suffix_info entry (char (*), char (*), ptr, fixed
      bin(35));

call fs_util_$suffix_info (dir_name, entryname, suffix_info_ptr, code);
```

ARGUMENTS

dir_name
is the absolute pathname of the directory containing the entry. (Input)

entryname
is the name of the entry. (Input)

suffix_info_ptr
is a pointer to the suffix_info structure. (Input)

code
is a standard system status code. (Output)

NOTES

The suffix_info structure and the named constant SUFFIX_INFO_VERSION_1 are defined in the include file suffix_info.incl.pl1.

The suffix_info structure is defined as follows:

```
1 suffix_info                aligned based (suffix_info_ptr),
  2 version                  char (8),
  2 type                     char (32) unaligned,
  2 type_name                char (32) unaligned,
  2 plural_name              char (32) unaligned,
  2 flags                    unaligned,
    3 standard_object        bit (1) unaligned,
    3 extended_acl           bit (1) unaligned,
    3 has_switches           bit (1) unaligned.
    3 mbz1                   bit (33) unaligned,
  2 modes                    char (36),
  2 max_mode_len             fixed bin,
  2 num_ring_brackets        fixed bin,
  2 copy_flags               like copy_flags,
  2 info_pathname            char (168) unaligned;
```

STRUCTURE ELEMENTS

version
is the current version of this structure and has the value of the named constant SUFFIX_INFO_VERSION_1.

type
is either the suffix if extended, or one of the name constants for standard entry types found in the include file suffix_info.incl.pl1. (Input)

type_name
is the singular name of the entry type (e.g., "mailbox").

plural_type
is the plural name of the entry type (e.g., "mailboxes").

standard_object
is set to indicate that the entry is to be handled by fs_util_ itself.

extended_acl
is a switch indicating whether or not the entry type supports an extended Access Control List. The switch should be on if the type supports extended ACLs, and off otherwise.

has_switches
is on if the entry type supports the get_switch and set_switch entries.

mbz1
is reserved for future use and must be zero.

modes
is a string containing the access modes for the entry type. This string contains one character for each mode bit. The position of the character in the string indicates which bit in the ACL represents that mode.

max_mode_len
is the maximum number of modes on a single entry of this type. This is used by the list_acl command for formatting.

num_ring_brackets
is the number of ring brackets on an entry.

copy_flags
for its format, see the copy_flags structure described above under the copy entry point.

The flags configuration provided by suffix_info define what copy operations are valid for the extended entry type. During the copy operation, these flags are ANDed with the copy flags provided with the call to fs_util_. Only the operations allowed by suffix_info and requested by the copy call are performed. fs_util_ does not notify its caller that certain flags were ignored; however, the identity $copy these flags is computable via a call to suffix_info.

info_pathname
is the pathname of an info segment containing more information about the extended entry type, meanings of its modes, switches, and so forth.

Entry: fs_util_$suffix_info_for_type

This entry point returns information about the characteristics of an entry that is of a given type. It behaves exactly as the suffix_info entrypoint except that a directory and entry name are not used to determine the type for which suffix info is to be returned.

USAGE

```
declare fs_util_$suffix_info_for_type entry (char(*), ptr, fixed
      bin(35));

call fs_util_$suffix_info_for_type (type, suffix_info_ptr, code);
```

ARGUMENTS

type
: is either the suffix if extended, or one of the name constants for standard entry types found in the include file suffix_info.incl.pl1. (Input)

suffix_info_ptr
: is a pointer to the suffix_info structure. (Input)

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

NOTES

The suffix_info structure and the named constant SUFFIX_INFO_VERSION_1 are defined in the include file suffix_info.incl.pl1. The suffix_info structure is described in the suffix_info entrypoint above.

Name: ft_menu_

The ft_menu_ subroutine allows a FORTRAN program to use the Multics menu facility (menu_). Through ft_menu_ a FORTRAN program may create a menu object, display the menu, and get a user-entered selection from a menu. Once a menu object has been created, the FORTRAN program can use this menu object by referencing it via a menu-id returned to the caller when the menu object was created or when a stored menu object was retrieved.

The functionality available is provided through the various entry points defined below. Also refer to the FORTRAN include file at the end of this section.

Entry: ft_menu_$create

Utilized to create a menu object. It returns a menu identifier (menu_id) which is subsequently used to reference the menu object.

USAGE

declarations:

```
     character*n1   choices (m1)
     character*n2   headers (m2)
     character*n3   trailers (m3)
     character*1    keys (m4)
     character*1    pad_char
     integer        menu_format (6)
     integer        menu_needs (3)
     integer        menu_id
     integer        code

     call ft_menu_$create (choices, headers, trailers, pad_char,
           menu_format, key, menu_needs, menu_id, code)
```

STRUCTURE ELEMENTS

choices
: is an array of character variables which are the text of the options that the user wishes to display in the menu. (Input) n1 is the length, in characters, of the longest character string comprising the text of an option. m1 is the extent of the array, i.e., the number of options in the menu being described. This array must be at least of extent 1.

headers
: is an array of character variables to be displayed at the top of the menu. (Input) n2 is the length, in characters, of the longest header specified. m2 is the extent of the array, i.e., the number of headers (lines) desired. At least one header must be specified (if the first variable is set to blanks, no headers will be used).

trailers
: is an array of trailers (displayed immediately below the menu). (Input) n3, m3, are analogous to n2, m2 respectively.

menu_format
is an array, which specifies the format of the menu being created. (Input) Prior to calling this entry point, the FORTRAN programmer is responsible for setting the following variables:

```
menu_format(menu_version) = version number of menu_
        (currently, only version 1 is defined).
menu_format(max_width) = maximum width of the window
         on which the menu will be displayed.
menu_format(max_heigth) = maximum height of window
          on which menu is to be displayed.
menu_format(no_of_columns) = number of columns to be used
          by the menu manager to display the options.
menu_format(center_headers) = 0 or 1;  0 = no, 1 = yes.
menu_format(center_trailers) = 0 or 1;  0 = no, 1 = yes.
```

pad_char
is the character that the menu facility will display at the right and left of a centered header or trailer to fill out the line. (Input)

keys
is an array (maximum value of m4 is 61) that identifies the keystroke to be associated with each choice. (Input) This array must be at least as long as the number of choices in the menu. Each element in the array must be unique.

menu_needs
an array that contains menu related information on successful execution of call. (Output)

```
Returned information:

menu_needs(lines_needed)   the number of lines required
                           to display the menu.
menu_needs(width_needed)   the number of columns required
                           to display the menu.
menu_needs(no_of_options)  the number of options defined
                           in the menu.
```

menu_id
the menu identifier (i.e., the menu object "identifier"). (Output) It must not be altered in any way by the application program.

code
return code. (Output) (See Appendix B.)

Entry: ft_menu_$delete

Deletes a menu object from a given value segment. (See ft_menu_$store.)

USAGE

declarations:

```
     character*168  dir_name
     character*32   entry_name
     character*32   menu_name
     integer        code


     call ft_menu_$delete (dir_name, entry_name, menu_name, code)
```

STRUCTURE ELEMENTS

dir_name
pathname of the directory containing the menu object. (Input)

entry_name
entry name of value segment containing the menu object. (Input) The suffix "value" need not be specified.

menu_name
name used to identify the menu object when the menu object was stored. (Input)

code
return code. (Output) (See Appendix B.)

Entry: ft_menu_$describe

Returns information about a menu object. It returns the number of options in the menu, the number of lines and number of columns required to display the menu. It is primarily used to determine if the menu can be displayed in a given window.

USAGE

declarations:

```
     integer  menu_id
     integer  menu_needs(3)
     integer  code

     call ft_menu_$describe (menu_id, menu_needs, code)
```

STRUCTURE ELEMENTS

menu_id
: the menu identifier returned by ft_menu_$create or ft_menu_$retrieve. (Input)

menu_needs
: an array into which menu related information is returned. (Output)

```
Returned information:
menu_needs(lines_needed)        the number of lines required
                                to display the menu.

menu_needs(width_needed)        the number of columns needed
                                to display the menu.

menu_needs(no_of_options)       the number of options defined
                                in the menu.
```

code
: return code. (Output) (See Appendix B.)

Entry: ft_menu_$destroy

Invoked to delete a menu object from storage. (Not to be confused with ft_menu_$delete, which deletes the menu object from a value segment.) Deleting the menu object has no effect on the screen contents.

USAGE

```
declarations:

     integer  menu_id
     integer  code

     call ft_menu_$destroy (menu_id, code);
```

STRUCTURE ELEMENTS

menu_id
: menu identifier returned by ft_menu_$create or ft_menu_$retrieve. (Input/Output) Set to an invalid value on return to prevent the old menu_id from being accidentally used.

code
: return code. (Output) (See Appendix B.)

Entry: ft_menu_$display

Invoked to display a menu in a given window.

USAGE

```
declarations:

     integer  window_id
     integer  menu_id
     integer  code

     call ft_menu_$display (window_id, menu_id, code)
```

STRUCTURE ELEMENTS

window_id
: a window identifier returned by ft_window_$create. (Input) If usage_mode = 0 this argument will be ignored (see ft_menu_$init2).

menu_id
: menu identifier returned when the menu object was created or retrieved. (Input)

code
: return code. (Output) (See Appendix B.)

Entry: ft_menu_$get_choice

Returns the choice made by the user, i.e., an integer representing either the menu item chosen or the function key (or its equivalent escape sequence) entered.

USAGE

```
declarations:

     character*nl  function_key_info
     integer       window_id
     integer       menu_id
     integer       fkeys
     integer       selection
     integer       code

     call ft_menu_$get_choice (window_id, menu_id, function_key_info,
          fkeys, selection, code)
```

STRUCTURE ELEMENTS

window_id
a window identifier returned by ft_window_$create. (Input) If usage_mode = 0 this argument will be ignored. (see ft_menu_$init2)

menu_id
menu identifier returned by ft_menu_$create or ft_menu_$retrieve. (Input)

function_key_info
a character variable (nl as required) used to specify the role of function keys (if they exist for the terminal being used) or an equivalent set of escape sequences if the terminal does not have function keys or not the function keys required by the application. (Input) The objective is to let the application use the terminal's function keys if possible, else specify key sequences to be used to simulate function keys. Each character in the string corresponds to one function key. If the character is a space, then it is not relevant if the corresponding function key exists or not. If the character is not a space, that character will be used to simulate a function key if the terminal does not have function keys. If the terminal does not have a function key for every non-space character in the string, then function keys will be simulated. Thus, the string " ?p q" means that the caller does not care whether the terminal has function key 0 or 3, but the caller does wish to use function keys 1,2, and 4. If any of these 3 function keys is not present on the terminal, then esc-? will substitute for F1, esc-p will substitute for F2, and esc-q will substitute for F4.

fkeys
if fkeys = 1 user entered a function key or escape sequence if fkeys = 0 user selected an option (Output)

selection
is an integer representing the choice made by the user. (Output) If the user has chosen an option, it is a number between 1 and the highest defined option. If the user has entered a function key, or escape sequence simulating a function key, it is the number associated with the function key.

code
return code. (Output) (See Appendix B.)

Entries: ft_menu_$init1, ft_menu_$init2

These must be the first calls made to the menu manager. They set up the necessary environment for the menu application and return information concerning the user i/o window.

USAGE

```
declarations:
     integer   code
     integer   usage_mode

     call ft_menu_$init1 ()

     call ft_menu_$init2
          (usage_mode,user_window_lines,user_window_columns,
          user_window_id,code)
```

STRUCTURE ELEMENTS

usage_mode
: usage_mode = 0 means that the caller does not wish to do any window management at all. (Input) When he/she wishes to display a menu, the window required will be automatically created. This means that the application will operate in a two window mode, the window containing the menu, and the user_io window. Both windows will be managed automatically for the user. If the user specifies this mode, all calls to the ft_window_ subroutine will be ignored and will return an appropriate error code. See Error Code Handling (Appendix B), below. All calls to the ft_menu_ subroutine that require a window identifier will ignore the user provided window_id.

  usage_mode = 1 means that the user wishes to define the number and characteristics of the windows to be used in the application. Thus, calls to ft_window_ will be supported and, for the entry points of ft_menu_ that require a window identifier, the caller must use a legal window_id (returned by ft_window_$create).

user_window_lines
: the number of lines (rows) in the user i/o window at the time the user invokes ft_menu_$init (which must be the first call to the menu manager in the application). (Output) Undefined if usage_mode = 0.

user_window_columns
: the number of columns of the user i/o window when ft_menu_$init invoked. (Output) Undefined if usage_mode = 0.

user_window_id
: window identifier of the user i/o window. (Output) Undefined if usage_mode = 0.

code
: return code (See Appendix B.) (Output)

Entry: ft_menu_$list

Used to list the menu object(s) stored in value segment. The names selected are those that match a user provided string.

USAGE

```
declarations:
      character*168  dir_name
      character*32   names_array(m1)
      character*32   entry_name
      character*32   match_string
      integer        no_of_matches
      integer        code

      call ft_menu_$list (dir_name, entry_name, match_string,
            no_of_matches, names_array, code)
```

STRUCTURE ELEMENTS

dir_name
: pathname of directory containing the menu object. (Input)

entry_name
: entry name of value segment containing the menu object. (Input) The suffix "value" need not be specified.

match_string
: a character variable that is to be used as the selection criteria to determine what menu object, if any, is contained in the specified value segment that match (or contain) this string. (Input) If set to space(s), all names returned.

no_of_matches
: the number of matches found. (Output) If none, then is is 0.

names_array
: an array, of extent m1. (Output) The user should insure that m1 is sufficiently large to contain all matches that may be found. Contains the names of all menu objects, in the specified value segment, that match the character string match_string.

code
: return code. (Output) (See Appendix B.)

Entry: ft_menu_$retrieve

Used to retrieve a menu object previously stored via the ft_menu_$store. Once retrieved, the user can reference the menu object via the menu identifier (menu_id).

USAGE

```
declarations:
     character*168  dir_name
     character*32   entry_name
     character*32   menu_name
     integer        menu_id
     integer        code

     call ft_menu_$retrieve (dir_name, entry_name, menu_name, menu_id,
          code)
```

STRUCTURE ELEMENTS

dir_name
pathname of the directory containing the menu object. (Input)

entry_name
entry name of value segment containing menu object. (Input) The suffix "value" need not be specified.

menu_name
name of the menu object used when the object was stored. (Input)

menu_id
is the menu id returned by the call. (Output) It is used as the menu object identifier.

code
return code. (Output) (See Appendix B.)

Entry: ft_menu_$store

Used to store a menu object in a specified value segment.

USAGE

declarations:

```
character*168  dir_name
character*32   entry_name
character*32   menu_name
integer        create_seg
integer        menu_id
integer        code

call ft_menu_$store (dir_name,entry_name, menu_name, create_seg,
      menu_id, code)
```

STRUCTURE ELEMENTS

dir_name
pathname of directory into which the menu object is to be placed. (Input)

entry_name
entry name of value segment into which the menu object is to be placed. (Input) The suffix "value" need not be specified.

menu_name
it is the name to be assigned to the stored menu object. (Input)

create_seg
create_seg = 0 means do not store if value segment identified by entry_name does not already exist. (Input)

create_seg = 1 means create value segment, if it does not already exist, and store menu object in it.

menu_id
it is the menu object identifier returned when ft_menu_$create or ft_menu_$retrieve was called. (Input)

code
return code. (Output) (See Appendix B.)

Entry: ft_menu_$terminate

Must be the last call to the menu manager in the menu application. It will remove the special environment created by ft_menu_$init1 and ft_menu_$init2.

USAGE

declarations: none

```
call ft_menu_$terminate ()
```

FORTRAN INCLUDE FILE

This include file contains the following declarations:

```
external ft_menu_$create (descriptors)
external ft_menu_$delete (descriptors)
external ft_menu_$describe (descriptors)
external ft_menu_$destroy (descriptors)
external ft_menu_$display (descriptors)
external ft_menu_$get_choice (descriptors)
external ft_menu_$init1 (descriptors)
external ft_menu_$init2 (descriptors)
external ft_menu_$list (descriptors)
external ft_menu_$retrieve (descriptors)
external ft_menu_$store (descriptors)
external ft_window_$change (descriptors)
external ft_window_$create (descriptors)
external ft_window_$destroy (descriptors)

integer menu_version
integer max_width
integer max_height
integer no_of_columns
integer lines_needed
integer width_needed
integer no_of_options
integer center_headers
integer center_trailers
integer user_window_id
integer user_window_lines
integer user_window_columns

parameter (menu_version = 1)
parameter (max_width = 2)
parameter (max_height = 3)
parameter (no_of_columns = 4)
```

```
parameter (center_headers = 5)
parameter (center_trailers = 6)
parameter (lines_needed = 1)
parameter (width_needed = 2)
parameter (no_of_options = 3)
```

Name: ft_window_

This is the basic video interface subroutine to be used by FORTRAN to create/destroy/change windows. (This subroutine should not be called if usage_mode = 0 (see ft_menu_$init2)).

Its facilities are available through the following entry points.

Entry: ft_window_$change

This entry point is used to change the size of an existing window. The size of a window can always be "shrunk", however it can be increased only it does not overlap with another defined window. (This entry point should not be called if usage_mode = 0 (see ft_menu_$init2).)

USAGE

```
declarations:
      integer  window_id
      integer  first_line
      integer  height
      integer  code

      call ft_window_$change (window_id, first_line, heigth, code)
```

STRUCTURE ELEMENTS

window_id
: window identifier returned by ft_window_$create (or by ft_menu_$init in the case of the user i/o window). (Input)

first_line
: new first line number for the window being changed. (Input) Positive integer.

height
: new height for the window being changed. (Input) Positive integer.

code
return code. (Output) (See Appendix B.)

Entry: ft_window_$clear_window

Used to clear a specified window.

USAGE

```
declarations:
      integer       window_id
      integer       code

      call ft_window_$clear_window (window_id, code)
```

STRUCTURE ELEMENTS

window_id
The window identifier (returned by ft_window_$create) of the window to be cleared. (Input)

code
return code. (Output) (See Appendix B.)

Entry: ft_window_$create

Used to create a new window on the terminal screen. (This entry point should not be called if usage_mode = 0.) (see ft_menu_$init2)

USAGE

```
declarations:

      character*32  switch_name
      integer       window_id
      integer       first_line
      integer       height
      integer       code

      call ft_window_$create (first_line, height, switch_name,
            window_id, code)
```

STRUCTURE ELEMENTS

first_line
is the line number where the window is to start. (Input)

height
the number of lines used by the window, i.e., its height. (Input)

switch_name
the name that the caller wishes to associate with the switch. (Input) (The caller may use the switch name, for example, in the FORTRAN "open" statement.)

window_id
the returned id of the window just created. (Output) It must not be altered in any way by the application program.

code
return code. (Output) (See Appendix B.)

Entry: ft_window_$destroy

Used to destroy a previously created window. (This entry point should not be called if usage_mode = 0 (see ft_menu_$init2).)

USAGE

```
declarations:
      integer  window_id
      integer  code

      call ft_window_$destroy (window_id, code)
```

STRUCTURE ELEMENTS

window_id
window identifier (returned by the ft_window_$create). (Input/Output) It is reset to an illegal value by this call.

code
return code. (Output) (See Appendix B.)

FORTRAN MENU APPLICATION EXAMPLES

In the following two FORTRAN examples, a "Message" menu application is created that allows you to display, print, discard, or forward messages. Example 1 is a simple FORTRAN program that interfaces with the Multics menu manager via the ft_menu_ routine. Note in Example 1 that window management functions are called automatically through arguments in the ft_menu_$init2 subroutine.

Example 2 is a FORTRAN program that interfaces with the Multics menu manager through ft_menu_routine; in example 2, however, window management functions are performed by the ft_window_ routine.

EXAMPLE 1:

In this example, all window management is done automatically.

```
subroutine testcase1 ()

    %include ft_menu_dcls

    external ft_menu_$init1 (descriptors)
    external ft_menu_$init2 (descriptors)
    character*15   choices(6)
    character*12   headers(1)
    character*27   trailers(1)
    character*1    keys(6)
    character*168  dir_name
    character*32   entry_name
    character*32   menu_name
    character*12   function_key_info
    character*32   switch_name
    character*9    ME
    integer        create_seg
    integer        no_of_matches
    integer        window_id
    integer        fkeys
    integer        selection
    integer        usage_mode
    integer        menu_format(6)
    integer        menu_needs(3)
    integer        menu_id
    integer        code
    integer        zero

    external com_err_(descriptors)

    integer        too_few_keys
    integer        bad_arg
    integer        keys_not_unique
```

```
      ME = "testcase1"
      zero = 0

      choices(1) = "Display Message"
      choices(2) = "Print Message"
      choices(3) = "Discard Message"
      choices(4) = "Forward Message"
      choices(5) = "Reply Message"
      choices(6) = "List Messages"
      headers(1) = "READ MAIL"
      trailers(1) = "Press F1 (or esc-q) to quit"
      keys(1) = "1"
      keys(2) = "2"
      keys(3) = "3"
      keys(4) = "4"
      keys(5) = "5"
      keys(6) = "6"
      pad_char = "-"
      menu_format(menu_version) = 1
      menu_format(max_width) = 79
      menu_format(max_height) = 10
      menu_format(no_of_columns) = 2
      menu_format(center_headers) = 1
      menu_format(center_trailers) = 1


      code = 0
      usage_mode = 0   !  Window management will be done automatically
                       !  by the system, i.e., usage_mode is set to 0.
                       !  by the system, i.e., usage_mode is set to 0.
      call ft_menu_$init1 ()
      call ft_menu_$init2 (usage_mode,user_window_lines,user_window_columns,
                           user_window_id,code)
                         !  Calling ft_menu_$init MUST
                         !  be the first call to ft_menu_ in the program.

      if (code .eq. zero) go to 5
      call com_err_ (code, ME, " (calling ft_menu_$init2)")
      print, "Unable to set up the appropriate environment for the application."
      go to 999

c     The following calls to cv_error_$name are used retrieve and store
c     the error codes associated with certain errors of interest returned
c     by calls to the menu manager or the system.

5       call cv_error_$name ("error_table_$bad_arg", bad_arg, code)
      if (code .eq. zero) go to 10
      call com_err_ (code, ME, "error_table_$bad_arg")
      go to 999
10      call cv_error_$name ("menu_et_$too_few_keys",too_few_keys,code)
```

```
      if (code .eq. zero) go to 20
      call com_err_ (code, ME, "menu_et_$too_few_keys")
      go to 999
20      call cv_error_$name ("menu_et_$keys_not_unique", keys_not_unique, code)
      if (code .eq. zero) go to 40
      call  com_err_ (code, ME, "menu_et_$keys_not_unique")
      go to 999

40      call ft_menu_$create (choices,headers,trailers,pad_char,menu_format,
&                             keys,menu_needs,menu_id,code)

c   This call creates the menu object and returns the menu object identifier,
c   "menu_id".

      if (code .eq. zero) go to 45
      call com_err_ (code, ME, " (calling ft_menu_$ceate)")
      print, "The menu could not be created."
      go to 999

c   The created menu is now stored for future use.

45      dir_name = ">udd>m>ri"            ! pathname of directory
      entry_name = "menus_seg"          ! entry name of "value" segment
      menu_name = "ft_read_mail_menu"   ! name of menu
      create_seg = 1        ! create "value" seg if it does not already exist.

      call ft_menu_$store (dir_name, entry_name, menu_name,
                           create_seg, menu_id, code)
      if (code .eq. zero) go to 50

      call com_err_ (code, ME, " (calling ft_menu_$store)")
      print, "The menu could not be stored."
      go to 999
50      window_id = 0
      call ft_menu_$display(window_id,menu_id,code)   !  This call displays
            ! the menu in its own window at top of screen. Since the usage_mode
            ! was set to 0, the program does not have to create the window
            ! before calling ft_menu_$display. The window_id argument is ignored.

      if (code .eq. zero) go to 60
      call com_err_ (code, ME, " (calling ft_menu_$display)")
      print, "The menu could not be displayed."
      go to 999

60      function_key_info = "q"  ! Defines the function key requirements, i.e,
            !   if the terminal has function key 1 (F1) then F1 will be used
          !   to "quit", otherwise "esc_q" will be used to "quit".

61      call ft_menu_$get_choice(window_id,menu_id,function_key_info,fkeys,
&                                 selection,code)
```

```
c  This call accepts the user input from the menu. On return, the variable
c  "selection" will contain a number (1, 2, 3 , or 4) representing the option
c  chosen by user.
c  Note: if the user entered anything other than 1 or 2 or 3 or 4
c  the terminal "beeped", and the user input was ignored.
c  Since usage_mode is 0, the window_id argument is ignored.

      if (code .eq. zero) go to 90
      if (code .ne. too_few_keys)  go to 70
      call com_err_ (0, ME, "Number of keys less than number of options.")
      go to 999
70      if (code .ne. keys_not_unique) go to 80
      call com_err_ (0, ME, "Option keys not unique.")
      go to 999
80      call com_err_ (code, ME, " (calling ft_menu_$get_choice).
                   An internal programming error has occurred.")
      go to 999
90      if (fkeys .eq. zero) go to 110
      if (fkeys .eq. 1) go to 100
      print, "An internal program error has occurred. Quitting."
      go to 999
100     if (selection .ne. 1) go to 61
      print, "You entered ""F1"" or ""esc q"". Quitting."
      go to 999
110     print 103,selection
103     format ("You selected option "i1)
      go to 50

999 call ft_menu_$terminate()
    return
    end
```

EXAMPLE 2:

In this example, FORTRAN interfaces with the Multics menu manager and the Multics window manager via the ft_menu_ and ft_window_ subroutines.

```
subroutine testcase2 ()

      %include ft_menu_dcls

      external ft_menu_$init1(descriptors)
      external ft_menu_$init2(descriptors)
      external ft_window_$clear_window (descriptors)
      character*9     choices_one(2)
      character*21    choices_three(4)
      character*21    headers(1)
      character*49    trailers(1)
```

```
character*1     keys(6)
character*168   dir_name
character*32    entry_name
character*32    menu_name
character*12    function_key_info
character*32    match_string
character*32    names_array(10)
character*32    switch_name
character*9     ME
integer         create_seg
integer         no_of_matches
integer         window_id1
integer         window_id2
integer         fkeys
integer         selection
integer         usage_mode
integer         menu_format(6)
integer         menu_needs_one(3)
integer         menu_needs_two(3)
integer         menu_needs_three(3)
integer         curr_window_id
integer         menu_id1
integer         menu_id2
integer         menu_id3
integer         code
integer         zero

external com_err_(descriptors)

integer         bad_window_id
integer         nonexistent_window
integer         insuff_room_for_window

ME = "testcase2"
zero = 0

choices_one(1)  = "Read Mail"
choices_one(2)  = "Send Mail"
choices_three(1)  = "Send New Messsage"
choices_three(2)  = "Send Deferred Message"
choices_three(3)  = "Print Sent Message"
choices_three(4)  = "Save Sent Message"
trailers(1)= "F1 (or esc-q) = quit ; F2 (or esc-f) = first menu"
keys(1)  = "1"
keys(2)  = "2"
keys(3)  = "3"
keys(4)  = "4"
keys(5)  = "5"
keys(6)  = "6"
pad_char = "-"
```

```
      menu_format(menu_version) = 1
      menu_format(max_width) = 79
      menu_format(max_height) = 8
      menu_format(no_of_columns) = 2
      menu_format(center_headers) = 1
      menu_format(center_trailers) = 1

      code = 0
      call ft_menu_$initl ()
      usage_mode = 1     Window management will be done by user
      call ft_menu_$init2 (usage_mode,user_window_lines,user_window_columns,
&       user_window_id,code)      Calling ft_menu_$init MUST be the
                                first call to ft_menu_ in the program.

      if (code .eq. 0) go to 5
      call com_err_ (code, ME, " (calling ft_menu_$init)")
      print, "Unable to set up the appropriate environment for the
&              application."
      go to 999

c          The following calls to cv_error_$name are used retrieve and store
c     the error codes associated with certain errors of interest returned
c     by calls to the menu manager or the system.

5       call cv_error_$name ("video_et_$bad_window_id", bad_window_id, code)
      if (code .eq. zero) go to 10
      call com_err_ (code, ME, "video_et_$bad_window_id")
      go to 999
10      call cv_error_$name ("video_et_$nonexistent_window",
                              nonexistent_window,code)
      if (code .eq. zero) go to 20
      call com_err_ (code, ME, "video_et_$nonexistent_window")
      go to 999
20      call cv_error_$name ("video_et_$insuff_room_for_window",
&                               insuff_room_for_window, code)
      if (code .eq. zero) go to 40
      call  com_err_ (code, ME, "video_et_$insuff_room_for_window")
      go to 999

c     Create first menu

40      headers(1) = "MULTICS MAIL"
      call ft_menu_$create (choices_one,headers,trailers,pad_char,menu_format,
&                              keys,menu_needs_one,menu_id1,code)

c   This call creates the menu object and returns the menu object identifier.
c   This menu is referenced by menu_id1.
```

```
      if (code .eq. 0) go to 41
      call com_err_ (code, ME, " (calling ft_menu_$ceate)")
      print, "The first menu could not be created."
      go to 999

c   For the second menu use a menu object which was stored in a "value" seg.
c   First determine if menu object exists.

41    dir_name = ">udd>m>ri"
      entry_name = "menus_seg"
      match_string = "ft_read_mail_menu"
      call ft_menu_$list (dir_name,entry_name,match_string,no_of_matches,
&                         names_array,code)
      if (code .eq. zero) go to 42
      call com_err_ (code, ME, " (calling ft_menu_$list)")
      go to 999
42    if (no_of_matches .eq. zero) then
      print, "Stored menu not found."
      go to 999
      else
      if (no_of_matches .eq. 1) go to 43
      print, "Internal error. Quitting."
      go to 999
      end if

c   Retrieve stored menu.

43    menu_name = "ft_read_mail_menu"
      call ft_menu_$retrieve (dir_name,entry_name,menu_name,menu_id2,code)
      if (code .eq. zero) go to 44
      call com_err_ (code, ME, " (calling ft_menu_$retrieve)")
      go to 999

c   Get attributes of retrieved menu.

44    call ft_menu_$describe (menu_id2,menu_needs_two,code)
      if (code .eq. zero) go to 45
      call com_err_ (code, ME, " (calling ft_menu_$describe)")
      go to 999

c    Create third menu

45    headers(1) = "SEND MAIL"
      call ft_menu_$create (choices_three,headers,trailers,pad_char,
&                  menu_format,keys,menu_needs_three,menu_id3,code)

      if (code .eq. 0) go to 50
      call com_err_ (code, ME, " (calling ft_menu_$ceate)")
      print, "The third menu could not be created."
      go to 999
```

```
50    curr_window_id = -1     "-1" indicates that there is no current menu
                              being displayed; otherwise, curr_window_id
                              contains the menu window id

52    call change_menu (user_window_id,curr_window_id,menu_id1,menu_needs_one,
&                       user_window_lines,window_id1,code)
    if (code) 51,53,51
51    call com_err_ (code,"change_menu","Internal error while changing menus.")
    go to 999
53    call ft_window_$clear_window (user_window_id, code)

60    call get_choice (menu_id1,window_id1,fkeys,selection,code)

c  This call accepts the user input from the menu. On return, the variable
c  "selection" will contain a number (0, 1, 2) representing the option or
c  the function key (or its equivalent escape sequence) entered by the user.
c  If fkeys = 1 then the user entered F1 or F2 (or esc-q or esc-f):
c      if F1 (or esc-q) was entered, then selection = 0
c      if F2 (or esc-f) was entered, then selection = 1
c  If fkeys = 0 then the user selected option:
c      if first option was chosen, then selection = 1
c      if second option was chosen, then selection = 2
c  Note: if the user entered anything other than F1 or F2 or 1 or 2
c  the terminal "beeped", and the user input was ignored.

    if (code .eq. zero) go to 70
    call com_err_ (0, "get_choice", "Internal program error
                                    while getting user choice")
    go to 999
70    if (fkeys .eq. zero) go to 90    user selected an option
    if (fkeys .eq. 1) then
    go to 80    user entered function key
    else      Something is wrong
    print, "An internal program error has occurred. Quitting."
    go to 999
    end if
80    go to (81,82), selection
    call com_err_ (code, ME, "An internal program has occurred. Quitting.")
    go to 999
81    print, "Exiting"     (user has entered F1 or esc-q. Wants to exit)
    go to 999
82    print, "You already are in the first menu."    User want to go to
                                                     first menu
    go to 60
90    go to (100,170), selection   Display either "Read Mail" or "Send Mail"
                                   menu
    call com_err_ (code, ME, "Internal program error. Quitting.")
    go to 999
100   call change_menu (user_window_id,window_id1,menu_id2,menu_needs_two,
&                       user_window_lines, window_id2, code)
```

```
     if (code .eq. zero) go to 110
     call com_err_ (code, "change_menu", "Internal error occurred
                                          while switching menus")
     go to 999
110    call get_choice (menu_id2, window_id2, fkeys, selection, code)
     if (code .ne. zero) then
     call com_err_ (code, "get_choice", "Internal error
                                         while getting user choice")
     go to 999
     end if
     go to (160,150), fkeys + 1
     call com_err_ (code,ME, "Internal program error. Quitting.")
     go to 999
150    go to (151,152), selection     user entered function key
     go to 110
151    print, "Exiting at your request"
     go to 999
152    curr_window_id = window_id2
     go to 52
160    print 300, selection
300    format ("You selected option "i1)
     go to 110

c      User chose "Send Mail" option

170   call change_menu (user_window_id, window_id1,menu_id3,menu_needs_three,
&                       user_window_lines,window_id2,code)
     if (code) 171,180,171
171    call com_err_ (code, "change_menu", "Internal error
                          while changing menus")
     go to 999
180    call get_choice (menu_id3,window_id2,fkeys,selection,code)
     if (code) 181,190,181
181    call com_err_ (code, "get_choice", "Internal error
                          while getting user choice")
     go to 999
190    go to (210,200), fkeys + 1
     print, "Internal error. Quitting"
     go to 999
200    go to (201,202), selection
     go to 180
201    print, "Exiting at your request."
     go to 888
202    curr_window_id = window_id2
     go to 52
210    print 301, selection
301    format ("You selected option "i1)
     go to 180
c   Delete second menu from the value seg.
```

```
888   call ft_menu_$delete (dir_name,entry_name,menu_name,code)
    if (code .eq. zero) go to 999
    print, "Menu could not be deleted from value segment."
999   call ft_menu_$terminate()
    return
    end

subroutine get_choice (menu_id,window_id,fkeys,selection,code)

    external ft_menu_$get_choice (descriptors)

    character*2    function_key_info
    integer        fkeys
    integer        selection
    integer        menu_id
    integer        window_id
    integer        code

    code = 0

    function_key_info = "qf"    Defines the function key requirements, i.e,
           if the terminal has function keys 1 and 2 (F1 and F2) then F1
           will be used  to "quit" and F2 to switch to the first menu,
           otherwise "esc_q" will be used to "quit" and "esc-f" to switch
           to the first menu

    call ft_menu_$get_choice(window_id,menu_id,function_key_info,fkeys,
&                                 selection,code)
    return
    end

subroutine change_menu (user_window_id,curr_window_id,menu_id,menu_needs,
                        user_window_lines,window_id,code)

    external ft_window_$change (descriptors)
    external ft_window_$create (descriptors)
    external ft_window_$destroy (descriptors)
    external ft_menu_$display (descriptors)
    external com_err_ (descriptors)

    character*32  switch_name

    integer  menu_needs(3)
    integer  user_window_id
    integer  user_window_columns
    integer  user_window_lines
    integer  curr_window_id
    integer  menu_id
```

```
    integer   window_id
    integer   code
    integer   first_line
    integer   height

    parameter (lines_needed = 1)


c  Destroy the current menu-window

    if (curr_window_id + 1) 90,100,90
90    call ft_window_$destroy (curr_window_id,code)
    if (code) 999,100,999

c  Change the size of the user i/o window to accomodate the new menu-window

100   first_line = 1 + menu_needs(lines_needed)
    height = user_window_lines - menu_needs(lines_needed)
    call ft_window_$change (user_window_id,first_line,height,code)
    if (code) 999,110,999

c   Create window for new menu

110   switch_name = "menu_window"
    call ft_window_$create (1,menu_needs(lines_needed),switch_name,window_id,
&                             code)
    if (code) 999,120,999

c   Display the menu in the menu-window

120   call ft_menu_$display (window_id,menu_id,code)

999   return
    end
```

Name: generic_math_

The generic_math_ subroutine is used to perform basic arithmetic operations on the generic numeric data types. The operations that can be performed are: addition, subtraction, multiplication, division, and negation. There are separate entrypoints for each variation of the types: real and complex, binary and decimal.

Entry: generic_math_$negate_decimal

This entrypoint negates a generic decimal number.

USAGE

```
declare generic_math_$negate_decimal entry(bit(576), bit(576));

call generic_math_$negate_decimal (num1, result);
```

ARGUMENTS

num1
is a generic decimal number. (Input)

result
is the generic decimal value that is the negation of num1. (Output)

Entry: generic_math_$negate_decimal_complex

This entrypoint negates a generic complex decimal number.

USAGE

```
declare generic_math_$negate_decimal_complex entry(bit(1152),
      bit(1152));

call generic_math_$negate_decimal_complex (num1, result);
```

ARGUMENTS

num1
is a generic complex decimal number. (Input)

result
is the generic complex decimal value that is the negation of num1. (Output)

Entry: generic_math_$add_decimal

This entrypoint adds two generic decimal numbers.

USAGE

```
declare generic_math_$add_decimal entry(bit(576), bit(576), bit(576));

call generic_math_$add_decimal (num1, num2, result);
```

ARGUMENTS

num1
is a generic decimal number. (Input)

num2
is a generic decimal number. (Input)

result
is the generic decimal value that is the result of adding num1 and num2. (Output)

Entry: generic_math_$add_decimal_complex

This entrypoint adds two generic complex decimal numbers.

USAGE

```
declare generic_math_$add_decimal_complex entry(bit(1152), bit(1152),
     bit(1152));

call generic_math_$add_decimal_complex (num1, num2, result);
```

ARGUMENTS

num1
is a generic complex decimal number. (Input)

num2
is a generic complex decimal number. (Input)

result
is the generic complex decimal value that is the result of adding num1 and num2. (Output)

Entry: generic_math_$subtract_decimal

This entrypoint subtracts two generic decimal numbers.

USAGE

```
declare generic_math_$subtract_decimal entry(bit(576), bit(576),
     bit(576));

call generic_math_$subtract_decimal (num1, num2, result);
```

ARGUMENTS

num1
is a generic decimal number. (Input)

num2
is a generic decimal number. (Input)

result
is the generic decimal value that is the result of subtracting num1 and num2. (Output)

Entry: generic_math_$subtract_decimal_complex

This entrypoint subtracts two generic complex decimal numbers.

USAGE

```
declare generic_math_$subtract_decimal_complex entry(bit(1152),
      bit(1152), bit(1152));

call generic_math_$subtract_decimal_complex (num1, num2, result);
```

ARGUMENTS

num1
is a generic complex decimal number. (Input)

num2
is a generic complex decimal number. (Input)

result
is the generic complex decimal value that is the result of subtracting num2 from num1. (Output)

Entry: generic_math_$multiply_decimal

This entrypoint multiplies two generic decimal numbers.

USAGE

```
declare generic_math_$multiply_decimal entry(bit(576), bit(576),
      bit(576));

call generic_math_$multiply_decimal (num1, num2, result);
```

ARGUMENTS

num1
is a generic decimal number. (Input)

num2
is a generic decimal number. (Input)

result
is the generic decimal value that is the result of multiplying num1 and num2. (Output)

Entry: generic_math_$multiply_decimal_complex

This entrypoint multiplies two generic complex decimal numbers.

USAGE

```
declare generic_math_$multiply_decimal_complex entry(bit(1152),
      bit(1152), bit(1152));

call generic_math_$multiply_decimal_complex (num1, num2, result);
```

ARGUMENTS

num1
is a generic complex decimal number. (Input)

num2
is a generic complex decimal number. (Input)

result
is the generic complex decimal value that is the result of multiplying num1 by num2. (Output)

Entry: generic_math_$divide_decimal

This entrypoint divides two generic decimal numbers.

USAGE

```
declare generic_math_$divide_decimal entry(bit(576), bit(576),
      bit(576));

call generic_math_$divide_decimal (num1, num2, result);
```

ARGUMENTS

num1
: is a generic decimal number. (Input)

num2
: is a generic decimal number. (Input)

result
: is the generic decimal value that is the result of dividing num1 by num2. (Output)

Entry: generic_math_$divide_decimal_complex

This entrypoint divides two generic complex decimal numbers.

USAGE

```
declare generic_math_$divide_decimal_complex entry(bit(1152), bit(1152),
      bit(1152));

call generic_math_$divide_decimal_complex (num1, num2, result);
```

ARGUMENTS

num1
: is a generic complex decimal number. (Input)

num2
: is a generic complex decimal number. (Input)

result
: is the generic complex decimal value that is the result of dividing num1 by num2. (Output)

Entry: generic_math_$negate_binary

This entrypoint negates a generic binary number.

USAGE

```
declare generic_math_$negate_binary entry(bit(108), bit(108));

call generic_math_$negate_binary (num1, result);
```

ARGUMENTS

num1
is a generic binary number. (Input)

result
is the generic binary value that is the negation of num1. (Output)

Entry: generic_math_$negate_binary_complex

This entrypoint negates a generic complex binary number.

USAGE

```
declare generic_math_$negate_binary_complex entry(bit(252), bit(252));

call generic_math_$negate_binary_complex (num1, result);
```

ARGUMENTS

num1
is a generic complex binary number. (Input)

result
is the generic complex binary value that is the negation of num1. (Output)

Entry: generic_math_$add_binary

This entrypoint adds two generic binary numbers.

USAGE

```
declare generic_math_$add_binary entry(bit(108), bit(108), bit(108));

call generic_math_$add_binary (num1, num2, result);
```

ARGUMENTS

num1
is a generic binary number. (Input)

num2
is a generic binary number. (Input)

result
is the generic binary value that is the result of adding num1 and num2. (Output)

Entry: generic_math_$add_binary_complex

This entrypoint adds two generic complex binary numbers.

USAGE

```
declare generic_math_$add_binary_complex entry(bit(252), bit(252),
      bit(252));

call generic_math_$add_binary_complex (num1, num2, result);
```

ARGUMENTS

num1
: is a generic complex binary number. (Input)

num2
: is a generic complex binary number. (Input)

result
: is the generic complex binary value that is the result of adding num1 and num2. (Output)

Entry: generic_math_$subtract_binary

This entrypoint subtracts two generic binary numbers.

USAGE

```
declare generic_math_$subtract_binary entry(bit(108), bit(108),
      bit(108));

call generic_math_$subtract_binary (num1, num2, result);
```

ARGUMENTS

num1
: is a generic binary number. (Input)

num2
: is a generic binary number. (Input)

result
: is the generic binary value that is the result of subtracting num2 from num1. (Output)

Entry: generic_math_$subtract_binary_complex

This entrypoint subtracts two generic complex binary numbers.

USAGE

```
declare generic_math_$subtract_binary_complex entry(bit(252), bit(252),
      bit(252));

call generic_math_$subtract_binary_complex (num1, num2, result);
```

ARGUMENTS

num1
: is a generic complex binary number. (Input)

num2
: is a generic complex binary number. (Input)

result
: is the generic complex binary value that is the result of subtracting num2 from num1. (Output)

Entry: generic_math_$multiply_binary

This entrypoint multiplies two generic binary numbers.

USAGE

```
declare generic_math_$multiply_binary entry(bit(108), bit(108),
      bit(108));

call generic_math_$multiply_binary (num1, num2, result);
```

ARGUMENTS

num1
: is a generic binary number. (Input)

num2
: is a generic binary number. (Input)

result
: is the generic binary value that is the result of multiplying num1 by num2. (Output)

Entry: generic_math_$multiply_binary_complex

This entrypoint multiplies two generic complex binary numbers.

USAGE

```
declare generic_math_$multiply_binary_complex entry (bit(252), bit(252),
      bit(252));

call generic_math_$multiply_binary_complex (num1, num2, result);
```

ARGUMENTS

num1
is a generic complex binary number. (Input)

num2
is a generic complex binary number. (Input)

result
is the generic complex binary value that is the result of multiplying num1 by num2. (Output)

Entry: generic_math_$divide_binary

This entrypoint divides two generic binary numbers.

USAGE

```
declare generic_math_$divide_binary entry (bit(108), bit(108), bit(108));

call generic_math_$divide_binary (num1, num2, result);
```

ARGUMENTS

num1
is a generic binary number. (Input)

num2
is a generic binary number. (Input)

result
is the generic binary value that is the result of dividing num1 by num2. (Output)

Entry: generic_math_$divide_binary_complex

This entrypoint divides two generic complex binary numbers.

USAGE

```
declare generic_math_$divide_binary_complex entry(bit(252), bit(252),
      bit(252));

call generic_math_$divide_binary_complex (num1, num2, result);
```

ARGUMENTS

num1
  is a generic complex binary number. (Input)

num2
  is a generic complex binary number. (Input)

result
  is the generic complex binary value that is the result of dividing num1 by num2. (Output)

Name: get_bound_seg_info_

The get_bound_seg_info_ subroutine is used by several object display programs concerned with bound segments to obtain information about a segment as a bound segment as well as general object information.

USAGE

```
declare get_bound_seg_info_ entry (ptr, fixed bin(24), ptr, ptr, ptr,
      fixed bin(35));

call get_bound_seg_info_ (obj_ptr, bit_count, oi_ptr, bm_ptr, sblk_ptr,
      code);
```

ARGUMENTS

obj_ptr
  is a pointer to the beginning of the segment. (Input)

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

oi_ptr
is a pointer to the object format structure to be filled in by the object_info_$display entry point (see structure declaration in the description of the object_info_ subroutine). (Input)

bm_ptr
is a pointer to the bind map. (Output)

sblk_ptr
is a pointer to the base of the symbol block containing the bindmap. (Output)

code
is a standard status code. (Output)

NOTES

If obj_ptr points to an object segment but no bindmap is found, two possible codes are returned. One is error_table_$not_bound, indicating that the segment is not bound. The other is error_table_$oldobj, indicating that the segment was bound before the binder produced internal bind maps. If either one of these is returned, the structure pointed to by oi_ptr contains valid information.

Name: get_default_wdir_

The get_default_wdir_ function returns the pathname of the user's current default working directory.

USAGE

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

default_wdir = get_default_wdir_ ();
```

ARGUMENTS

default_wdir
is the pathname of the user's current default working directory. (Output)

Name: get_definition_

The get_definition_ subroutine returns a pointer to a specified definition within an object segment.

USAGE

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

call get_definition_ (def_section_ptr, segname, entryname, def_ptr,
      code);
```

ARGUMENTS

def_section_ptr
: is a pointer to the definition section of the object segment. This pointer can be obtained via the object_info_ subroutine. (Input)

segname
: is the name of the object segment. (Input)

entryname
: is the name of the desired entry point. (Input)

def_ptr
: is a pointer to the definition for the entry point. (Output)

code
: is a standard status code. If the entry point is found, code is 0. (Output)

Name: get_ec_version_

The get_ec_version_ subroutine returns the version number of an exec_com, and the character position of the first character after the &version statement, if any.

USAGE

```
dcl get_ec_version_ entry (char(*), char(*), fixed bin, fixed bin(21),
      fixed bin(35));

call get_ec_version_ (dn, en, version, text_pos, code);
```

ARGUMENTS

dn
is the directory containing the exec_com. (Input)

en
is the name of the exec_com. (Input)

version
is the version number of the exec_com. (Output)

text_pos
is the character position of the first character following the &version statement, if any, or 1. (Output)

code
is a standard status code. (Output)

ACCESS REQUIRED

The user must have read access on the exec_com.

Name: get_entry_arg_descs_

This subroutine returns information about the calling sequence of a procedure entry point. Archive component pathnames are supported.

Entry: get_entry_arg_descs_$get_entry_arg_desc_

The get_entry_arg_descs_ entry point, given a pointer to the entry sequence or segdef of a procedure entry point, returns a list of argument descriptors describing the parameters of the entry point.

USAGE

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

call get_entry_arg_descs_ (entry_ptr, nargs, desc_ptrs, code);
```

ARGUMENTS

entry_ptr
points to the entry sequence or segdef of the procedure entry point whose parameter descriptors are to be described. (Input)

nargs
is the number of parameters declared in the procedure entry point. (Output)

desc_ptrs
is an array of pointers to the argument descriptors describing the declared parameters of the entry point. If dimension (desc_ptrs, 1) is less than nargs, the pointers identify the first dimension (desc_ptrs, 1) parameter descriptors. (Output)

code
is a standard status code. It can be:
error_table_$nodescr
the entry point did not have parameter descriptors. (Output)

NOTES

For some version 0 object segments, a code of zero is returned, nargs is set, but the descriptor pointers in desc_ptrs are null.

Entry: get_entry_arg_descs_$info

This entry point, given a pointer to the entry sequence or segdef of a procedure entry point, returns a list of argument descriptors describing the parameters of the entry point, plus a set of entry sequence flags which further describe the entry point.

USAGE

```
declare get_entry_arg_descs_$info entry (ptr, fixed bin, (*) ptr, ptr,
     fixed bin(35));

call get_entry_arg_descs_$info (entry_ptr, nargs, desc_ptrs,
     entry_desc_info_ptr, code);
```

ARGUMENTS

entry_ptr
points to the entry sequence or segdef of the procedure entry point whose parameter descriptors are to be described. (Input)

nargs
is the number of parameters declared in the procedure entry point. (Output)

desc_ptrs
is an array of pointers to the argument descriptors describing the declared parameters of the entry point. If dimension (desc_ptrs, 1) is less than nargs, the pointers identify the first dimension (desc_ptrs, 1) parameter descriptors. (Output)

entry_desc_info_ptr
points to the entry_desc_info structure described under "Notes" below. (Input)

code
is a standard status code. It can be:
error_table_$nodescr
the entry point did not have parameter descriptors. (Output)

NOTES

The entry_desc_info_ptr argument of get_entry_arg_descs_$info points to the structure shown below. This structure is declared in entry_desc_info.incl.pl1.

```
dcl 1 entry_desc_info           aligned based(entry_desc_info_ptr),
      2 version                 fixed bin,
      2 flags,
       (3 basic_indicator,
        3 revision_1,
        3 has_descriptors,
        3 variable,
        3 function)             bit(1) unaligned,
        3 pad                   bit(13) unaligned,
      2 object_ptr              ptr,
      2 bit_count               fixed bin(24);

dcl entry_desc_info_version_2  fixed bin int static
                                 options(constant) init(2),
    entry_desc_info_ptr         ptr;
```

STRUCTURE ELEMENTS

version
is the version number of this structure. The current version number is 2. The named constant, entry_desc_info_version_2, should be used to set this version number.

flags
are the flags which further describe the procedure entry point.

basic_indicator
is on if the entry point is in a program written in the BASIC language.

revision_1
is on if the entry sequence has version 1 descriptor data.

has_descriptors
is on if the entry sequence has argument descriptors describing its parameters.

variable
is on if the entry point accepts an undefined number of arguments, and has been declared with the options(variable) attribute. This flag will usually be off for entry points in command and active function procedures, even though these procedures accept a variable number of arguments. Command and active function procedures usually do not declare their entry points with explicit parameters or with the options(variable) attribute.

function
is on if the procedure entry point is a function which returns a value. The final parameter argument descriptor describes this return value.

object_ptr
if the entry descriptor is being taken from an archive, this is the pointer to the base of the archive component. Otherwise, this is null. (Output)

bit_count
if the entry descriptor is being taken from an archive, this is the bit count of the archive component. Otherwise, this is zero. (Output)

entry_desc_info_version_2
is a named constant which the caller should use to set the version number in the structure above.

entry_desc_info_ptr
points to the structure above.

Entry: get_entry_arg_descs_$text_only

This entry point, given a pointer to the entry sequence of a procedure entry point, returns a list of argument descriptors describing the parameters of the entry point. It differs from the get_entry_arg_descs_ entry point, in that it assumes that it is given a pointer to an entry sequence in the text section of the procedure, rather than checking to see if it was given a pointer to a segdef.

USAGE

```
declare get_entry_arg_descs_$text_only entry (ptr, fixed bin, (*) ptr,
     fixed bin(35));

call get_entry_arg_descs_$text_only (entry_ptr, nargs, desc_ptrs, code);
```

ARGUMENTS

The arguments are the same as for the get_entry_arg_descs_ entry point above. If entry_ptr does not point to an entry point in the text section, then error_table_$nodescr is returned as the value of code.

Entry: get_entry_arg_descs_$text_only_info

This entry point, given a pointer to the entry sequence of a procedure entry point, returns a list of argument descriptors describing the parameters of the entry point, plus a set of entry sequence flags which further describe the entry point. It differs from the get_entry_arg_descs_$info entry point, in that it assumes that it is given a pointer to an entry sequence in the text section of the procedure, rather than checking to see if it was given a pointer to a segdef.

USAGE

```
declare get_entry_arg_descs_$text_only_info entry (ptr, fixed bin, (*)
      ptr, ptr, fixed bin(35));

call get_entry_arg_descs_$text_only_info (entry_ptr, nargs, desc_ptrs,
      entry_desc_info_ptr, code);
```

ARGUMENTS

The arguments are the same as for the get_entry_arg_descs_$info entry point above.

Name: get_entry_name_

The get_entry_name_ subroutine, given a pointer to an externally defined location or entry point in a segment, returns the associated name.

USAGE

```
declare get_entry_name_ entry (ptr, char(*), fixed bin(18), char(8)
      aligned, fixed bin(35));

call get_entry_name_ (entry_ptr, symbolname, segno, lang, code);
```

ARGUMENTS

entry_ptr
is a pointer to a procedure entry point. (Input)

symbolname
is the name corresponding to the location specified by entry_ptr. The maximum length is 256 characters. (Output)

segno
is the segment number of the object segment where symbolname is found. It is useful when entry_ptr does not point to a text section. (Output)

lang
is the language in which the segment or component pointed to by entry_ptr was compiled. (Output)

code
is a standard status code. (Output)

Name: get_entry_point_dcl_

The get_entry_point_dcl_ subroutine returns attributes needed to construct a PL/I declare statement for external procedure entry points and for error_table_ codes and other system-wide external data. The program obtains the attributes from data files declaring all unusual procedure entry points (e.g., ALM segments), and system-wide data values (e.g., sys_info$max_seg_size), and from the argument descriptors describing the entry point's parameters that are included with the entry point itself.

The get_entry_point_dcl_ entry point returns the declaration for an external value, either from one of the data files, or by using the parameter argument descriptors associated with the procedure entry point. It makes a special case of error_table_ values by always returning 'fixed bin(35) ext static' for them. For example, given the name iox_$put_chars, it might return:

```
entry (ptr, ptr, fixed bin(21), fixed bin(35))
```

Note that neither the name of the external value nor any trailing semicolon (;) is returned as part of the declaration.

Archive component pathnames are supported.

USAGE

```
dcl get_entry_point_dcl_ entry (char(*), fixed bin, fixed bin,
      char(*) varying, char(32) varying, fixed bin(35));

call get_entry_point_dcl_ (name, dcl_style, line_length, dcl, type,
      code);
```

ARGUMENTS

name
: is the name of the external entry point or data item whose declaration must be obtained. (Input)

dcl_style
: is the style of indentation to be performed for the name. See "Notes" below for a list of allowed values. (Input)

line_length
: is the maximum length to which lines in return value are allowed to grow when indentation is performed. (Input)

dcl
: is the declaration that was obtained. (Output)

type
: is the type of declaration. In the current implementation, this is always a null string. (Output)

code
: is a standard status code describing any failure to obtain the declaration. (Output)

NOTES

Three styles of declaration indentation are supported by the dcl_style argument described above. Style 0 (dcl_style = 0) involves no indentation. The declaration is returned as a single line.

Style 1 (dcl_style = 1) indents the declaration in the format similar to the indent command. Long declarations are broken into several lines. For example, a declare statement for hcs_$initiate_count would appear as:

```
dcl  hcs_$initiate_count entry (char(*), char(*), char(*),
     fixed bin(24), fixed bin(2), ptr, fixed bin(35));
```

when the string "dcl hcs_$initiate_count" is concatenated with the value returned by get_entry_point_dcl_, and a semicolon (;) is appended to this value.

Style 2 (dcl_style = 2) indents the declaration in an alternate format that makes the name of the entry point stand out from its declaration. It assumes that the name of the entry point begins in column 11 (indented one horizontal tab stop from left margin), and the declaration begins in column 41. In style 2, the declare statement for hcs_$initiate_count would appear as:

```
dcl       hcs_$initiate_count             entry (char(*), (char(*), (char(*),
                                                   fixed bin(24), fixed bin(2),
                                                   ptr, fixed bin(35));
```

Most command and active function entry points do not declare arguments in their procedure statements since they accept a variable number of arguments. Neither do they use the options(variable) attribute in their procedure statements. Therefore, when get_entry_point_dcl_ encounters a procedure entry point with no declared arguments and without options(variable), it assumes the options(variable) attribute required for commands and active functions and returns:

```
entry options(variable)
```

This page intentionally left blank.

It distinguishes between such assumed options(variable) entries and those that explicitly use the options(variable) attribute in their procedure statement by returning "entry" for the assumed case and "entry()" for the explicit case. Thus, for the display_entry_point_dcl command, which explicitly uses options(variable) in its procedure statement, get_entry_point_dcl_ returns:

```
entry() options(variable)
```

For procedures which use structures as arguments, certain structure declarations are inexactly returned as parameter declarations because the mechanism for encoding argument descriptors does not provide an adequate description of the alignment of a structure. The descriptor only determines whether the overall structure is packed or not, and does not specify whether or not it was originally declared with the aligned attribute.

The following structures generate the same argument descriptors, even though PL/I treats the level 1 structures as having different attributes:

```
dcl 1 s2 structure aligned,
      2 el1 fixed bin aligned,
      2 el2 fixed bin aligned;

dcl 1 s2 structure,
      2 el1 fixed bin aligned,
      2 el2 fixed bin aligned;
```

get_entry_point_dcl_ reproduces the declaration for s2 when either s1 or s2 are used as parameters for an entry point. In order to bypass this problem, declare the subroutine properly in your personal .dcl segment (see "User-Provided Data Files" below), and place this segment in your "declare" search paths.

SEARCH LIST

The get_entry_point_dcl_ subroutine uses the "declare" search list, which has the synonym "dcl", to find data files describing unusual procedure entry points. For more information about search lists, see the descriptions of the search facility commands and, in particular, the add_search_paths command description. Type:

```
!  print_search_paths declare
```

to see what the current declare search list is. The default search list identifies the data file:

```
>sss>pl1.dcl
```

USER-PROVIDED DATA FILES

Users may provide data files that redeclare standard system entry points (e.g., redeclaring a subroutine as a function), or that declare their own entry points or external data items. The add_search_paths command can be used to place user-provided data files in the "declare" search list. For example:

```
! add_search_paths declare [hd]>my_pl1.dcl -first
```

Declarations have the general form of:

```
virtual_entry declaration
```

For example:

```
ioa_ entry options(variable)
```

Note that the word "dcl" is not included in the data item, nor does the declaration end with a semicolon (;). External data values are declared in a similar fashion. For example:

```
iox_$user_output ptr external static
```

Name: get_equal_name_

The get_equal_name_ subroutine accepts an entryname and an equal name as its input and constructs a target name by substituting components or characters from the entryname into the equal name, according to the Multics equal convention. Refer to "Constructing and Interpreting Names" in the Programmer's Reference Manual for a description of the equal convention and for the rules used to construct and interpret equal names.

USAGE

```
declare get_equal_name_ entry (char (*), char (*), char (32),
     fixed bin(35));

call get_equal_name_ (entryname, equal_name, target_name, code);
```

ARGUMENTS

entryname
: is the entryname from which the target is to be constructed. Trailing blanks in the entryname character string are ignored. (Input)

equal_name
: is the equal name from which the target is to be constructed. Trailing blanks in the equal name character string are ignored. (Input)

target_name
: is the target name that is constructed. (Output)

code
: is a standard status code. (Output) It can be one of the following:
  error_table_$bad_equal_name
    the equal name has a bad format.
  error_table_$badequal
    there is no letter or component in the entryname that corresponds to a percent character (%) or an equal sign (=) in the equal name.
  error_table_$longeql
    the target name to be constructed is longer than 32 characters.

NOTES

If the error_table_$badequal status code is returned, then a target_name is returned in which null character strings are used to represent the missing letter or component of entryname.

If the error_table_$longeql status code is returned, then the first 32 characters of the target name to be constructed are returned as target_name.

The entryname argument that is passed to get_equal_name_ can also be used as the target_name argument, as long as the argument has a length of 32 characters.

Entry: get_equal_name_$component

This entry point accepts an archive and component name and two equal names as input and constructs a target archive and component name by substituting components or characters from the archive and component names into the equal names, according to the Multics archive component pathname equal convention. Refer to the Programmer's Reference Manual for a description of archive component pathnames and the equal convention.

USAGE

```
declare get_equal_name_$component entry (char (*), char (*), char (*),
      char (*), char (32), char (32), fixed bin(35));

call get_equal_name_$component (entryname, component, equal_entryname,
      equal_component, target_entryname, target_component, code);
```

ARGUMENTS

entryname
is the archive name from which the target archive name is constructed, or is the entryname from which the target component name is constructed if the source pathname is not an archive component pathname. (Input)

component
is the component name from which the target component name is constructed or is a null string if the source pathname is not an archive component pathname. (Input)

equal_entryname
is the equal name from which the target archive name is constructed or is the equal name from which the target entryname is constructed if the target pathname is not an archive component pathname. (Input)

equal_component
is the equal name from which the target component name is constructed or is a null string if the target pathname is not an archive component pathname. (Input)

target_entryname
is the target archive name that is constructed or is the target entryname that is constructed if the target pathname is not an archive component pathname. (Output)

target_component
is the target component name that is constructed or is a null string if the target pathname is not an archive component pathname. (Output)

code
is a standard status code. (Output) It can be one of the following:
error_table_$bad_equal_name
either equal_entryname or equal_component has a bad format.
error_table_$badequal
there is no letter or component in the archive or component name that corresponds to a percent character (%) or an equal sign (=) in the appropriate equal name.

error_table_$longeql
: the target archive or component name to be constructed is longer than 32 characters.

error_table_$no_archive_for_equal
: the target pathname has an equal name in the archive name position but the source pathname is not an archive component pathname.

NOTES

If the error_table_$badequal status code is returned, the name returned in the appropriate output argument is constructed using null character strings to represent the letters or component names missing from the source name.

If the error_table_$longeql status code is returned, the first 32 characters of the constructed name are returned in the appropriate output argument.

The two pairs of input arguments to this subroutine are expected to be the output arguments from two calls to expand_pathname_$component, one call for the source pathname and one for the pathname containing the equal names.

The output arguments of this subroutine should be used in a call to the initiate_file_$component subroutine. For example:

```
call expand_pathname_$component (arg1, source_dir, source_ename,
          source_comp, code);
if code ^= 0 then ...

call expand_pathname_$component (arg2, target_dir, equal_entry,
          equal_component, code);
if code ^= 0 then ...

call get_equal_name_$component (source_ename, source_comp, equal_entry,
          equal_component, target_ename, target_comp, code);
if code ^= 0 then ...

call initiate_file_$component (source_dir, source_ename, source_comp,
          R_ACCESS, source_ptr, source_bit_count, code);
if code ^= 0 then ...

call initiate_file_$component (target_dir, target_ename, target_comp,
          R_ACCESS, target_ptr, target_bit_count, code);
if code ^= 0 then ...
```

Name: get_external_variable_

The get_external_variable_ subroutine obtains the location and size of an external variable.

USAGE

```
declare get_external_variable_ entry (char(*), ptr, fixed bin(19), ptr,
     fixed bin(35));

call get_external_variable_ (vname, vptr, vsize, vdesc_ptr, code);
```

ARGUMENTS

vname
: is the name of the external variable. (Input)

vptr
: is a pointer to the current allocation of the external variable. (Output)

vsize
: is the size (in words) of the external variable. (Output)

vdesc_ptr
: is a pointer to a standard argument descriptor array describing the external variable. If the external variable does not have descriptor information associated with it, a null pointer is returned. (Output)

code
: is a standard status code. (Output)

Name: get_group_id_

The get_group_id_ function 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_ ();
```

ARGUMENTS

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

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 ();
```

ARGUMENTS

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

Name: get_initial_ring_

The get_initial_ring_ subroutine returns the current value of the ring number in which the process was initialized.

USAGE

```
declare get_initial_ring_ entry (fixed bin(3));

call get_initial_ring_ (i_ring);
```

ARGUMENTS

i_ring
is the initial ring for the process. (Output)

Name: get_line_length_

The get_line_length_ function 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);

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

ARGUMENTS

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

code
: is a standard status code. (Output)

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);

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

ARGUMENTS

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

code
: is a standard status code. (Output)

line_length
    is the line length of switch_name. (Output)

Name: get_lock_id_

The get_lock_id_ subroutine returns the 36-bit unique lock identifier to be used by a process in setting locks. By using this lock identifier, a convention can be established so that a process wishing to lock a data base and finding it already locked can verify that the lock is set by an existing process.

USAGE

```
declare get_lock_id_ entry (bit(36) aligned);

call get_lock_id_ (lock_id);
```

ARGUMENTS

lock_id
    is the unique identifier of this process used in locking. (Output)

NOTES

For a more detailed discussion of locking see the set_lock_ subroutine description.

Name: get_pdir_

The get_pdir_ function returns the absolute pathname of the user's process directory. For a discussion of process directories, see the Programmer's Reference Manual.

USAGE

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

process_dir = get_pdir_ ();
```

ARGUMENTS

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

Name: get_privileges_

The get_privileges_ function returns the access privileges of the process. (See "Access Control" in the Programmer's Reference Manual for more information on access privileges.)

USAGE

```
declare get_privileges_ entry returns (bit(36) aligned);

privilege_string = get_privileges_ ();
```

ARGUMENTS

privilege_string
is a bit string with a bit set ("1"b) for each access privilege the process has. (Output)

NOTES

The individual bits in privilege_string are defined by the following PL/I structure:

```
dcl 1 privileges    unaligned,
      2 ipc         bit(1),
      2 dir         bit(1),
      2 seg         bit(1),
      2 soos        bit(1),
      2 ring1       bit(1),
      2 rcp         bit(1),
      2 mbz         bit(30);
```

STRUCTURE ELEMENTS

ipc
indicates whether the access isolation mechanism (AIM) restrictions for sending/receiving wakeups to/from any other process are bypassed for the calling process.
"1"b yes
"0"b no

dir
indicates whether the AIM restrictions for accessing any directory are bypassed for the calling process.
"1"b yes
"0"b no

seg
    indicates whether the AIM restrictions for accessing any segment are bypassed for the calling process.
    "1"b yes
    "0"b no

soos
    indicates whether the AIM restrictions for accessing directories that have been set security-out-of-service are bypassed for the calling process.
    "1"b yes
    "0"b no

ring1
    indicates whether the AIM restrictions for accessing any ring 1 system segment are bypassed for the calling process.
    "1"b yes
    "0"b no

rcp
    indicates whether the AIM restrictions for accessing resources through RCP resource management are bypassed for the calling process.
    "1"b yes
    "0"b no

mbz
    is unused and is "0"b.

Name: get_process_access_class_

The get_process_access_class_ function returns the AIM access class contained in the current process authorization.

USAGE

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

access_class = get_process_access_class_ ();
```

ARGUMENTS

access_class
    is the access class derived from the process login authorization. (Output)

Name: get_process_authorization_

The get_process_authorization_ function returns the process' current authorization. This includes the login authorization and any privileges that have been enabled.

USAGE

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

authorization = get_process_authorization_ ();
```

ARGUMENTS

authorization
    is is the current process authorization, including privileges. (Output)

Name: get_process_id_

The get_process_id_ function 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_ ();
```

ARGUMENTS

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

Name: get_process_max_authorization_

The get_process_max_authorization_ function returns the maximum AIM authorization of the process. See the Programmer's Reference Manual for additional information on AIM.

USAGE

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

max_authorization = get_process_max_authorization_ ();
```

ARGUMENTS

max_authorization
is the returned maximum authorization. (Output)

Name: get_ring_

The get_ring_ function returns to the caller the number of the protection ring in which the caller is executing. For a discussion of rings see "Intraprocess Access Control" in the Programmer's Reference Manual.

USAGE

```
declare get_ring_ entry returns (fixed bin(3));

ring_no = get_ring_ ();
```

ARGUMENTS

ring_no
is the number of the ring in which the caller is executing. (Output)

Name: get_shortest_path_

Shortens the specified pathname by replacing each directory level with the shortest name on the directory. If the caller does not have access to get the names of a directory, the original name of that directory is left intact.

USAGE

```
dcl get_shortest_path_ entry (char (*)) returns (char (168));

short_path = get_shortest_path_ (original_path);
```

ARGUMENTS

original_path
is the pathname of a storage system entry. (Input)

NOTES

When more than one name qualify as the shortest name for a directory, an attempt is made to select the name containing all lower case characters. If more than one name still qualifies, these names are compared to the primary name of the directory. The first name found with the same first character as the primary name is chosen. This comparison is case independent.

Name: get_system_aim_attributes_

This subroutine returns a structure describing the AIM attributes defined on this system.

USAGE

```
declare get_system_aim_attributes_ entry (ptr, char(8), ptr,
      fixed bin(35));

call get_system_aim_attributes_ (area_ptr, version_wanted,
      aim_attributes_ptr, code);
```

ARGUMENTS

area_ptr
: is a pointer to an area in which the aim_attributes structure is allocated. (Input)

version_wanted
: is the version of the structure that the caller expects get_system_aim_attributes_ to return. The only supported version at present is given by the value of the named constant AIM_ATTRIBUTES_VERSION_1 defined in the system include file aim_attributes.incl.pl1. (Input)

aim_attributes_ptr
: is set to locate the aim_attributes structure allocated by this program. (Output)

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

  0
  : the aim_attributes structure was successfully allocated.

  error_table_$unimplemented_version
  : the version of the structure requested by the caller is not implemented by get_system_aim_attributes_.

  error_table_$noalloc
  : there was not sufficient room in the caller's area to allocate the aim_attributes structure.

NOTES

The aim_attributes structure is defined in the system include file aim_attributes.incl.pl1 and has the following format:

```
dcl 1 aim_attributes               aligned based,
      2 version                    char (8) unaligned,
      2 access_class_ceiling       bit(72),
      2 levels (0 : 7),
        3 long_name                char (32) unaligned,
        3 short_name               char (8) unaligned,
      2 categories (18),
        3 long_name                char (32) unaligned,
        3 short_name               char (8) unaligned,
```

STRUCTURE ELEMENTS

version
    is the version of this structure (currently AIM_ATTRIBUTES_VERSION_1).

access_class_ceiling
    is the maximum authorization or access class in terms of the AIM attributes.

levels
    are the sensitivity levels defined on this system. Only the entries from levels(0) through levels(highest_level) contain definitions. The remaining entries are all blank.

    long_name
        is the long name of this sensitivity level.

    short_name
        is the short name of this sensitivity level.

categories
    are the access categories defined on this system. Only the first n_categories entries of this substructure contain definitions. The remaining entries are all blank.

    long_name
        is the long name of this sensitivity level.

    short_name
        is the short name of this sensitivity level.

Name: get_system_free_area_

The get_system_free_area_ function returns a pointer to the system free area for the ring in which it was called. Allocations by system programs are performed in this area.

USAGE

```
declare get_system_free_area_ entry returns (ptr);

area_ptr = get_system_free_area_ ();
```

ARGUMENTS

area_ptr
    is a pointer to the system free area. (Output)

Name: get_temp_segment_

The get_temp_segment_ subroutine acquires a temporary segment in the process directory. The segment returned to the caller is 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 of creating a segment each time one is needed.

If more than one temporary segment is required, use the get_temp_segments_ subroutine.

USAGE

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

call get_temp_segment_ (program, temp_seg_ptr, code);
```

ARGUMENTS

program
    is a 32-character field identifying the program on whose behalf the temporary segment is to be used. This field is displayed by the list_temp_segments command. Besides giving the name of the command or subroutine invoked by the user, it can also briefly describe how the temporary segment is used; for example, "sort_seg (sort indexes)". (Input)

temp_seg_ptr
    is a returned pointer to the requested temporary segment. (Output)

code
is a standard status code. (Output)

NOTES

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

The list_temp_segments command can be used to list the temporary segments being used by a process.

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 of creating a segment each time one is needed.

USAGE

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

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

ARGUMENTS

program
is a 32-character field identifying the program on whose behalf the temporary segment is to be used. This field is displayed by the list_temp_segments command. Besides giving the name of the command or subroutine invoked by the user, it can also briefly describe how the temporary segment is used; for example, "sort_seg (sort indexes)". (Input)

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

code
is a standard 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.NNNN suffix, where NNNN is the segment number of the segment in octal. See the description of the release_temp_segments_ or the release_temp_segment_ 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.

The list_temp_segments command (described in the the Commands manual) can be used to list the temporary segments being used by a process.

Name: get_wdir_

The get_wdir_ function returns the absolute pathname of the user's current working directory. For a discussion of working directories, see "System Directories" in the Programmer's Reference Manual.

USAGE

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

declare working_dir character (168);

working_dir = get_wdir_ ();
```

ARGUMENTS

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

NOTES

Working directories are per-ring. If get_wdir_ is invoked in a ring for which a working directory has never been set, it will use the sub_err_ mechanism to signal an error (see the sub_err_ subroutine). The sub_err_ action code given is "ACTION_CANT_RESTART". The status code is error_table_$no_wdir. See the Programmer's Reference Manual for more information on ring protection.

Name: hash_

The hash_ subroutine is used to maintain a hash table. It contains entry points that initialize a hash table and insert, delete, and search for entries in the table.

A hash table is used to locate entries in another data table when the length of the data table or the frequency with which its entries are referenced makes linear searching uneconomical.

A hash table entry contains a name and a value. The name is a character string (of up to 32 characters) that is associated in some way with a data table entry. The value is a fixed binary number that can be used to locate that data table entry (for example, an array index or an offset within a segment). The entries in the hash table are arranged so that the location of any entry can be computed by applying a hash function to the corresponding name.

It is possible for several names to hash to the same location. When this occurs, a linear search from the hash location to the first free entry is required, to find a place for a new entry (if adding), or to find out whether an entry corresponding to the name exists (if searching). The more densely packed the hash table, the more likely this occurrence is. To maintain a balance between efficiency and table size, hash_ keeps a hash table approximately 75 percent full, by rehashing it (i.e. rebuilding it in a larger space) when it becomes too full.

The number of entries is limited only by the available space. The table uses eight words per entry plus ten words for a header. If an entire segment is available to hold the table, it can have over 32,000 entries.

Entry: hash_$in

This entry point adds an entry to a hash table. If the additional entry makes the table too full, the table is rehashed before the new entry is added (see the description of the rehash_ subroutine).

USAGE

```
declare hash_$in entry (ptr, char(*), bit(36) aligned, fixed bin(35));

call hash_$in (table_ptr, name, value, code);
```

ARGUMENTS

table_ptr
    is a pointer to the hash table. (Input)

name
    is a name associated with a data table entry. It can be up to 32 characters long. (Input)

value
    is the locator (e.g., index or offset) of the data table entry associated with name. (Input)

code
    is a standard system error code with the following values: (Output)
    0
        entry added successfully.
    error_table_$segnamedup
        entry already exists, with same value.
    error_table_$namedup
        entry already exists, with different values.
    error_table_$full_hashtbl
        hash table is full and there is no room to rehash it into a larger space.

Entry: hash_$inagain

This entry point adds an entry to a hash table. It is identical to the hash_$in entry except that it never tries to rehash the table. The new entry is added unless the table is completely full. This entry point is used by the rehash_ subroutine to avoid loops. It can also be used by an application that has a hash table embedded in a larger data base, where automatic rehashing would damage the data base.

USAGE

```
declare hash_$inagain entry (ptr, char(*), bit(36) aligned, fixed
      bin(35));

call hash_$inagain (table_ptr, name, value, code);
```

ARGUMENTS

table_ptr
    is a pointer to the hash table. (Input)

name
    is a name associated with a data table entry. It can be up to 32 characters long. (Input)

value
    is the locator (e.g., index or offset) of the data table entry associated with name. (Input)

code
    is a standard system error code with the following values: (Output)
    0
        entry added successfully.
    error_table_$segnamedup
        entry already exists, with same value.

error_table_$namedup
    entry already exists, with different values.
error_table_$full_hashtbl
    hash table is full and there is no room to rehash it into a larger space.

Entry: hash_$make

This entry point initializes an empty hash table. The caller must provide a segment to hold it, and must specify its initial size (see hash_$opt_size).

USAGE

```
declare hash_$make entry (ptr, fixed bin, fixed bin(35));

call hash_$make (table_ptr, size, code);
```

ARGUMENTS

table_ptr
    is a pointer to the table to be initialized. (Input)

size
    is the initial number of entries. (Input). It is recommended that the value returned by hash_$opt_size be used.

code
    is a standard status code. (Output). It can be:
    0
        if there is no error.
    error_table_$invalid_elsize
        if size is too large.

Entry: hash_$opt_size

This entry point, given the number of entries to be placed in a new hash table initially, returns the optimal size for the new table. This function is used when rehashing a full hash table, and should be used when making a new hash table.

USAGE

```
declare hash_$opt_size entry (fixed bin) returns (fixed bin);

size=hash_$opt_size (n_entries);
```

ARGUMENTS

n_entries
is the number of entries to be added. (Input)

size
is the optimal table size for that number of entries. (Output)

Entry: hash_$out

This entry point deletes a name from the hash table.

USAGE

```
declare hash_$out entry (ptr, char(*), bit(36) aligned, fixed bin(35));

call hash_$out (table_ptr, name, value, code);
```

ARGUMENTS

table_ptr
is a pointer to the hash table. (Input)

name
is the name to be deleted. (Input). Its maximum length is 32 characters.

value
is the locator value corresponding to name. (Input)

code
is a standard status code. (Output). It can be:
0
name was found and deleted.
error_table_$noentry
name was not found in the hash table.

Entry: hash_$search

This entry point searches a hash table for a given name and returns the corresponding locator value.

USAGE

```
declare hash_$search entry (ptr, char(*), bit(36) aligned, fixed
     bin(35));

call hash_$search (table_ptr, name, value, code);
```

ARGUMENTS

table_ptr
is a pointer to the hash table. (Input)

name
is the name to be searched for. (Input). It can be up to 32 characters long.

value
is the locator value corresponding to name. (Output)

code
is a standard status code. (Output). It can be:
0
name was found.
error_table_$noentry
name was not found in the hash table.

Name: hash_index_

The hash_index_ subroutine returns the value of a hash function of a character string.

USAGE

```
declare hash_index_ entry (ptr, fixed bin(21), fixed bin, fixed bin)
     returns (fixed bin);

hash_value = hash_index_ (string_ptr, string_len, mbz, table_size);
```

ARGUMENTS

string_ptr
is a pointer to the character string to be hashed. This character string must be aligned. (Input)

string_len
is the length of the character string. (Input)

mbz
is reserved and must be zero. (Input)

table_size
is the number of entries in the hash table. (Input)

NOTES

The value returned is between zero and table_size-1, inclusive.

Name: hcs_$add_acl_entries

This 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);
```

ARGUMENTS

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

entryname
: is the entry name of the segment. (Input)

acl_ptr
: points to a user-filled segment_acl_array structure (see "Entry Information" below). (Input)

acl_count
: contains the number of ACL entries in the segment_acl_array structure (see "Entry Information" below). (Input)

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

ENTRY INFORMATION

The segment_acl_array structure should be declared in the following way:

```
dcl 1 segment_acl_array (acl_count) aligned like segment_acl_entry;
```

The segment_acl_entry structure (declared in the include file acl_structures.incl.pl1) is as follows:

```
dcl 1 segment_acl_entry       aligned based,
      2 access_name           char(32) unaligned,
      2 mode                  bit(36) aligned,
      2 extended_mode         bit(36) aligned,
      2 status_code           fixed bin(35);
```

STRUCTURE ELEMENTS

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

mode
: contains the modes for this access name. The first three bits correspond to the modes read, execute, and write. The remaining bits must be zeros. For example, rw access is expressed as "101"b. The include file access_mode_values.incl.pl1 defines mnemonics for these values:

```
dcl (N_ACCESS                init ("000"b),
     R_ACCESS                init ("100"b),
     E_ACCESS                init ("010"b),
     W_ACCESS                init ("001"b),
     RE_ACCESS               init ("110"b),
     REW_ACCESS              init ("111"b),
     RW_ACCESS               init ("101"b)),
   bit (3) internal static options (constant);
```

extended_mode
: should contain the value "0"b. (This field is for use with extended access and should only be used in subsystems defining extended access modes).

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

NOTES

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 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

This 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);
```

ARGUMENTS

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

entryname
is the entryname of the directory. (Input)

acl_ptr
points to a user-filled dir_acl_array structure (see "Entry Information" below). (Input)

acl_count
contains the number of ACL entries in the dir_acl_array structure (see "Entry Information" below). (Input)

code
is a storage system status code. (Output)

ENTRY INFORMATION

The dir_acl_array structure should be declared in the following way:

```
dcl 1 dir_acl_array (acl_count) aligned like dir_acl_entry;
```

The dir_acl_entry structure (declared in the include file acl_structures.incl.pl1) is as follows:

```
dcl 1 dir_acl_entry           based,
      2 access_name           char (32) unaligned,
      2 mode                  bit(36) aligned,
      2 status_code           fixed bin(35);
```

STRUCTURE ELEMENTS

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

mode
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 zeros. For example, status permission is expressed as "100"b. The include file access_mode_values.incl.pl1 defines mnemonics for these values:

```
dcl (S_ACCESS              init ("100"b),
     M_ACCESS              init ("010"b),
     A_ACCESS              init ("001"b),
     SA_ACCESS             init ("101"b),
     SM_ACCESS             init ("110"b)),
     SMA_ACCESS            init ("111"b)),
   bit(3) internal static options (constant);
```

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

NOTES

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_$add_dir_inacl_entries

This entry point adds specified directory access modes to the initial access control list (initial ACL) for new directories created for the specified ring within the specified directory. If an access name already appears on the initial ACL of the directory, its mode is changed to the one specified by the call.

USAGE

```
declare hcs_$add_dir_inacl_entries entry (char(*), char(*), ptr,
      fixed bin, fixed bin(3), fixed bin(35));

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

ARGUMENTS

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

entryname
is the entryname of the directory. (Input)

acl_ptr
points to a user-filled dir_acl_array structure (see "Entry Information" below). (Input)

acl_count
contains the number of initial ACL entries in the dir_acl_array structure (see "Entry Information" below). (Input)

ring
is the ring number of the initial ACL. (Input)

code
is a storage system status code. (Output)

ENTRY INFORMATION

The dir_acl_array structure should be declared in the following way:

```
dcl 1 dir_acl_array (acl_count) aligned like dir_acl_entry;
```

The dir_acl_entry structure (declared in the include file acl_structures.incl.pl1) is as follows:

```
dcl 1 dir_acl_entry            based,
      2 access_name            char(32) unaligned,
      2 mode                   bit(36) aligned,
      2 status_code            fixed bin(35);
```

STRUCTURE ELEMENTS

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

mode
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 zeros. For example, status permission is expressed as "100"b. The include file access_mode_values.incl.pl1 defines mnemonics for these values:

```
dcl (S_ACCESS                init ("100"b),
     M_ACCESS                init ("010"b),
     A_ACCESS                init ("001"b),
     SA_ACCESS               init ("101"b),
     SM_ACCESS               init ("110"b)),
     SMA_ACCESS              init ("111"b)),
  bit(3) internal static options (constant);
```

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

NOTES

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

Name: hcs_$add_inacl_entries

This entry point adds specified access modes to the initial access control list (initial ACL) for new segments created for the specified ring within the specified directory. If an access name already appears on the initial ACL of the segment, its mode is changed to the one specified by the call.

USAGE

```
declare hcs_$add_inacl_entries entry (char(*), char(*), ptr, fixed bin,
      fixed bin(3), fixed bin(35));

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

ARGUMENTS

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

entryname
is the entryname of the directory. (Input)

acl_ptr
points to a user-filled segment_acl_array structure (see "Entry Information" below). (Input)

acl_count
contains the number of initial ACL entries in the segment_acl_array structure (see "Entry Information" below). (Input)

ring
is the ring number of the initial ACL. (Input)

code
is a storage system status code. (Output)

ENTRY INFORMATION

The segment_acl_array structure should be declared in the following way:

```
dcl 1 segment_acl_array (acl_count) aligned like segment_acl_entry;
```

The segment_acl_entry structure (declared in the include file acl_structures.incl.pl1) is as follows:

```
dcl 1 segment_acl_entry        aligned based,
      2 access_name            char(32) unaligned,
      2 mode                   bit(36) aligned,
      2 extended_mode          bit(36) aligned,
      2 status_code            fixed bin(35);
```

STRUCTURE ELEMENTS

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

mode
: contains the modes for this access name. The first three bits correspond to the modes read, execute, and write. The remaining bits must be zeros. For example, rw access is expressed as "101"b. The include file access_mode_values.incl.pl1 defines mnemonics for these values:

```
dcl (N_ACCESS              init ("000"b),
     R_ACCESS              init ("100"b),
     E_ACCESS              init ("010"b),
     W_ACCESS              init ("001"b),
     RE_ACCESS             init ("110"b),
     REW_ACCESS            init ("111"b),
     RW_ACCESS             init ("101"b)),
  bit (3) internal static options (constant);
```

extended_mode
: should contain the value "0"b. (This field is for use with extended access and should only be used in subsystems defining extended access modes).

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

NOTES

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

Name: hcs_$append_branch

The hcs_$append_branch entry point creates a segment in the specified directory, initializes the access control list (ACL) of the segment 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 ACL of the segment with the mode specified.

USAGE

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

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

ARGUMENTS

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

entryname
    is the entryname of the segment. (Input)

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

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. (Ring brackets are described in the Programmer's Reference Manual).

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_$append_branchx entry point to create a storage system entry with values other than the defaults listed above. Also see the description of the hcs_$append_branchx entry point for values of the access mode argument.

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 access control list (ACL) of the subdirectory 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 ACL of the segment 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);
```

ARGUMENTS

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

entryname
: is the entryname of the segment or subdirectory. (Input)

mode
: is the user's access mode (see "Notes" below). (Input)

rings
: is a three-element array that specifies the ring brackets of the new segment or subdirectory. (Input) If a subdirectory is to be created, the third element is ignored. (Ring brackets are described in the Programmer's Reference Manual).

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

dir_sw
: is the branch's directory switch. (Input)
  1 if a directory is being created
  0 if a segment is being created

copy_sw
: is the value of the copy switch to be placed in the branch. (Input) See the Programmer's Reference Manual for an explanation of the copy switch.

bit_count
: is the segment length (in bits). (Input)

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 zero. For example, rw access is 01010b in binary form.

The include file access_mode_values.incl.pl1 defines mnemonics for these values:

```
dcl (N_ACCESS_BIN         init (00000b),
     R_ACCESS_BIN         init (01000b),
     E_ACCESS_BIN         init (00100b),
     W_ACCESS_BIN         init (00010b),
     RW_ACCESS_BIN        init (01010b),
     RE_ACCESS_BIN        init (01100b),
     REW_ACCESS_BIN       init (01110b),

     S_ACCESS_BIN         init (01000b),
     M_ACCESS_BIN         init (00010b),
     A_ACCESS_BIN         init (00001b),
     SA_ACCESS_BIN        init (01001b),
     SM_ACCESS_BIN        init (01010b),
     SMA_ACCESS_BIN       init (01011b))

     fixed bin (5) internal static options (constant);
```

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.

USAGE

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

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

ARGUMENTS

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

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

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

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_$change_bc

This entry point provides a method of changing the bitcount of a segment. It is an indivisible operation in that only one process can perform it at a time; thus, if several processes try to change the bitcount, each one will get a different output. This can be used when several processes must write into a segment; if they use the change_bc entrypoint to determine where to write, they will never overwrite each other's data, and they will also never have to explicitly manipulate locks.

USAGE

```
declare hcs_$change_bc entry (char (*), char(*), fixed bin (24),
     fixed bin(24), fixed bin(24), fixed bin (35));

call hcs_$change_bc (dir_name, entryname, change, old_bc, new_bc, code);
```

ARGUMENTS

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

entryname
    is the entry name of the segment. (Input)

change
    is the amount by which the bitcount will be changed. (Input)

old_bc
    is the bitcount before the change was applied. (Output)

new_bc
    is the bitcount after the change was applied. (Output)

code
    is a storage system status code. (Output)

NOTES

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

The hcs_$change_bc_seg entrypoint performs the same function, but it takes a pointer to the segment rather than the pathname.

Name: hcs_$change_bc_seg

This entry point provides a method of changing the bitcount of a segment. It is an indivisible operation in that only one process can perform it at a time; thus, if several processes try to change the bitcount, each one will get a different output. This can be used when several processes must write into a segment; if they use the change_$bc_seg entrypoint to determine where to write, they will never overwrite each other's data, and they will also never have to explicitly manipulate locks.

USAGE

```
declare hcs_$change_bc_seg entry (pointer, fixed bin(24), fixed bin(24),
      fixed bin(24), fixed bin(35));

call hcs_$change_bc_seg (seg_ptr, change, old_bc, new_bc, code);
```

ARGUMENTS

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

change
is the amount by which the bitcount will be changed. (Input)

old_bc
is the bitcount before the change was applied. (Output)

new_bc
is the bitcount after the change was applied. (Output)

code
is a storage system status code. (Output)

NOTES

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

The hcs_$change_bc entry point performs the same function, but it takes the pathname of the segment rather than a pointer to it.

Name: hcs_$chname_file

This entry point changes the entry name 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);
```

ARGUMENTS

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

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

oldname
is the name to be deleted from the entry. (Input) It can be a null character string ("") in which case no name is deleted. If oldname is null, then newname must not be null.

newname
is the name to be added to the entry. (Input) 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.

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>work` exists and that it also has the entryname Work. Then the following call to hcs_$chname_file:

```
call hcs_$chname_file (">my_dir", "work", "Work", "work2", code);
```

changes the entryname Work to work2. The entry now has the names work and work2. Another call:

```
call hcs_$chname_file (">my_dir", "work2", "work2", "", code);
```

removes the entryname work2. Either work or work2 could be used in the second argument position. The entry now has only the name work. And finally, the call:

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

adds the entryname wk. The entry now has the names work and wk.

Name: hcs_$chname_seg

This 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);
```

ARGUMENTS

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

oldname
: is the name to be deleted from the entry. (Input) 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.

newname
: is the name to be added to the entry. (Input) 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.

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 call to hcs_$chname_seg:

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

changes the entryname beta to gamma. The segment now has the names alpha and gamma. Another call:

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

removes the entryname gamma. Now the segment only has an entryname of alpha. Finally, the call:

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

adds the entryname delta. The segment now has the names alpha and delta.

Name: hcs_$create_branch_

This entry point 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 entry point.) If a subdirectory is created, then the access control list (ACL) of the subdirectory is initiated by copying the initial ACL for directories that is stored in the specified directory; otherwise, the ACL of the segment 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);
```

ARGUMENTS

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

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

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

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 info_ptr pointer points to a structure of the following form (found in the include file, create_branch_info.incl.pl1):

```
dcl 1 create_branch_info     aligned based,
      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 parent_ac_sw       bit(1) unaligned,
        3 mbz1               bit(31) unaligned,
      2 mode                 bit(3) unaligned,
      2 mbz2                 bit(33) unaligned,
      2 rings                (3) fixed bin(3),
      2 userid               char(32),
      2 bitcnt               fixed bin(24),
      2 quota                fixed bin(18),
      2 access_class         bit(72);
```

STRUCTURE ELEMENTS

version
    is a number representing the version of the create_branch_info structure being used. The caller should set this to create_branch_info_version_1 before making the call.

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

copy_sw
    is the copy switch of the created segment.
    "1"b make a copy whenever the segment is written, if write access is not already present.
    "0"b do not make a copy--use the original.

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.

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

parent_ac_sw
indicates whether the access class of the parent directory is to be used for the created branch.
"1"b use the access class of the parent.
"0"b use the access class specified (by access_class described below).

mbz1
must be (31)"0"b.

mode
is the ACL mode desired for access_name. The meanings of the bits are defined in the description of hcs_$add_acl_entries for segments and hcs_$add_dir_acl_entries for directories.

mbz2
must be (33)"0"b.

rings
are the desired ring brackets of the new segment or subdirectory. If a subdirectory is to be created, the third element is ignored. Ring brackets are described in the Programmer's Reference Manual.

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

bitcnt
is the length of the segment (in bits).

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.

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 or the parent_ac_sw switch is set. (See the hcs_$get_access_class entry point.)

Name: hcs_$delete_acl_entries

This entry point deletes 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);
```

ARGUMENTS

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

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

acl_ptr
: points to a user-filled delete_acl_array structure (see "Entry Information" below). (Input)

acl_count
: is the number of ACL entries in the delete_acl_array structure (see "Entry Information" below). (Input)

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

ENTRY INFORMATION

The delete_acl_array structure should be declared in the following way:

```
dcl 1 delete_acl_array (acl_count) aligned like delete_acl_entry;
```

The delete_acl_entry structure (declared in the include file acl_structures.incl.pl1) is as follows:

```
dcl 1 delete_acl_entry        aligned based,
      2 access_name           char(32) unaligned,
      2 status_code           fixed bin(35);
```

STRUCTURE ELEMENTS

access_name
is the access name (in the form of Person_id.Project_id.tag) that identifies the ACL entry to be deleted.

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

NOTES

If code is returned as error_table_$argerr, then the erroneous ACL entries in the delete_acl_array 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 ACL of the segment, then the status_code for that ACL entry in the delete_acl_array structure is set to error_table_$user_not_found. Processing continues to the end of the delete_acl_array structure and code is returned as 0.

Name: hcs_$delete_dir_acl_entries

This entry point is used to delete specified entries from an access control list (ACL) for a directory. The delete_acl_array 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);
```

ARGUMENTS

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

entryname
is the entryname of the directory. (Input)

acl_ptr
points to a user-filled delete_acl_array structure. (Input)

acl_count
is the number of ACL entries in the delete_acl_array structure. (Input)

code
  is a storage system status code (see "Notes" below). (Output)

NOTES

If code is returned as error_table_$argerr, then the erroneous ACL entries in the delete_acl_array 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 ACL of the segment, then the status_code for that ACL entry in the delete_acl_array structure is set to error_table_$user_not_found. Processing continues to the end of the delete_acl_array structure and code is returned as 0.

Name: hcs_$delete_dir_inacl_entries

This entry point is used to delete specified entries from an initial access control list (initial ACL) for new directories created for the specified ring within the specified directory. The delete_acl_array structure used by this subroutine is described in the hcs_$delete_acl_entries entry point.

USAGE

```
declare hcs_$delete_dir_inacl_entries entry (char(*), char(*), ptr,
      fixed bin, fixed bin(3), fixed bin(35));

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

ARGUMENTS

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

entryname
  is the entryname of the directory. (Input)

acl_ptr
  points to the user-filled delete_acl_array structure. (Input)

acl_count
  is the number of initial ACL entries in the delete_acl_array structure. (Input)

ring
  is the ring number of the initial ACL. (Input)

code
is a storage system status code. (Output)

NOTES

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

If an access_name in the delete_acl_array structure cannot be matched to one existing on the initial ACL, then the status_code of that initial ACL entry in the delete_acl_array structure is set to error_table_$user_not_found. Processing continues to the end of the delete_acl_array structure and code is returned as 0.

Name: hcs_$delete_inacl_entries

This entry point is called to delete specified entries from an initial access control list (initial ACL) for new segments created for the specified ring within the specified directory. The delete_acl_array structure used by this subroutine is discussed in the hcs_$delete_acl_entries entry point.

USAGE

```
declare hcs_$delete_inacl_entries entry (char(*), char(*), ptr,
      fixed bin, fixed bin(3), fixed bin(35));

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

ARGUMENTS

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

entryname
is the entryname of the directory. (Input)

acl_ptr
points to the user-filled delete_acl_array structure. (Input)

acl_count
contains the number of initial ACL entries in the delete_acl_array structure. (Input)

ring
is the ring number of the initial ACL. (Input)

code
is a storage system status code. (Output)

NOTES

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

If an access_name in the delete_acl_array structure cannot be matched to one existing on the initial ACL, then the status_code of that initial ACL entry in the delete_acl_array structure is set to error_table_$user_not_found. Processing continues to the end of the delete_acl_array structure and code is returned as 0.

Name: hcs_$force_write

This entry point causes the supervisor to force modified pages out of main memory protecting data against an unrecoverable main memory crash.

USAGE

```
declare hcs_$force_write entry (ptr, bit(36), fixed bin(35));

call hcs_$force_write (segp, flags, code);
```

ARGUMENTS

segp
is a pointer to the segment whose modified pages are to be written. (Input)

flags
specify a set of options. (Input) Currently only one option is defined. The following structure (also defined in the system include file force_write_flags.incl.pl1) defines the options:

```
dcl 1 force_write_options        based (addr (flags)) unaligned,
      2 mbz1                     bit(1),
      2 serial_write             bit(1),
      2 mbz2                     bit(34);
```

serial_write:
"0"b queue write requests for all modified pages in parallel, up to the maximum permitted by the supervisor's force-write limit (see shcs_$set_force_write_limit).
"1"b queue write requests for all modified pages serially (i.e., one at a time).

mbz1, mbz2
: these fields must be zero.

code
: is a standard status code. (Output) It can be one of the following:
error_table_$bad_ring_brackets
: the segment is an inner ring segment.
error_table_$moderr
: the user does not have write access to the segment.
error_table_$invalidsegno
: the segment is not known, not active, or a hardcore segment. This should not be treated as an error because the user has no control over whether or not the segment is active.

NOTES

Use of this entry point may introduce substantial real time delay into execution, since the caller must wait for the movement of the disk; other usage of the system, meanwhile, may cause further delay.

Name: hcs_$fs_get_access_modes

This entry point returns the user's access mode and extended access mode on a specified segment at the current validation level. For a discussion of access modes see "Access Control" in the Programmer's Reference Manual.

USAGE

```
declare hcs_$fs_get_access_modes entry (ptr, bit (36) aligned, bit (36)
      aligned, fixed bin (35));

call hcs_$fs_get_access_modes (seg_ptr, modes, ex_modes, code);
```

ARGUMENTS

seg_ptr
: is a pointer to the segment whose access mode is to be returned. (Input)

modes
: is the returned access mode. See the description of the hcs_$append_branchx entry point for the values of the mode argument. (Output)

ex_modes
: is the returned extended access mode. (Output)

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" in the Programer's Reference Manual.

Name: hcs_$fs_get_mode

This entry point returns the user's access mode on a specified segment at the current validation level. For a discussion of access modes see the Programmer's Reference Manual.

USAGE

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

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

ARGUMENTS

seg_ptr
is a pointer to the segment whose access mode is to be returned. (Input)

mode
is the returned access mode (see "Notes" below). (Output)

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 the Programmer's Reference Manual.

See the description of the hcs_$append_branchx entry point for the values of the mode argument.

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. For a definition of "primary name" refer to "Glossary of Multics Terms" in the Programmer's Reference Manual.

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);
```

ARGUMENTS

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

dir_name
: is the pathname of the containing directory. (Output) 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.

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

entryname
: is the primary entryname of the segment. (Output) 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.

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

Name: hcs_$fs_get_ref_name

This entry point returns a specified (i.e., first, second, etc.) reference name for a specified segment.

USAGE

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

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

ARGUMENTS

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

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)

ref_name
: is the desired reference name. (Output)

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

NOTES

If the count argument is larger than the total number of names, the name which the segment was originally made known is returned and code is set to error_table_$refname_count_too_big.

Name: hcs_$fs_get_seg_ptr

This entry point, given a reference name of a segment, returns a pointer to the base of the segment.

USAGE

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

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

ARGUMENTS

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

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

code
is a storage system status code. (Output)

NOTES

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 the Programmer's Reference Manual.

Name: hcs_$fs_move_file

This 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);
```

ARGUMENTS

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

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

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

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

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

code
is a storage system status code. (Output) It can have the value error_table_$no_move for any of the reasons 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. Either to_entry or from_entry is not a segment.

2. The user does not have rw access to to_entry.

3. The user does not have read access to from_entry.

4. The max_length of to_entry is less than the length of from_entry.

5. There is not enough quota in to_dir to perform the move.

Name: hcs_$fs_move_seg

This entry point moves the data associated with one segment in the hierarchy to another segment, given pointers to the segments in question. The old segment remains, but with a zero length.

USAGE

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

call hcs_$fs_move_seg (from_ptr, to_ptr, trun_sw, code);
```

ARGUMENTS

from_ptr
: is a pointer to the segment from which data is to be moved. (Input)

to_ptr
: is a pointer to the target segment. (Input)

trun_sw
: indicates whether the segment specified by to_ptr is to be truncated (if it is not already zero length) before performing the move. (Input)
: 0 returns code error_table_$clnzero if the segment is not already zero length
: 1 truncates the segment before moving

code
: is a storage system status code. (Output) It can have the value error_table_$no_move or error_table_$clnzero.

NOTES

The hcs_$fs_move_file entry point performs the same function given the pathnames of the segments instead of the pointers.

Name: hcs_$get_access_class

This entry point returns the access class of a segment or directory in the storage hierarchy. For information on access classes, see the Programmer's Reference Manual.

USAGE

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

call hcs_$get_access_class (dir_name, entryname, access_class, code);
```

ARGUMENTS

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

entryname
is the entryname of the segment or directory. (Input)

access_class
is the access class of the segment or directory. (Output)

code
is a storage system status code. (Output)

NOTES

If the value of entryname is null, dir_name is assumed to be a full pathname.

The user must have status permission on the directory (the dir_name argument) or nonnull access to the entry (the entryname argument).

Name: hcs_$get_access_class_seg

This entry point, given a pointer, returns the access class of that pointer's corresponding segment. For information on access classes, see the Programmer's Reference Manual.

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);
```

ARGUMENTS

seg_ptr
is the pointer to the segment. (Input)

access_class
is the access class of the segment. (Output)

code
is a storage system status code. (Output)

Name: hcs_$get_access_info

This entry point returns all of the access attributes of a storage system object, given its pathname.

USAGE

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

call hcs_$get_access_info (dir_name, entryname, chase,
      entry_access_info_ptr, code);
```

ARGUMENTS

dir_name
: the directory containing the object about which information is to be returned. (Input)

entryname
: the entryname of the object. (Input)

chase
: indicates the action hcs_$get_access_info is to perform if the object is a link. (Input) Its possible values are:
: 0 do not chase the link
: 1 chase the link.

entry_access_info_ptr
: points to a structure containing all the access information for the object. (Input)

code
: is a standard system status code. (Output) Its possible values are:

error_table_$link
: if chase was not specified, then the pathname supplied is a link. Otherwise, the pathname supplied eventually points to a null link.

error_table_$null_info_ptr
: the entry_access_info_ptr supplied was null.

error_table_$unimplemented_version
: the requested version of entry_access_info is not supported.

NOTES

The following structure, declared in entry_access_info.incl.pl1, is pointed to by entry_access_info_ptr:

```
dcl 1 entry_access_info aligned based (entry_access_info_ptr),
      2 version                  char (8),
      2 type                     fixed bin,
      2 dir_name                 char (168) unaligned,
      2 entryname                char (32) unaligned,
      2 uid                      bit (36) aligned,
      2 ring_brackets            (3) fixed bin (3),
      2 extended_ring_brackets   (3) fixed bin (3),
      2 effective_access_modes   bit (36) aligned,
      2 extended_access_modes    bit (36) aligned,
      2 access_class             bit (72) aligned,
      2 parent_access_class      bit (72) aligned,
      2 multiclass               bit (1) aligned;
```

STRUCTURE ELEMENTS

version
    must be set to ENTRY_ACCESS_INFO_VERSION_1.

type
    specifies the type of the object. Its possible values are:
    0 link
    1 segment
    2 directory

dir_name
    the pathname of the entry's parent.

entryname
    primary name of the entry

uid
    the entry's unique identifer

ring_brackets
    the entry's ring brackets. For directories, ring_brackets(3) is not used.

extended_ring_brackets
    this has not been implemented.

effective_access_modes
    the user's effective access to this entry, taking into account ACLs, ring brackets, and AIM authorization.

extended_access_modes
the user's extended access modes to this entry.

access_class
the access class of the object. If it is a multiclass object, the maximum access class from which the object can be referenced.

parent_access_class
the access class of the object's parent. For a multiclass object, the minimum access class from which the object can be referenced.

multiclass
is "1"b if the object is multiclass

ACCESS REQUIRED

This entrypoint requires s access to the containing directory of the object, or non-null access to the object itself.

Name: hcs_$get_access_info_seg

This entry point returns all of the access attributes of a storage system object, given a pointer to the segment.

USAGE

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

call hcs_$get_access_info_seg (seg_ptr, entry_access_info_ptr, code);
```

ARGUMENTS

seg_ptr
is a pointer to the segment about which information is to be returned.

entry_access_info_ptr
points to a structure containing all the access information for the object. (Input)

code
is a standard system status code. (Output) Its possible values are:

error_table_$null_info_ptr
the entry_access_info_ptr supplied was null.

error_table_$unimplemented_version
the requested version of entry_access_info is not supported.

NOTES

The following structure, declared in entry_access_info.incl.pl1, is pointed to by entry_access_info_ptr:

```
dcl 1 entry_access_info aligned based (entry_access_info_ptr),
    2 version                  char (8),
    2 type                     fixed bin,
    2 dir_name                 char (168) unaligned,
    2 entryname                char (32) unaligned,
    2 uid                      bit (36) aligned,
    2 ring_brackets             (3) fixed bin (3),
    2 extended_ring_brackets    (3) fixed bin (3),
    2 effective_access_modes   bit (36) aligned,
    2 extended_access_modes    bit (36) aligned,
    2 access_class             bit (72) aligned,
    2 parent_access_class      bit (72) aligned,
    2 multiclass               bit (1) aligned;
```

STRUCTURE ELEMENTS

version
: must be set to ENTRY_ACCESS_INFO_VERSION_1.

type
: specifies the type of the object. Its possible values are:
  1 segment
  2 directory

dir_name
: the pathname of the entry's parent.

entryname
: primary name of the entry

uid
: the entry's unique identifer

ring_brackets
: the entry's ring brackets. For directories, ring_brackets(3) is not used.

extended_ring_brackets
: this has not been implemented.

effective_access_modes
: the user's effective access to this entry, taking into account ACLs, ring brackets, and AIM authorization.

extended_access_modes
: the user's effective extended access mode to this entry.

access_class
the access class of the object. If it is a multiclass object, the maximum access class from which the object can be referenced.

parent_access_class
the access class of the object's parent. For a multiclass object, the minimum access class from which the object can be referenced.

multiclass
is "1"b if the object is multiclass

ACCESS REQUIRED

This entrypoint requires s access to the containing directory of the object, or non-null access to the object itself.

Name: hcs_$get_author

This entry point returns the author of a segment, directory, multisegment file, or link.

USAGE

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

call hcs_$get_author (dir_name, entryname, chase, author, code);
```

ARGUMENTS

dir_name
is the pathname of the directory. (Input)

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

chase
if entryname refers to a link, this flag indicates whether to return the author of the link or the author of the segment, directory, or multisegment file to which the link points. (Input)
0 return link author.
1 return segment, directory, or multisegment file author.

author
is the author of the segment, directory, multisegment file, or link in the form of Person_id.Project_id.tag with a maximum length of 32 characters. (Output) An error is not detected if the string is too short on hold the author.

code
is a storage system status code. (Output)

NOTES

The user must have status permission on the directory or non-null access on the entry.

This page intentionally left blank.

Name: hcs_$get_bc_author

This entry point returns the bit count author of a segment or directory. The bit count author is the name of the user who last set the bit count of the segment or directory.

USAGE

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

call hcs_$get_bc_author (dir_name, entryname, bc_author, code);
```

ARGUMENTS

dir_name
: is the pathname of the directory. (Input)

entryname
: is the entry name of the segment or directory. (Input)

bc_author
: is the bit count author of the segment or directory in the form of Person_id.Project_id.tag with a maximum length of 32 characters. (Output) An error is not detected if the string is too short to hold the bit count author.

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

NOTES

The user must have status permission on the directory or non-null access on the entry.

Name: hcs_$get_dir_ring_brackets

This entry point, when given the pathname of a containing directory and the entryname of a subdirectory, returns the value of that subdirectory's ring brackets.

USAGE

```
declare hcs_$get_dir_ring_brackets entry (char (*), char (*),
      (2) fixed bin(3), fixed bin(35));

call hcs_$get_dir_ring_brackets (dir_name, entryname, drb, code);
```

ARGUMENTS

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

entryname
is the entry name of the subdirectory. (Input)

drb
is a two-element array that contains the directory's ring brackets. (Output) The first element contains the level required for modify and append permission; the second element contains the level required for status permission.

code
is a storage system status code. (Output)

NOTES

The user must have status permission on the directory or non-null access on the entry.

Ring brackets are discussed in "Intraprocess Access Control" in the Programmer's Reference Manual.

Name: hcs_$get_exponent_control

This entry point returns the current settings of the flags that control the system's handling of exponent overflow and underflow conditions. For more information on exponent control see the description of hcs_$set_exponent_control.

USAGE

```
declare hcs_$get_exponent_control entry (bit(1) aligned, bit(1) aligned,
      float bin(63));

call hcs_$get_exponent_control (restart_underflow, restart_overflow,
      overflow_value);
```

ARGUMENTS

restart_underflow
is "1"b if underflows are currently being automatically restarted, and "0"b otherwise. (Output)

restart_overflow
is "1"b if overflows are currently being automatically restarted, and "0"b otherwise. (Output)

overflow_value
is the value used for the result of the computation in the case of overflow. (Output)

Name: hcs_$get_initial_ring

This entry point returns the ring at which the process was logged in.

USAGE

```
declare hcs_$get_initial_ring entry (fixed bin);

call hcs_$get_initial_ring (initial_ring);
```

ARGUMENTS

initial_ring
the ring number at which the process began execution. (Output)

ACCESS REQUIRED

No access is required.

Name: hcs_$get_ips_mask

This entry point returns the value of the current ips mask.

USAGE

```
declare hcs_$get_ips_mask entry (bit(36) aligned);

call hcs_$get_ips_mask (old_mask);
```

ARGUMENTS

old_mask
is the current value of the ips mask. (Output)

NOTES

A "1"b in any position in the mask means that the corresponding ips interrupt is enabled.

The thirty-sixth (rightmost) bit of old_mask does not correspond to an interrupt, but is used as a control bit, giving a positive indication that a particular masking or unmasking operation has taken place. No ips interrupts can occur in the time interval between the requested mask modification and the returning of the old_mask, with the control bit set appropriately.

Entry points used at the beginning of a critical section of code, to disable some or all ips interrupts, return a value of "1"b for the control bit, while those that are used at the end of a critical section of code, to re-enable those interrupts, return a value of "0"b for the control bit. Thus, a condition handler can interpret a value of "1"b in the control bit as meaning that execution was in a critical section of code, and the ips mask has been modified.

The control bit in the mask returned by this entry point is always "0"b.

Name: hcs_$get_link_target

This entry point returns the pathname of the ultimate target of a link if the ultimate target exists, or what that pathname would be if the target did exist.

USAGE

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

call hcs_$get_link_target (dir_name, entryname, link_dir_name,
      link_entryname, code);
```

ARGUMENTS

dir_name
: is the directory name containing the link. (Input)

entryname
: is the entryname of the link for which target information is desired. (Input)

link_dir_name
: is the directory name of the link target with a maximum length of 168 characters. (Output)

link_entryname
: is the entryname of the link target with a maximum length of 32 characters. (Output)

code
: is a standard status code. (Output)

NOTES

This entry chases the link to its ultimate target. The ultimate target of a link must be a directory or segment, which may or may not exist. If the immediate target of a link is another link, the chasing of links continues toward the ultimate target directory or segment until it is encountered or found to be nonexistent. If the ultimate target of the link exists, the user must either have nonnull permission on the directory containing the target or nonnull access to the target itself in order to determine its pathname. If appropriate access exists, the code is zero, and link_dir_name and link_entryname are set. If not, an error code is returned, and the link_dir_name and link_entryname are returned as blank.

If the ultimate target does not exist, the target pathname of the last link encountered while chasing links will be returned if the user has nonnull permission on the directory that would have contained that target pathname. In this case, the returned code is error_table_$noentry, and the link_dir_name and link_entryname are set.

This page intentionally left blank.

In all other cases, an error code is returned to indicate the lack of access, and link_dir_name and link_entryname are returned as blanks.

Name: hcs_$get_max_length

This entry point, given a directory name and entryname, returns the maximum length (in words) of the segment.

USAGE

```
declare hcs_$get_max_length entry (char(*), char(*), fixed bin(19),
      fixed bin(35));

call hcs_$get_max_length (dir_name, entryname, max_length, code);
```

ARGUMENTS

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

entryname
    is the entryname of the segment. (Input)

max_length
    is the maximum length of the segment in words. (Output)

code
    is a storage system status code. (Output)

NOTES

The user must have status permission on the directory containing the segment or nonnull access to the segment.

Name: hcs_$get_max_length_seg

This entry point, given a pointer to a segment, returns the maximum length (in words) of the segment.

USAGE

```
declare hcs_$get_max_length_seg entry (ptr, fixed bin(19),
      fixed bin(35));

call hcs_$get_max_length_seg (seg_ptr, max_length, code);
```

ARGUMENTS

seg_ptr
is a pointer to the segment whose maximum length is to be returned. (Input)

max_length
is the maximum length of the segment in words. (Output)

code
is a storage system status code. (Output)

NOTES

The user must have status permission on the directory containing the segment or nonnull access to the segment.

Name: hcs_$get_page_trace

The hcs_$get_page_trace entry point returns information about recent paging activity.

USAGE

```
declare hcs_$get_page_trace entry (ptr);

call hcs_$get_page_trace (data_ptr);
```

ARGUMENTS

data_ptr
is a pointer to a user data space where return information is stored. (Input)

NOTES

The format of the data structure returned by hcs_$get_page_trace is described below. The amount of data returned cannot be known in advance other than that there are less than 1024 words returned.

```
dcl 1 trace              aligned based(tp)
      2 next_available   bit(18) aligned,
      2 size             bit(18) aligned,
      2 time             fixed bin(71),
      2 pad1             fixed bin(35),
      2 index            bit(17),
      2 pad2             fixed bin(71),
      2 data             (512 refer(divide (trace.size,2,17,0))),
        3 info           bit(36) aligned,
        3 type           bit(6) unaligned
        3 pageno         bit(12) unaligned,
        3 time_delta     bit(18) unaligned;
```

STRUCTURE ELEMENTS

next_available
: is a relative pointer (relative to the first trace entry) to the next entry to be used in the trace list.

size
: is the number of words in the trace array and, hence, twice the number of entries in the array.

time
: is the real-time clock reading at the time the last trace entry was entered in the list.

pad1
: is unused.

index
: is a relative pointer to the first trace entry entered in the last quantum. Thus, all events traced in the last quantum can be determined by scanning from trace.index to trace.next_available (minus 1) with the obvious check for wrap-around.

pad2
: is unused.

info
: is information about the particular trace entry.

type
specifies what kind of a trace entry it is. The following types are currently defined:

0 page fault
2 segment fault begin
3 segment fault end
4 linkage fault begin
5 linkage fault end
6 bound fault begin
7 bound fault end
8 signaller event
9 restarted signal
10 reschedule
11 user marker
12 interrupt

pageno
is the page number associated with the fault. Certain trace entries do not fill in this field.

time_delta
is the amount of real time elapsed between the time this entry was entered and the previous entry was entered. The time value is in units of 64 microseconds.

Name: hcs_$get_process_usage

This entry point returns information on system resource usage by the requesting process.

USAGE

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

call hcs_$get_process_usage (process_usage_pointer, code);
```

ARGUMENTS

process_usage_pointer
is a pointer to the structure described in "Notes" below. (Input)

code
is a standard status code. (Output)

NOTES

The following structure, declared in process_usage.incl.pl1, is pointed to by process_usage_pointer:

```
dcl 1 process_usage        based (process_usage_pointer),
      2 number_wanted      fixed bin,
      2 number_can_return  fixed bin,
      2 cpu_time           fixed bin (71),
      2 paging_measure     fixed bin (71),
      2 page_faults        fixed bin (34),
      2 pd_faults          fixed bin (34),
      2 virtual_cpu_time   fixed bin (71),
      2 segment_faults     fixed bin (34),
      2 bounds_faults      fixed bin (34),
      2 vtoc_reads         fixed bin (34),
      2 vtoc_writes        fixed bin (34);
```

STRUCTURE ELEMENTS

number_wanted
: specifies how much information is to be returned in the structure. It must be set prior to the call to hcs_$get_process_usage, and its interpretation is given below. It is the only input parameter in the structure; all other items are output from hcs_$get_process_usage or are ignored, depending on the value of number_wanted.

number_can_return
: is the number of system resource values which can be returned. It corresponds to the number of level 2 items in the structure following number_can_return. This is returned for all values of number_wanted.

cpu_time
: is the cumulative central processor time for the process. It includes all time spent executing instructions outside of ring 0, all time spent executing instructions in ring 0 as the result of explicit calls to ring 0, and all overhead time while executing instructions in the address space of this process (e.g., processing page faults for this process and interrupts where this process was interrupted). This is returned if number_wanted is 1 or greater.

paging_measure
: is the cumulative memory usage for the process in billable memory units. This is returned if number_wanted is 2 or greater.

page_faults
: is the cumulative number of page faults by the process. This number represents the number of times a page was referenced which was not in main memory. This is returned if number_wanted is 3 or greater.

pd_faults
is the cumulative number of paging device faults by the process. This number is always zero. This is returned if number_wanted is 4 or greater.

virtual_cpu_time
is the cumulative virtual time for the process. This includes all time spent executing instructions outside of ring 0 and all time spent executing instructions in ring 0 as the result of explicit calls to ring 0. It does not include overhead time, such as the time spent processing page faults, segment faults, or interrupts. This is returned if number_wanted is 5 or greater.

segment_faults
is the cumulative number of segment faults by the process. This represents the number of times a segment was referenced whose page table was not in main memory. This is returned if number_wanted is 6 or greater.

bounds_faults
is the cumulative number of bounds faults by the process. This represents the number of times an address within a segment was referenced that was beyond the segment bound. This occurs most commonly when a segment expands to the point where it requires a larger page table. This is returned if number_wanted is 7 or greater.

vtoc_reads
is the number of read I/Os done by the process to Volume Table of Contents Entries (VTOCEs). This is returned if number_wanted is 8 or greater.

vtoc_writes
is the number of write I/Os done by the process to VTOCEs. This is returned if number_wanted is 9 or greater.

NOTES

In the above description, cumulative activity by the requesting process is defined to mean all activity since login or since the most recent new_proc.

Name: hcs_$get_ring_brackets

This entry point, given the directory name and entryname of a segment, returns the value of that segment's ring brackets.

USAGE

```
declare hcs_$get_ring_brackets entry (char(*), char(*), (3) fixed
      bin(3),
      fixed bin(35));

call hcs_$get_ring_brackets (dir_name, entryname, rb, code);
```

ARGUMENTS

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

entryname
is the entryname of the segment. (Input)

rb
is a three-element array that contains the segment's ring brackets. (Output) Ring brackets and validation levels are discussed in "Intraprocess Access Control" in the Programmer's Reference Manual.

code
is a storage system status code. (Output)

NOTES

The user must have status permission on the directory or non-null access to the entry.

Name: hcs_$get_ring_brackets_seg

This entry point, given a pointer to a segment, will return the ring brackets of the segment.

USAGE

```
declare hcs_$get_ring_brackets entry (ptr, (3) fixed bin(3), fixed
      bin(35));

call hcs_$get_ring_brackets (seg_ptr, brackets, code);
```

ARGUMENTS

seg_ptr
is a pointer to the segment in question. (Input)

brackets
is a three-element array that contains the segment's ring brackets. (Output) Ring brackets and validation levels are discussed in "Intraprocess Access Control" in the Programmer's Reference Manual.

code
is a storage system status code. (Output)

ACCESS REQUIRED

The user must have status permission on the directory or non-null access to the object.

Name: hcs_$get_safety_sw

This entry point, given a directory name and an entryname, returns the value of the safety switch of a directory or a segment.

USAGE

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

call hcs_$get_safety_sw entry (dir_name, entryname, safety_sw, code);
```

ARGUMENTS

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

entryname
is the entryname of the directory or segment. (Input)

safety_sw
is the value of the safety switch. (Output)
"0"b the segment or directory can be deleted.
"1"b the segment or directory cannot be deleted.

code
is a storage system status code. (Output)

NOTES

The user must have status permission on the containing directory or nonnull access to the directory or segment.

Name: hcs_$get_safety_sw_seg

This entry point, given a pointer to the segment, returns the value of the safety switch of a segment.

USAGE

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

call hcs_$get_safety_sw_seg (seg_ptr, safety_sw, code);
```

ARGUMENTS

seg_ptr
: is a pointer to the segment whose safety switch is to be examined. (Input)

safety_sw
: is the value of the segment safety switch. (Output)
"0"b the segment can be deleted.
"1"b the segment cannot be deleted.

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

This page intentionally left blank.

NOTES

The user must have status permission on the directory containing the segment or must have nonnull access to the segment.

Name: hcs_$get_search_rules

This entry point returns the search rules currently in use in the caller's process.

USAGE

```
declare hcs_$get_search_rules entry (ptr);

call hcs_$get_search_rules (search_rules_ptr);
```

ARGUMENTS

search_rules_ptr
is a pointer to a user-supplied search rules structure. (Input) See "Note" below.

NOTES

The structure pointed to by search_rules_ptr is declared as follows:

```
dcl 1 search_rules            aligned,
      2 number                fixed bin,
      2 names                 (21) char(168) aligned;
```

STRUCTURE ELEMENTS

number
is the number of search rules in the array.

names
are the names of the search rules. They can be absolute pathnames of directories or keywords. (See the hcs_$initiate_search_rules entry point for a detailed description of the search rules.)

Name: hcs_$get_system_search_rules

This entry point provides the user with the values of the site-defined search rule keywords accepted by hcs_$initiate_search_rules.

USAGE

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

call hcs_$get_system_search_rules (search_rules_ptr, code);
```

ARGUMENTS

search_rules_ptr
is a pointer to the structure described in "Notes" below. (Input)

code
is a storage system status code. (Output)

NOTES

The structure pointed to by search_rules_ptr is declared as follows:

```
dcl 1 drules                  based aligned,
      2 ntags                 fixed bin,
      2 nrules                fixed bin,
      2 tags (10),
        3 name                char(32),
        3 flag                bit(36),
      2 rules (50),
        3 name                char(168),
        3 flag                bit(36);
```

STRUCTURE ELEMENTS

ntags
is the number of tags.

nrules
is the number of rules.

tags
is an array of keywords.

tags.name
is the keyword.

tags.flag
is a bit field with one bit on.

rules
  is an array of directory names.

rules.name
  is the absolute pathname of the directory.

rules.flag
  is a bit field with bits on for every tag that selects this directory.

Name: hcs_$get_uid_file

This entry point returns the unique identifier of a storage system entry. If the input arguments refer to a link, the uid of the target is returned.

USAGE

```
declage hcs_$get_uid_file entry (char(*), char(*), bit(36) aligned,
     fixed bin(35));

call hcs_$get_uid_file (dir_name, entry_name,uid, code);
```

ARGUMENTS

dir_name
  is the name of the directory containing the entry. (Input)

entry_name
  is the name of the entry whose unique identifier is to be returned. (Input)

uid
  is the unique identifier of the entry. (Output)

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

Name: hcs_$get_uid_seg

This entry point, when given a pointer to a segment, returns the unique identifier associated with the segment.

USAGE

```
declare hcs_$get_uid_seg entry (ptr, bit (36) aligned, fixed bin (35));

call hcs_$get_uid_seg (seg_ptr, unique_id, code);
```

ARGUMENTS

seg_ptr
is a pointer to the segment whose unique identifier is to be determined. (Input)

unique_id
is the unique identifier associated with the segment. (Output)

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

Name: hcs_$get_user_access_modes

This entry point returns the user's effective access mode and extended access mode on a branch. For a description of access modes, see "Effective Access" in the *Multics Programmer's Reference Manual,* Order No. AG91.

USAGE

```
declare hcs_$get_user_access_modes entry (char (*), char (*), char (*),
      fixed bin, bit (36) aligned, bit (36) aligned, fixed bin(35));

call hcs_$get_user_access_modes (dir_name, entryname, user_id, ring,
      mode, ex_mode, code);
```

ARGUMENTS

dir_name
is the directory name of the branch. (Input)

entryname
is the entry name of the branch. (Input)

user_id
: is the access name of the user in the form Person_id.Project_id_.tag. (Input) This is limited to 32 characters. If null, the access name of the calling process is used.

ring
: is the validation level that is to be used in computing effective access. (Input) It must be a value between 0 and 7 inclusive, or -1. If the ring value is -1, the default value of the validation level of the calling process is used. This default should be used in all cases except those in which a different ring's access is explicitly required.

mode
: is the effective access mode of the user on the branch (see "Notes" below). (Output)

ex_mode
: is the extended access mode of the user on the branch. (Output)

code
: is a standard status code. (Output)

ACCESS REQUIRED

The user must have status permission on the containing directory, unless the access name supplied is that of the calling process or is null.

NOTES

The include file access_modes_values.incl.pl1 defines mnemonics for the different values of mode. Extended access modes are defined by the subsystem owning the branch.

Name: hcs_$get_user_access_modes_ptr

This entry point returns the effective access mode and extended access mode of a user to a segment, given a pointer to the segment, the name of the user, and the validation level (ring number) of the user. (For a description of this mode, see "Effective Access" in the *Multics Programmer's Reference Manual,* Order No. AG91.)

USAGE

```
declare hcs_$get_user_access_modes entry (pointer, fixed bin, bit (36)
      aligned, bit (36) aligned, fixed bin (35));

call hcs_$get_user_access_modes (segment_ptr, user_id, ring, mode,
      ex_mode, code);
```

ARGUMENTS

segment_ptr
: is a pointer to the segment for which access will be returned (Input)

user_id
: is the access name fo the user in the form Person_id.Project_id.tag. (Input) This is limited to 32 characters. If null, the access name of the calling process is used.

ring
: is the validation level that is to be used in computing effective access. (Input) It must be a value between 0 and 7 inclusive, or −1. If the ring value is −1, a default value of the validation level of the calling process is used. This default should be used in all cases except those in which a different ring's access is explicitly required.

mode
: is the effective access mode of the user to the branch (see "Notes" below). (Output)

ex_mode
: is the extended access mode of the user to the branch. (Output)

code
: is a standard status code. (Output)

ACCESS REQUIRED

The user must have status permission on the containing directory, unless the access name supplied is that of the calling process or null.

NOTES

The include file access_modes_values.incl.pl1 defines mnemonics for the different values of mode. Extended access modes are defined by the subsystem owning the branch.

Name: hcs_$get_user_effmode

This entry point returns a user's effective access mode on a branch. (For a description of access modes, see "Effective Access" in the Programmer's Reference Manual.)

USAGE

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

call hcs_$get_user_effmode (dir_name, entryname, user_id, ring, mode,
     code);
```

ARGUMENTS

dir_name
: is the directory name of the branch. (Input)

entryname
: is the entry name of the branch. (Input)

user_id
: is the access name of the user in the form Person_id.Project_id_.tag. (Input) This is limited to 32 characters. If null, the access name of the calling process is used.

ring
: is the validation level that is to be used in computing effective access. (Input) It must be a value between 0 and 7 inclusive, or -1. If the ring value is -1, a default value of the validation level of the calling process is used. This default should be used in all cases except those in which a different ring's access is explicitly required.

mode
: is the effective access mode of the user to the branch (see "Notes" below). (Output)

code
: is a standard status code. (Output)

NOTES

The mode argument is a fixed binary number where the desired mode is encoded with one access mode specified by each bit. The modes for segments 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 modes for directories 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)
```

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. The access_mode_values.incl.pl1 include file defines mnemonics for these values:

```
dcl (N_ACCESS_BIN        init (00000b),
     R_ACCESS_BIN        init (01000b), 8
     E_ACCESS_BIN        init (00100b), 4
     W_ACCESS_BIN        init (00010b), 2
     RW_ACCESS_BIN       init (01010b), 10
     RE_ACCESS_BIN       init (01100b), 12
     REW_ACCESS_BIN      init (01110b), 14

     S_ACCESS_BIN        init (01000b), 8
     M_ACCESS_BIN        init (00010b), 2
     A_ACCESS_BIN        init (00001b), 1
     SA_ACCESS_BIN       init (01001b), 9
     SM_ACCESS_BIN       init (01010b), 10
     SMA_ACCESS_BIN      init (01011b)) 11
     fixed bin (5) internal static options (constant);
```

The user must have status permission on the containing directory, unless the access name supplied is that of the calling process or null.

Name: hcs_$high_low_seg_count

This entry point returns information about the lowest and highest segment numbers used in the process, excluding hardcore segments.

USAGE

```
declare hcs_$high_low_seg_count entry (fixed bin, fixed bin);

call hcs_$high_low_seg_count (nonhardcore_seg_count,
      lowest_nonhardcore_segno);
```

ARGUMENTS

nonhardcore_seg_count
    is the number of nonhardcore segment numbers being used. (Output)

lowest_nonhardcore_segno
is the lowest nonhardcore segment number. (Output)

ACCESS REQUIRED

No access is required.

Name: hcs_$history_regs_get

This entry point returns the current state of the per-process history register switch.

USAGE

```
declare hcs_$history_regs_get entry (bit (1) aligned);

call hcs_$history_regs_get (current_state);
```

ARGUMENTS

current_state
is the current state of the per-process history register switch. (Output)

This page intentionally left blank.

Name: hcs_$history_regs_set

This entry point controls the state of the per-process switch. If this per-process switch is set on ("1"b), then history registers of the processor that a process was executing on at the time of a signalable fault (e.g., illegal_procedure) are stored by the fault module (fim) and copied into the signaller's stack frame (return_to_ring_0_). If the per-process switch is set off and the per-system switch (wired_hardcore_data$global_hregs) is off, then the history register block in the signaller's stack frame is set to all zeros.

USAGE

```
declare hcs_$history_regs_set entry (bit (1) aligned);

call hcs_$history_regs_set (desired_state);
```

ARGUMENTS

desired_state
: is the desired state of the per-process switch. (Input)
"1"b on
"0"b off

Name: hcs_$initiate

This entry point, when given a pathname and a reference name, makes known the segment defined by the pathname, initiates the given reference name, and increments the count of initiated reference names for the segment.

Use this entry point when you need to initiate a file with a nonblank (nonnull) reference name, and use the initiate_file_ subroutine when you need to initiate a file with a blank (null) reference name.

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);
```

ARGUMENTS

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

entryname
: is the entry name of the segment. (Input)

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

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

copy_ctl_sw
    is obsolete, and should be set to zero. (Input)

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

code
    is a storage system status code. (Output)

NOTES

The user must have nonnull access on the segment (the entryname argument) in order to make it known.

If a segment is concurrently initiated more than a system-defined number of times, the usage count of the segment is said to be in an overflowed condition, and further initiations do not affect the usage count. This affects the use of the hcs_$terminate_noname and hcs_$terminate_name entry points. 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.

If entryname cannot be made known, a null pointer is returned for seg_ptr and the returned value of code indicates the reason for failure. Thus, the usual way to test whether the call was successful is to check the pointer, not the code, since the code may be nonzero even if the segment was successfully initiated. If entryname is already known to the user's process, code is returned as error_table_$segknown and the seg_ptr argument contains a nonnull pointer to entryname. If ref_name has already been initiated in the current ring, the code is returned as error_table_$namedup. The seg_ptr argument contains a valid pointer to the segment being initiated. If entryname is not already known, and no problems are encountered, seg_ptr contains a valid pointer and code is 0.

Name: hcs_$initiate_count

The hcs_$initiate_count entry point, when given a pathname and a reference name, causes the segment defined by the pathname to be made known and the given reference name initiated. A segment number is assigned and returned as a pointer and the bit count of the segment is returned. *

Use this entry point when you need to initiate a file with a nonblank (nonnull) reference name, and use the initiate_file_ subroutine when you need to initiate a file with a blank (null) reference name.

USAGE

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

call hcs_$initiate_count (dir_name, entryname, ref_name, bit_count,
      copy_ctl_sw, seg_ptr, code);
```

ARGUMENTS

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

entryname
: is the entry name of the segment. (Input)

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

bit_count
: is the bit count of the segment. (Output)

copy_ctl_sw
: is obsolete, and should be set to zero. (Input)

seg_ptr
: is a pointer to the segment. (Output)

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

NOTES

The user must have nonnull access on the segment (the entryname argument) in order to make it known.

If entryname cannot be made known, a null pointer is returned for seg_ptr and the returned value of code indicates the reason for failure. Thus, the usual way to test whether the call was successful is to check the pointer, not the code, since the code may be nonzero even if the segment was successfully initiated. If entryname is already known to the user's process, code is returned as error_table_$segknown and the seg_ptr argument contains a nonnull pointer to entryname. If entryname is not already known, and no problems are encountered, seg_ptr contains a valid pointer and code is 0. If ref_name has already been initiated in the current ring, the code is returned as error_table_$namedup. The seg_ptr argument contains a valid pointer to the segment being 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_$initiate_search_rules

This entry point provides the user with a subroutine interface for specifying the search rules that he wants to use in his process.

USAGE

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

call hcs_$initiate_search_rules (search_rules_ptr, code);
```

ARGUMENTS

search_rules_ptr
is a pointer to a structure containing the new search rules. See "Information Structure" below. (Input)

code
is a storage system status code. (Output)

INFORMATION STRUCTURE

The structure pointed to by search_rules_ptr is declared as follows:

```
dcl 1 search_rules              aligned,
      2 number                  fixed bin,
      2 names                   (21) char(168) aligned;
```

STRUCTURE ELEMENTS

number
: is the number of search rules contained in the array. The current maximum number of search rules the user can define is 21.

names
: are the names of the search rules. Two types of search rules are permitted: absolute pathnames of directories to be searched or keywords.

LIST OF KEYWORDS

initiated_segments
: search for the already initiated segments.

referencing_dir
: search the containing directory of the segment making the reference.

working_dir
: search the working directory.

process_dir
: search the process directory.

home_dir
: search the home directory.

set_search_directories
: insert the directories following this keyword into the default search rules after working_dir, and make the result the current search rules.

site-defined keywords
: may also be specified. These keywords may expand into one or more directory pathnames. The keyword, default, is always defined to be the site's default search rules.

NOTES

The set_search_directories keyword, when used, must be the first search rule specified and the only keyword used. If this keyword is used, hcs_$initiate_search_rules sets the default search rules, and then inserts the specified directories in the search rules after the working directory.

Some of the keywords, such as set_search_directories, are expanded into more than one search rule. The limit of 21 search rules applies to the final number of search rules to be used by the process as well as to the number of rules contained in the array.

The search rules remain in effect until this entry point is called with a different set of rules or the process is terminated.

Codes that can be returned from this entry point are:

error_table_$bad_string (not a pathname or keyword)
error_table_$notadir
error_table_$too_many_sr

Additional codes can be returned from other procedures that are called by hcs_$initiate_search_rules.

For the values of the site-defined keywords, the user may call the hcs_$get_system_search_rules entry point.

Name: hcs_$list_acl

This 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.

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);
```

ARGUMENTS

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

entryname
is the entryname of the segment. (Input)

area_ptr
points to an area in which the list of ACL entries, which make up the entire ACL of the segment, is allocated. (Input) 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).

area_ret_ptr
points to the start of the allocated list of ACL entries. (Output)

acl_ptr
if area_ptr is null, then acl_ptr points to an ACL structure, segment_acl_array, into which mode information is placed for the access names specified in that same structure. The segment_acl_array structure is discussed in the description of hcs_$add_acl_entries. (Input)

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_array structure allocated in the area pointed to by area_ptr, if area_ptr is not null.

code
is a storage system status code. (Output)

NOTES

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_array 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.

Name: hcs_$list_dir_acl

This 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. *

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);
```

ARGUMENTS

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

entryname
is the entryname of the directory. (Input)

area_ptr
points to an area in which the list of ACL entries, which make up the entire ACL of the directory, is allocated. (Input) 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).

area_ret_ptr
points to the start of the allocated list of ACL entries. (Output)

acl_ptr
if area_ptr is null, then acl_ptr points to an ACL structure, dir_acl_array, into which mode information is placed for the access names specified in that same structure. The dir_acl_array structure is discussed in the description of hcs_$add_dir_acl_entries. (Input)

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_array structure allocated in the area pointed to by area_ptr, if area_ptr is not null.

code
is a storage system status code. (Output)

NOTES

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_array structure either has status_code set to 0 and contains the mode of the directory or has status_code set to error_table_$user_not_found and contains a mode of 0.

Name: hcs_$list_dir_inacl

This entry point is used either to list the entire initial access control list (initial ACL) for new directories created for the specified ring within the specified directory or to
* return the access modes for specified initial ACL entries.

USAGE

```
declare hcs_$list_dir_inacl entry (char(*), char(*), ptr, ptr, ptr,
      fixed bin, fixed bin(3), fixed bin(35));

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

ARGUMENTS

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

entryname
is the entryname of the directory. (Input)

area_ptr
: points to an area into which the list of initial ACL entries, which makes up the entire initial ACL of the directory, is allocated. (Input) If area_ptr is null, then the user wants access modes for certain initial ACL entries; these will be specified by the structure pointed to by acl_ptr (see below).

area_ret_ptr
: points to the start of the allocated list of initial ACL entries. (Output)

acl_ptr
: if area_ptr is null, then acl_ptr points to an initial ACL structure, dir_acl_array, into which mode information is placed for the access names specified in that same structure. The dir_acl_array structure is discussed in the description of hcs_$add_dir_inacl_entries. (Input)

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

ring
: is the ring number of the initial ACL. (Input)

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

NOTES

If acl_ptr is used to obtain modes for specified access names (rather than obtaining modes for all access names on the initial ACL), then each initial ACL entry in the dir_acl_array 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.

Name: hcs_$list_inacl

This entry point is used either to list the entire initial access control list (initial ACL) for new segments created for the specified ring within the specified directory or to * return the access modes for specified initial ACL entries.

USAGE

```
declare hcs_$list_inacl entry (char(*), char(*), ptr, ptr, ptr,
      fixed bin, fixed bin(3), fixed bin(35));

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

ARGUMENTS

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

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

area_ptr
: points to an area into which the list of initial ACL entries, which makes up the entire initial ACL of the directory, is allocated. (Input) If area_ptr is null, then the user wants access modes for certain initial ACL entries; these will be specified by the structure pointed to by acl_ptr (see below).

area_ret_ptr
: points to the start of the allocated list of initial ACL entries. (Output)

acl_ptr
: if area_ptr is null, then acl_ptr points to an initial ACL structure, segment_acl_array, into which mode information is to be placed for the access names specified in that same structure. The segment_acl_array structure is discussed in the description of hcs_$add_inacl_entries. (Input)

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

ring
: is the ring number of the initial ACL. (Input)

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

NOTES

If acl_ptr is used to obtain modes for specified access names (rather than obtaining modes for all access names on the initial ACL), then each initial ACL entry in the segment_acl_array 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.

Name: hcs_$lv_attached

This entry point checks to see if a logical volume is attached and available for use in this process.

USAGE

```
dcl hcs_$lv_attached entry (bit(36) aligned) returns (fixed bin(35));

code = hcs_$lv_attached (lvid);
```

ARGUMENTS

lvid
is the logical volume id. Use mdc_$find_lvid to get the logical volume identifier for a given logical volume name. (Input)

code
is a standard system status code. (Output) Its possible values are:

0
the volume is attached and available for use.

error_table_$logical_volume_not_connected
the volume is private and has not been attached in this process.

error_table_$logical_volume_not_defined
the volume specified by lvid is not known to the system.

ACCESS REQUIRED

No access is required.

Name: hcs_$make_entry

This entry point, when given a reference name and an entry point name, returns the value of a specified entry point. If the reference name has not yet been initiated, the search rules are used to find a segment or multisegment file (MSF) with a name the same as the reference name. The file is made known and the reference name initiated. Use hcs_$make_ptr to have a pointer returned.

USAGE

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

call hcs_$make_entry (ref_ptr, entryname, entry_point_name, entry_point,
      code);
```

ARGUMENTS

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

entryname
: is the entryname or reference name of the file. (Input)

entry_point_name
: is the name of the entry point to be located. (Input)

entry_point
: is the value of the segment entry point specified by entryname and entry_point_name. (Output)

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. Search rules are described in the Programmer's Reference Manual. Normally ref_ptr is null. If the segment pointed to by ref_ptr is a component of an object MSF, the directory used as the referencing dir is the directory containing the MSF, rather than the MSF itself.

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 an entry value referring to the base of the segment is returned. In any case, the segment or MSF 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.

Name: hcs_$make_ptr

This 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 or multisegment file (MSF) with a name the same as the reference name. The file is made known and the reference name initiated. Use hcs_$make_entry to have entry values returned.

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);
```

ARGUMENTS

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

entryname
: is the entryname of the file. (Input)

entry_point_name
: is the name of the entry point to be located. (Input)

entry_point_ptr
: is the pointer to the segment entry point specified by entryname and entry_point_name. (Output)

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. For a discussion of standard search rules, refer to the Programmer's Reference Manual. Normally ref_ptr is null. If the segment pointed to by ref_ptr is a component of an object MSF, the directory used as the referencing dir is the directory containing the MSF, rather than the MSF itself.

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 or MSF 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.

Name: hcs_$make_seg

This 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);
```

ARGUMENTS

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

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

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

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

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

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.

If the segment cannot be created or made known, a null pointer is returned for seg_ptr and the returned value of code indicates the reason for failure. Thus, the usual way to test whether the call was successful is to check the pointer, not the code, since the code may be nonzero even if the segment was successfully initiated.

Name: hcs_$quota_move

This entry point moves all or part of a quota between two directories, one of which is immediately inferior to the other.

USAGE

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

call hcs_$quota_move (dir_name, entryname, quota_change, code);
```

ARGUMENTS

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

entryname
is the entryname of the directory. (Input)

quota_change
is the number of records of secondary storage quota to be moved between the superior directory and the inferior directory. (Input) (See "Notes" below.)

This page intentionally left blank.

code
    is a storage system status code. (Output)

NOTES

The entryname specified by the entryname argument must be a directory.

The user must have modify permission on both directories.

After the quota change, the remaining quota in each directory must be greater than the number of records used in that directory.

The quota_change argument can be either a positive or negative number. If it is positive, the quota is moved from dir_name to entryname. If it is negative, the move is from entryname to dir_name. If the change results in zero quota left on entryname, that directory is assumed to no longer contain a terminal quota and all of its used records are reflected up to the used records on dir_name. It is a restriction that no quota in any of the directories superior to entryname can be modified from a nonzero value to a zero value by this subroutine.

Name: hcs_$quota_read

This entry point returns the segment record quota and accounting information for a directory.

USAGE

```
declare hcs_$quota_read entry (char(*), fixed bin(18), fixed bin(71),
     bit(36) aligned, bit(36), fixed bin(1), fixed bin(18),
     fixed bin(35));

call hcs_$quota_read (dir_name, quota, trp, tup, sons_lvid, tacc_sw,
     used, code);
```

ARGUMENTS

dir_name
    is the pathname of the directory for which quota information is desired. (Input)

quota
    is the segment record quota in the directory. (Output)

trp
    is the time-record product (trp) charged to the directory. (Output) This double-precision number is in units of record-seconds.

tup
is the time, expressed in storage system time format (the high-order 36 bits of the 52-bit time returned by the clock_ subroutine), that the trp was last updated. (Output)

sons_lvid
is the logical volume ID for segments contained in this directory. (Output)

tacc_sw
is the terminal account switch. (Output) The setting of this switch determines how charges are made.
1 records are charged against the quota in this directory
0 records are charged against the quota in the first superior directory with a terminal account

used
is the number of records used by segments in this directory and by segments in nonterminal inferior directories. (Output)

code
is a storage system status code. (Output)

NOTES

If the directory contains a nonterminal account, the quota, trp, and tup are all zero. The variable specified by used, however, is kept up-to-date and represents the number of records in this directory and inferior, nonterminal directories.

Name: hcs_$release_segment_numbers

This entry point releases reserved segment numbers which are not associated with segments.

USAGE

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

call hcs_$release_segment_numbers (first_segno, block_size, code);
```

ARGUMENTS

first_segno
is the first segment number of the reserved block. (Input)

block_size
    is the number of segment numbers to be released. (Input)

code
    is error_table_$invalidsegno if any portion of the segment number range is an invalid segment number. It is error_table_$segknown if any of the segment numbers is known. (Output)

NOTES

This entry should be used in the cleanup handler of programs which reserve segment number blocks using hcs_$reserve_segment_numbers. If code is non-zero, no segment numbers were released. For example, suppose that a program reserves a block of ten segment numbers and a cleanup condition occurs after only four of these segment numbers have been used in calls to hcs_$initiate. The cleanup handler should call hcs_$release_segment_numbers for the last six segments of the block and then individually terminate the first four segments.

Name: hcs_$replace_acl

This 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_array 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);
```

ARGUMENTS

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

entryname
    is the entryname of the segment. (Input)

acl_ptr
    points to the user supplied segment_acl_array structure that is to replace the current ACL. (Input)

acl_count
    is the number of entries in the segment_acl_array structure. (Input)

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_array entries are added. (Input)
"0"b adds rw *.SysDaemon.* to ACL.
"1"b replaces the existing ACL with only the user-supplied segment_acl_array.

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_array entries is performed top to bottom, allowing later entries to overwrite previous ones if the access_name in the segment_acl_array structure is identical.

If the segment is a gate 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

This 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_array 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);
```

ARGUMENTS

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

entryname
is the entryname of the directory. (Input)

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

acl_count
: contains the number of entries in the dir_acl_array structure. (Input)

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_array entries are added. (Input)
"0"b adds sma *.SysDaemon.* to ACL.
"1"b replaces the existing ACL with only the user-supplied dir_acl_array.

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_array entries is performed top to bottom, allowing later entries to overwrite previous ones if the access_name in the dir_acl_array structure is identical.

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

Name: hcs_$replace_dir_inacl

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

USAGE

```
declare hcs_$replace_dir_inacl entry (char(*), char(*), ptr, fixed bin,
     bit(1), fixed bin(3), fixed bin(35));

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

ARGUMENTS

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

entryname
is the entryname of the directory. (Input)

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

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

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

ring
is the ring number of the initial ACL. (Input)

code
is a storage system status code. (Output)

NOTES

If acl_count is zero, then the existing initial 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_array entries is performed top to bottom, allowing later entries to overwrite previous ones if the access_name in the dir_acl_array structure is identical.

Name: hcs_$replace_inacl

This entry point replaces an entire initial access control list (initial ACL) for new segments created for the specified ring within a specified directory with a user-provided initial ACL, and can optionally add an entry for *.SysDaemon.* with mode rw to the new initial ACL. The segment_acl_array structure described in the hcs_$add_acl_entries entry point is used by this entry point.

USAGE

```
declare hcs_$replace_inacl entry (char (*), char (*), ptr, fixed bin,
      bit(1), fixed bin(3), fixed bin(35));

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

ARGUMENTS

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

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

acl_ptr
: points to the user-supplied segment_acl_array structure that is to replace the current initial ACL. (Input)

acl_count
: contains the number of entries in the segment_acl_array structure. (Input)

no_sysdaemon_sw
: is a switch that indicates whether the rw *.SysDaemon.* entry is to be put on the initial ACL after the existing initial ACL is deleted and before the user-supplied segment_acl_array entries are added. (Input)
"0"b adds rw *.SysDaemon.* entry
"1"b replaces the existing initial ACL with only the user-supplied segment_acl_array

ring
: is the ring number of the initial ACL. (Input)

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

NOTES

If acl_count is zero, then the existing initial 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_array entries is performed top to bottom, allowing later entries to overwrite previous ones if the access_name in the segment_acl_array structure is identical.

Name: hcs_$reserve_segment_numbers

This entry point allows a user to reserve a block of contiguous segment numbers.

USAGE

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

call hcs_$reserve_segment_numbers (block_size, first_segno, code);
```

ARGUMENTS

block_size
: is the number of segments in the segment number group. (Input)

first_segno
: is the number of the first segment in the group assigned. (Output)

code
: is 0 if the allocation succeeded. It is error_table_$nrmkst if there is no group of block_size contiguous segment numbers available. (Output)

NOTES

This entry removes the contiguous segment number group from the available free segment numbers. The assigned segment numbers are available only by using hcs_$initiate and specifying a reserved segment number. See also the description of hcs_$release_segment_numbers.

Name: hcs_$reset_ips_mask

This entry point replaces the entire ips mask with a specified mask, and returns the previous value of the mask with a control bit of "0"b. It can be used at the end of a critical section of code to restore the mask to its former value. See "Notes" in the description of the create_ips_mask_ subroutine for a discussion of the control bit.

USAGE

```
declare hcs_$reset_ips_mask entry (bit(36) aligned, bit(36) aligned);

call hcs_$reset_ips_mask (mask, old_mask);
```

ARGUMENTS

mask
is the new ips mask, to replace the current one. (Input) A "1" bit in a mask position enables the corresponding ips interrupt.

old_mask
is the former value of the ips mask, with a control bit of "0"b. (Output)

NOTES

The create_ips_mask_ subroutine can be used to create a mask, given a set of ips names.

This entry point can be used at the end of a critical section of code to undo the mask changes made by the hcs_$set_ips_mask entry point. The old_mask returned by the latter entry point should be used as the value of the new mask set by this entry point.

Name: hcs_$set_256K_switch

This entry point sets the per-process switch which controls whether or not segments of maximum length 256K (262144 words) can be used. The standard maximum length for a segment, defined as sys_info$max_seg_size, is 255K (261120 words). The only supported use of 256K segments is for Fortran Very Large Arrays.

USAGE

```
declare hcs_$set_256K_switch entry (bit(2) aligned, bit(2) aligned,
     fixed bin(35));

call hcs_$set_256K_switch (new_switch, old_switch, code);
```

ARGUMENTS

new_switch
is the new control value. (Input) If it is "11"b, the process may use segments having a maximum length of 256K words. It it is "10"b, the process may not use segments having a maximum length larger than sys_info$max_seg_size (255K words).

old_switch
is the previous control value. (Output) It is always set, even if an error occurs.

code
is 0 if the operation was successful. It is error_table_$action_not_performed if new_switch was neither "11"b or "10"b or if the callers validation level is greater than the process initial ring.

NOTES

The following code sequence provides correct resetting of the switch by a cleanup handler.

```
dcl old_switch bit (2) aligned;
dcl  cleanup condition

  old_switch = ""b;                /*invlaid switch*/
  on cleanup begin;
     call hcs_$set_256K_switch (old_switch, (""b), ignore_code);
  end;
  call_hcs_$set_256K_switch ("11"b, old_switch, code),
                                   /*enable big segments*/
```

Name: hcs_$set_bc

This 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);
```

ARGUMENTS

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

entryname
is the entry name of the segment. (Input)

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

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_dir_ring_brackets

This entry point, given the pathname of the containing directory and the entryname of the subdirectory, sets the subdirectory's ring brackets.

USAGE

```
declare hcs_$set_dir_ring_brackets entry (char(*), char(*),
      (2) fixed bin(3), fixed bin(35));

call hcs_$set_dir_ring_brackets (dir_name, entryname, drb, code);
```

ARGUMENTS

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

entryname
: is the entryname of the subdirectory. (Input)

drb
: is a two-element array specifying the ring brackets of the directory. (Input) The first element contains the level required for modify and append permission; the second element contains the level required for status permission.

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

NOTES

The user must have modify permission on the containing directory. Also, the validation level must be less than or equal to both the present value of the first ring bracket and the new value of the first ring bracket that the user wishes set.

Ring brackets and validation levels are discussed in "Intraprocess Access Control" in the Programmer's Reference Manual.

Name: hcs_$set_dnzp_sw

This entry point allows the dnzp switch associated with the specified segment to be changed. The "don't null zero pages" (dnzp) switch indicates how zero pages of a segment are written to disk.

USAGE

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

call hcs_$set_dnzp_sw (dir_name, entryname, dnzp_sw, code);
```

ARGUMENTS

dir_name
: is the pathname of the directory containing the segment. (Input)

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

dnzp_sw
: is the new value of the dnzp switch. (Input) Its possible values are:
"0"b zero pages are nulled and not written to disk or charged against quota
"1"b zero pages are written to disk and charged against quota

code
: is a standard system status code. (Output) Its possible values are:

error_table_$bad_ring_brackets
: the ring brackets of the segment are less than the validation level of the user.

ACCESS REQUIRED

The user must have status and modify permission on the containing directory and w effective access to the segment.

Name: hcs_$set_dnzp_sw_seg

This entry point allows the dnzp switch associated with the specified segment to be changed. The "don't null zero pages" (dnzp) switch indicates how zero pages of a segment are written to disk.

USAGE

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

call hcs_$set_dnzp_sw_seg (seg_ptr, dnzp_sw, code);
```

ARGUMENTS

seg_ptr
is a pointer to the segment whose dnzp switch is to be modified. (Input)

dnzp_sw
is the new value of the dnzp switch. (Input) Its possible values are:
"0"b zero pages are nulled and not written to disk or charged against quota
"1"b zero pages are written to disk and charged against quota

code
is a standard system status code. (Output) Its possible values are:

error_table_$bad_ring_brackets
the ring brackets of the segment are less than the validation level of the user.

ACCESS REQUIRED

The user must have status and modify permission on the containing directory and w effective access to the segment.

Name: hcs_$set_entry_bound

This entry point, given a directory name and an entryname, sets the entry point bound of a segment.

The entry point bound attribute provides a way of limiting which locations of a segment may be targets of a call. This entry point allows the caller to enable or disable a hardware check of calls to a given segment from other segments. If the mechanism is enabled, all calls to the segment must be made to an entry point whose offset is less than the entry point bound.

In practice, this attribute is most effective when all of the entry points are located at the base of the segment. In this case, the entry point bound is the number of callable words.

USAGE

```
declare hcs_$set_entry_bound entry (char (*), char (*), fixed bin(14),
      fixed bin(35));

call hcs_$set_entry_bound (dir_name, entryname, entry_bound, code);
```

ARGUMENTS

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

entryname
is the entryname of the segment. (Input)

entry_bound
is the new value in words for the entry point bound of the segment. (Input) If the value of entry_bound is 0, then the mechanism is disabled.

code
is a storage system status code. (Output) (See "Notes" below.)

NOTES

A directory cannot have its entry point bound changed.

The user must have modify permission on the containing directory.

If an attempt is made to set the entry point bound of a segment greater than the system maximum of 16383, code is set to error_table_$argerr.

The hcs_$set_entry_bound_seg entry point can be used when a pointer to the segment is given, rather than a pathname.

Name: hcs_$set_entry_bound_seg

This entry point, given a pointer to a segment, sets the entry point bound of the segment.

The entry point bound attribute provides a way of limiting which locations of a segment may be targets of a call. This entry point allows the caller to enable or disable a hardware check of calls to a given segment from other segments. If the mechanism is enabled, all calls to the segment must be made to an entry point whose offset is less than the entry point bound.

In practice, this attribute is most effective when all of the entry points are located at the base of the segment. In this case, the entry point bound is the number of callable words.

USAGE

```
declare hcs_$set_entry_bound_seg entry (ptr, fixed bin(14),
     fixed bin(35));

call hcs_$set_entry_bound_seg (seg_ptr, entry_bound, code);
```

ARGUMENTS

seg_ptr
is a pointer to the segment whose entry point bound is to be changed. (Input)

entry_bound
is the new value in words for the entry point bound of the segment. (Input) If the value of entry_bound is 0, then the mechanism is disabled.

code
is a storage system status code. (Output) (See "Notes" below.)

NOTES

A directory cannot have its entry point bound changed.

The user must have modify permission on the containing directory.

If an attempt is made to set the entry point bound of a segment to greater than the system maximum of 16383, code is set to error_table_$argerr.

The hcs_$set_entry_bound entry point can be used when a pathname of the segment is given, rather than a pointer.

Name: hcs_$set_exponent_control

This entry point changes the current settings of the flags that control the system's handling of exponent overflow and underflow conditions. For more information on exponent control see "Notes".

USAGE

```
declare hcs_$set_exponent_control entry (bit(1) aligned, bit(1) aligned,
      float bin(63), fixed bin (35));

call hcs_$set_exponent_control (restart_underflow, restart_overflow,
      overflow_value, code);
```

ARGUMENTS

restart_underflow
: is "1"b if underflows should be automatically restarted, and "0"b otherwise. (Input)

restart_overflow
: is "1"b if overflows should be automatically restarted, and "0"b otherwise. (Input)

overflow_value
: is the value used for the result of the computation in the case of overflow. (Input)

code
: is a standard status code. (Output)

NOTES

When either of the two flags are set to zero, the corresponding error condition causes the appropriate fault condition to be signalled. If a flag is set to one, then the computation resulting in the error is automatically restarted. In the case of underflow its result is set to zero. In the case of positive overflow, its value is set to the value specified in overflow_value. In the case of negative overflow, the negative of overflow_value is used. The default value is the largest representable positive number, available as Default_exponent_control_overflow_value in the include file exponent_control.incl.pl1.

This subroutine affects only the system's handling of exponent overflow and underflow when the overflow condition or the underflow condition is raised. In certain cases, the error condition is raised instead; this subroutine does not affect the system's handling of such cases.

In programs not written in PL/I, the exponent_control_ subroutine should be used in place of hcs_$set_exponent_control.

Name: hcs_$set_ips_mask

This entry point replaces the entire ips mask with a supplied value, and returns the previous value of the mask with a control bit of "1"b. It can be used at the beginning of a critical section of code, to disable one or more ips interrupts, and turn on the control bit to indicate that some interrupts are disabled. See "Notes" below.

USAGE

```
declare hcs_$set_ips_mask entry (bit(36) aligned, bit(36) aligned);

call hcs_$set_ips_mask (mask, old_mask);
```

ARGUMENTS

mask
: is the new value to replace the ips mask. (Input) A "1" bit in each mask position enables the corresponding ips interrupt.

old_mask
: is the former value of the ips mask, with a control bit of "1"b. (Output)

NOTES

The IPS mask is a 36 bit quantity, 35 of the bits represent individual signals. The 36'th bit is the control bit, used to indicate the fact that the mask has been changed fro its "normal" value. hcs_$set_ips_mask returns an old_mask with the 36'th bit ON. hcs_$reset_ips_mask returns with the 36'th bit set OFF.

Masked IPS signals can have a serious effect on the behavior of a process. Therefore, it is important to have a reliable clean_up handler and any_other handler in effect while IPS signals are masked, to make sure that they are unmasked before the program returns to command level.

The handlers should obey the convention in the following piece of code. the important steps of the algorithm are as follows:

1) Clear the saved_mask.

2) Establish the handlers.

3) Mask IPS signals with hcs_$set_ips_mask.

4) Perform the critical operations.

5) Call hcs_$reset_ips_mask ONLY if the 35'th bit of the saved_mask is "1"b.

6) Revert the handlers.

This protocol insures that the handlers will never call hcs_$reset_ips_mask when hcs_$set_ips_mask has not been called, or fail to call it when hcs_$set_ips_mask has been called.

```
      .....
   saved_mask = (36)"0"b;  /* clear saved mask so 36'th bit is off */
   on cleanup call clean_up;
   on any_other call fix_up_and_continue;
   call hcs_$set_ips_mask (MASK, saved_mask);
   call DO_PROTECTED_CODE;
   call hcs_$reset_ips_mask (saved_mask, saved_mask);
   revert any_other, cleanup;
      .....
clean_up:  procedure;
   if substr (saved_mask, 36, 1) then call hcs_$reset_ips_mask (saved_mask,
      saved_mask);
      .....                  /* clean things up */
   return;
   end clean_up;

fix_up_and_continue:  procedure;
   if substr (saved_mask, 36, 1) then call hcs_$reset_ips_mask (saved_mask,
      saved_mask);
      .....                  /* close down critical operations */
   call continue_to_signal_ (0);
   end fix_up_and_continue;
```

Name: hcs_$set_max_length

This entry point, given a pathname, sets the maximum length (in words) of a segment.

USAGE

```
declare hcs_$set_max_length entry (char(*), char(*), fixed bin(19),
      fixed bin(35));

call hcs_$set_max_length (dir_name, entryname, max_length, code);
```

ARGUMENTS

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

entryname
    is the entryname of the segment. (Input)

max_length
is the new value in words for the maximum length of the segment. (Input)

code
is a storage system status code. (Output) (See "Notes" below.)

NOTES

A directory cannot have its maximum length changed.

The user must have modify permission on the containing directory.

The maximum length of a segment is accurate to units of 1024 words, and if max_length is not a multiple of 1024 words, it is set to the next multiple of 1024 words.

If an attempt is made to set the maximum length of a segment to greater than the system maximum, sys_info$max_seg_size, code is set to error_table_$argerr. The sys_info data base is described in the Programmer's Reference Manual. It is possible to set a segment maximum length as high as sys_info$seg_size_256K, but this is intended for Fortran only. General system support is still restricted to segments of length sys_info$max_seg_size

If an attempt is made to set the maximum length of a segment to less than its current length, code is set to error_table_$invalid_max_length.

The hcs_$set_max_length_seg entry point can be used when the pointer to the segment is given, rather than a pathname.

Name: hcs_$set_max_length_seg

This entry point, given the pointer to the segment, sets the maximum length (in words) of a segment.

USAGE

```
declare hcs_$set_max_length_seg entry (ptr, fixed bin(19),
     fixed bin(35));

call hcs_$set_max_length_seg (seg_ptr, max_length, code);
```

ARGUMENTS

seg_ptr
is the pointer to the segment whose maximum length is to be changed. (Input)

max_length
is the new value in words for the maximum length of the segment. (Input)

code
is a storage system status code. (Output) (See "Notes" below.)

NOTES

A directory cannot have its maximum length changed.

The user must have modify permission on the containing directory.

The maximum length of a segment is accurate to units of 1024 words, and if max_length is not a multiple of 1024 words, it is set to the next multiple of 1024 words.

If an attempt is made to set the maximum length of a segment to greater than the system maximum, sys_info$max_seg_size, code is set to error_table_$argerr. The sys_info data base is described in the Programmer's Reference Manual.

If an attempt is made to set the maximum length of a segment to less than its current length, code is set to error_table_$invalid_max_length.

The hcs_$set_max_length entry point can be used when a pathname of the segment is given, rather than the pointer.

Name: hcs_$set_ring_brackets

This entry point, given the directory name and entryname of a nondirectory segment, sets the segment's ring brackets.

USAGE

```
declare hcs_$set_ring_brackets entry (char (*), char (*),
      (3) fixed bin(3), fixed bin(35));

call hcs_$set_ring_brackets (dir_name, entryname, rb, code);
```

ARGUMENTS

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

entryname
is the entryname of the segment. (Input)

rb
is a three-element array specifying the ring brackets of the segment; see "Notes" below. (Input)

code
is a storage system status code. (Output)

NOTES

Ring brackets must be ordered as follows:

rb1 <= rb2 <= rb3

The user must have modify permission on the containing directory. Also, the validation level must be less than or equal to both the present value of the first ring bracket and the new value of the first ring bracket that the user wishes set.

Ring brackets and validation levels are discussed in "Intraprocess Access Control" in the Programmer's Reference Manual.

Name: hcs_$set_safety_sw

This entry point allows the safety switch associated with a segment or directory to be changed. The segment is designated by a directory name and an entryname. See "Segment, Directory, and Link Attributes" in the Programmer's Reference Manual for a description of the safety switch.

USAGE

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

call hcs_$set_safety_sw (dir_name, entryname, safety_sw, code);
```

ARGUMENTS

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

entryname
is the entryname of the segment or directory. (Input)

safety_sw
is the new value of the safety switch. (Input)
"0"b if the segment can be deleted.
"1"b if the segment cannot be deleted.

code
is a storage system status code. (Output)

NOTES

The user must have modify permission on the containing directory.

The hcs_$set_safety_sw_seg entry point can be used when the pointer to the segment is given, rather than a pathname.

Name: hcs_$set_safety_sw_seg

This entry point, given a pointer to a segment, sets the safety switch of the segment. See "Segment, Directory, and Link Attributes" in the Programmer's Reference Manual for a description of the safety switch.

USAGE

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

call hcs_$set_safety_sw_seg (seg_ptr, safety_sw, code);
```

ARGUMENTS

seg_ptr
is the pointer to the segment. (Input)

safety_sw
is the new value of the safety switch. (Input)
"0"b if the segment can be deleted.
"1"b if the segment cannot be deleted.

code
    is a storage system status code. (Output)

NOTES

The user must have modify permission on the containing directory.

The hcs_$set_safety_sw entry point can be used when a pathname of the segment is given, rather than the pointer.

Name: hcs_$star_

This entry point is the star convention handler for the storage system. It is called with a directory name and an entryname that is a star name (contains asterisks or question marks). The directory is searched for all entries that match the given entryname. Information about these entries is returned in a structure. If the entryname is **, information on all entries in the directory is returned.

This entry point returns the storage system type and all names that match the given entryname. The hcs_$star_dir_list_ and hcs_$star_list_ entry points described below return more information about each entry. The hcs_$star_dir_list_ entry point returns only information kept in the directory branch, while the hcs_$star_list_ entry point returns information kept in the volume table of contents (VTOC). Accessing the VTOC is an additional expense, and it can be quite time consuming to access the VTOC entries for all branches in a large directory. Further, if the volume is not mounted, it is impossible to access the VTOC. Therefore, use of the hcs_$star_dir_list_ entry point is recommended for all applications in which information from the VTOC is not essential.

Status permission is required on the directory to be searched.

USAGE

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

call hcs_$star_ (dir_name, star_name, star_select_sw, area_ptr,
      star_entry_count, star_entry_ptr, star_names_ptr, code);
```

ARGUMENTS

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

star_name
    is the entryname that can contain asterisks or question marks. (Input)

star_select_sw
indicates what information is to be returned. (Input) It can be:
star_LINKS_ONLY (=1)
information is returned about link entries only.
star_BRANCHES_ONLY (=2)
information is returned about segment and directory entries only.
star_ALL_ENTRIES (=3)
information is returned about segment, directory, and link entries.

area_ptr
is a pointer to the area in which information is to be returned. If the pointer is null, star_entry_count is set to the total number of selected entries. See "Notes" below. (Input)

star_entry_count
is a count of the number of entries that match the entryname. (Output)

star_entry_ptr
is a pointer to the allocated structure in which information on each entry is returned. (Output)

star_names_ptr
is a pointer to the allocated array of all the entrynames in this directory that match star_name. See "Notes" below. (Output)

code
is a storage system status code. See "Status Codes" below. (Output)

NOTES

Even if area_ptr is null, star_entry_count is set to the total number of entries in the directory that match star_name. The setting of star_select_sw determines whether star_entry_count is the total number of link entries, the total number of segment and directory entries, or the total number of all entries.

If area_ptr is not null, the entry information structure and the name array are allocated in the user-supplied area.

This data structure is declared in star_structures.incl.pl1. The entry information structure is as follows:

```
dcl 1 star_entries (star_entry_count) aligned based (star_entry_ptr),
      2 type                          fixed bin (2) unsigned unaligned,
      2 nnames                        fixed bin (16) unsigned unaligned,
      2 nindex                        fixed bin (18) unsigned unaligned;
```

STRUCTURE ELEMENTS

type
specifies the storage system type of entry (the following named constants are declared in star_structures.incl.pl1):
star_LINK (=0)
star_SEGMENT (=1)
star_DIRECTORY (=2)

nnames
specifies the number of names for this entry that match star_name.

nindex
specifies the offset in star_names of the first name returned for this entry.

NOTES

All of the names that are returned for any one entry are stored consecutively in an array of all the names allocated in the user-supplied area. The first name for any one entry begins at the nindex offset in the array.

The names array, allocated in the user-supplied area and declared in star_structures.incl.pl1, is as follows:

```
dcl star_names (sum (star_entries (*).nnames)) char(32)
     based (star_names_ptr);
```

The user must provide an area large enough for the hcs_$star_ entry point to store the requested information.

STATUS CODES

If no match with star_name was found in the directory, code will be returned as error_table_$nomatch.

If star_name contained illegal syntax with respect to the star convention, code will be returned as error_table_$badstar.

If the user did not provide enough space in the area to return all requested information, code will be returned as error_table_$notalloc. In this case, the total number of entries (for hcs_$star_) or the total number of branches and the total number of links (for hcs_$star_list_ and hcs_$star_dir_list_) will be returned, to provide an estimate of space required.

USING THE INCLUDE FILE

A program using star_structures.incl.pl1 should declare addr, binary, and sum to be builtin. The arguments star_entry_count, star_entry_ptr, and star_names_ptr are declared in the include file along with named constants for the value of star_select_sw and the storage system type. One of the named constants for star_select_sw can be passed as an argument to hcs_$star_ along with star_entry_count, star_entry_ptr and star_names_ptr.

Name: hcs_$star_dir_list_

This entry point returns information about the selected entries, such as the mode and bit count for branches, and link pathnames for links. It returns only information kept in directory branches, and does not access the VTOC entries for branches. This entry point is more efficient than the hcs_$star_list_ entry point.

USAGE

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

call hcs_$star_dir_list_ (dir_name, star_name, star_select_sw, area_ptr,
      star_branch_count, star_link_count, star_list_branch_ptr,
      star_list_names_ptr, code);
```

ARGUMENTS

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

star_name
: is the entryname that can contain asterisks or question marks. (Input)

star_select_sw
: indicates what information is to be returned. (Input) It can be:
  star_LINKS_ONLY (=1)
  : information is returned about link entries only.
  star_BRANCHES_ONLY (=2)
  : information is returned about segment and directory entries only.
  star_ALL_ENTRIES (=3)
  : information is returned about segment, directory, and link entries.
  star_LINKS_ONLY_WITH_LINK_PATHS (=5)
  : information is returned about link entries only, including the pathname associated with each link entry.
  star_ALL_ENTRIES_WITH_LINK_PATHS (=7)
  : information is returned about segment, directory, and link entries, including the pathname associated with each link entry.

area_ptr
: is a pointer to the area in which information is to be returned. If the pointer is null, star_branch_count and star_link_count are set to the total number of selected entries. See "Notes" below. (Input)

star_branch_count
: is a count of the number of segments and directories that match the entryname. (Output)

star_link_count
: is a count of the number of links that match the entryname. (Output)

star_list_branch_ptr
: is a pointer to the allocated structure in which information on each entry is returned. (Output)

star_list_names_ptr
: is a pointer to the allocated array in which selected entrynames and pathnames associated with link entries are stored. (Output)

code
: is a storage system status code. See "Status Codes" above in the description of hcs_$star_ entry point. (Output)

NOTES

The names star_LINKS_ONLY through STAR_ALL_ENTRIES_WITH_LINK_PATHS are declared in star_structures.incl.pl1. The star_LINKS_ONLY, star_BRANCHES_ONLY, and star_ALL_ENTRIES are declared fixed bin (2) for compatability with hcs_$star and the star_LINKS_ONLY_WITH_LINK_PATHS and star_ALL_ENTRIES_WITH_LINK_PATHS are declared as fixed bin (3).

Even if area_ptr is null, star_branch_count and star_link_count may be set. If information on segments and directories is requested, star_branch_count is set to the total number of segments and directories that match star_name. If information on links is requested, star_link_count is the total number of links that match star_name.

If area_ptr is not null, an array of entry information structures and the names array, as described in the hcs_$star_ entry point above, are allocated in the user-supplied area. Each element in the structure array may be either of the structures described below (the star_links structure for links or the star_list_branch structure for segments and directories). The correct structure is indicated by the type item, the first item in both structures.

If the system is unable to access the VTOC entry for a branch, values of zero are returned for records used, date_time_contents_modified, and date_time_used, and no error code is returned. Callers of this entry point should interpret zeros for all three of these values as an error indication, rather than as valid data.

The first three items in each structure are identical to the ones in the structure returned by the hcs_$star_ entry point.

The following structure, declared in star_structures.incl.pl1, is used if the entry is a link:

```
dcl 1 star_links (star_branch_count + star_link_count)
                         aligned based (star_list_branch_ptr),
      2 type             fixed binary(2) unsigned unaligned,
      2 nnames           fixed binary(16) unsigned unaligned,
      2 nindex           fixed binary(18) unsigned unaligned,
      2 dtem             bit(36) unaligned,
      2 dtd              bit(36) unaligned,
      2 pathname_len     fixed binary(18) unsigned unaligned,
      2 pathname_index   fixed binary(18) unsigned unaligned;
```

STRUCTURE ELEMENTS

type
: specifies the storage system type of entry:
  star_LINK (=0)
  star_SEGMENT (=1)
  star_DIRECTORY (=2)

nnames
: specifies the number of names for this entry that match star_name.

nindex
: specifies the offset in star_list_names of the first name returned for this entry.

dtem
: is the date and time the link was last modified.

dtd
is the date and time the link was last dumped.

pathname_len
is the number of significant characters in the pathname associated with the link.

pathname_index
is the index in star_list_names of the link pathname.

If the pathname associated with each link was requested, the pathname is placed in the names array and occupies as many units as are needed. The index of the first unit is specified by pathname_index in the links array. The length of the pathname is given by pathname_len in the links array.

The following structure is the array of names. It is declared in star_structures.incl.pl1.

```
dcl star_list_names char (32) based (star_list_names_ptr)
     dimension (star_links (star_branch_count + star_link_count).nindex
      + star_links (star_branch_count + star_link_count).nnames
      + divide (star_links (star_branch_count + star_link_count).pathname_len
        + 31, 32, 17, 0)
        * binary (
          (star_links (star_branch_count + star_link_count).type = star_LINK)
          & (star_select_sw >= star_LINKS_ONLY_WITH_LINK_PATHS), 1));
```

The following based variable is used to get the pathname associated with link star_linkx in the star_links array. It is declared in star_structures.incl.pl1.

```
dcl star_link_pathname char (star_links (star_linkx).pathname_len) based
     (addr (star_list_names (star_links (star_linkx).pathname_index)));
```

Use the following structure if the entry is a segment or a directory. The star_dir_list_branch structure is the same as the star_list_branch structure except for the dtem and bit-count fields. This structure is declared in star_structures.incl.pl1.

```
dcl 1 star_dir_list_branch     (star_branch_count + star_link_count)
                             aligned based (star_list_branch_ptr),
      2 type                 fixed binary(2) unsigned unaligned,
      2 nnames               fixed binary (16) unsigned unaligned,
      2 nindex               fixed binary (18) unsigned unaligned,
      2 dtem                 bit(36) unaligned,
      2 pad                  bit(36) unaligned,
      2 mode                 bit(5) unaligned,
      2 raw_mode             bit(5) unaligned,
      2 master_dir           bit(1) unaligned,
      2 bit_count            fixed binary(24) unaligned;
```

STRUCTURE ELEMENTS

type
specifies the storage system type of entry:
star_LINK (=0)
star_SEGMENT (=1)
star_DIRECTORY (=2)

nnames
specifies the number of names for this entry that match star_name.

nindex
specifies the offset in star_list_names of the first name returned for this entry.

dtem
is the date and time the directory entry for the segment or directory was last modified.

pad
is unused space in this structure.

mode
is the current user's access mode to the segment or directory. See the "Notes" section in the description of hcs_$get_user_effmode in this manual for a more detailed description of access modes.

raw_mode
is the current user's access mode before ring brackets and access isolation are considered.

master_dir
specifies whether entry is a master directory:
"1"b yes
"0"b no

bit_count
is the bit count of the segment or directory.

NOTES

The star_links structure described for hcs_$star_list is used if the entry is a link.

Name: hcs_$star_list_

This entry point returns more information about the selected entries, such as the mode and records used for segments and directories and link pathnames for links. This entry point obtains the records used and the date of last modification and last use from the VTOC, and is, therefore, more expensive to use than the hcs_$star_dir_list_ entry point.

USAGE

```
declare hcs_$star_list_ entry (char(*), char(*), fixed bin(3), ptr,
      fixed bin, fixed bin, ptr, ptr, fixed bin(35));

call hcs_$star_list_ (dir_name, star_name, star_select_sw, area_ptr,
      star_branch_count, star_link_count, star_list_branch__ptr,
      star_list_names_ptr, code);
```

ARGUMENTS

The arguments are exactly the same as those for the hcs_$star_dir_list_ entry point above.

NOTES

The notes for hcs_$star_dir_list_ also apply to this entry.

The following structure, declared in star_structures.incl.pl1, is used if the entry is a segment or a directory:

```
dcl 1 star_list_branch (star_branch_count + star_link_count)
                       aligned based (star_list_branch_ptr),
      2 type           fixed binary(2) unsigned unaligned,
      2 nnames         fixed binary(16) unsigned unaligned,
      2 nindex         fixed binary(18) unsigned unaligned,
      2 dtcm           bit(36) unaligned,
      2 dtu            bit(36) unaligned,
      2 mode           bit(5) unaligned,
      2 raw_mode       bit(5) unaligned,
      2 master_dir     bit(1) unaligned,
      2 pad            bit(7) unaligned,
      2 records        fixed bin(18) unsigned unaligned;
```

STRUCTURE ELEMENTS

type
specifies the storage system type of entry:
star_LINK (=0)
star_SEGMENT (=1)
star_DIRECTORY (=2)

nnames
    specifies the number of names for this entry that match star_name.

nindex
    specifies the offset in star_list_names of the first name returned for this entry.

dtcm
    is the date and time the contents of the segment or directory were last modified.

dtu
    is the date and time the segment or directory was last used.

mode
    is the current user's access mode to the segment or directory.

raw_mode
    is the current user's access mode before ring brackets and access isolation are considered.

master_dir
    specifies whether entry is a master directory:
        "1"b yes
        "0"b no

pad
    is unused space in the structure.

records
    is the number of 1024-word records of secondary storage that have been assigned to the segment or directory.

Name: hcs_$status_

This entry point returns the most often needed information about a specified directory entry. Other entry points (hcs_$status_long, hcs_$status_minf, hcs_$status_mins) return more or less detailed information, and should be selected as the user requires.

USAGE

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

call hcs_$status_ (dir_name, entryname, chase_sw, status_ptr,
      status_area_ptr, code);
```

ARGUMENTS

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

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

chase_sw
: 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.

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

status_area_ptr
: is a pointer to the area in which names are returned. (Input) If the pointer is null, no names are returned (see "Notes" below).

code
: is a storage system status code. (Output) One of the possible codes returned can be:
  error_table_$no_s_permission
  The user lacks status permission on the containing directory, but has non-null access to the object. When this code is returned, all values in the status structure are valid, except for the offset of the array of names--no information on names is available if this error code is returned.

ENTRY INFORMATION

The status_ptr argument points to the following structure (defined in the include file status_structures.incl.pl1) if the entry is a segment or directory:

```
dcl 1 status_branch       aligned based (status_ptr),
      2 short             aligned,
        3 type            fixed bin (2) unaligned unsigned,
        3 nnames          fixed bin (16) unaligned unsigned,
        3 names_relp      bit (18) unaligned,
        3 dtcm            bit (36) unaligned,
        3 dtu             bit (36) unaligned,
        3 mode            bit (5) unaligned,
        3 raw_mode        bit (5) unaligned,
        3 pad1            bit (8) unaligned,
        3 records_used    fixed bin (18) unaligned unsigned;
```

STRUCTURE ELEMENTS

type
specifies the type of entry:
0 link
1 segment
2 directory
the named constants Link, Segment and Directory are declared in status_structures.incl.pl1.

nnames
specifies the number of names for this entry. It is set to zero if no names are allocated.

names_relp
is a pointer (relative to the base of the segment containing the user-specified free storage area) to an array of names. It is set to zero if no names are allocated.

dtcm
contains the date and time the contents of the segment or directory were last modified.

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

mode
contains the effective mode of the segment with respect to the current user's validation level. See the hcs_$append_branchx entry point for a description of modes. The values of these bits are the same as for the fixed bin (5) mode argument of the hcs_$append_branchx entry point.

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.

pad1
is unused space in this structure.

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

The status_ptr argument points to the following structure, (defined in the include file status_structures.incl.pl1) if the entry is a link:

```
dcl 1 status_link          aligned based (status_ptr),
      2 type               fixed bin (2) unaligned unsigned,
      2 nnames             fixed bin (16) unaligned unsigned,
      2 names_relp         bit (18) unaligned,
      2 dtem               bit (36) unaligned,
      2 dtd                bit (36) unaligned,
      2 pathname_length    fixed bin (17) unaligned,
      2 pathname_relp      bit (18) unaligned;
```

STRUCTURE ELEMENTS

type
specifies the type of entry:
0 link
1 segment
2 directory
the named constants Link, Segment and Directory are declared in status_structures.incl.pl1.

nnames
specifies the number of names for this entry. It is set to zero if no names are allocated.

names_relp
is a pointer (relative to the base of the segment containing the user-specified storage area) to an array of names. It is set to zero if no names are allocated.

dtem
contains the date and time the link was last modified.

dtd
contains the date and time the link was last dumped by the hierarchy dumper.

pathname_length
specifies the length in characters of the link pathname. It is set to zero if the pathname is not allocated.

pathname_relp
is a pointer (relative to the base of the segment containing the user-specified free storage area) to the link pathname. It is set to zero if the pathname is not allocated.

NOTES

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

If the status_area_ptr argument is not null, entrynames are returned in the following structure (defined in include file status_structures.incl.pl1) allocated in the user-specified area:

```
dcl status_entry_names (status_branch.nnames) character (32) aligned
      based (pointer (status_area_ptr, status_branch.names_relp));
```

The first name in this array is defined as the primary name of the entry. The user must provide an area that is large enough to accommodate a reasonable number of names. If for any reason the entrynames cannot be allocated, status_branch.names_relp will be zero.

Link pathnames are returned using the following declaration (defined in include file status_structure.incl.pl1) allocated in the user-specified area:

```
dcl status_pathname character (status_link.pathname_length) aligned
      based (pointer (status_area_ptr, status_link.pathname_relp));
```

If for any reason the link pathname cannot be allocated. status_link.pathname_relp will be zero.

For compatability with older programs, if entryname is given as a null string, status is returned on the directory dir_name.

ACCESS REQUIREMENTS

The user must have either status permission on the containing directory or nonnull access to the object to obtain complete information. Entrynames, however, are not returned unless the user has status permission on the containing directory.

Name: hcs_$status_long

This entry point returns most user-accessible information about a specified entry.

USAGE

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

call hcs_$status_long (dir_name, entryname, chase_sw, status_ptr,
      status_area_ptr, code);
```

ARGUMENTS

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

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

chase_sw
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

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

status_area_ptr
is a pointer to the area in which names are returned. (Input) If the pointer is null, no names are returned (see "Entry Information" below).

code
is a storage system status code (see "Access Requirements" below). (Output) One of the possible codes returned can be:
error_table_$no_s_permission
The user lacks status permission on the containing directory, but has non-null access to the object. When this code is returned, all values in the status structure are valid, except for the offset of the array of names--no information on names is available if this error code is returned.

ACCESS REQUIREMENTS

The user must have either status permission on the containing directory or nonnull access to the object to obtain complete information. Entrynames, however, are not returned unless the user has status permission on the containing directory.

NOTES

For compatability with older programs, if entryname is given as a null string, status is returned on the directory dir_name.

ENTRY INFORMATION

The entry_ptr argument points to the same structure as described under the hcs_$status_ entry point if the entry is a link. If the entry is a segment, directory, or multisegment file, it points to the following structure (defined in include file status_structures.incl.pl1):

```
dcl 1 status_branch         aligned based (status_ptr),
      2 short               aligned,
        3 type              fixed bin (2) unaligned unsigned,
        3 nnames            fixed bin (16) unaligned unsigned,
        3 names_relp        bit (18) unaligned,
        3 dtcm              bit (36) unaligned,
        3 dtu               bit (36) unaligned,
        3 mode              bit (5) unaligned,
        3 raw_mode          bit (5) unaligned,
        3 pad1              bit (8) unaligned,
        3 records_used      fixed bin (18) unaligned unsigned,
      2 long                aligned,
        3 dtd               bit (36) unaligned,
        3 dtem              bit (36) unaligned,
        3 lvid              bit (36) unaligned,
        3 current_length    fixed bin (12) unaligned unsigned,
        3 bit_count         fixed bin (24) unaligned unsigned,
        3 pad2              bit (8) unaligned,
        3 copy_switch       bit (1) unaligned,
        3 tpd_switch        bit (1) unaligned,
        3 mdir_switch       bit (1) unaligned,
        3 damaged_switch    bit (1) unaligned,
        3 pad3              bit (6) unaligned,
        3 ring_brackets     (0:2) fixed bin (6) unaligned unsigned,
        3 uid               bit (36) unaligned;
```

STRUCTURE ELEMENTS

type
: specifies the type of entry:
  0 link
  1 segment
  2 directory
  the named constants Link, Segment and Directory are declared in status_structures.incl.pl1.

nnames
: specifies the number of names for this entry. It is set to zero if no names are allocated.

names_relp
: is a pointer (relative to the base of the segment containing the user-specified free storage area) to an array of names. It is set to zero if no names are allocated.

dtcm
: contains the date and time the contents of the segment or directory were last modified.

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

mode
contains the effective mode of the segment with respect to the current user's validation level. For directory entries, the 4-bit is 1 (00100b).

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.

pad1
is unused space in this structure.

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

dtd
is the date and time the segment was last dumped by the hierarchy dumper.

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

lvid
is the ID of the logical volume on which this entry resides.

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

bit_count
is the bit count associated with the segment if type is 1. If type is 2, then this is zero for a directory, otherwise it is the number of components for a multisegment file.

pad2
is unused space in this structure.

copy_switch
contains the setting of the segment copy switch.
"0"b the default action on initiate in not to produce a copy.
"1"b the default action on initiate is to produce a copy.

tpd_switch
is obsolete and always returned as "0"b.

mdir_switch
is the master directory switch. It can be:
"0"b directory is not a master directory.
"1"b directory is a master directory.

damaged_switch
contains the setting of the damaged switch for the segment.
"0"b segment is undamaged or damage is undetected or user has previously reset the switch.
"1"b system has detected damage to the contents of the segment.

pad3
is unused space in this structure.

ring_brackets
contains the ring brackets of the segment.

uid
is the segment unique identifier.

NOTES

See the hcs_$status_ entry point for a description of the status_entry_names array.

Name: hcs_$status_minf

This entry point returns the bit count and entry type given the name of a directory and an entry. 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_sw, type, bit_count,
     code);
```

ARGUMENTS

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

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

chase_sw
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.

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

bit_count
is the bit count. (Output)

code
is a storage system status code. (Output)

Name: 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);
```

ARGUMENTS

seg_ptr
points to the segment about which information is desired. (Input)

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

bit_count
is the bit count. (Output)

code
is a storage system status code. (Output)

Name: hcs_$truncate_file

This 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 zeros in the words beyond the specified length.

USAGE

```
declare hcs_$truncate_file entry (char(*), char(*), fixed bin(19),
      fixed bin(35));

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

ARGUMENTS

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

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

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

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 zeros.

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

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

Name: hcs_$validate_processid

This entry determines whether a 36-bit quantity is the unique identifier of a process which is currently active on the system.

USAGE

```
declare hcs_$validate_processid entry (bit(36) aligned, fixed bin(35));

call hcs_$validate_processid (processid, code);
```

ARGUMENTS

processid
: contains a quantity which may be the unique identifier of an active process. (Input)

code
: is a standard status code. (Output) If processid is the unique identifier of an active process, the value returned is zero. Otherwise, the value is error_table_$process_unknown.

Name: hcs_$wakeup

This entry point sends an interprocess communication wakeup signal to a specified process over a specified event channel. If that process has previously called the ipc_$block entry point, it is awakened. See the ipc_ subroutine description.

USAGE

```
declare hcs_$wakeup entry (bit(36) aligned, fixed bin(71),
     fixed bin(71), fixed bin(35));

call hcs_$wakeup (process_id, channel_id, message, code);
```

ARGUMENTS

process_id
: is the process identifier of the target process. (Input)

channel_id
: is the identifier of the event channel over which the wakeup is to be sent. (Input)

message
: is the event message to be interpreted by the target process. (Input)

code
    is a standard status code. (Output)

Name: heap_manager_

Entry: heap_manager_$push_heap_level

This entry point creates a new heap level, allocates the heap header and chains the previous heap to the current heap. If the stack_header_ptr is null an error of error_table_$null_info_ptr is returned.

USAGE

```
declare heap_manager_$push_heap_level entry (pointer,
      fixed bin(17), fixed bin(35));

call heap_manager_$push_heap_level (stack_header_ptr, exe_level, code);
```

ARGUMENTS

stack_header_ptr
    is a pointer to the stack header. This can be obtained via the PL/1 builtin stackbaseptr(). (Input)

exe_level
    is the new execution level after the new heap is created. (Output)

code
    is a standard status code. (Output)

Entry: heap_manager_$pop_heap_level

This entry point resets the heap to the previous level freeing the old heap and any variables allocated therein.

USAGE

```
declare heap_manager_$pop_heap_level entry (pointer,
      fixed bin(35));

call heap_manager_$pop_heap_level (stack_header_ptr, code);
```

ARGUMENTS

stack_header_ptr
is a pointer to the stack header. This can be obtained via the PL/1 builtin stackbaseptr(). (Input)

code
is a standard status code. (Output)

Entry: heap_manager_$get_heap_header

This entry point returns a pointer to the heap header for the specified execution level. If the execution level does not exist an error of error_table_$no_heap_defined is returned.

USAGE

```
declare heap_manager_$get_heap_header entry (pointer,
      fixed bin(17), pointer, fixed bin(35));

call heap_manager_$get_heap_header (stack_header_ptr, exe_level,
      heap_header_ptr, code);
```

ARGUMENTS

exe_level
is the execution level of the heap required. If a -1 is passed then the current execution level is used. (Input)

stack_header_ptr
is a pointer to the stack header. This can be obtained via the PL/1 builtin stackbaseptr(). (Input)

heap_header_ptr
is a pointer to the heap header for the passed execution level. (Output).

code
is a standard status code. (Output)

Entry: heap_manager_$get_heap_level

This entry point returns the current execution level from the current heap header. If the heap does not exist an execution level of -1 is returned.

USAGE

```
declare heap_manager_$get_heap_level entry (pointer)
       returns (fixed bin(17));

exe_level = heap_manager_$get_heap_level (stack_header_ptr);
```

ARGUMENTS

stack_header_ptr
    is a pointer to the stack header. This can be obtained via the PL/1 builtin stackbaseptr(). (Input)

Entry: heap_manager_$get_heap_area

This entry point returns a pointer to the heap area for the specified level. The area is max_segsize - 50 words. If the heap level specified does not exist an error of error_table_$no_heap_defined is returned.

USAGE

```
declare heap_manager_$get_heap_area entry (pointer, fixed bin(17),
     pointer, fixed bin(35));

call heap_manager_$get_heap_area (stack_header_ptr, exe_level,
     heap_area_ptr, code);
```

ARGUMENTS

exe_level
    is the execution level of the heap area required. If a -1 is passed then the current execution level is used. (Input)

stack_header_ptr
    is a pointer to the stack header. This can be obtained via the PL/1 builtin stackbaseptr(). (Input)

heap_area_ptr
    is pointer to the heap area for the passed level. (Output)

code
    is a standard status code. (Output)

Name: help_

The help_ subroutine performs the basic work of the help command The help_ subroutine is called to print selected information from one or more info segments. The caller may select: what information is to be printed; what search list is to be used to find the info segments; what suffix the info segments must have. Thus, the help_ provides an interface for implementing a subsystem help command.

Several entry points in the help_ subroutine are described below. help_$init must be called before calling the help_ or help_$check_info_segs entry points. The help_ or help_$check_info_segs entry points may then be called one or more times. When the caller no longer needs the help_args structure, help_$term must be called to release the temporary segment containing the help_args structure.

The help_ entry point searches for info segments, selects information blocks (infos), and prints the information. The caller provides information in the help_args structure (obtained in the call to help_$init) to select the infos to be printed and the type of information to be printed.

The help_ subroutine may ask the user questions about how much information should be printed. These questions and the responses the user may give are in the description of the help command. Questions are asked using the command_query_ subroutine.

USAGE

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

call help_ (caller, Phelp_args, suffix, progress, code);
```

ARGUMENTS

caller
: is the name of the calling program, on whose behalf the temporary segment containing the help_args structure is obtained. (Input)

Phelp_args
: is a pointer to the help_args structure, described under "Information Structure" below. (Output)

suffix
: is the suffix which must appear in the entrynames of info segments to be processed by this invocation of help_. This suffix is also assumed when omitted from the (final or only) entryname of values given for help_args.path.value in the help_args structure (see "Information Structure" below). If a null string is given, then no suffix is required in info segment entrynames, and none is assumed in values of help_args.path.value. (Input)

progress

is a special status code that indicates which stage of processing help_ was performing when an error occurs. (Output) The following values may be returned:

1 the Phelp_args argument points to an unimplemented version of the help_args structure.

2 help_args.Npaths is not positive, indicating that no info_names were given. help_ is unable to select info segments for printing.

3 an error is encountered while evaluating one or more of the help_args.path.value values. help_args.path.code indicates the particular error encountered in each value.

4 no fatal errors are encountered. Some infos matching help_args.path were found. Any nonfatal errors encountered while finding the infos are diagnosed to the user, unless help_args_.Sctl.inhibit_errors is on. A list of infos to be compared with the -section and -search criteria is created.

5 infos matching the -section and -search criteria are printed. A nonzero code argument is returned only when no infos match the -section and -search criteria. help_ does not report such an error to the user. The caller is responsible for doing this.

code

is a standard status code. (Output) When progress is 1, the code may have the following value:

error_table_$unimplemented_version

help_ does not support the version of the help_args structure pointed to by the Phelp_args pointer argument.

When progress is 2, the code may have the following value:

error_table_$noarg

help_args.Npaths was not positive.

When progress is 3, the code may have any value returned by expand_pathname_$add_suffix or check_star_name_$entry, or it may have the following value:

error_table_$inconsistent

a star name was given when help_args.Sctl.ep = "1"b, or when a value of help_args.path.value contains a subroutine entry point name.

When progress is 4, the code may have the following value:

error_table_$nomatch

no info segments match any of the help_args.path elements. For each help_args.path.value element, help_ prints an error message when no matching info segments are found.

When progress is 5, the code may have the following value:

error_table_$nomatch

none of the infos selected by help_args.path contain sections whose titles match the selection criteria given in help_args.scn, or paragraphs that match the selection criteria given in help_args.srh. help_ does not report this error to the user. The caller of help_ must do this.

INFORMATION STRUCTURE

The Phelp_args argument points to the following structure, which is declared in help_args_.incl.pl1:

```
dcl 1 help_args                         aligned based (Phelp_args),
      2 version                         fixed bin,
      2 Sctl,
       (3 he_only,
        3 he_pn
        3 he_info_name,
        3 he_counts,
        3 title,
        3 scn,
        3 srh,
        3 bf,
        3 ca,
        3 ep,
        3 all,
        3 inhibit_errors)               bit(1) unal,
        3 mbz1                          bit(24) unal,
      2 Nsearch_dirs                    fixed bin,
      2 Npaths                          fixed bin,
      2 Ncas                            fixed bin,
      2 Nscns                           fixed bin,
      2 Nsrhs                           fixed bin,
      2 min_Lpgh                        fixed bin,
      2 max_Lpgh                        fixed bin,
      2 Lspace_between_infos            fixed bin,
      2 min_date_time                   fixed bin(71),
      2 sci_ptr                         ptr,
      2 pad2 (8)                        fixed bin,
      2 search_dirs (0 refer (help_args.Nsearch_dirs)) char(168) unal,
      2 path (0 refer (help_args.Npaths)),
        3 value                         char(425) varying,
        3 info_name                     char(32) unal,
        3 dir (1)                       char(168) unal,
        3 ent                           char(32) unal,
        3 ep                            char(32) varying,
        3 code                          fixed bin(35),
        3 S,
         (4 pn_ctl_arg,
          4 info_name_not_starname,
          4 less_greater,
          4 starname_ent,
          4 starname_info_name,
          4 separate_info_name)         bit(1) unal,
          4 pad3                        bit(30) unal,
```

```
    2 ca (0 refer (help_args.Ncas))    char(32) varying,
    2 scn (0 refer (help_args.Nscns)) char(80) varying,
    2 srh (0 refer (help_args.Nsrhs)) char(80) varying,
  Phelp_args                          ptr,
  Vhelp_args_2                        fixed bin int static
                                        options(constant) init(2);
```

STRUCTURE ELEMENTS

version
: is the version number of this structure (currently 2). The variable Vhelp_args_2 should be used when checking this version number.

Sctl
: are flags controlling the operations which help_ performs on the info segments. help_$init sets all of these flags to "0"b.

Sctl.he_only
: help_ prints only a heading line identifying matching info segments. The heading line includes the info heading, plus heading fields selected by Sctl.he_pn, Sctl.he_info_name and Sctl.he_counts. No other information is printed. This flag is mutually exclusive with all other Sctl flags except those named above, Sctl.scn and Sctl.srh.

Sctl.he_pn
: help_ includes the info pathname in all heading lines. help_ prints other information along with the heading line, as requested by the other Sctl flags. If no other flags are set, help_ prints the heading line followed by the first paragraph of information.

Sctl.he_info_name
: help_ includes the info_name in all heading lines. This info_name is included only when help_args.path identifies an info segment containing more than one information block (info). help_ prints other information along with the heading line, as requested by other Sctl flags. If no other flags are set, help_ prints the heading line followed by the first paragraph of information.

Sctl.he_counts
: help_ includes info line counts and subroutine info entry point counts in all heading lines. help_ prints other information along with the heading line, as requested by other Sctl flags. If no other flags are set, help_ prints the heading line followed by the first paragraph of information.

Sctl.title
: help_ prints all section titles (including section line counts), then asks if the user wants to see the first paragraph. Normally, help_ just begins printing the first paragraph.

Sctl.scn
help_ searches section titles for one containing all of the substrings given in help_args.scn. If a matching title is found, help_ begins printing information requested by other Sctl flags. If no other flags are set, help_ prints the first paragraph of the matching section. If no matching title is found, help_ skips the info without comment.

Sctl.srh
help_ searches all paragraphs for one containing all of the substrings given in help_args.srh. If a matching paragraph is found, help_ begins printing information requested by other Sctl flags. If no other flags are set, help_ prints the matching paragraph. If no matching paragraph is found, help_ skips the info without comment. If Sctl.scn is also "1"b, then only paragraphs from the matching section to the end of the info are searched.

Sctl.bf
help_ prints only a brief summary of an info describing a command, active function, or subroutine. This flag is mutually exclusive with all other Sctl flags except Sctl.he_pn, Sctl.he_info_name, Sctl.he_counts, Sctl.ca, Sctl.scn and Sctl.srh.

Sctl.ca
for an info describing a command, active function, or subroutine, help_ prints only the descriptions of one or more arguments or control arguments identified by the substrings in help_args.ca. This flag is mutually exclusive with all other Sctl flags except Sctl.he_pn, Sctl.he_info_name, Sctl.he_counts, Sctl.bf, Sctl.scn and Sctl.srh.

Sctl.ep
help_ prints information describing the main entry point of a subroutine, rather than information describing the general characteristics of all subroutine entry points.

Sctl.all
help_ prints all of the info without asking the user any questions.

Sctl.inhibit_errors
help_ suppresses error messages which it normally prints to diagnose failure to find a given info or entrypoint within a subroutine info. If no matching infos are found, then help_ still returns a code of error_table_$nomatch.

Sctl.mbz1
is reserved for future use. help_$init sets this field to ""b.

Nsearch_dirs
is the number of directories help_ searches for info segments. The directory pathnames are given in help_args.search_dirs. This number is set by help_$init to the number of paths in the search list named in the call to help_$init, but the caller may change it before calling help_.

Npaths
is the number of info names help_ searches for. The names are given in help_args.path. The caller must set this number before calling help_. help_$init initializes it to zero.

Ncas
is the number of substrings help_ uses in searching for argument or control argument descriptions when help_args.Sctl.ca is given. The substrings are given in help_args.ca. help_$init initializes this number to zero.

Nscns
is the number of substrings help_ uses in searching for a matching section title when help_args.Sctl.scn is given. The substrings are given in help_args.scn. help_$init initializes this number to zero.

Nsrhs
is the number of substrings help_ uses in searching for a matching paragraph when help_args.Sctl.srh is given. The substrings are given in help_args.srh. help_$init initializes this number to zero.

min_Lpgh
is the length (in lines) of the shortest paragraph that help_ will consider as a distinct unit. Paragraphs shorter than this may be printed with their preceding paragraph, rather than asking the user if he wants to see the short paragraph. help_$init initializes this number to 4.

max_Lpgh
is the maximum number of lines of information that help_ allows in grouped paragraphs before asking the user whether he wants to see more. help_ will never group short paragraphs with their preceding paragraph if the total number of lines to be printed (including 2 blank lines between paragraphs) would exceed this number. help_$init initializes this number to 15.

Lspace_between_infos
is the number of blank lines which help_ prints between the last paragraph of one info and the heading line (or first paragraph) of the next. help_$init initializes this number to 2.

min_date_time
is a Multics clock value. Only infos modified on or after the time given in this clock value are selected. Info modification time is based upon the date_time_entry_modified of the segment containing the info. When an info segment contains more than one info, any date given in the info heading is used as the modification date for that info. help_$init initializes this number to -1, indicating that all infos are eligible for selection.

sci_ptr

is a pointer to the subsystem control structure for an ssu invocation. It is used by help_ to report error messages on behalf of the subsystem. This should be set by subsystems which are calling help_ directly. If an sci_ptr exists from a prior call to ssu_$create_invocation, then set help_args.sci_ptr to this value. Otherwise, after calling help_args_$init, call ssu_$create_standalone_invocation passing it help_args.sci_ptr; before calling help_$term, call ssu_$destroy_invocation passing it help_args.sci_ptr.

pad2

is reserved for future use. This field should not be set or referenced. help_$init sets this field to 0.

search_dirs

is an array of absolute pathnames specifying directories that help_ will look in for named infos. help_ searches for an info unless help_args.path.value contains less-than (<) or greater-than (>) characters, or unless help_args.path.S.pn_ctl_arg = "1"b. help_$init sets this array to the pathnames given for the search list named by its search_list_name argument. The caller can change this list before calling help_. Note that the search_dirs are absolute pathnames which are expanded from the rules in a search list. If the working directory may have changed between calls to help_, then the search list rules must be reevaluated before each call to help_. This can be accomplished by calling help_$init before each call to help_, and help_$term after each call.

path

is an array of minor structures that identify the infos to be printed.

path.value

is a value used to select one or more info segments. A relative or absolute pathname may be given, or just an entryname. The (final or only) entryname may be a starname. A subroutine entry point name may follow the entryname. For example:

```
ioa_$rsnnl
```

or:

```
my_info_dir>extend_subr$init
```

A starname may not be given with a subroutine entry point name or when Sctl.ep = "1"b. A proper suffix (as defined by the suffix argument to the help_ entry point) is assumed if not given. If path.value contains a less-than or greater-than character, it is assumed to be the pathname of an info to be printed. Otherwise, path.value is assumed to be the entryname of an info which is searched for in directories named in the search_dirs array. Note that path.value has a maximum length of 425 characters to accommodate a maximum size pathname (168 characters), a maximum size entry point name (256 characters), plus a dollar sign ($) separator.

path.info_name
selects an info within the info segments found by path.value. Normally, the caller of help_ sets the info_name to a null string, causing help_ to use the (final or only) entryname from path.value (without its suffix) as the info_name. help_ then searches for an info segment having the info_name (with an appropriate suffix) as one of its segment names. help_ looks inside the segment to see if it is divided into different information blocks (infos). Lines of the form:

```
:Info: info_name1: ...info_nameN:  date info_heading
```

divide the segment into infos. For each info segment containing multiple infos, help_ searches for infos having an info_namei matching the info_name and prints only those infos.

When the caller of help_ gives a nonnull value for path.info_name, then the info_name need not be a name on the info segment itself. This is sometimes useful for subsystems which want to store all of their infos in a single info segment (to reduce storage costs, simplify maintenance of the infos or facilitate printing all of the information), but which do not want to add all of the info_names to the segment. This avoids the need for many names on the segment, and also prevents the system help command from accessing the infos whose names do not appear on the info segment. The star convention may be used in the path.info_name. Note that the info_namei given in a :Info: line of an info segment correspond to names on the info segment when a null path.info_name is given. However, when a nonnull path.info_name is given, the info_namei need not be unique within the info segment. help_ selects all infos having a matching info_namei in the order in which they appear in the info segment, even when path.info_name is not a star name. If path.info_name is set to a nonnull value, the pathS.info_name_not_starname must also be set.

path.dir
is the directory part of a pathname given as the value of path.value. help_ sets this value, and the caller of help_ need not set this value. The variable is a one-dimensional array so that it can be used interchangeably with the search_dirs array in searching for info segments.

path.ent
is the entryname part of a pathname given as the value of path.value. help_ sets this value, and the caller of help_ need not set this value.

path.ep
is the entry point name part of a name given in path.value. help_ sets this value, and the caller of help_ need not set this value.

path.code
is a standard status code associated with processing the value given in path.value. When help_ returns to its caller with a progress argument value of 3 and a nonzero status code argument, the caller of help_ should: examine each path.code; for nonzero values, report an error in path.value. path.code may have any of the values listed above for the code argument returned by help_ when the progress argument is 3.

path.S
are flags controlling the interpretation of path.value.

path.S.pn_ctl_arg
is "1"b if path.value is to be interpreted as a relative or absolute pathname, rather than as an entryname which should be searched for using the search_dirs. If the flag is "0"b, then help_ interprets path.value as a pathname only if it contains a less-than or greater-than character. The caller of help_ must set this flag to the appropriate value.

path.S.info_name_not_starname
is "1"b if path.info_name is not a star name, even though it may contain * or ? characters. A value of "0"b causes path.info_name to be treated as a star name if it contains * or ? characters. If the caller sets path.info_name to a nonnull value, then this switch must be set.

path.S.less_greater
is a flag that help_ uses to record that path.value contains less-than or greater-than characters, or that path.S.pn_ctl_arg was set. The caller of help_ need not set this flag.

path.S.starname_ent
is a flag that help_ uses to record the fact that the (final or only) entryname in path.value is a star name. The caller of help_ need not set this value.

path.S.starname_info_name
is a flag that help_ uses to record that path.info_name is a star name. The caller of help_ need not set this flag.

path.S.separate_info_name
is a flag that help_ uses to record that path.info_name was supplied by the caller of help_, rather than being extracted from path.value by help_. The caller of help_ need not set this flag.

path.S.pad3
is a reserved field. The caller of help_ must set this field to "0"b.

ca
is the array of substrings help_ uses in searching for argument or control argument descriptions when help_args.Sctl.ca is given. If any of these strings appears in the argument name line of an argument or control argument description, then help_ prints the entire description.

scn
is the array of substrings help_ uses in searching for a matching section title when help_args.Sctl.scn is given. All of these substrings must appear (in any order) in a matching section title. Comparisons are made after all substrings are translated to lowercase, so the letter case of the substrings does not matter.

srh
is the array of substrings help_ uses in searching for a matching paragraph when help_args.Sctl.srh is given. All of the substrings must appear (in any order) in a matching paragraph. Comparisons are made after all substrings are translated to lowercase, so the letter case of substrings does not matter.

Phelp_args
is a pointer to the help_args structure. help_$init returns a value for this pointer argument. help_, help_$check_info_segs and help_$term require the pointer as an input argument.

Vhelp_args_2
is a named constant which the caller of help_$init should use for the required_version argument. This constant can also be used to check the value of help_args.version.

NOTES

The structure above is somewhat complex, due to the many options provided by the help_ subroutine. Callers of help_ or help_$check_info_segs can use the following steps to set structure elements:

1. Set the Sctl flags to the required values. Set min_Lpgh, max_Lpgh, Lspace_between_infos, and min_date_time values if you wish to change the defaults supplied by help_$init.

2. If any of the search_dirs are to be set (or changed from the pathnames given in the search list named in the call to help_$init), then set Nsearch_dirs to the correct value, and set the search_dir array elements to the desired values.

3. Set Npaths to the number of info pathname/info_name input values. Set the elements of help_args.path for each of these input values. If the values are arguments in a subsystem help request, they can be placed in the help_args.path structure as each argument is processed. In this case, add 1 to Npaths as each argument is processed, then set help_args.path(Npaths) to the appropriate input values.

4. Provide substrings used in searching for argument or control argument descriptions, if any. Set Ncas to the appropriate value, then store the substrings in the ca array.

5. Provide substrings used in searching for section titles, if any. Set Nscns to the appropriate value, then store the substrings in the scn array.

6. Provide substrings used in searching for matching paragraphs, if any. Set Nsrhs to the appropriate value, then store the substrings in the srh array.

Note that when substrings for argument and control argument matching, section title matching, or paragraph matching are not provided, Ncas, Nscns, or Nsrhs above need not be set. help_$init initializes these values to zero.

Entry: help_$check_info_segs

This entry point searches for info segments modified since a given date. It returns a sorted list of info segments matching the selection criteria. The list is sorted by directory name, and within a directory by entryname. In addition, the help_$check_info_segs entry point flags entrynames found in more than one directory. All but the first such duplicate segment are marked with a cross reference flag and are sorted after all unique info segments. The caller provides the selection criteria in the help_args structure, obtained by calling help_$init. In particular, help_args.min_date_time specifies the info segment modification threshold.

USAGE

```
declare help_$check_info_segs entry (char(*), ptr, char(*), fixed bin,
      fixed bin(35), ptr);

call help_$check_info_segs (caller, Phelp_args, suffix, progress, code,
      PPDinfo_seg);
```

ARGUMENTS

caller
: is the name of the calling program, on whose behalf the temporary segment containing the help_args structure is obtained. (Input)

Phelps_args
: is a pointer to the help_args structure, described under "Notes" above. (Output)

suffix
: is the suffix which must appear in the entrynames of info segments to be processed by this invocation of help_. This suffix is also assumed when omitted from the (final or only) entryname of values given for help_args.path.value in the help_args structure (see "Notes" above). If a null string is given, then no suffix is required in info segment entrynames, and none is assumed in values of help_args.path.value. (Input)

progress
: is a special status code that indicates which stage of processing help_ was performing when an error occurs. (Output) See the help_ entry point.

code
: is a standard status code. (Output) See the help_ entry point.

This page intentionally left blank.

PPDinfo_seg
points to the PDinfo_seg structure, described under "Notes" below. This structure contains a sorted list of pointers to descriptors for the selected info segments. (Output)

NOTES

The PPDinfo_seg argument points to the PDinfo_seg structure that follows. This structure is declared in help_cis_args_.incl.pl1. All structure values are set by help_$check_info_segs.

```
dcl 1 PDinfo_seg              aligned based(PPDinfo_seg),
      2 version               fixed bin,
      2 N                     fixed bin(24),
      2 P  (0 refer  (PDinfo_seg.N))
                              ptr unal,
    PPDinfo_seg               ptr,
    VPDinfo_seg_1             fixed bin int static
                                   options(constant)  init(1);
```

Each pointer PDinfo_seg.P points to the following info segment descriptor structure, which is also declared in help_cis_args_.incl.pl1.

```
dcl 1 Dinfo_seg               aligned based,
      2 Scross_ref            bit(36) aligned,
      2 dir                   char(168) unal,
      2 ent                   char(32) unal,
      2 info_name             char(32) unal,
      2 ep                    char(32) var,
      2 uid                   bit(36),
      2 I                     fixed bin(21),
      2 L                     fixed bin,
      2 date                  fixed bin(71),
     (2 segment_type          bit(2),
      2 mode                  bit(3),
      2 pad1                  bit(31)) unal,
      2 code                  fixed bin(35);
```

STRUCTURE ELEMENTS

version
is the version number of the PDinfo_seg and Dinfo_seg structures (currently 1). The variable VPDinfo_seg_1 should be used when checking this version number.

N
is the number of info segments found.

P
is the array of pointers to the Dinfo_seg structures which describe the info segments found by the selection criteria.

PPDinfo_seg
is a pointer to the PDinfo_seg structure.

VPDinfo_seg_1
is a named constant which the caller of help_$check_info_segs should use when testing the value of PDinfo_seg.version.

Dinfo_seg
is the structure which describes each info segment found by the selection criteria.

Scross_ref
is an info segment cross-reference flag. If the flag equals "1"b, then several info segments were found having the same entryname but residing in different directories, and the info segment identified by this structure was not the first such duplicate.

dir
is the directory part of the pathname of the info segment.

ent
is the final entryname part of the pathname of the info segment.

info_name
is reserved for use by help_, and is always a null character string.

ep
is the subroutine entry point name given in the selection criteria for the info segment.

uid
is reserved for use by help_, and is always 0.

I
is reserved for use by help_, and is always 0.

L
is the length (in characters) of the info segment.

date
is the date_time_entry_modified of the info segment.

segment_type
is the type of storage system entry identified by Dinfo_seg.dir and Dinfo_seg.ent. It may have one of the following values:

"00"b	link
"01"b	segment

mode
is the user's access mode to the info segment. The three bits correspond to read, execute and write access mode. For example, rw access is expressed as "101"b.

pad1
    is reserved for future use.

code
    is a standard status code encountered while processing this info segment. It may have any of the following values:
    error_table_$noentry
        Dinfo_seg.dir and Dinfo_seg.ent identify a link whose target does not exist.
    error_table_$zero_length_seg
        the info segment is empty.
    error_table_$bad_syntax
        the info segment has a bit count which is not evenly divisible by 9. Therefore, the info segment does not contain a whole number of characters.

Entry: help_$init

This entry point obtains a pointer to the help_args structure (see "Notes" above). This structure is used to pass information from the caller to the help_ and help_$check_info_segs entry points. The structure is a based structure containing several arrays with adjustable extents. The help_$init entry point creates the structure in a temporary segment so that these arrays can be grown incrementally by the caller as information is added to the structure.

The help_ subroutine selects and prints info segments based upon the information given in the help_args structure. It also uses space in the temporary segment following the help_args structure for a work area. For this reason, space for help_args must be obtained by calling the help_$init entry point.

The help_$init entry point obtains the paths defined in a search list named by the caller. It stores these paths in the help_args structure for use by the help_ subroutine. Several other help_args elements are set, as described under "Notes" above.

USAGE

```
declare help_$init entry (char(*), char(*), char(*), fixed bin, ptr,
      fixed bin(35));

call help_$init (caller, search_list_name, search_list_ref_dir,
      required_version, Phelp_args, code);
```

ARGUMENTS

caller
is the name of the calling program, on whose behalf the temporary segment containing the help_args structure is obtained. (Input)

search_list_name
is the name of the search list to be used in searching for info segments. A null string may be given if no search list is to be used. (Input)

search_list_ref_dir
is the pathname of the directory to be used when expanding the referencing_dir search rule in the search list. If a null string is given, the referencing_dir search rule is omitted from the search list. (Input)

required_version
is the version number of the help_args structure which the caller is prepared to accept. This argument should be set to the value of the Vhelp_args_1 constant, described under "Notes" above. (Input)

Phelp_args
is a pointer to the help_args structure, described under "Notes" above. (Output)

code
is a standard status code reporting any failure in expanding the search list. (Output)

Entry: help_$term

This entry point releases the temporary segment in which the help_args structure (and the PDinfo_seg and Dinfo_seg structures of help_$check_info_segs) are created. This entry point should be called before calling help_$init again.

USAGE

```
declare help_$term entry (char(*), ptr, fixed bin(35));

call help_$term (caller, Phelp_args, code);
```

ARGUMENTS

The arguments are as described above for the help_ entry point.

Name: hphcs_$ips_wakeup

The hphcs_$ips_wakeup entry point sends a specified IPS signal to a specified process. That process is interrupted immediately unless it has the specified IPS signal masked off. See the description of the hcs_$get_ips_mask, hcs_$reset_ips_mask, and hcs$set_ips_mask entry points for a discussion of ips masking.

USAGE

```
declare hphcs_$ips_wakeup entry (bit(36) aligned, char(4) aligned);

call hphcs_$ips_wakeup (process_id, ips_name);
```

ARGUMENTS

process_id
    is the process identifier of the target process. (Input)

ips_name
    is the name of the ips signal to be sent to the target process. (Input)

NOTES

See the description of the set_ips_mask command for a list of valid ips signal names.

If the arguments are invalid (nonexistent process, undefined ips signal name) or are not properly aligned, the call is ignored; i.e., no signal is sent, and no error indication is given.

Name: hphcs_$read_partition

This entry point is used to read words of data from a specified disk partition on some mounted physical storage system disk.

USAGE

```
dcl hphcs_$read_partition entry (bit (36) aligned, char(*),
     fixed bin (35), pointer, fixed bin (19), fixed bin (35));

call hphcs_$read_partition (pvid, partition_name, offset, data_pointer,
     word_count, code);
```

ARGUMENTS

pvid
: is the physical volume id of the disk from which to read. (Input). The physical volume id is used instead of the volume name because this is a ring zero interface, and volume names are not accessible by ring zero; hence, all ring zero interfaces that reference physical volumes use the pvid. A pvname can be converted to a pvid by a call to mdc_$find_pvname, or the pvid can be obtained from find_partition_.

partition_name
: is the name of the disk partition to be read from. (Input). It must be four characters long or shorter.

offset
: is the offset in words, from the first word of the partition, of the first location to be read. (Input). It must be nonnegative and less than the number of words in the partition.

data_ptr
: is a pointer to the user-supplied buffer into which the data is to be read. (Input). It must be aligned on a word boundary.

word_count
: is the number of words to be read. (Input). The sum of offset and word_count must be less than or equal to the number of words in the partition. The sum of word_count and binary (rel (data_ptr)) must also be less than or equal to sys_info$max_seg_size, in order to avoid accessing past the end of the segment pointed to by data_ptr.

code
: is a nonstandard status code. (Output). It can be one of the following:
  0
  : indicates that the data was successfully read.

  error_table_$pvid_not_found
  : indicates that the specified physical volume is not presently mounted.

  error_table_$entry_not_found
  : indicates that the specified partition could not be found.

  error_table_$out_of_bounds
  : indicates that read request attempts to access data outside the partition; that is, the sum of offset and word_count is too large.

  an integer between 1 and 10
  : indicates that a physical disk error occurred while trying to read the label. Error messages for physical disk errors are declared in the include file fsdisk_errors.incl.pl1, in the array fsdisk_error_message.

Name: hphcs_$write_partition

This entry point is used to write words of data into a specified disk partition on some mounted physical storage system disk. No protection is provided against simultaneous use of this entry point by several processes writing to the same partition; thus, care must be exercised when using it.

USAGE

```
dlc hphcs_$write_partition entry (bit (36) aligned, char (*),
      fixed bin (35), pointer, fixed bin (18), fixed bin (35));

call hphcs_$write_partition (pvid, partition_name, offset, data_pointer,
      word_count, code);
```

ARGUMENTS

pvid
: is the physical volume id of the disk on which to write. (Input). The physical volume id is used instead of the volume name because this is a ring zero interface, and volume names are not accessible by ring zero; hence, all ring zero interfaces that reference physical volumes use the pvid. A pvname can be converted to a pvid by a call to mdc_$find_pvname, or the pvid can be obtained from find_partition_.

partition_name
: is the name of the disk partition to be written. (Input). It must be four characters long or shorter.

offset
: is the offset in words, from the first word of the partition, of the first location to be written. (Input). It must be nonnegative and less than the number of words in the partition.

data_ptr
: is a pointer to the data which is written into the partition from the user-supplied buffer. (Input). It must be aligned on a word boundary.

word_count
: is the number of words to be written. (Input). The sum of offset and word_count must be less than or equal to the number of words in the partition. The sum of word_count and binary (rel (data_ptr)) must also be less than or equal to sys_info$max_seg_size, in order to avoid accessing past the end of the segment pointed to by data_ptr.

code
is a nonstandard status code. (Output). It can be one of the following:
0
indicates that the data was successfully written.
error_table_$pvid_not_found
indicates that the specified physical volume is not presently mounted.
error_table_$entry_not_found
indicates that the specified partition could not be found.
error_table_$out_of_bounds
indicates that write request attempts to access data outside the partition; that is, the sum of offset and word_count is too large.
an integer between 1 and 10
indicates that a physical disk error occurred while trying to read the label. Error messages for physical disk errors are declared in the include file fsdisk_errors.incl.pl1, in the array fsdisk_error_message.

Name: initiate_file_

The initiate_file_ subroutine contains entry points for making a segment or archive component known with a null reference name.

The initiate_file_ entry point, given a directory name, entry name, and access mode, checks that the user's process has at least the desired access on the specified segment. If so, the segment is initiated with a null reference name. This entry point returns a pointer to the base of the segment and the bit count of the segment.

USAGE

```
declare initiate_file_ entry (char (*), char (*), bit (*), pointer,
      fixed binary (24), fixed binary (35));

call initiate_file_ (dirname, entryname, mode, seg_ptr, bit_count,
      code);
```

ARGUMENTS

dirname
is the pathname of the containing directory. (Input)

entryname
is the entryname of the segment. (Input)

mode
is the required access mode to the segment. (Input) The first three bits correspond to the modes read, execute, and write. The remaining bits, if any, must be zero. Named constants for the access modes are declared in access_mode_values.incl.pl1, and defined in the description of the hcs_$add_acl_entries entry point.

seg_ptr
if the segment was made known, this is a pointer to the base of the segment. (Output) Otherwise, this is null.

bit_count
is the bit count of the segment. (Output)

code
is a standard status code. (Output) It may have one of the following values:
error_table_$no_r_permission
read permission was required but not present.
error_table_$no_e_permission
execute permission was required but not present.
error_table_$no_w_permission
write permission was required but not present.

NOTES

The specified segment must exist, and the user must have nonnull access to it, as well as the required modes, in order to make it known.

If making the segment known encounters the error_table_$segknown status code, a zero status code is returned instead. This enables the user of this entry point to test either the returned pointer, or the status code, to indicate whether the segment was made known or not.

The hcs_$terminate_noname entry point or the terminate_file_ subroutine should be used to make the segment unknown.

Entry: initiate_file_$component

This entry point can make either a segment or an archive component known with a null reference name.

If the component name is null, this entry point is identical to initiate_file_.

Otherwise, the directory name and entry name arguments are assumed to specify an achive, and the component name specifies a component within that archive. If the user's process has at least the desired access on the archive segment, and the user desires no more than read access, then the archive is made known with a null reference name, and a segment number is assigned. This entry point returns a pointer to the base of the component and the bit count of the component.

USAGE

```
declare initiate_file_$component entry (char (*), char (*), char (*),
     bit (*), pointer, fixed binary (24), fixed binary (35));

call initiate_file_$component (dirname, entryname, component_name, mode,
     component_ptr, bit_count, code);
```

ARGUMENTS

dirname
: is the pathname of the containing directory. (Input)

entryname
: if component_name is null, this is the entryname of the segment. (Input) Otherwise, this is the entryname of an archive. The archive suffix must be supplied.

component_name
: is null, or is the name of a component in the archive. (Input)

mode
: is the required access mode to the segment. (Input) The first three bits correspond to the modes read, execute, and write. The remaining bits, if any, must be zero. Named constants for the access modes are declared in access_mode_values.incl.pl1.

component_ptr
: if the segment was made known, this is a pointer to the base of the segment or the base of the archive component. (Output) Otherwise, this is null.

bit_count
: is the bit count of the segment or archive component. (Output)

code
: is a standard status code. (Output) In addition to the above values, it can be:
: error_table_$archive_component_modification
: write permission was required on an archive component.

NOTES

The notes for the initiate_file_ entry point apply to this entry point also.

If a nonnull component name is specified, the following constraints apply to the use of this entry point:

1. The component may not be modified. Only read access is permitted.

2. The component is guaranteed to be contiguous and aligned on a word boundary. It is not guaranteed to have any other alignment.

3. No explicit dependence on the format of archives is permitted. This means that only the data starting at the pointer and extending as far as the bit count may be referenced. No data before or after the component may be referenced.

Entry: initiate_file_$create

This entry point initiates the specified segment with a null reference provided that the user's process has at least the desired access to the segment. If the segment does not exist, it will be created .

USAGE

```
declare initiate_file_$create entry (char (*), char (*), bit (*),
      pointer, bit (1) aligned, fixed binary (24), fixed binary (35));

call initiate_file_$create (dirname, entryname, mode, seg_ptr, created,
      bit_count, code);
```

ARGUMENTS

dirname
    is the pathname of the containing directory. (Input)

entryname
    is the entry name of the segment. (Input)

mode
    is the required access mode to the segment. The first three bits correspond to the modes read, execute, and write. The remaining bits, if any, must be zero. Named constants for the access modes are declared in access_mode_values.incl.pl1. (Input)

seg_ptr
    is set to a pointer to the base of the segment if successful; otherwise, it is set to null. (Output)

created
is set to "1"b if the segment did not exist and was created by this call; otherwise, it is set to "0"b. (Output)

bit_count
is set to the bit count of the segment. (Output)

code
is a standard status code. (Output) It can have one of the following values:
error_table_$no_m_permission
the segment did not exist and could not be created with the required access.
error_table_$no_r_permission
read permission was required but not present.
error_table_$no_e_permission
execute permission was required but not present.
error_table_$no_w_permission
write permission was required but not present.
error_table_$moderr
the user has null access to the segment.

NOTES

If making the segment known encounters the error_table_$segknown status code, a zero status code is returned instead. This enables the user of this entry point to test either the returned pointer or the status code to indicate whether the segment was made known or not.

The terminate_file_ subroutine should be used to make the segment unknown. If the segment was create by this call and the caller terminates abnormally or its cleanup handler is invoked, the caller can use the delete option of terminate_file_ to remove the segment that was created.

Name: interpret_resource_desc_

The interpret_resource_desc_ subroutine provides a facility for displaying the contents of an RCP resource description in a format similar to that used by the resource_status command.

USAGE

```
declare interpret_resource_desc_ entry (pointer, fixed bin, char(*),
      bit(36) aligned, bit(1) aligned, char(*) varying, fixed bin(35));

call interpret_resource_desc_ (resource_desc_ptr, nth, callername,
      string (rst_control), return_noprint, return_string, code);
```

ARGUMENTS

resource_desc_ptr
is a pointer to the structure containing the RCP resource description to be displayed. (See the resource_control_ subroutine.) (Input)

nth
specifies which element of the resource description is to be displayed (the index to the array resource_descriptions.item). If nth is zero, all elements will be displayed. (Input)

callername
is the name of the command invoking interpret_resource_desc_. It is used in printing any necessary error messages. (Input)

rst_control
is declared in the include file rst_control.incl.pl1. (See "Display Control" below.) (Input)

return_noprint
specifies, if "0"b, that information about the resource description is to be written to the user_output I/O switch. If "1"b, the information is returned in return_string, nth must not be zero, and the elements of the structure rst_control must be set so that exactly one item of information is requested. (Input)

return_string
contains, if return_noprint is "1"b, a printable representation of the information requested. Otherwise, its contents are undefined. (Output)

code
is a standard status code. (Output)

DISPLAY CONTROL

The rst_control structure (declared in the include file rst_control.incl.pl1) is defined as follows:

```
dcl 1 rst_control             aligned,
      2 default               bit (1) unaligned,
      2 name                  bit (1) unaligned,
      2 uid                   bit (1) unaligned,
      2 potential_attributes  bit (1) unaligned,
      2 attributes            bit (1) unaligned,
      2 desired_attributes    bit (1) unaligned,
      2 potential_aim_range   bit (1) unaligned,
      2 aim_range             bit (1) unaligned,
      2 owner                 bit (1) unaligned,
      2 acs_path              bit (1) unaligned,
      2 location              bit (1) unaligned,
      2 comment               bit (1) unaligned,
      2 charge_type           bit (1) unaligned,
      2 mode                  bit (1) unaligned,
      2 usage_lock            bit (1) unaligned,
      2 release_lock          bit (1) unaligned,
      2 awaiting_clear        bit (1) unaligned,
      2 user_alloc            bit (1) unaligned,
      2 given_flags           bit (1) unaligned,
      2 mbz                   bit (16) unaligned,
      2 any_given_item        bit (1) unaligned;
```

STRUCTURE ELEMENTS

default
: if "1"b, signifies that certain items of information are to be displayed only if they are not in the most common state. This bit should not be used by non-system commands.

name
: is "1"b if item.name is to be displayed.

uid
: is "1"b if item.uid is to be displayed.

potential_attributes
: is "1"b if item.potential_attributes is to be displayed.

attributes
: is "1"b if item.attributes is to be displayed.

desired_attributes
: is "1"b if item.desired_attributes is to be displayed.

potential_aim_range
is "1"b if item.potential_aim_range is to be displayed.

aim_range
is "1"b if item.aim_range is to be displayed.

owner
is "1"b if item.owner is to be displayed.

acs_path
is "1"b if item.acs_path is to be displayed.

location
is "1"b if item.location is to be displayed.

comment
is "1"b if item.comment is to be displayed.

charge_type
is "1"b if item.charge_type is to be displayed.

mode
is "1"b if item.mode is to be displayed.

usage_lock
is "1"b if item.usage_lock is to be displayed.

release_lock
is "1"b if item.release_lock is to be displayed.

awaiting_clear
is "1"b if item.awaiting_clear is to be displayed.

user_alloc
is "1"b if item.user_alloc is to be displayed.

given_flags
is "1"b if the state of all the flags in the structure item.given is to be displayed.

mbz
is unused and must be "0"b.

any_given_item
is "1"b to display any field in the item structure for which the corresponding bit in the item.given structure is "1"b.

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.

The ioa_ subroutine has several entry points in order to provide options concerning the formatting and disposition of the resulting string. Since any entry point can be called with various different arguments, each must be declared (in PL/I) 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 suppress this character, most types of ioa_ calls have a corresponding entry point with "nnl" (for no newline character); this entry point does the same editing.

Entries: ioa_, ioa_$nnl

These two entry points format the input data according to the control string and write the resulting string on the user_output I/O switch.

USAGE

```
call ioa_ (control_string, argl, ..., argN);
```

ARGUMENTS

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

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)

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(21), bit(1) aligned, bit(1) aligned);

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

ARGUMENTS

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

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

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

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

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

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

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

Entry: ioa_$general_rs_control_string

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

USAGE

```
declare ioa_$general_rs_control_string entry (ptr, char(*), fixed bin
      char(*), fixed bin(21), bit(1) aligned, bit(1) aligned);

call ioa_$general_rs_control_string (arglist_ptr, control_string,
      ff_argno, ret_string, len, pad_sw, nl_sw);
```

ARGUMENTS

arglist_ptr
: is a pointer to the argument list from which the format arguments are to be taken. (Input)

control_string
: is the control string. (Input)

ff_argno
: is the argument number of the first format argument in the argument list pointed to by arglist_ptr.

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

len
: specifies the number of significant characters in ret_string.

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

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

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 switch_name argument in the parameter list.

USAGE

```
call ioa_$ioa_stream (switch_name, control_string, argl, ..., argN);
```

ARGUMENTS

switch_name
    is the name of the I/O switch (char(*)) to which the resulting character string is to be written. (Input)

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

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_$ioa_switch, ioa_$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, argl, ..., argN);
```

ARGUMENTS

iocb_ptr
    is a pointer to the control block of the switch. (Input)

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

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)

This page intentionally left blank.

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, argl, ..., argN);
```

ARGUMENTS

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

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

len
: is the length of the returned string (fixed bin(21)). (Output)

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)

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. As a rule, 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	^Np	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
^[		start an if/then/else or case selection group

^]	limit the scope of a ^[
^; ^N;	used as a clause delimiter between ^[and ^]
^Nt ^N.Mt	insert enough space characters to reach column N

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 is interpreted as a numeric value, and is used to obtain the actual value. If this argument happens to be negative or not a number, 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 that appears after the argument list is exhausted, is inserted into the output string unchanged.

The numeric control codes (^d, ^i, ^f, ^e) take any PL/I numeric data type, including a character string that represents a numeric value, 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 ^Nd
: takes any numeric argument, including a character string that represents a numeric value, and edits it as a decimal integer. If N is not specified, the number is printed with no leading spaces or zeros. 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 ^Ni
: is the same as ^d, for compatibility with FORTRAN and PL/I formats.

^f ^Nf ^N.Df ^.Df

takes any numeric argument, including a character string that represents a numeric value, 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 zeros omitted. If D is specified, the fractional part of the number is rounded, or padded with extra zeros to achieve the desired result. If N is not specified, the number is printed with no leading spaces or zeros (except for a zero before the decimal point for numbers less than 1). If N is specified, the number is right-justified with leading spaces.

^e ^Ne

takes any numeric argument, including a character string that represents a numeric value, 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,.The standard format that is always used is:

```
N.ddddeN
```

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 zeros in the mantissa are omitted. The exponent sign is omitted if positive. Leading zeros in the exponent are also omitted.

^o ^No

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

^w ^Nw

takes any argument and edits one machine word in octal. Leading zeros 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 ^Na

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, to make it take up N columns. If the string (without any trailing spaces) is wider than N columns, the field width is ignored. If the string contains a newline or formfeed, no trailing spaces are ever added.

^b ^Nb ^N.Db ^.Db

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 ^Np

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 zeros suppressed. If the bit offset is zero, the (bb) portion of the pointer is omitted. If a field width is specified, the pointer is left justified in a field of width N.

^s ^Ns

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.

^(^N(

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 interation 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 ^N(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.

^[

starts an if/then/else or case selection group. A ^[takes a fixed binary or a bit-string argument, and must have a matching ^] to limit its scope. Using ^; as the delimiter, the text between the ^[and the ^] may be divided into clauses. If ^[is given a fixed-binary argument of N, the Nth clause between the ^[and the ^] is expanded; all other clauses are ignored. If there is no Nth clause (N too large or <1), all the text between the ^[and the ^] is ignored. If the argument to ^[is a nonzero bit string, the first clause is expanded (equivalent to a fixed-bin argument of 1 or "then"). If the argument to ^[is an all-zero bit string, the second clause is expanded (the "else" case). The ^[controls may be nested up to four deep. Null clauses are permitted. The arguments to ^[may also be the character strings "true" or "false", which correspond to "1"b and "0"b, or a character string containing ASCII digits, which are converted to a fixed binary argument.

^]

limits the scope of a ^[. See above.

^; ^N;

is used as a clause delimiter between ^[and ^]. See above. A ^N; is equivalent to N repetitions of ^;.

^Nt ^N.Mt

inserts enough space characters to reach column N. The column number is defined as follows: it is set to 1 when the ioa_ subroutine is entered, and whenever a newline character is placed in the output string; it is reduced by 1 whenever a backspace character is placed in the output string (but it is never reduced below 1); it is increased to the next tabstop column (11, 21,...) whenever a horizontal tab character is placed in the output string; and it is increased by 1 when any other character is placed in the output string. If M is specified, it is the minimum number of spaces that are to be placed in the output string. The default value of M is 1; a value of 0 is permitted. If the current column number is greater than N - M, then M spaces are placed in the output string, even though this causes the column number to become greater than N. However, if the next character string placed in the output string contains leading spaces, then ioa_ attempts to force that string into its proper column by deleting enough of the leading spaces to counteract the overflow in the previous field. Thus, in some cases the desired columnar alignment of data is preserved even when some of the data exceed the width of the columns reserved for them.

ioa_ resets the column count to 0 on each invocation, regardless of whether or not the entry point called was a $nnl. This causes ^t to be passing useless with any of the ioa_$Xnnl entry points.

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 # 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:   ######oct=6703#hex=DC3


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

Result:   1. #1.e0 #1.e-10 #1.00


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

Source:   sw="0"b;
          call ioa_ ("a=^d ^[b=^d^;^s^] c=^d",5,sw,7,9);

Result:   a=5 c=9

Source:   sw="1"b;
          call ioa_ ("a=^d ^[b=^d^;^s^] c=^d",5,sw,7,9);

Result:   a=5 b=7 c=9

Source:   dir=">"; ename="foo";
          call ioa_ ("Error in segment ^a^[>^]^a", dir,
               (dir ^=">"), ename);

Result:   Error in segment >foo
```

```
Source:    dir=">foo"; ename="bar";
           call ioa_ ("Error in segment ^a^[>^]^a", dir,
                (dir ^= ">"), ename);

Result:    Error in segment >foo>bar


Source:    option=2;  /* Assume following call is on one line*/
           call ioa_ ("Insurance option selected:
                ^[no fault^;bodily injury^;property damage^]", option);

Result:    Insurance option selected: bodily injury


Source:    name(1)="Jones"; name(2)="Morganstern"; name(3)="Shaughnessey";
           amt(1)=594.27; amt(2)=365.25; amt(3)=1.79;
           do i = 1 to 3; call ioa_ ("^a^12.1t^6.2f",name(i),amt(i)); end;

Result:    Jones      594.27
           Morganstern 365.25
           Shaughnessey 1.79
```

Name: iod_info_

The iod_info_ subroutine extracts information from the I/O daemon tables needed by those commands and subroutines that submit I/O daemon requests.

Entry: iod_info_$driver_access_name

This entry point returns the driver access name for a specified request type as defined in the I/O daemon tables. For example, the driver access name for the "printer" request type might be "IO.SysDaemon.*".

USAGE

```
declare iod_info_$driver_access_name entry (char(*), char(32),
     fixed bin(35));

call iod_info_$driver_access_name (request_type, access_name, code);
```

ARGUMENTS

request_type
is the name of a request type as defined in the I/O daemon tables. (Input)

access_name
is the driver access name for the above request type. (Output)

code
is a standard status code. If the specified request type is not found, the code error_table_$id_not_found is returned. (Output)

Entry: iod_info_$generic_type

This entry point returns the generic type of a specified request type as defined in the I/O daemon tables. For example, the generic type for the "unlined" request type might be "printer". Refer to the print_request_types command for information on generic types available for specific request types.

USAGE

```
declare iod_info_$generic_type entry (char(*), char(32), fixed bin(35));

call iod_info_$generic_type (request_type, generic_type, code);
```

ARGUMENTS

request_type
is the name of a request type as defined in the I/O daemon tables. (Input)

generic_type
is the name of the generic type of the above request type. (Output)

code
is a standard status code. If the specified request type is not found, the code error_table_$id_not_found is returned. (Output)

Entry: iod_info_$queue_data

This entry point examines the I/O daemon tables and returns the default queue and maximum number of queues for a given request type.

USAGE

```
declare iod_info_$queue_data entry (char(*), fixed bin, fixed bin,
      fixed bin(35);

call iod_info_$queue_data entry (request_type, default_q, max_queues,
      code);
```

ARGUMENTS

request_type
: is the name of the request type as defined in the I/O daemon tables. (Input)

default_q
: is the number of the default queue for the request type. (Output)

max_queues
: is the number of queues for the request type. (Output)

code
: is a standard status code. If the specified request type is not found, the code error_table_$id_not_found is returned. (Output)

Entry: iod_info_$rqt_list

This entry point examines the I/O daemon tables and returns a list of request types of a given generic type.

USAGE

```
declare iod_info_$rqt_list entry (char(32), (*) char(32), fixed bin,
      fixed bin(35));

call iod_info_$rqt_list entry (gen_type, q_list, n_queues, code);
```

ARGUMENTS

gen_type
: is the generic type of request types to be listed. If the string is blank, then all request types are listed. (Input)

q_list
is an array that is filled in with the request type names to be returned. If the size of this array is less than the number of names to be returned, the code error_table_$too_many_names will be returned, with the partial list. (Output)

n_queues
is the number of entries returned in the q_list array. (Output)

code
is a standard status code. If there are no matching entries, the code error_table_$no_entry is returned. (Output)

Name: iox_

The iox_ subroutine performs I/O operations and some related functions. I/O operations are described in the Programmer's Reference Manual.

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 switch to be in the attached state, the description of the function of the entry point applies only when the switch is attached. For other states, see the description of the particular I/O module. (The standard system I/O modules are described in Section 3 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_$not_open is returned. If the I/O switch is open but the operation is not allowed for that opening mode, the state of the switch is not changed, and the code that is returned is error_table_$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.

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. All entry points in iox_ are declared in the include file iox_dcls.incl.pl1.

Entry: iox_$attach_loud

This entry point is the same as iox_$attach_ptr except that a call to this entry turns on the com_err_ switch of the I/O module. This means that the attach routine of the I/O module calls com_err_ when an error is detected.

USAGE

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

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

ARGUMENTS

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

atd
    is the attach description. (Input)

ref_ptr
    is a pointer to the referencing procedure. It is used by the search rules to find an I/O module. (Input) (See hcs_$make_ptr for more information about ref_ptr.)

code
is an I/O system status code.

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.

Entry: iox_$attach_name

This entry point attaches an I/O switch which is designated by name and returns a pointer to its control block. The control block is created if it does not already exist. The form of an attach description is given in the Programmer's Reference Manual. If the switch is not in the detached state, its state is not changed and the code error_table_$not_detached is returned.

The I/O module is located using the current search rules.

USAGE

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

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

null()

ARGUMENTS

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

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

atd
    is the attach description. (Input)

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

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

Entry: iox_$attach_ptr

This entry point attaches an I/O switch in accordance with a specified attach description.

USAGE

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

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

ARGUMENTS

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

atd
    is the attach description. (Input)

ref_ptr
    is a pointer to the referencing procedure. It is used by the search rules to find an I/O module. (Input) (See hcs_$make_ptr for more information about ref_ptr.)

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

NOTES

The ref_ptr argument can be used to specify a particular I/O module if one by that name is not already initiated, for example:

```
call iox_$attach_ptr (iocb_ptr, "discard_",
     addr (my_discard_$my_discard_attach), code);
```

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);
```

ARGUMENTS

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

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

Entry: iox_$close_file

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.

This entry point differs from the iox_$close entry point due to the addition of the close description argument. For those I/O modules that support the close_file entry, the close description offers a means of providing file closing parameters such as a closing comment, where to position to upon closing, etc.

USAGE

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

call iox_$close_file (iocb_ptr, cld, code);
```

ARGUMENTS

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

cld
    is the close description. (Input)

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);
```

ARGUMENTS

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

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

info_ptr
    is null or points to data whose form depends on the I/O module and control order. (Input)

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);
```

ARGUMENTS

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

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

Entry: iox_$destroy_iocb

This entry point frees the storage used by the control block for an I/O switch. The switch must be in the detached state. Any existing pointers to the control block become invalid.

USAGE

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

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

ARGUMENTS

iocb_ptr
    points to the I/O control block to be freed. (Input)

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

Entry: iox_$detach

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.

This entry point differs from the iox_$detach_iocb entry point due to the addition of the detach description argument. For those I/O modules that support the detach entry, the detach description offers a means of providing detach time parameters such as a resource disposition comment to be sent to the system operator.

USAGE

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

call iox_$detach (iocb_ptr, dtd, code);
```

ARGUMENTS

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

dtd
: is the detach description. (Input)

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);
```

ARGUMENTS

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

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

Entries: iox_$err_no_operation, iox_$err_not_attached, iox_$err_not_closed, iox_$err_not_open

These entry points accept any number of arguments, the last of which is fixed bin (35). Each entry point sets the last argument to the respective code; error_table_$no_operation, error_table_$not_attached, error_table_$not_closed, or, error_table_$not_open. These entry points are assigned to entry variables in the I/O control block in order to return an error code when that entry variable is called. See the information on user-written I/O modules in the Programmer's Reference Manual for instructions on when to assign this entry point to such an entry value.

USAGE

```
declare iox_$err_no_operation entry options (variable);

call iox_$err_no_operation (argl, ..., argN, code);
```

ARGUMENTS

argI
: is a user-supplied argument. (Input)

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 (switch_name, iocb_ptr, code);
```

ARGUMENTS

switch_name
: is the name of the I/O switch. (Input)

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

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

NOTES

If the IOCB is for one of the four standard I/O switches, use one of the four standard external static pointers, iox_$user_io, iox_$user_input, iox_$user_output, or iox_$error_output. These pointers are declared in iox_dcls.incl.pl1.

Entry: iox_$find_iocb_n

This entry point may be used to find all existing I/O control blocks, whether attached or detached. It returns a pointer to the Nth control block in the calling ring, the numbering being arbitrary. If there are fewer than N control blocks, a null pointer and the code error_table_$no_iocb are returned. Creating or destroying control blocks during a sequence of calls to this entry point should be avoided, as it causes unpredictable changes to the numbering.

USAGE

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

call iox_$find_iocb_n (n, iocb_ptr, code);
```

ARGUMENTS

n
is the number of the I/O control block. (Input)

iocb_ptr
is a pointer to the control block. (Output)

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.

It is possible to write a program which takes certain actions if a call to iox_ takes longer than a certain amount of time. See the timed_io_$get_chars entry point.

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);
```

ARGUMENTS

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

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

n
is the number of bytes to be read where n>=0. (Input)

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

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, or if the read operation occurred with the input buffer length at zero, 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.

It is possible to write a program which takes certain actions if a call to iox_ takes longer than a certain amount of time. See the timed_io_$get_line entry point.

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);
```

ARGUMENTS

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

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

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

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

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

Entry: iox_$init_standard_iocbs

This entry point attaches the standard switches for a user process. These are currently user_input, user_output, and error_output, and they are attached with attach descriptions of:

syn_ user_i/o -inhibit close,put_chars
syn_ user_i/o -inhibit close,get_line,get_chars
syn_ user_i/o -inhibit close,get_line,get_chars

The variables iox_$user_input, iox_$user_output, and iox_$error_output are set to the iocb pointers for these switches.

USAGE

```
declare iox_$init_standard_iocbs entry ();

call iox_$init_standard_iocbs;
```

NOTES

Should the standard attachments change, this program will change to establish whatever they are. It should therefore be used in any direct process overseer that wishes to establish standard attachments.

Entry: iox_$look_iocb

This entry point returns a pointer to the control block for a specified I/O switch. If the control block does not exist, it is not created, and a null pointer and the code error_table_$no_iocb are returned.

USAGE

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

call iox_$look_iocb (switch_name, iocb_ptr, code);
```

ARGUMENTS

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

iocb_ptr
    is a pointer to the control block. (Output)

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);
```

ARGUMENTS

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

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

old_modes
is the string of modes in force when the call is made. (Output) If this argument has length zero, this information is not returned. This argument should be declared large enough to accommodate all the modes supported by the I/O module; 512 characters should be large enough for all system-supplied I/O modules.

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

Entry: iox_$move_attach

This entry point moves an attachment from one I/O switch, s1, to another I/O switch, s2. The s1 I/O 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 I/O switch to the s2 I/O switch. All pointer values and entry values are copied from the control block of the s1 I/O switch to the control block of the s2 I/O switch. Additional information on I/O control blocks is provided in the Programmer's Reference Manual. Attach and open data blocks maintained by the I/O module (if the s1 I/O switch is attached) are not affected. Finally, the s1 I/O switch is set to the detached state and iox_$propagate is called for both I/O switches.

USAGE

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

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

ARGUMENTS

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

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

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 the information on File Input/Output in the Programmer's Reference Manual for details.

USAGE

```
                  entry
declare iox_$open (ptr, fixed bin, bit (1) aligned, fixed bin(35));

call iox_$open (iocb_ptr, mode, unused, code);
```

ARGUMENTS

iocb_ptr
: is a pointer to the control block. (Input)

mode
: is the number assigned to the mode, e.g., 1 for stream_input, 2 for stream_output. (Input) Numbers associated with all allowed I/O modes are described in the Programmer's Reference Manual as part of the information on Input/Output Facilities. Named constant values for these modes are defined in iox_modes.incl.pl1.

unused
: must be "0"b. (Input)

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

Entry: iox_$open_file

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.

This entry point differs from the iox_$open entry point due to the addition of the open description argument. For those I/O modules that support the open_file entry, the open description offers a means of providing file opening parameters such as format, block size, record size, etc. The open description also allows the logical separation of attachment of resources, such as tape volumes, with the iox_$attach_name and iox_$attach_ptr entry points, and file specific operations for those I/O modules that deal with multi-file resources.

USAGE

```
declare iox_$open_file (ptr, fixed bin, char (*), bit (1) aligned,
      fixed bin(35));

call iox_$open_file (iocb_ptr, mode, opd, unused, code);
```

ARGUMENTS

iocb_ptr
: is a pointer to the control block. (Input)

mode
: is the number assigned to the mode as shown in the Programmers' Reference Manual under "Input and Output Facilities". (Output) For example, 1 for stream_input, 2 for stream_output. Named constant values for these modes are defined in iox_modes.incl.pl1.

opd
: is the open description. (Input)

unused
: must be "0"b. (Input)

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);
```

ARGUMENTS

iocb_ptr
: is a pointer to the control block. (Input)

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
2 positions to an absolute character or record
3 skip characters (stream input only)

n
: is the number of lines, records, or characters to be skipped (forward skip) or the negative of that number (backward skip), or the absolute position. (Input) It may be 0.

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

NOTES

Absolute positioning moves the next byte or record position to the location specified by n. Skipping characters moves the next byte position forward or backward over the specified, n, number of characters. If the file contains too few characters, the next byte position is at the end of file (forward skip) or beginning of file (backwards skip) and error_table_$end_of_info is returned.

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_$propagate

This entry point adjusts certain pointers and entry variables in an I/O control block as required when changing between the states detached, attached-closed, and attached-open. It also reflects modifications to a control block to other control blocks that are synonyms (immediate or chained) for it. This entry point must be called at certain points in the code of an I/O module, and it must not be called in any other circumstances.

USAGE

```
declare iox_$propagate entry (ptr);

call iox_$propagate (iocb_ptr);
```

ARGUMENTS

iocb_ptr
is a pointer to the control block. (Input)

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 3 for details.

It is possible to write a program which takes certain actions if a call to iox_ takes longer than a certain amount of time. See the timed_io_$put_chars entry point.

USAGE

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

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

ARGUMENTS

iocb_ptr
is a pointer to the control block. (Input)

buff_ptr
points to a byte-aligned buffer containing the bytes to be written. (Input)

n
is the number of bytes to be written where n>=0. (Input)

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);
```

ARGUMENTS

iocb_ptr
is a pointer to the control block. (Input)

key
is the next record's key. (Output)

rec_len
is the next record's length in bytes. (Output)

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);
```

ARGUMENTS

iocb_ptr
is a pointer to the control block. (Input)

rec_len
is the next record's length in bytes. (Output)

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);
```

ARGUMENTS

iocb_ptr
: is a pointer to the control block. (Input)

buff_ptr
: points to a byte-aligned buffer into which the record is to be read. (Input)

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

rec_len
: is the length of the record in bytes. (Output)

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);
```

ARGUMENTS

iocb_ptr
: is a pointer to the control block. (Input)

buff_ptr
: points to a byte-aligned buffer containing the new record. (Input)

rec_len
: is the length of the new record. (Input)

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);
```

ARGUMENTS

iocb_ptr
: is a pointer to the control block. (Input)

key
: contains the given key. (Input) All trailing blanks are removed from key to obtain the given key, and the result may be the null string.

rec_len
: is the length in bytes of the record with the given key. (Output)

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_update
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);
```

ARGUMENTS

iocb_ptr
: is a pointer to the control block. (Input)

buff_ptr
: points to a byte-aligned buffer containing the new record. (Input)

rec_len
: is the length of the new record in bytes. (Input)

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

Name: ipc_

The Multics system supports an interprocess communication facility. The basic purpose of the facility is to provide control communication (by means of stop and go signals) between processes.

The ipc_ subroutine is the user's interface to the Multics interprocess communication facility. Briefly, that facility works as follows: a process establishes event channels in the current protection ring and waits for an event on one or more channels.

Event channels can be thought of as numbered slots in the interprocess communication facility tables. Each channel is either an event-wait or event-call channel. An event-wait channel receives events that are merely marked as having occurred and awakens the process if it is blocked waiting for an event on that channel. On an event-call channel, the occurrence of an event causes a specified procedure to be called if (or when) the process is blocked waiting for an event on any channel. Naturally, the specific event channel must be made known to the process that expected to notice the event. For an event to be noticed by an explicitly cooperating process, the event channel identifier value is typically placed in a known location of a shared segment. For an event to be noticed by a system module, a subroutine call is typically made to the appropriate system module. A process can go blocked waiting for an event to occur or can explicitly check to see if it has occurred. If an event occurs before the target process goes blocked, then it is immediately awakened when it does go blocked.

The user can operate on an event channel only if his ring of execution is the same as his ring when the event channel was created (for a discussion of rings see the Programmer's Reference Manual.)

The hcs_$wakeup entry point is used to wake up a blocked process for a specified event.

Entry: ipc_$block

This entry point blocks the user's process until one or more of a specified list of events has occurred.

USAGE

```
declare ipc_$block entry (ptr, ptr, fixed bin(35));

call ipc_$block (event_wait_list_ptr, event_wait_info_ptr, code);
```

ARGUMENTS

event_wait_list_ptr
is a pointer to a structure that specifies the channels on which events are being awaited. (Input) This structure is declared in event_wait_list.incl.pl1.

```
dcl 1 event_wait_list     based aligned (event_wait_list_ptr),
      2 n_channels        fixed bin,
      2 pad               bit(36),
      2 channel_id     (event_wait_list_n_channels refer
                       (event_wait_list.n_channels)) fixed bin(71);
```

STRUCTURE ELEMENTS

n_channels
is the number of channels. This item must be allocated on an even-word boundary.

pad
must be zero.

channel_id
is an array of channel identifiers selecting the channels to wait on.

Frequently ipc_$block is called with only one channel in the wait list. In this case, the following structure may be used. It is declared in event_wait_channel.incl.pl1.

```
dcl 1 event_wait_channel   aligned,
      2 n_channels         fixed bin initial (1),
      2 pad                bit(36),
      2 channel_id         (1) fixed bin(71);
```

where:

event_wait_info_ptr
is a pointer to a structure into which the ipc_$block entry point can put information about the event that caused it to return (i.e., that awakened the process). This structure is declared in event_wait_info.incl.pl1. (Input)

```
dcl 1 event_wait_info       based aligned  (event_wait_info_ptr),
      2 channel_id          fixed bin(71),
      2 message             fixed bin(71),
      2 sender              bit(36),
      2 origin,
        3 dev_signal        bit(18) unaligned,
        3 ring              fixed bin(17) unaligned,
      2 channel_index       fixed bin;
```

STRUCTURE ELEMENTS

channel_id
is the identification of the event channel.

message
is an event message as specified to the hcs_$wakeup entry point.

sender
is the process identifier of the sending process.

dev_signal
indicates whether this event occurred as the result of an I/O interrupt.
"1"b yes
"0"b no

ring
is the sender's validation level.

channel_index
is the index of channel_id in the event_wait_list structure above.

code
is a standard status code. (Output)

Entry: ipc_$create_event_channel

This entry point creates an event channel of the specified type with the specified parameters. This entry replaces the ipc_$create_ev_chn and ipc_$decl_event_call_chn sequence to create normal call channels. This entry is the only way to create an async event call channel.

USAGE

```
dcl ipc_$create_event_channel entry (ptr, fixed bin (71),
      fixed bin (35));

call ipc_$create_event_channel (arg_ptr, channel_id, code);
```

ARGUMENTS

arg_ptr
is a pointer to ipc_create_arg_structure described below. (Input)

channel_id
is the identifier of the event channel created. (Output)

code
is a standard system status code. (Output)

NOTES

The following structure contains the arguments to the create_event_channel entry. All of the fields of the structure are to be filled in on input. The structure is declared in ipc_create_arg.incl.pl1:

```
dcl  1 ipc_create_arg_structure      aligned
                        based (ipc_create_arg_structure_ptr),
       2 version                     char (8) unaligned,
       2 channel_type                fixed bin,
       2 call_entry                  variable entry (ptr),
       2 call_data_ptr               ptr,
       2 call_priority               fixed bin (17);
```

STRUCTURE ELEMENTS

version
: is the version of the structure. It should be set to the constant: ipc_create_arg_structure_v1. (Input)

channel_type
: is the type of event channel that is to be created. Constant values for the type can be found in event_channel_types.incl.pl1 as follows: (Input)

1 FAST_EVENT_CHANNEL_TYPE
2 WAIT_EVENT_CHANNEL_TYPE
3 CALL_EVENT_CHANNEL_TYPE
4 ASYNC_CALL_EVENT_CHANNEL_TYPE

call_entry
: is the procedure entry point invoked when an event occurs on the specified channel. (Input)

call_data_ptr
: is a pointer to data to be passed to and interpreted by the procedure entry point. (Input)

call_priority
: is a number indicating the priority of an event call channel as compared to other event call channels declared by this process for this ring. If, upon interrogating all the appropriate event call channels, more than one is found to have received an event, the lowest-numbered priority is honored first, and so on. Synchronous and asynchronous call channels are the same with respect to this priority. (Input)

Entry: ipc_$cutoff

This entry point inhibits the reading of events on a specified event channel. Any pending events are not affected. More can be received, but do not cause the process to wake up.

USAGE

```
declare ipc_$cutoff entry (fixed bin(71), fixed bin(35));

call ipc_$cutoff (channel_id, code);
```

ARGUMENTS

channel_id
    is the identifier of the event channel. (Input)

code
    is a standard status code. (Output)

*

Entry: ipc_$decl_ev_wait_chn

This entry point changes an event-call channel into an event-wait channel.

USAGE

```
declare ipc_$decl_ev_wait_chn entry (fixed bin(71), fixed bin(35));

call ipc_$decl_ev_wait_chn (channel_id, code);
```

ARGUMENTS

channel_id
    is the identifier of the event channel. (Input)

code
    is a standard status code. (Output)

Entry: ipc_$delete_ev_chn

This entry point destroys an event channel previously created by the process.

USAGE

```
declare ipc_$delete_ev_chn entry (fixed bin(71), fixed bin(35));

call ipc_$delete_ev_chn (channel_id, code);
```

ARGUMENTS

channel_id
    is the identifier of the event channel. (Input)

code
    is a standard status code. (Output)

This page intentionally left blank.

Entry: ipc_$drain_chn

This entry point resets an event channel so that any pending events (i.e., events that have been received but not processed for that channel) are removed.

USAGE

```
declare ipc_$drain_chn entry (fixed bin(71), fixed bin(35));

call ipc_$drain_chn (channel_id, code);
```

ARGUMENTS

channel_id
    is the identifier of the event channel. (Input)

code
    is a standard status code. (Output)

Entry: ipc_$mask_ev_calls

This entry point causes the ipc_$block entry point to completely ignore event-calls occurring in the user's ring at the time of this call. This call causes a mask counter to be incremented. Event calls are masked if this counter is greater than zero.

USAGE

```
declare ipc_$mask_ev_calls entry (fixed bin(35));

call ipc_$mask_ev_calls (code);
```

ARGUMENTS

code
    is a standard status code. (Output)

Entry: ipc_$read_ev_chn

This entry point reads the information about an event on a specified channel if the event has occurred.

USAGE

```
declare ipc_$read_ev_chn entry (fixed bin(71), fixed bin, ptr,
      fixed bin(35));

call ipc_$read_ev_chn (channel_id, ev_occurred, event_wait_info_ptr,
      code);
```

ARGUMENTS

channel_id
: is the identifier of the event channel. (Input)

ev_occurred
: indicates whether an event occurred on the specified channel. (Output)
0 no event occurred
1 an event occurred

event_wait_info_ptr
: is a pointer to a structure into which the ipc_$block entry point can put information about the event that caused it to return (i.e., that awakened the process). This structure is declared in event_wait_info.incl.pl1. (Input) See the description in the ipc_$block entry point.

code
: is a standard status code. (Output)

Entry: ipc_$reconnect

This entry point enables the reading of events on a specified event channel for which reading had previously been inhibited (using the ipc_$cutoff entry point). All pending signals, whether received before or during the time reading was inhibited, are henceforth available for reading.

USAGE

```
declare ipc_$reconnect entry (fixed bin(71), fixed bin(35));

call ipc_$reconnect (channel_id, code);
```

ARGUMENTS

channel_id
: is the identifier of the event channel. (Input)

code
: is a standard status code. (Output)

Entry: ipc_$set_call_prior

This entry point causes event-call channels to be given priority over event-wait channels when several channels are being interrogated; e.g., upon return from being blocked and waiting on any of a list of channels. Only event channels in the current ring are affected. By default, event-call channels have priority.

USAGE

```
declare ipc_$set_call_prior entry (fixed bin(35));

call ipc_$set_call_prior (code);
```

ARGUMENTS

code
    is a standard status code. (Output)

Entry: ipc_$set_wait_prior

This entry point causes event-wait channels to be given priority over event-call channels when several channels are being interrogated; e.g., when a process returns from being blocked and is waiting on any of a list of channels. Only event channels in the current ring are affected.

USAGE

```
declare ipc_$set_wait_prior entry (fixed bin(35));

call ipc_$set_wait_prior (code);
```

ARGUMENTS

code
    is a standard status code. (Output)

Entry: ipc_$unmask_ev_calls

This entry point causes the event-call mask counter to be decremented. Event calls remain masked as long as the counter is greater than zero. To force event calls to become unmasked, call this entry point repeatedly, until a nonzero code is returned.

USAGE

```
declare ipc_$unmask_ev_calls entry (fixed bin(35));

call ipc_$unmask_ev_calls (code);
```

ARGUMENTS

code
: is a standard status code. A nonzero code is returned if event calls were not masked at the time of the call. (Output)

INVOKING AN EVENT-CALL PROCEDURE

When a process is awakened on an event-call channel, control is immediately passed to the procedure specified by the ipc_$decl_event_call_channel entry point. The procedure is called with one argument, a pointer to the following structure. This structure is declared in event_call_info.incl.pl1.

```
dcl 1 event_call_info       based aligned  (event_call_info_ptr),
      2 channel_id          fixed bin(71),
      2 message             fixed bin(71),
      2 sender              bit(36),
      2 origin,
        3 dev_signal        bit(18) unaligned,
        3 ring              fixed bin(17) unaligned,
      2 data_ptr            ptr;
```

STRUCTURE ELEMENTS

channel_id
: is the identifier of the event channel.

message
: is an event message as specified to the hcs_$wakeup entry point.

sender
: is the process identifier of the sending process.

dev_signal
: indicates whether the event occurred as the result of an I/O interrupt.
"1"b yes
"0"b no

ring
: is the sender's validation level.

data_ptr
: points to further data to be used by the called procedure.

Name: lex_string_

The lex_string_ subroutine provides a facility for parsing an ASCII character string into tokens (character strings delimited by break characters) and statements (groups of tokens). It supports the parsing of comments and quoted strings. It parses an entire character string during one invocation, creating a chain of descriptors for the tokens and statements in a temporary segment. The cost per token of lex_string_ is significantly lower than that of parse_file_ because the overhead of calling parse_file_ to obtain each token is eliminated. Therefore, the lex_string_ subroutine is recommended for translators that deal with moderate to large amounts of input.

The descriptors generated when the lex_string_ subroutine parses a character string can be used as input to translators generated by the reductions command, as well as in other applications. In addition, the information in the statement and token descriptors can be used in error messages printed by the lex_error_ subroutine.

Refer to the the reductions and lex_error_ descriptions for details on the use of these facilities.

Entry: lex_string_$init_lex_delims

This entry point constructs two character strings from the set of break characters and comment, quoting, and statement delimiters: one string contains the first character of every delimiter or break character defined by the language to be parsed; the second string contains a character of control information for each character in the first string. These two character strings form the break tables that the lex_string_ subroutine uses to parse an input string. It is intended that these two (delimiter and control) character strings be internal static variables of the program that calls lex_string_, and that they be initialized only once per process. They can then be used in successive calls to lex_string_$lex, as described below.

USAGE

```
declare lex_string_$init_lex_delims entry (char(*), char(*), char(*),
     char(*), char(*), bit(*), char(*) varying aligned,
     char(*) varying aligned, char(*) varying aligned,
     char(*) varying aligned);

call lex_string_$init_lex_delims (quote_open, quote_close, comment_open,
     comment_close, statement_delim, Sinit, break_chars,
     ignored_break_chars, lex_delims, lex_control_chars);
```

max_severity_no
is the severity number (fixed bin) of the highest severity error message that has been printed by the lex_error_ subroutine. (Input/Output). Before the lex_error_ is invoked by a translator, max_severity_no should be initialized to 0. Each time it is called, the lex_error_ subroutine compares this value with the severity_no of the current message and sets max_severity_no to the higher of these two numbers.

Pstmt
is a pointer to the statement descriptor generated by the lex_string_ subroutine for the statement that is to be printed after the error message. (Input). The line number and statement number given in this statement descriptor are included in the error message.

Ptoken
is a pointer to the token descriptor of the token that is in error. (Input). If Pstmt is null, then the number of the line that contains the token described by the descriptor is included in the error message. If both Pstmt and Ptoken are null, then no line number is included in the error message.

Scontrol
is a control bit string (bit(*)) that determines whether the message character string or the brief_message character string is used in the error message. (Input). The interpretation of the bits in this string is described in "Notes" below.

error_message_text
is an ioa_ control string (char(*) or char(*) varying) that contains the long form of the error message text. (Input)

brief_message_text
is an ioa_ control string (char(*) or char(*) varying) that contains the brief form of the error message text. (Input)

argN
are optional arguments that are substituted into the ioa_ message texts, in place of the ioa_ control characters. (Input)

NOTES

The error messages that are generated by the lex_error_ subroutine have the form shown below.

```
prefix error_number, SEVERITY severity_no IN STATEMENT k OF LINE l.
error_message_text
SOURCE:
statement_in_error
```

For example,

```
ERROR 7, SEVERITY 2 IN STATEMENT 2 OF LINE 2.
A bad track specification was given in a Volume statement.
9track has been assumed.
SOURCE:
Volume: 70082, 8track;
```

The severity_no associated with an error controls the prefix that is placed in the error message, as shown in the list below.

0 COMMENT
Comment. The error message is a comment, which does not indicate that an error has occurred, but merely provides information for the user.

1 WARNING
Warning only. The error message warns of a statement that may or may not be in error, but compilation continues without ill effect.

2 ERROR
Correctable error. The message diagnoses an error that the translator can correct, probably without ill effect. Compilation continues, but correct results cannot be guaranteed.

3 FATAL ERROR
An uncorrectable but recoverable error. The translator has detected an error that it cannot correct. Translation continues in an attempt to diagnose further errors, but no output is produced by the translation.

4 TRANSLATOR ERROR
An unrecoverable error. The translator cannot continue beyond this error. The translation is aborted after the error message is printed.

The phrase "IN STATEMENT k OF LINE l" appears in the error message only if Pstmt is a nonnull pointer. Pstmt is assumed to point to a statement descriptor generated by the lex_string_ subroutine. The values for k and l come from this descriptor. If the error occurred in the first statement of line l, then the phrase "STATEMENT k OF" is omitted from the error message.

If Pstmt is null, then "STATEMENT k OF" is omitted from the error message, and l is the line number on which the token described by Ptoken appears. If Ptoken is a null pointer, "IN STATEMENT k OF LINE l" is omitted altogether.

Currently, only the first two bits of the Scontrol bit string have meaning, as shown in the table below.

Scontrol	*Interpretation*
"00"b	The printed error contains the error_message_text the first time the error occurs, and the brief_message_text for subsequent occurrences of that error during a given translation.
"10"b	The printed error always contains the error_message_text.
"11"b	The printed error always contains the error_message_text.
"01"b	The printed error always contains the brief_message_text.

If Serror_printed is "1"b, then the lex_error_ subroutine assumes the text of the error message has already been printed in a previous message. It uses the long or brief error message text, according to the value of Scontrol.

If Pstmt points to a statement descriptor, then the lex_error_ subroutine sets the error_in_stmt switch in the statement descriptor. It also checks the value of the output_in_err_msg switch in the descriptor. If this switch is "0"b, the lex_error_ subroutine sets it to "1"b and prints the character string representation of the statement in the error message. If it is already "1"b, then the lex_error_ subroutine assumes that the statement has already appeared in another error message and omits the "SOURCE:" phrase from the error message.

If max_severity_no is less than severity_no, then the lex_error_ subroutine sets max_severity_no equal to severity_no.

Refer to the lex_string_ subroutine for a description of statement and token descriptors.

NOTES

A user should be familiar with interprocess communication in Multics and the pitfalls of writing programs that can run asynchronously within a process. For example, if a program does run asynchronously within a process and it does input or output with the tty_ I/O module, then the program should issue the start control order of tty_ before it returns. This is necessary because a wakeup from tty_ may be intercepted by the asynchronous program.

If a program establishes an event-call channel, and the procedure associated with the event-call channel uses static storage, then the event-call procedure should have the perprocess_static attribute. This is not necessary if the procedure is part of a limited subsystem in which run units cannot be used. See the description of the run command in the Commands manual for more information on run units and perprocess_static.

Name: lex_error_

The lex_error_ subroutine generates compiler-style error messages on the error_output I/O switch for translators generated by the reductions command and for other procedures that process tokens generated by the lex_string_ subroutine. See "Notes" below for a description of the error message format.

USAGE

```
declare lex_error_ entry options (variable);

call lex_error_ (error_number, Serror_printed, severity_no,
     max_severity_no, Pstmt, Ptoken, Scontrol, message, brief_message,
     arg1, ..., argN);
```

ARGUMENTS

error_number
: is the error number (fixed bin), as it should appear in the error message. (Input)

Serror_printed
: is a switch (bit(1) unaligned) that is "1"b if the text of the error message has been printed in a previous error and "0"b, otherwise. (Input/Output). If Serror_printed is "1"b, the text is omitted from the error message. Otherwise, text is included and the switch is set to "1"b to suppress this text in any subsequent occurrence of the same error.

severity_no
: is the severity number (fixed bin) of the error. (Input). It must have a value from 0 through 4. See "Notes" below for an interpretation of the severity_no value.

ARGUMENTS

quote_open
: is the character string delimiter that begins a quoted string. (Input). It can contain up to four characters. If it is a null character string, then quoted strings are not supported during the parsing of an input string.

quote_close
: is the character string delimiter that ends a quoted string. (Input). It can be the same character string as quote_open, and can contain up to four characters.

comment_open
: is the character string delimiter that begins a comment. (Input). It can contain up to four characters. If it is a null character string, then comments are not supported during the parsing of a character string.

comment_close
: is the character string delimiter that ends a comment. (Input). It can be the same character string as comment_open, and can contain up to four characters.

statement_delim
: is the character string delimiter that ends a statement. (Input). It can contain up to four characters. If it is a null character string, then statements are not delimited during the parsing of a character string.

Sinit
: is a bit string that controls the creation of statement descriptors, and the creation of token descriptors for quoting delimiters. (Input). The bit string consists of two bits in the order listed below.

  Ssuppress_quoting_delims
  : is "1"b if token descriptors for the quote opening and closing delimiters of a quoted string are to be suppressed. A token descriptor is still created for the quoted string itself, and the quoted_string switch in this descriptor is turned on. If Ssuppress_quoting_delims is "0"b, then token descriptors are returned for the quote opening and closing delimiters, as well as for the quoted string.

  Ssuppress_stmt_delims
  : is "1"b if the token descriptor for a statement delimiter is to be suppressed. The end_of_stmt switch in the descriptor of the token that precedes the statement delimiter is turned on, instead. If Ssuppress_stmt_delims is "0"b, then a token descriptor is returned for a statement delimiter, and the end_of_stmt switch in this descriptor is turned on.

break_chars
: is a character string containing all of the characters that can be used to delimit tokens. (Input). The string can include characters used also in the quoting, comment, or statement delimiters, and should include any ASCII control characters that are to be treated as delimiters.

ignored_break_chars
: is a character string containing all of the break_chars that can be used to delimit tokens but that are not tokens themselves. (Input). No token descriptors are created for these characters.

lex_delims
: is an output character string containing all of the delimiters that the lex_string_ subroutine uses to parse an input string. (Output). This string is constructed by the init_lex_delims entry from the preceding arguments. It must be long enough to contain all of the break_chars, plus the first character of the quote_open delimiter, the comment_open delimiter, and the statement_delim delimiter, plus 30 additional characters. This length must not exceed 128 characters, the number of characters in the ASCII character set.

lex_control_chars
: an output character string containing one character of control information for each character in lex_delims. (Output). This string is also constructed by init_lex_delims from the preceding arguments. It must be as long as lex_delims.

Entry: lex_string_$lex

This entry point parses an input string, according to the delimiters, break characters, and control information given as its arguments. The input string consists of two parts: the first part is a set of characters, which are to be ignored by the parser except for the counting of lines; the second part is the characters to be parsed. It is necessary to count lines in the part that is otherwise ignored so that accurate line numbers can be stored in the token and statement descriptors for the parsed section of the string.

USAGE

```
declare lex_string_$lex entry (ptr, fixed bin(21), fixed bin(21), ptr,
      bit(*), char(*), char(*), char(*), char(*), char(*),
      char(*) varying aligned, char(*) varying aligned,
      char(*) varying aligned, char(*) varying aligned, ptr, ptr,
      fixed bin(35));

call lex_string_$lex entry (Pinput, Linput, Lignored_input, Psegment,
      Slex, quote_open, quote_close, comment_open, comment_close,
      statement_delim, break_chars, ignored_break_chars, lex_delims,
      lex_control_chars, Pfirst_stmt_desc, Pfirst_token_desc, code);
```

ARGUMENTS

Pinput
is a pointer to the string to be parsed. (Input)

Linput
is the length (in characters) of the second part of the input string, the part that is actually to be parsed. (Input)

Lignored_input
is the length (in characters) of the first part of the input string, the part that is ignored except for line counting. (Input). This length can be 0 if none of the input characters are to be ignored.

Psegment
is a pointer to a temporary segment created by the translator_temp_ subroutine. (Input)

SLex
is a bit string that controls the creation of statement and comment descriptors, the handling of doubled quotes within a quoted string, and the interpretation of a comment_close delimiter that equals the statement_delim. (Input). The bit string consists of four bits in the order listed below.

Sstatement_desc
is "1"b if statement descriptors are to be created along with the token descriptors. If Sstatement_desc is "0"b, or if the statement delimiter is a null character string, then no statement descriptors are created.

Sscomment_desc
is "1"b if comment descriptors are to be created for any comments that appear in the input string. When Scomment_desc is "0"b, comment_open is a null character string, or statement descriptors are not being created, then no comment descriptors are created.

Sretain_doubled_quotes
is "1"b if doubled quote_close delimiters that appear within a quoted string are to be retained. If Sretain_doubled_quotes is "0"b, then a copy of each quoted string containing doubled quote_close delimiters is created in the temporary segment with all doubled quote_close delimiters changed to single quote_close delimiters.

Sequate_comment_close_stmt_delim
is "1"b if the comment_close and statement_delim character strings are the same, and if the closing of a comment is to be treated as the ending of the statement containing the comment. It could be used when parsing line-oriented languages that have only one statement per line and one comment per statement.

quote_open
is the character string delimiter that begins a quoted string. (Input). It can contain up to four characters. If it is a null character string, then quoted strings are not supported during the parsing of an input string.

quote_close
is the character string delimiter that ends a quoted string. (Input). It can be the same character string as quote_open, and can contain up to four characters.

comment_open
is the character string delimiter that begins a comment. (Input). It can contain up to four characters. If it is a null character string, then comments are not supported during the parsing of a character string.

comment_close
is the character string delimiter that ends a comment. (Input). It can be the same character string as comment_open, and can contain up to four characters.

statement_delim
is the character string delimiter that ends a statement. (Input). It can contain up to four characters. If it is a null character string, then statements are not delimited during the parsing of a character string.

break_chars
is a character string containing all of the characters that can be used to delimit tokens. (Input). The string can include characters used also in the quoting, comment, or statement delimiters, and should include any ASCII control characters that are to be treated as delimiters.

ignored_break_chars
is a character string containing all of the break_chars that can be used to delimit tokens but that are not tokens themselves. (Input). No token descriptors are created for these characters.

lex_delims
is the character string initialized by lex_string_$init_lex_delims. (Input)

lex_control_chars
is the character string initialized by lex_string_$init_lex_delims. (Input)

Pfirst_stmt_desc
is a pointer to the first in the chain of statement descriptors. (Output). This is a null pointer on return if no statement descriptors have been created.

Pfirst_token_desc
is a pointer to the first in the chain of token descriptors. (Output). This is a null pointer on return if no tokens were found in the input string.

code
: is one of the following status codes: (Output)

  0
  : the parsing was completed successfully.

  error_table_$zero_length_seg
  : no tokens were found in the input string.

  error_table_$no_stmt_delim
  : the input string did not end with a statement delimiter, when statement delimiters were used in the parsing.

  error_table_$unbalanced_quotes
  : the input string ended with a quoted string that was not terminated by a quote_close delimiter.

NOTES

Any character can be used in the quoting, comment, and statement delimiter character strings, including such characters as new line and the space character. A quoted string is defined in the PL/I sense, as a string of characters that is treated as a single token, even though some of the characters can be delimiters or break characters. The string must begin with a quote_open delimiter, and must end with a quote_close delimiter. Two consecutive quote_close delimiters can be used to represent a quote_close delimiter within the quoted string. The lex_string_$lex entry point provides the option of retaining any doubled quote_close delimiters in the quoted string token, or of copying the quoted string into the temporary segment, changing double quote_close to single quote_close delimiters, and treating the modified copy as the quoted string token. Switches in the token descriptor of a quoted string are turned on: to indicate that the token is a quoted string; to indicate whether any quote_close delimiters appear within the quoted string; and to indicate whether doubled quote_close delimiters have been retained in the token.

Statements are defined as groups of tokens that are terminated by a statement delimiter token. The lex_string_$lex subroutine can optionally return a token descriptor for the statement delimiter or it can suppress the token descriptor of the statement delimiter. It always turns on the end_of_stmt switch in the final token descriptor of each statement, even if the token descriptor of the statement delimiter has been suppressed. Besides, it can optionally return a statement descriptor that points to the descriptors for the first and last tokens of a statement, contains a pointer to and the length of the part of the input string containing the entire statement, and describes various other characteristics of the statement. These descriptors are described in the next section.

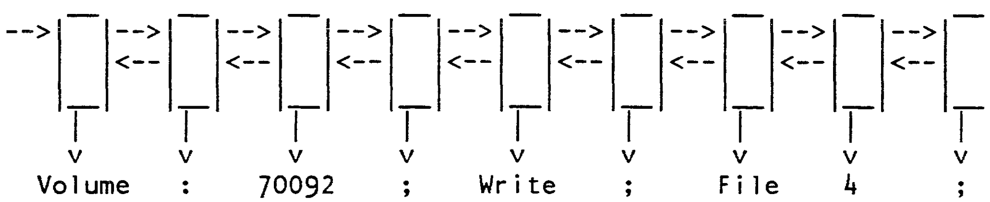
Comments are defined in the PL/I sense, as a string of characters that begin with a comment_open delimiter, and that end with a comment_close delimiter. Comments that appear in the input string act as breaks between tokens. The lex_string_$lex entry point can optionally create descriptors for each comment that appears in a statement. These descriptors are chained off of the statement descriptor for that statement. Switches are set in each comment descriptor of a given statement to indicate whether the comment appears before any of the tokens in that statement, and whether any tokens intervene between this comment and any previous comments in that statement.

The lex_string_ subroutine uses the translator_temp_ facility to allocate space for the descriptors in the temporary segment. Refer to the translator_temp_ subroutine description for details on the use of these temporary segments.

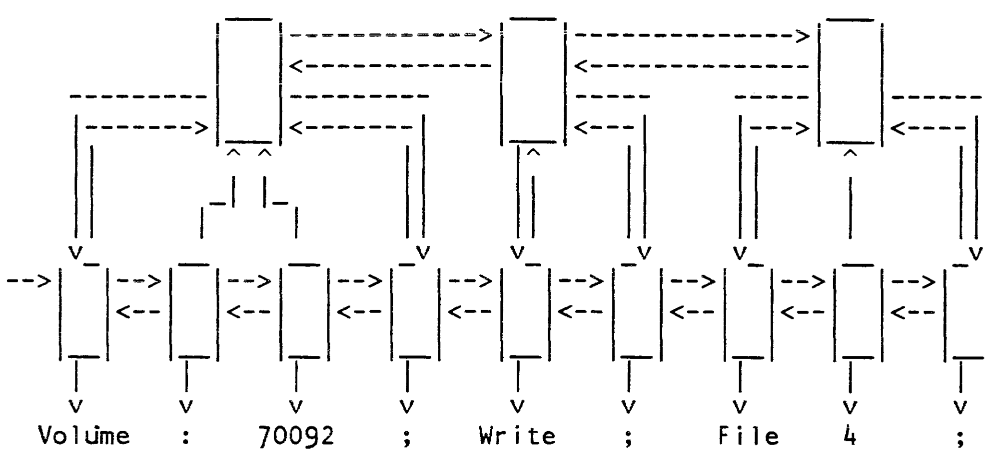
DESCRIPTORS

If the lex_string_$lex entry point were invoked using standard PL/I parsing conventions to parse the input shown below, then tokens and token descriptors created by the lex_string_ subroutine would have the following form:

```
Volume: 70092;
Write;
File 4;              /* Process 4th file on the tape.         */
/* END */
```



If statement descriptors were being created by the lex_string_ subroutine, then the output would have the following form:



Below is a declaration for the token descriptor structure:

```
dcl 1 token                   aligned based (Ptoken),
      2 group1                unaligned,
        3 version             fixed bin(17),
        3 size                fixed bin(17),
      2 Pnext                 ptr unal,
      2 Plast                 ptr unal,
      2 Pvalue                ptr unal,
      2 Lvalue                fixed bin(18),
      2 Pstmt                 ptr unal,
      2 Psemant               ptr unal,
      2 group2                unaligned,
        3 Itoken_in_stmt      fixed bin(17),
        3 line_no             fixed bin(17),
        3 Nvalue              fixed bin(35),
        3 S,
          4 end_of_stmt       bit(1),
          4 quoted_string     bit(1),
          4 quotes_in_string  bit(1),
          4 quotes_doubled    bit(1),
          4 pad2              bit(32);

dcl Ptoken                    ptr;
dcl token_value               char(token.Lvalue) based (token.Pvalue);
```

STRUCTURE ELEMENTS

version
: is the version number of the structure. The structure shown above is version 1.

size
: is the size of the structure, in words.

Pnext
: is a pointer to the descriptor for the next token in the input. If this is the last token descriptor, then the pointer is null.

Plast
: is a pointer to the descriptor for the previous token in the input. If this is the first token descriptor, then the pointer is null.

Pvalue
: is a pointer to the token character string.

Lvalue
: is the length of the token character string, in characters.

Pstmt
is a pointer to the statement descriptor for the statement that contains this token. If statement descriptors are not being created, then this pointer is null.

Psemant
is a pointer available for use by the caller of lex_string_. It is initialized as a null pointer. It might be used to chain a structure defining the semantic value of the token to the token's descriptor.

Itoken_in_stmt
is the position of the token with respect to the other tokens in the statement containing this token. If no statement delimiters are being used in the parsing, then this is the position of the token with respect to all other tokens in the input.

line_no
is the line_no on which this token appears.

Nvalue
is a number available for use by the caller of lex_string_. It is initialized to 0. It might be set to the numeric value of a token that is the character string representation of an integer.

end_of_stmt
is "1"b if this is the last token of a statement.

quoted_string
is "1"b if this token appeared in the input as a quoted string.

quotes_in_string
is "1"b is quote_close delimiters appear within this quoted string token.

quotes_doubled
is "1"b if quote_close delimiters that appear in a quoted string token are still represented by doubled quote_close delimiters, rather than having been converted to single quote_close delimiters.

pad2
is available for use by the caller of lex_string_. It is initialized to ""b by lex_string_.

Ptoken
is a pointer to a token descriptor.

token_value
is the character string representation of the token described by the token descriptor pointed to by Ptoken.

Statement descriptors are declared by the structure shown below.

```
dcl 1 stmt                      aligned based (Pstmt),
      2 group1                  unaligned,
        3 version               fixed bin(17),
        3 size                  fixed bin(17),
      2 Pnext                   ptr unal,
      2 Plast                   ptr unal,
      2 Pvalue                  ptr unal,
      2 Lvalue                  fixed bin(18),
      2 Pfirst_token            ptr unal,
      2 Plast_token             ptr unal,
      2 Pcomments               ptr unal,
      2 Puser                   ptr unal,
      2 group2                  unaligned,
        3 Ntokens               fixed bin(17),
        3 line_no               fixed bin(17),
        3 Istmt_in_line         fixed bin(17),
        3 semant_type           fixed bin(17),
        3 S,
          4 error_in_stmt       bit(1),
          4 output_in_err_msg   bit(1),
          4 pad                 bit(34);

dcl Pstmt                       ptr;
dcl stmt_value                  char(stmt.Lvalue) based (stmt.Pvalue);
```

STRUCTURE ELEMENTS

version
: is the version number of this structure. The structure declared above is version 1.

size
: is the size of this structure, in words.

Pnext
: is a pointer to the statement descriptor for the next statement. If this is the descriptor for the last statement, then this pointer is null.

Plast
: is a pointer to the descriptor for the previous statement. If this is the descriptor for the first statement, then the pointer is null.

Pvalue
: is a pointer to the character string representation of the statement as it appears in the input, excluding any leading newline characters or leading comments.

Lvalue
: is the length of the character string representation of the statement, in characters.

Pfirst_token
is a pointer to the descriptor of the first token in the statement.

Plast_token
is a pointer to the descriptor of the last token in the statement.

Pcomments
is a pointer to a chain of comment descriptors associated with this statement.

Puser
is a pointer available for use by the caller of lex_string_.

Ntokens
is a count of the tokens in this statement.

line_no
is the line number on which the first token of this statement appears in the input.

semant_type
is a number available for use by the caller of lex_string_. It is initialized to 0 by lex_string_. It might be used to classify the statement by its semantic type.

error_in_stmt
is "1"b if an error has occurred while processing this statement. This switch is never set by lex_string_, but it is set by lex_error_ when a statement descriptor is used to generate an error message.

output_in_err_msg
is "1"b if the statement has already been output in another error message. This switch is referenced and set by lex_error_ to prevent a statement from being printed in more than one error message.

pad
is available for use by the caller of lex_string_. It is initialized to ""b by lex_string_.

Pstmt
is a pointer to a statement descriptor.

stmt_value
is the character string value of the statement, as it appears in the input, excluding any leading newline characters or leading comments.

Comment descriptors are declared as follows.

```
dcl 1 comment                 aligned based (Pcomment),
      2 group1                unaligned,
        3 version             fixed bin(17),
        3 size                fixed bin(17),
      2 Pnext                 ptr unal,
      2 Plast                 ptr unal,
      2 Pvalue                ptr unal,
      2 Lvalue                fixed bin(18),
      2 group2                unaligned
        3 line_no             fixed bin(17),
        3 S,
          4 before_stmt       bit(1),
          4 contiguous        bit(1),
          4 pad               bit(16);

dcl Pcomment              ptr;
dcl comment_value         char(comment.Lvalue) based (comment.Pvalue);
```

STRUCTURE ELEMENTS

version
: is the version number of this structure. The structure declared above is version 1.

size
: is the size of this structure, in words.

Pnext
: is a pointer to the descriptor for the next comment associated with the statement containing this comment. If there are no more comments associated with that statement, then the pointer is null.

Plast
: is a pointer to the descriptor for the previous comment associated with the statement containing this comment. If this is the first comment associated with the statement, then the pointer is null.

Pvalue
: is a pointer to the character string value of the comment string, exactly as it appears in the input, excluding the comment_open and comment_close delimiters.

Lvalue
: is the length of the character string value of the comment, in characters.

line_no
: is the line number on which the comment begins.

before_stmt
: is "1"b if the comment appears in its statement before any tokens.

contiguous
is "1"b if no tokens appear between this comment and the previous comment associated with this statement.

pad
is available for use by lex_string_'s caller.

Pcomment
is a pointer to a comment descriptor structure.

comment_value
is the character string value of a comment.

The above declarations are available for inclusion in PL/I programs in lex_descriptors_.incl.pl1.

Name: list_dir_info_

The list_dir_info_ subroutine is used by the list_dir_info, rebuild_dir, and comp_dir_info commands to list the values in a single entry in a directory information segment created by the save_dir_info command.

USAGE

```
declare list_dir_info_ entry (ptr, fixed bin, char(1));
```

```
call list_dir_info_ (ptr, mode, prefix);
```

ARGUMENTS

ptr
points to an entry in the dir_info segment (created by invoking the save_dir_info command). (Input)

mode
is the verbosity desired. (Input). It can be 0, 1, or 2 (where 0 is the least verbose).

prefix
is a one-character prefix for every line printed. (Input)

NOTES

Output from the list_dir_info_ subroutine is written on the user_output I/O switch. It consists of a series of lines, each of the form:

```
item_name:  value
```

The prefix character is appended to the beginning of each line.

The list below gives the output items for each verbosity level, for segments, directories, and links. Verbosity level 1 returns information listed in 0 and 1; verbosity level two returns information listed in 0, 1, and 2.

For segments:

```
0.  names              1.  date branch modified       2.  ACL
    type                   records used                   data dumped
    date used              bit count                      current length
    date modified          bit count author               device ID
                           max length                     move device ID
                           safety switch                  copy switch
                           property list                  ring brackets
                                                          unique ID
                                                          author
```

For directories:

```
0.  names              1.  date branch modified       2.  ACL
    type                   bit count                      initial seg ACL
    date used              records used                   initial dir ACL
    date modified          quota
                           date dumped
                           current length
                           device ID
                           move device ID
                           copy switch
                           ring brackets
                           unique ID
                           author
                           bit count author
                           max length
                           safety switch
                           property list
```

For links:

```
0.  names              1.  date link modified         2.  date link dumped
    type
    target
```

Name: match_star_name_

The match_star_name_ subroutine implements the Multics star convention by comparing an entryname with a name which may contain stars or question marks, called a starname.

USAGE

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

call match_star_name_ (entryname, starname, code);
```

ARGUMENTS

entryname
: is the string to be compared with the starname. Trailing spaces in this string are ignored. (Input)

starname
: is the string with which the entryname is compared. Trailing spaces in this string are ignored. (Input)

code
: is one of the standard status codes listed below. (Output)

LIST OF STATUS CODES

0
: the entryname matches the starname.

error_table_$nomatch
: the entryname does not match the starname.

error_table_$badstar
: the starname does not have an acceptable format.

NOTES

See the description of the hcs_$star_ entrypoint in hcs_ to find how to list the directory entries that match a given starname. See check_star_name_ to find how to validate a starname. See starname.gi.info for the rules governing the formation and interpretation of starnames.

Name: mdc_

The mdc_ subroutine (actually a ring 1 gate) provides a series of entry points for manipulation of master directories.

Entry: mdc_$check_mounted

This entry point determines whether a logical volume is mounted and available for use.

USAGE

```
declare mdc_$check_mounted entry (char (*), fixed bin(35));

call mdc_$check_mounted (lv_name, code);
```

ARGUMENTS

lv_name
    is the name of the logical volume. (Input)

code
    is a standard system status code. (Output) Its possible values are:

    0
        the volume is mounted and ready for use. This does not mean it is attached to the calling process (see Notes, below.)

    error_table_$mount_not_ready
        the volume is not mounted.

NOTES

Use hcs_$lv_attached to determine if a logical volume is both mounted and attached to the calling process.

ACCESS REQUIRED

No special access is required.

Entry: mdc_$create_dir

This entry point is used to create a new master directory. Its arguments are roughly analogous to the hcs_$append_branchx entry point.

USAGE

```
declare mdc_$create_dir entry (char(*), char(*), char(*),
      bit(36) aligned, (3) fixed bin(3), char(*), fixed bin,
      fixed bin(35));

call mdc_$create_dir (dir_name, entryname, volume, mode, rings, user_id,
      quota, code);
```

ARGUMENTS

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

entryname
    is the entryname of the subdirectory. (Input)

volume
    is the name of the logical volume that is to contain segments created in the new directory. (Input)

mode
    is the user's access mode. (Input)

rings
    are the ring brackets of the directory. (Input) Only the first values are used.

user_id
    is an access control name. (Input)

quota
    is the quota to be placed on the new directory. (Input)

code
    is a standard status code. (Output)

Entry: mdc_$create_dirx

This entry point is an extension of the mdc_$create_dir entry point, which is similiar to hcs_$create_branch_ entry point.

USAGE

```
declare mdc_$create_dirx entry (char(*), char(*), char(*), ptr,
      fixed bin(35));

call mdc_$create_dirx (dir_name, entryname, volume, info_ptr, code);
```

ARGUMENTS

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

entryname
: is the entryname of the subdirectory. (Input)

volume
: is the name of the logical volume that is to contain segments created in the new directory. (Input)

info_ptr
: is a pointer to the create_branch_info structure as described under the hcs_$create_branch_ entry point. (Input)

Entry: mdc_$delete_dir

This entry point is used to delete a master directory.

USAGE

```
declare mdc_$delete_dir entry (char(*), char(*), fixed bin(35));

call mdc_$delete_dir (dir_name, entryname, code);
```

ARGUMENTS

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

entryname
: is the entryname of the subdirectory. (Input)

code
is a standard status code. (Output)

Entry: mdc_$find_lvid

This entry point returns the logical volume identifier for a given logical volume name.

USAGE

```
declare mdc_$find_lvid entry (char (*), bit(36), fixed bin (35));

call mdc_$find_lvid (lv_name, lvid, code);
```

ARGUMENTS

lvname
is the logical volume name whose identifier is to be returned. (Output)

lvid
is a logical volume identifier. (Input)

code
is a standard status code. (Output)

Entry: mdc_$get_lv_access

This entry point gets the calling process' effective access to manipulate a logical volume.

USAGE

```
declare mdc_$get_lv_access entry (char(*), fixed bin(3), fixed bin(5),
     bit (1) aligned, fixed bin (35));

call mdc_$get_lv_access (lv_name, ring, mode, public, code);
```

ARGUMENTS

lv_name
is the logical volume name. (Input)

ring
is the validation level for which access is to be calculated. (Input)

mode
is either REW_ACCESS_BIN for a volume executive, RW_ACCESS_BIN for a user with access to use the volume, or N_ACCESS_BIN for a user with no access to the volume. (Output) These values are declared in access_mode_values.incl.pl1.

public
 is "1"b if the volume is public.

code
 is a standard system status code. (Output)

ACCESS REQUIRED

No special access is required.

Entry: mdc_$pvname_info

This entry point gets various kinds of information about a specified storage-system physical volume.

USAGE

```
declare mdc_$pvname_info entry (char (*), bit (36) aligned, char (*),
      bit (36) aligned, fixed bin, fixed bin (35));

call mdc_$pvname_info (pvname, pvid, lvname, lvid, device_type, code);
```

ARGUMENTS

pvname
 is the name of the physical volume about which information is to be returned. (Input)

pvid
 is the physical volume id of the specified volume. It can be used as a parameter to ring-zero volume and partition interfaces. (Output)

lvname
 is the name of the logical volume to which the physical volume belongs. (Output)

lvid
 is the logical volume id of the logical volume to which the physical volume belongs. (Output)

device_type
 is a number indicating what type of device the specified physical volume is mounted on. The names and characteristics of these devices are listed in various arrays declared in the include file fs_dev_types.incl.pl1. (Output)

code
 is a standard system-status code. It is nonzero if the information about the volume cannot be obtained or if the volume does not exist. (Output)

Entry: mdc_$set_mdir_account

This entry point is used to set the quota account of a master directory.

USAGE

```
declare mdc_$set_mdir_account entry (char(*), char(*), char(*),
      fixed bin(35));

call mdc_$set_mdir_account (dir_name, entryname, account, code);
```

This page intentionally left blank.

ARGUMENTS

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

entryname
is the entryname of the subdirectory. (Input)

account
is the name of the new quota account. The directory quota is returned to the old account and redrawn from this new account.

code
is a standard system status code. (Output)

Entry: mdc_$set_mdir_owner

This entry point is used to set the owner name of a master directory.

USAGE

```
declare mdc_$set_mdir_owner entry (char (*), char (*), char (*),
     fixed bin(35));

call mdc_$set_mdir_owner (dir_name, entryname, owner, code);
```

ARGUMENTS

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

entryname
is the entryname of the subdirectory. (Input)

owner
is the new owner name of the master directory, in the form Person_id.Project_id.tag. (Input)

code
is a standard system status code. (Output)

Entry: mdc_$set_mdir_quota

This entry point is used to set the quota on a master directory.

USAGE

```
declare mdc_$set_mdir_quota entry (char(*), char(*), bit(1) aligned,
      fixed bin, fixed bin(35));

call mdc_$set_mdir_quota (dir_name, entryname, sw, quota, code);
```

ARGUMENTS

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

entryname
    is the entryname of the subdirectory. (Input)

sw
    is a switch indicating the kind of quota change. (Input)
    "0"b sets the directory quota to the quota parameter.
    "1"b algebraically adds the quota parameter to the current directory quota.

quota
    is the quota to be placed on the new directory. (Input)

code
    is a standard system status code. (Output)

Entry: mdc_$set_volume_quota

This entry point is used to set the volume quota for a quota account on a logical volume.

USAGE

```
declare mdc_$set_volume_quota entry (char(*), char(*), bit(1) aligned,
      fixed bin, fixed bin(35));

call mdc_$set_volume_quota (volume, account, sw, quota, code);
```

ARGUMENTS

volume
    is the name of the logical volume that is to contain segments created in the new directory. (Input)

account
is the name of the quota account in the form Person_id.Project_id.tag. The quota account name may contain stars. (Input)

sw
is a switch indicating the kind of quota change. (Input)
"0"b sets the directory quota to the quota parameter.
"1"b algebraically adds the quota parameter to the current directory quota.

quota
is the quota to be placed on the new directory. (Input)

code
is a standard system status code. (Output)

Name: menu_

The menu_ subroutine provides menu display and selection services. It can display a menu in a window and get a selection from the user. The entries work with menu objects. A menu object is a pointer to an internal description of a menu. The caller is expected to preserve the pointer, and to perform no operation on it other than comparison with the null pointer or with another menu object, except through the menu_ subroutine. Declarations for the entries and the associated structures are in the include file menu_dcls.incl.pl1 described below in "Data Structures".

Entry: menu_$create

This entry creates a menu object given its description. The menu data structure is allocated in a caller supplied area, and may be saved across processes by calling menu_$store. A pointer to the new menu is returned, also with the minimum size of a window to hold the menu.

USAGE

```
declare menu_$create entry ((*) char (*) varying, (*) char (*) varying,
      (*) char (*) varying, ptr, (*) char (1) unal, ptr, ptr, ptr,
      fixed bin (35));

call menu_$create (choices, headers, trailers, format_ptr, keys,
      area_ptr, needs_ptr, menu, code);
```

ARGUMENTS

choices
: is an array of the names of the options. (Input) If the maximum number of choices is exceeded, the code menu_et_$too_many_options is returned. The current maximum is 61.

headers
: is an array of headers. (Input) If the length of the first header is zero, then no headers are used. This allows the caller to specify no headers, without resorting to a zero-extent array, which is invalid PL/I.

trailers
: is an array of trailers. (Input) As for headers, a zero-length first trailer means that no trailers are displayed.

format_ptr
: points to a structure, menu_format, that controls formatting of the menu. (Input) This structure is described below in "Data Structures".

keys
: is an array specifying the keystroke for each option. (Input) The array must have at least as many elements as the array of option names. If not, the error code menu_et_$too_few_keys is returned. It may have more keys than choices. Each item of the array must be unique, or menu_et_$keys_not_unique is returned. If the valid keys (the keys for which there are choices) are either all upper case or all lower case, menu_$get_choice will treat upper and lower case letters identically.

area_ptr
: is a pointer to an area where the menu description is allocated. (Input) If the area is not large enough, the area condition is signalled. If this pointer is null, the system free area is used.

needs_ptr
: points to the menu_requirements structure giving requirements to display the menu. (Input) The structure is described below in "Data Structures". The caller supplies this structure and fills in the version number menu_requirements_version_1, the remaining members are output from this entry.

menu
: is a newly created menu object. (Output)

code
: is a standard system error code, or an error code from menu_et_. (Output)

Entry: menu_$delete

This entry deletes a menu object from a specified value segment.

USAGE

```
declare menu_$delete entry (char (*), char (*), char (*), fixed bin
     (35));

call menu_$delete (dirname, entryname, menu_name, code);
```

ARGUMENTS

dirname
: is the pathname of the containing directory. (Input)

entryname
: is the entryname of the segment. (Input) It must have the value suffix.

menu_name
: is the name that was assigned to the menu when it was stored (see the description of menu_$store). (Input)

code
: is a standard system error code. (Output)

Entry: menu_$describe

This entry fills in a caller-supplied data structure describing some of the aspects of a menu object. The caller can use this to ensure a window is sufficiently large to hold a menu.

USAGE

```
declare menu_$describe entry (ptr, ptr, fixed bin (35));

call menu_$describe (menu, needs_ptr, code);
```

ARGUMENTS

menu
: is the menu object to describe. (Input)

needs_ptr
: points to a structure declared like menu_requirements described in "Data Structures" below. (Input) The caller fills in the version to be menu_requirements_version_1, and the remaining members are filled in by this entry.

code
    is a standard system error code. (Output)

Entry: menu_$destroy

This entry is used to delete a menu object. The caller uses this to free storage of a menu, since the representation of a menu is not known outside the menu_ subroutine. This entry has no effect on screen contents or on stored menus.

USAGE

```
declare menu_$destroy entry (ptr, fixed bin (35));

call menu_$destroy (menu, code);
```

ARGUMENTS

menu
    is the menu object to destroy. (Input)

code
    is a standard system error code. (Output)

Entry: menu_$display

This entry displays a menu object on a supplied window.

USAGE

```
declare menu_$display entry (ptr, ptr, fixed bin (35));

call menu_$display (window, menu, code);
```

ARGUMENTS

window
    is a pointer to an IOCB for an I/O switch attached through window_io_. (Input) This window must be large enough to hold the menu. A menu window should be used ONLY for menu I/O, if redisplay optimizations are desired.

menu
    is the menu object to be displayed. (Input)

code
    is a standard system error code. (Output)

Entry: menu_$get_choice

This entry returns a choice from a menu. The menu is assumed to be already displayed in the window.

USAGE

```
declare menu_$get_choice entry (ptr, ptr, ptr, bit (1) aligned,
      fixed bin, fixed bin (35));

call menu_$get_choice (window, menu, function_key_info, fkey, selection,
      code);
```

ARGUMENTS

window
: is a pointer to the IOCB for the I/O switch used to display the menu. (Input)

menu
: is the menu object on display in the window. (Input)

function_key_info
: is a pointer to a data structure describing the function keys available on the terminal. (Input) This data structure is obtained by the caller from the ttt_info_$function_key_data subroutine. If this pointer is null, no function keys are used.

fkey
: returns a value of "1"b if a function key was hit instead of a menu selection. (Output)

selection
: gives the option number or function key number chosen by the user. For an option, it is a number between 1 and the highest defined option, inclusive. For a function key, it is the number of the function key.

code
: is a standard system error code. (Output)

NOTES

If a terminal has no function keys, the caller can define input escape sequences for function keys. These may be chosen to have mnemonic value to the end user. For example, if Function Key 1 is used to print a help file, the input sequence ESC h could replace it. In some applications, this will be easier for the end user to remember than an unlabelled function key. The caller can define these keys by allocating and filling in the same function key structure normally returned by the ttt_info_ subroutine.

If a key is hit that is not one of the option keys and is not a function key, then the terminal bell is rung.

Entry: menu_$list

This entry lists the menu objects stored in a specified value segment.

USAGE

```
declare menu_$list entry (char (*), char (*), char (*), ptr, fixed bin,
      ptr, fixed bin (35));

call menu_$list (dirname, entryname, menu_starname, area_ptr,
      menu_list_info_version, menu_list_info_ptr, code);
```

ARGUMENTS

dirname
: is the pathname of the containing directory. (Input)

entryname
: is the entryname of the segment. (Input) It must have the value suffix.

menu_starname
: is matched against the names of the menus stored in the segment. (Input) Only names that match menu_starname are returned. (see the description of menu_$store).

area_ptr
: is a pointer to an area in which to allocate the structure containing the menu names. (Input) If it is null, the system free area is used.

menu_list_info_version
: is the version of the menu_list_info structure that the caller expects. (Input) It must be a supported menu_list_info structure version. The only supported version is menu_list_info_version_1.

menu_list_info_ptr
: is a pointer to the menu_list_info structure, described below under "Data Structures". (Output)

code
: is a standard system error code. (Output)

Entry: menu_$retrieve

This entry retrieves a menu from a specified segment. The segment must be a value segment. The menu data structure is allocated in a caller-supplied area. The menu information is copied from the menu object stored in the segment into the newly allocated structure.

USAGE

```
declare menu_$retrieve entry (char (*), char (*), char (*), ptr, ptr,
      fixed bin (35));

call menu_$retrieve (dirname, entryname, menu_name, area_ptr, menu_ptr,
      code);
```

ARGUMENTS

dirname
: is the pathname of the containing directory. (Input)

entryname
: is the entryname of the segment. (Input) It must have the value suffix.

menu_name
: is the name that was assigned to the menu when it was stored (see the description of menu_$store). (Input)

area_ptr
: is a pointer to an area where the menu object is allocated. (Input) If this argument is null, the system free area is used. If the area is not large enough, the area condition is signalled.

menu_ptr
: is a pointer to the menu object that is retrieved from the segment. (Output)

code
: is a standard system error code. (Output)

Entry: menu_$store

This entry stores a menu object in a specified segment. The specified segment must be a value segment.

USAGE

```
declare menu_$store entry (char (*), char (*), char (*), bit(1) aligned,
      ptr, fixed bin (35));

call menu_$store (dirname, entryname, menu_name, create_sw, menu_ptr,
      code);
```

ARGUMENTS

dirname
: is the pathname of the containing directory. (Input)

entryname
: is the entryname of the segment. (Input) It must have the value suffix.

menu_name
: is a name to be assigned to the menu. (Input)

create_sw
: determines whether or not the segment is created if it does not already exist. If the segment does not exist, a value of "1"b will cause it to be created. (Input)

menu_ptr
: is a pointer to the menu object that is to be stored in the segment. (Input)

code
: is a standard system error code. (Output)

DATA STRUCTURES

A menu is described by the "menu_format" structure. It is declared in menu_dcls.incl.pl1.

```
dcl 1 menu_format           aligned based (menu_format_ptr),
      2 version             fixed bin,
      2 constraints,
        3 max_width         fixed bin,
        3 max_height        fixed bin,
      2 n_columns           fixed bin,
      2 flags,
        3 center_headers    bit (1) unal,
        3 center_trailers   bit (1) unal,
        3 pad               bit (34) unal,
      2 pad_char            char (1);
```

STRUCTURE ELEMENTS

menu_format
: specifies the format for menu display. (Input) It gives limits for number of lines and characters per line, specifies the number of columns (of options), and controls centering of headers and trailers.

version
: must be menu_format_version_1. (Input)

max_width
: is the width of the window the menu will be displayed on. (Input) This value is used for centering headers and aligning columns.

max_height
: is the maximum height of the window, in lines. (Input)

n_columns
: is the number of columns to use in displaying options. (Input)

center_headers
: if set, header lines will be centered using the window width supplied above. (Input) If not set, they are flush with the left edge of the window.

center_trailers
: same as center_headers, but for trailers. (Input)

pad
: must be "0"b. (Input)

pad_char
: is the character used for centering headers and/or trailers. (Input)

THE MENU_LIST_INFO STRUCTURE

This entry returns information in the menu_list_info structure, found in the include file menu_list_info.incl.pl1, shown below:

```
dcl 1 menu_list_info          aligned based (menu_list_info_ptr),
      2 version               fixed bin,
      2 n_names               fixed bin,
      2 name_string_length    fixed bin (21),
      2 names                 (menu_list_n_names refer
                                (menu_list_info.n_names)) aligned,
        3 position            fixed bin (21),
        3 length              fixed bin (21),
      2 name_string character (menu_list_name_string_length
                              refer (menu_list_info.name_string_length))
                              unaligned;
```

STRUCTURE ELEMENTS

version
is the version of this structure, menu_list_info_version_1. (Output)

n_names
is the number of menu object names that matched the supplied starname. (Output)

name_string_length
is the total length of all the names that matched the supplied starname, concatenated together. (Output)

names
is an array of information with one entry for each name that matched the specified starname. (Output)

position
is the position in the string menu_list_info.name_string of this menu name. (Output)

length
is the length of this menu name in the string menu_list_info.name_string. (Output)

name_string
contains all the returned names, concatenated together. (Output) The PL/I "defined" attribute can be used to advantage to refer to individual names. For example, we wish to print the menu name indexed by name_index.

```
begin;
 declare this_name character (menu_list_info.length (name_index))
                   defined (menu_list_info.name_string)
                   position (menu_list_info.position (name_index));

     call ioa_ ("The ^d'th menu name is: ^a", name_index, this_name);
 end;
```

THE MENU_REQUIREMENTS STRUCTURE

The requirements for a menu are specified by the menu_requirements structure. It is declared in menu_dcls.incl.pl1.

```
dcl 1 menu_requirements    aligned based (menu_requirements_ptr),
      2 version            fixed bin,
      2 lines_needed       fixed bin,
      2 width_needed       fixed bin,
      2 n_options          fixed bin;
```

STRUCTURE ELEMENTS

version
is set by the caller, and must be menu_requirements_version_1. (Input)

lines_needed
is the number of lines required. (Output) If the window does not have this number of lines, menu display will fail.

width_needed
is the number of columns needed. (Output)

n_options
is the number of options defined. (Output)

The include file, menu_dcls.incl.pl1, also provides an array of key characters that may be used in the menu to select options. This array can be used by the caller as input to the menu_$create entry. Its name is MENU_OPTION_KEYS.

Name: metering_util_

The metering_util_ subroutine contains several entry points useful for collecting hardcore metering data. In general, hardcore metering data elements can be categorized as samples, cumulative times, or cumulative counts (the latter two being cumulative since system initialization). Samples are snapshots of variables that describe the state of some system object (e.g, number of processes eligible at this instant). An example of a cumulative count is the total number of read I/Os issued to a particular disk device since system initialization, while an example of a cumulative time is the total busy time of a particular disk device while processing read I/Os. It is easy to compute average I/O time for a read to a particular device from these last two items. If a given set of metering data is sampled periodically, then more interesting, time-varying data can be computed. For example, the average I/O time for a read during a certain time interval can be computed. That interval is the time between any two samples of the data; subtracting the earlier cumulative count of I/Os from the later count yields the incremental count (i.e., the count of I/Os during the interval).

Multics metering commands are designed for interactive use, with the interval boundaries defined by the user in real time. Typically, metering commands support the following control arguments:

-report
prints a report of activity since the last interval boundary (or since system initialization, if no boundary has been defined).

-reset
: defines an interval boundary for this metering program; all further invocations of this command display data reflecting activity since this boundary.

-report_reset
: reports and then resets.

Under this scheme, each display of data, establishment of an interval boundary, etc., is done in a separate invocation of the same metering program. This allows the user to establish an interval boundary, exercise the system in some fashion, and then print data describing the system performance while it was being exercised. Additionally, a user can run any number of metering programs, each with independent interval boundaries. These considerations imply that metering data collection (which is sampling of hardcore data bases) should be global to the process (in order to exist through multiple invocations of the same metering command) and be distinguished among different metering programs.

The metering_util_ subroutine satisfies the above requirements in the following manner. On the first invocation of a metering program, the program calls metering_util_$define_region to define the hardcore data of interest; the collection of such data can be an arbitrary number of contiguous regions in an arbitrary number of hardcore data bases. On this first invocation, metering_util_ allocates sufficient static storage to maintain two copies of each hardcore region. This storage is allocated in a system free area in the process directory. A unique identifier in the form of a nonzero integer is assigned and returned to the invoker. This unique identifier is used in all further communications with metering_util_ by that metering program to identify the set of hardcore regions defined in this first call. Current buffers are filled by calls to metering_util_$fill_buffers, at which time the hardcore regions specified in the call to metering_util_$define_region are copied into the corresponding current buffers. Previous buffers are initially set to binary zeros. On calls to metering_util_$reset, the current buffers are copied into the previous buffers. On calls to metering_util_$fill_buffers, pointers to the current and previous buffers for each hardcore region are returned.

To use this subroutine, sufficient access to copy all hardcore regions specified is required. Access to the phcs_ gate is sufficient. If all hardcore regions specified are defined in >sl1>ring_zero_meters_limits.ascii, then access to metering_gate_ is sufficient.

Entry: metering_util_$define_regions

This entry is used to define a set of sections of hardcore data bases which are of interest to the invoker. Upon return, sufficient static storage has been allocated to contain two copies of each hardcore region specified in the call; this storage has also been initialized to zero.

USAGE

```
declare code fixed bin (35); declare unique_index fixed bin; declare
     metering_util_$define_regions entry options (variable);

call metering_util_$define_regions (unique_index, code,
          hardcore_seg_1, begin_region_1, end_region_1, ...  , ...  ,
          ...  hardcore_seg_n, begin_region_n, end_region_n);
```

ARGUMENTS

unique_index
: is a unique identifier for the set of regions. This identifier must be used in calls to other metering_util_ entry points. (Output)

code
: is a standard status code. The code error_table_$wrong_no_of_args is returned if the number of arguments remaining is not modulo 3. (Output)

The remaining arguments must be in groups of three, as shown in the calling sequence above. Each such group defines a hardcore region by specifying a hardcore segment and a contiguous region within the segment. The arguments in each group, in order, are the following:

hardcore_seg_i
: identifies the ring 0 data base. It may be of the form char (*), in which case it is assumed to be the name of a ring 0 segment; or of the form ptr aligned, in which case it is assumed to be a pointer to the segment. In the latter case, only the segment number is significant. (Input)

begin_region_i
: identifies the beginning of the region in the ring 0 data base. It may be of the form char (*), in which case it is assumed to be the name of an external symbol in hardcore_seg_i; or of the form fixed bin, in which case it is assumed to be a word offset into hardcore_seg_i. (Input)

end_region_i
: identifies the end of the region in the ring 0 data base. It may be of the form char (*), in which case it is assumed to be the name of an external symbol in hardcore_seg_i that refers to the next word beyond the end of the region; or of the form fixed bin, in which case it is assumed to be the length of the region in words. (Input)

NOTES

Any errors encountered by this entry point are reported to the user by means of the sub_err_ subroutine. Examples of such errors are invalid segment names or symbol names, or invalid region specification (e.g., nonpositive length). Errors of this sort are always programming errors, and are not external circumstances from which the calling program can be expected to recover.

Entry: metering_util_$fill_buffers

This entry is used to copy the current contents of all regions defined for the specified unique identifier into the current buffers for that unique identifier, and to return pointers to the current and previous buffers for these regions.

USAGE

```
declare metering_util_$fill_buffers entry (fixed bin, fixed bin(71),
      char(10), (*) ptr, (*) ptr, fixed bin(35));

call metering_util_$fill_buffers (unique_index, meter_time,
      formatted_time, current_ptrs, previous_ptrs, code);
```

ARGUMENTS

unique_index
: is the unique identifier returned by metering_util_$define_regions (above). (Input)

meter_time
: is the total metering time in microseconds. The total metering time is defined as the time between the last call to metering_util_$reset and this call. If metering_util_$reset has not been called, the total metering time is defined as the time between the last system bootload and this call. (Output)

formatted_time
: is the total metering time in a format suitable for printing. This format is

```
HHHH:MM:SS
```

where this represents the decomposition of total metering time into hours (HH), minutes (MM), and seconds (SS). (Output)

current_ptrs
: is an array of pointers that, on return, contain pointers to the current buffers for the hardcore regions defined in the call to metering_util_$define_regions. The number of elements in this array must be equal to the number of hardcore regions defined in the call to metering_util$define regions. The elements of this array are pointers to the current buffers for the corresponding hardcore regions. Specifically, current_ptrs (i) contains on return a pointer to the current buffer for hardcore_seg_i (defined above). (Output)

previous_ptrs
: is an array of pointers which, on return, contain pointers to the previous buffers for the hardcore regions defined in the call to metering_util_$define_regions. The number of elements in this array must be equal to the number of hardcore regions defined in the call to metering_util_$define regions. The elements of this array are pointers to the previous buffers for the corresponding hardcore regions. Specifically, previous_ptrs (i) contains on return a pointer to the previous buffer for hardcore_seg_i (defined above). (Output)

code
    is a standard status code. If either the array current_ptrs or the array previous_ptrs does not have the proper number of elements (see above), the code error_table_$invalid_array_size is returned, and no action is performed. (Output)

Entry: metering_util_$reset

This entry point is called to reset the metering interval to the time of this call. This is done by copying the current buffers into the previous buffers for all regions defined for the unique index specified.

USAGE

```
declare metering_util_$reset entry (fixed bin, fixed bin(35));

call metering_util_$reset (unique_index, code);
```

ARGUMENTS

unique_index
    is as above. (Input)

code
    is as above. (Output)

Name: meter_gate_

The meter_gate_ subroutine is an entry point (used by the meter_gate metering command) that returns data about specific gate entries to the caller.

USAGE

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

call meter_gate_ (gate_name, array_ptr, code);
```

ARGUMENTS

gate_name
    is the name of the gate whose entries are to be metered. (Input)

array_ptr
    is a pointer to an array described in "Notes" below. (Input)

code
    is a standard status code. (Output)

STRUCTURE

The second argument to meter_gate_ is a pointer to an array of entry names to be metered. This array has the following format:

```
dcl 1 arg_array            aligned based (array_ptr),
      2 num_ents           fixed bin,
      2 info               (0 refer (arg_array.num_ents)),
        3 name             char (32),
        3 calls            fixed bin,
        3 page_waits       fixed bin,
        3 time fixed       bin(71);
```

STRUCTURE ELEMENTS

num_ents
    is the number of entries in the array info.

name
    is the entryname.

calls
    is the number of calls to that entry.

page_waits
    is the number of page waits by that entry.

time
    is the CPU time in (microseconds) used by that entry.

Name: mhcs_

Entry: mhcs_$get_seg_usage

This entry point returns the number of page faults taken on a segment since its creation.

USAGE

```
declare mhcs_$get_seg_usage entry (char (*), char (*), fixed bin(35),
      fixed bin(35));

call mhcs_$get_seg_usage (dir_name, entryname, use, code);
```

ARGUMENTS

dir_name
    is the directory containing the segment. (Input)

entryname
    is the entry name of the segment. (Input)

use
    is the page fault count. (Output)

code
    is a standard status code. (Output)

NOTES

This entry point works for segments only and cannot be used to determine the page faults on a directory.

Entry: mhcs_$get_seg_usage_ptr

This entry point works the same as mhcs_$get_seg_usage except that it takes a pointer to the segment.

USAGE

```
declare mhcs_$get_seg_usage_ptr entry (ptr, fixed bin(35),
      fixed bin(35));
```

```
call mhcs_$get_seg_usage_ptr (s_ptr, use, code);
```

ARGUMENTS

s_ptr
    is a pointer to the segment. (Input)

use
    is the page fault count. (Output)

code
    is a standard status code. (Output)

Name: mlr_

The mlr_ subroutine moves a character string by copying the characters from left to right.

USAGE

```
declare mlr_ entry (ptr, fixed bin(21), ptr, fixed bin(21));

call mlr_ (input_ptr, input_lth, output_ptr, output_lth);
```

ARGUMENTS

input_ptr
is a pointer to the input string. (Input)

input_lth
is the length of the input string in characters. (Input)

output_ptr
is a pointer to the output string. (Input)

output_lth
is the length of the output string in characters. (Input)

NOTES

If the output string is shorter than the input string, only the first output_lth characters of the input string are moved. If the output string is longer than the input string, the output string is padded on the right with blanks.

The following call to mlr_ --

```
call mlr_ (addcharno (addr (text), start), lth,
           addcharno (addr (text), start-N), lth);
```

where N is a positive number is equivalent to the PL/I statement --

```
substr (text, start, lth) = substr (text, start+N, lth);
```

Name: mode_string_

The mode_string_ subroutine provides a set of entry points for handling mode strings. Mode strings are a way for a user to pass control information to a command or subsystem. A mode string is a character string which contains one or more modes, or is empty. A mode is a character string, separated from other modes by a comma. A mode specifies the name of a parameter and (implicitly) the data type and value of the parameter. Parameter names are character strings of one to 32 characters. Parameters may be one of the following types: Boolean, Numeric, or Character. The type and value of the parameter are determined in the following way:

If the first character in the mode is the circumflex character ("^"), then the parameter is a Boolean type whose value is "false" and whose name is all the remaining characters in the mode.

If the mode contains the equal character ("="), then the name of the parameter is given by all the characters before the equal character. If all characters after the equal character are decimal digits or sign characters ("+" or "-"), then the parameter is of type Numeric, and the value is the number given, which is of precision fixed binary (35,0). Otherwise, the type is Character, and the value is the character string beginning after the first equal character. The value may be the null string. The character string may be enclosed in quotes to distinguish it from a Numeric value, or if it contains a reserved character. Reserved characters are circumflex (^), comma, period, equals (=), double quote, and any character other than the 94 printing graphic characters in the Multics character set. Character values are limited to 32 characters in length.

If the mode does not contain the equal character, then if the last N characters are decimal digits or sign characters (where N > 1), then the parameter is of type Numeric, and those digits specify the value. Otherwise, the parameter is of type Boolean and the value is "true".

White space is permitted anywhere in a mode string. White space, however, is not insignificant; it separates tokens and may cause syntax errors if delimiter characters, such as comma and equal, are omitted. White space is defined as any number of the characters SPACE, TAB, NEWLINE, FORMFEED or VERTICAL TAB in any order. This definition is the same as the one used by the Multics command processor when scanning active function return values produced by the "|[" construct.

A period (.) is permitted at the end of the mode string to delimit it. If any nonwhite characters follow an unquoted period in the mode string, the mode string is in incorrect format.

Ambiguous modes are not permitted. An ambiguous mode is one which begins with one or more circumflexes (^) and which contains a Numeric or Character value.

EXAMPLES

mode	*name*	*type*	*value*
crecho	crecho	Boolean	true
^lfecho	lfecho	Boolean	false
^^tab	tab	Boolean	true
ll79	ll	Numeric	79
ll=79	ll	Numeric	79
indent=-5	indent	Numeric	-5
audit_trigger=@	audit_trigger	Character	@
more_mode=scroll	more_mode	Character	scroll
prompt=" -> "	prompt	Character	" -> "
prompt=	prompt	Character	""

illegal specifications

^ll79	^^ll79	^ll=79	^ll=foo	^ll="foo"
^x^	x=y=z	x y	"foo"	ll"foo"

INFO STRUCTURE

The mode_string_ entries describe a mode string with the following data structures, declared in mode_string_info.incl.pl1

```
dcl 1 mode_string_info         aligned based (mode_string_info_ptr),
      2 version                fixed bin,
      2 number                 fixed bin,
      2 modes                  (number_of_modes refer
                               (mode_string_info.number))
                               like mode_value;
```

STRUCTURE ELEMENTS

version
: gives the version of the structure. The most recent version is given by the constant mode_string_info_version_2, also declared in the include file. If the caller is supplying the structure as input, the caller must ensure that this value is set. If the structure is returned by one of the entries, the value will be set, and the caller should check it.

number
: gives the number of parameters in the mode string.

modes
: are the component modes of the mode string.

A parameter (mode value) is described by the following structure:

```
dcl 1 mode_value              aligned based (mode_value_ptr),
      2 version               fixed bin,
      2 mode_name             char (32) unal,
      2 flags,
        3 boolean_valuep      bit (1) unal,
        3 numeric_valuep      bit (1) unal,
        3 char_valuep         bit (1) unal,
        3 boolean_value       bit (1) unal,
        3 pad1                bit (32) unal,
      2 numeric_value         fixed bin (35),
      2 char_value            char (32) varying,
      2 code                  fixed bin (35),
      2 pad2                  bit (36);
```

STRUCTURE ELEMENTS

version
: gives the version of the structure. The most recent version is given by the constant mode_value_version_3, also declared in the include file.

mode_name
: is the name of the parameter

flags
: describe the parameter.

boolean_valuep
: is "1"b for a Boolean parameter.

numeric_valuep
: is "1"b for a Numeric parameter.

char_valuep
: is "1"b for a Character parameter.

boolean_value
: is valid only for a Boolean parameter, and holds its value.

pad1
: must be "0"b.

numeric_value
: is valid only for a Numeric parameter, and holds its value.

char_value
: is valid only for a Character parameter, and holds its value. Note that the string is varying to permit (quoted) trailing whitespace in a mode value.

code
is an error code for the particular mode, and normally is zero.

pad2
must be "0"b.

For all entry points in the mode_string_ subroutine, the following codes can be returned:

error_table_$bad_mode_syntax
for a mode string with bad syntax.
error_table_$undefined_mode
when a mode searched for is not found.
error_table_$mode_string_truncated
when mode string to be returned to the caller will not fit in the caller-supplied string. In this case, the string returned is truncated to the nearest whole mode.

Entry: mode_string_$combine

The mode_string_$combine entry point returns a mode string which represents the union of the modes defined in the two input arguments. The order of modes in the output string is not defined. If the same parameter is given in both structures, the type and value are taken from the second structure.

USAGE

```
declare mode_string_$combine entry (ptr, ptr, char(*), fixed bin(35));

call mode_string_$combine (mode_string_info_ptr1, mode_string_info_ptr2,
     modestr, code);
```

ARGUMENTS

mode_string_info_ptr1
points to the first mode_string_info structure. (Input)

mode_string_info_ptr2
points to the second mode_string_info structure. (Input) This pointer may be null, and the string is formed from the first structure only.

modestr
is a mode string. (Output)

code
is a standard system error code. (Output)

Entry: mode_string_$delete

The mode_string_$delete entry point returns a new mode string, with any mention of specified modes deleted. It is not an error if any of the specified modes are absent from the structure.

USAGE

```
declare mode_string_$delete entry (ptr, (*) char(*), char(*),
      fixed bin(35));

call mode_string_$delete (mode_string_info_ptr, excludes, modestr,
      code);
```

ARGUMENTS

mode_string_info_ptr
    is a pointer to the mode_string_info structure. (Input)

excludes
    is the array of names to be excluded. (Input) To exclude a single name, a scalar may be given. (Input)

modestr
    is a mode string. (Output)

code
    is a standard system error code. (Output)

Entry: mode_string_$get

The mode_string_$get entry point returns a mode string formed from the mode string info structure supplied it. If the caller supplied string is not long enough to hold the mode string, it is truncated at the nearest whole mode, and the error code error_table_$mode_string_truncated is returned. This ensures that the mode string returned is valid.

USAGE

```
declare mode_string_$get entry (ptr, char(*), fixed bin(35));

call mode_string_$get (mode_string_info_ptr, modestr, code);
```

ARGUMENTS

mode_string_info_ptr
is a pointer to the mode_string_info structure. (Input)

modestr
is a mode string. (Output)

code
is a standard system error code. (Output)

Entry: mode_string_$get_error

The mode_string_$get_error entry point is just like the mode_string_$get entry point except that the string returned only contains modes where mode_value.code was nonzero. This selection mechanism can be used to return a list of bad modes when a call to iox_$modes fails, for inclusion in an error message.

USAGE

```
declare mode_string_$get_error entry (ptr, char(*), fixed bin(35));

call mode_string_$get_error (mode_string_info_ptr, modestr, code);
```

ARGUMENTS

mode_string_info_ptr
is a pointer to the mode_string_info structure. (Input)

modestr
is a mode string. (Output)

code
is a standard system error code. (Output)

Entry: mode_string_$get_mode

The mode_string_$get_mode entry point parses a supplied mode string and extracts a single parameter from it, filling in a caller-supplied mode_value structure (remember to set the version), or returning an error code if the parameter is not present in the string.

USAGE

```
declare mode_string_$get_mode (char(*), char(*), ptr, fixed bin(35));

call mode_string_$get_mode (modestr, mode_name, mode_value_ptr, code);
```

ARGUMENTS

modestr
is a mode string. (Output)

mode_name
is the name of the mode to search for. (Input)

mode_value_ptr
is a pointer to a mode_value structure. (Input)

code
is a standard system error code. (Output)

Entry: mode_string_$parse

The mode_string_$parse entry point parses a mode string, allocating a structure giving the parameters specified in the string.

USAGE

```
declare mode_string_$parse entry (char (*), ptr, ptr, fixed bin(35));

call mode_string_$parse (modestr, areap, mode_string_info_ptr, code);
```

ARGUMENTS

modestr
is a mode string. (Input)

areap
points to an area where the mode string info structure may be allocated. (Input) If a null pointer is provided, the system area is used.

mode_string_info_ptr
is a pointer to a mode_string_info structure. (Output)

code
is a standard system error code. (Output)

NOTES

The error code error_table_$bad_mode_value has been provided for the use of callers of this interface to return when rejecting modes for incorrect type.

Name: mrl_

The mrl_ subroutine moves a character string by copying the characters from right to left.

USAGE

```
declare mrl_ entry (ptr, fixed bin(21), ptr, fixed bin(21));

call mrl_ (input_ptr, input_lth, output_ptr, output_lth);
```

ARGUMENTS

input_ptr
is a pointer to the input string. (Input)

input_lth
is the length of the input string in characters. (Input)

output_ptr
is a pointer to the output string. (Input)

output_lth
is the length of the output string in characters. (Input)

NOTES

If the output string is shorter than the input string, only the last output_lth characters of the input string are moved. If the output string is longer than the input string, the output string is padded on the left with blanks.

The following PL/I statement --

```
substr (text, start, lth) = substr (text, start+N, lth);
```

where N is a positive number will not execute properly as the code generated by the compiler moves the character string from left to right which destroys the contents of the string. Instead, the following call to mrl_ should be used --

```
call mrl_ (addcharno (addr (text), start), lth,
           addcharno (addr (text), start+N), lth);
```

Name: msf_manager_

The msf_manager_ subroutine provides a centralized and consistent facility for handling multisegment files. Multisegment files are files that can require more than one segment for storage. Examples of multisegment files are listings, data used through I/O switches, and APL workspaces. The msf_manager_ subroutine makes multisegment files almost as easy to use as single segment files in many applications.

A multisegment file is composed of one or more components, each the size of a segment, identified by consecutive unsigned integers. Any word in a single segment file can be specified by a pathname and a word offset. Any word in a multisegment file can be specified by a pathname, component number, and word offset within the component. The msf_manager_ subroutine provides the means for creating, accessing, and deleting components, truncating the multisegment file, and controlling access.

In this implementation, a multisegment file with only component 0 is stored as a single segment file, unless the msf_manager_$msf_get_ptr entrypoint was responsible for creating the file, in which case it is stored as a multisegment file with only one component. If components other than 0 are present, they are stored as segments with names corresponding to the ASCII representation of their component numbers in a directory with the pathname of the multisegment file.

The ACL of a multisegment file is maintained on each of its components. This ACL is translated into a similar directory ACL maintained on the directory portion of the multisegment file. The directory ACL is maintained such that all users have at least "s" access to the directory portion so that all users can determine their actual access mode to the multisegment file.

To keep information between calls, the msf_manager_ subroutine stores information about files in per-process data structures called file control blocks. The user is returned a pointer to a file control block by the entry point msf_manager_$open. This pointer, fcb_ptr, is the caller's means of identifying the multisegment file to the other msf_manager_ entry points. The file control block is freed by the msf_manager_$close entry point.

Entry: msf_manager_$acl_add

This entry point adds the specified access modes to the ACL of a multisegment file.

USAGE

```
declare msf_manager_$acl_add entry (ptr, ptr, fixed bin, fixed bin(35));

call msf_manager_$acl_add (fcb_ptr, acl_ptr, acl_count, code);
```

ARGUMENTS

fcb_ptr
is a pointer to the file control block. (Input)

acl_ptr
points to the user-supplied segment_acl_array structure (described under "Notes" below). (Input)

acl_count
is the number of ACL entries in the segment_acl_array structure. (Input)

code
is a storage system status code. (Output) It can be:
error_table_$argerr
the erroneous ACL entries in the segment_acl_array structure have status_code set to an appropriate error code. No processing is performed.

NOTES

The following is the segment_acl_array structure (declared in acl_structures.incl.pl1):

```
dcl 1 segment_acl_array (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);
```

STRUCTURE ELEMENTS

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

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. The include file access_mode_values.incl.pl1 defines mnemonics for these values:

```
dcl (N_ACCESS                init ("000"b),
     R_ACCESS                init ("100"b),
     E_ACCESS                init ("010"b),
     W_ACCESS                init ("001"b),
     RE_ACCESS               init ("110"b),
     REW_ACCESS              init ("111"b),
     RW_ACCESS               init ("101"b)),
  bit (3) internal static options (constant);
```

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

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

Entry: msf_manager_$acl_delete

This entry point deletes ACL entries from the ACL of a multisegment file.

USAGE

```
declare msf_manager_$acl_delete entry (ptr, ptr, fixed bin,
     fixed bin(35));

call msf_manager_$acl_delete (fcb_ptr, acl_ptr, acl_count, code);
```

ARGUMENTS

fcb_ptr
  is a pointer to the file control block. (Input)

acl_ptr
  points to a user-supplied delete_acl structure. See "Notes" below. (Input)

acl_count
  is the number of ACL entries in the delete_acl structure. (Input)

code
  is a storage system status code. (Output)

NOTES

The delete_acl structure (declared in acl_structures.incl.pl1) is as follows:

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

STRUCTURE ELEMENTS

access_name
  is the access name (in the form Person_id.Project_id.tag) of an ACL entry to be deleted.

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

NOTES

If code is error_table_$argerr, no processing is performed and status_code in each erroneous ACL entry is set to an appropriate error code.

If an access name matches no name already on the ACL, then the status_code for that delete_acl entry is set to error_table_$user_not_found. Processing continues to the end of the delete_acl structure and code is returned as 0.

Entry: msf_manager_$acl_list

This entry point returns the access control list (ACL) of a multisegment file.

USAGE

```
declare msf_manager_$acl_list entry (ptr, ptr, ptr, ptr, fixed bin,
      fixed bin(35));

call msf_manager_$acl_list (fcb_ptr, area_ptr, area_ret_ptr, acl_ptr,
      acl_count, code);
```

ARGUMENTS

fcb_ptr
    is a pointer to the file control block. (Input)

area_ptr
    points to an area in which the list of ACL entries, which make up the entire ACL of the multisegment file, 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. (Input)

area_ret_ptr
    points to the start of the allocated list of ACL entries. (Output)

acl_ptr
    if area_ptr is null, then acl_ptr points to an ACL structure, segment_acl_array, (described in the msf_manager_$acl_add entry point above) into which mode information is placed for the access names specified in that same structure. (Input)

acl_count
    is the number of entries in the segment_acl_array structure. (Input/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_array structure allocated in the area pointed to by area_ptr, if area_ptr is not null.

code
is a storage system status code. (Output)

NOTES

If acl_ptr is used to obtain modes for specified access names (rather than obtaining modes for all access names in area_ret_ptr), then each ACL entry in the segment_acl_array structure either has status_code set to 0 and contains the multisegment mode of the file or has status_code set to error_table_$user_not_found and contains a mode of 0.

Entry: msf_manager_$acl_replace

This entry point replaces the ACL of a multisegment file.

USAGE

```
declare msf_manager_$acl_replace entry (ptr, ptr, fixed bin, bit(1),
     fixed bin(35));

call msf_manager_$acl_replace (fcb_ptr, acl_ptr, acl_count,
     no_sysdaemon_sw code);
```

ARGUMENTS

fcb_ptr
is a pointer to the file control block. (Input)

acl_ptr
points to the user-supplied segment_acl_array structure (described in the msf_manager_$acl_add entry point above) that is to replace the current ACL. (Input)

acl_count
is the number of entries in the segment_acl_array structure. (Input)

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

code
is a storage system status code. (Output)

NOTES

If acl_count is zero, 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_array entries is performed top to bottom, allowing a later entry to overwrite a previous one if the access_name in the segment_acl_array structure is identical.

Entry: msf_manager_$adjust

The msf_manager_$adjust entry point optionally sets the bit count, truncates, and terminates the components of a multisegment file. The number of the last component and its bit count must be given. The bit counts of all components but the last are set to the first component's max_length*36. All components with numbers greater than the given component are deleted. All components that have been initiated are terminated. A 3-bit switch is used to control these actions.

USAGE

```
declare msf_manager_$adjust entry (ptr, fixed bin, fixed bin(24),
      bit(3), fixed bin(35));

call msf_manager_$adjust (fcb_ptr, component, bc, switch, code);
```

ARGUMENTS

fcb_ptr
: is a pointer to the file control block. (Input)

component
: is the number of the last component. (Input)

bc
: is the bit count to be placed on the last component. (Input)

switch
: is a 3-bit count/truncate/terminate switch. (Input)

  bit count
  - "0"b do not set the bit count.
  - "1"b set the bit count.

  truncate
  - "0"b do not truncate the given component.
  - "1"b truncate the given component to the length specified in the bc argument.

  terminate
  - "0"b do not terminate the component.
  - "1"b terminate the component.

code
    is a storage system status code. (Output)

Entry: msf_manager_$close

This entry point terminates all components that the file control block indicates are initiated and frees the file control block.

USAGE

```
declare msf_manager_$close entry (ptr);

call msf_manager_$close (fcb_ptr);
```

ARGUMENTS

fcb_ptr
    is the pointer to the file control block.

Entry: msf_manager_$get_ptr

This entry point returns a pointer to a specified component in a multisegment file. The component can be created if it does not exist. If the file is a single segment file, and a component greater than 0 is requested, the single segment is converted to a component 0. (See also the msf_manager_$msf_get_ptr entry point.)

USAGE

```
declare msf_manager_$get_ptr entry (ptr, fixed bin, bit(1), ptr,
      fixed bin(24), fixed bin(35));

call msf_manager_$get_ptr (fcb_ptr, component, create_sw, seg_ptr, bc,
      code);
```

ARGUMENTS

fcb_ptr
    is a pointer to the file control block. (Input)

component
    is the number of the component desired. (Input)

create_sw
    is the create switch. (Input)
    "1"b create the component if it does not exist.
    "0"b do not create the component if it does not exist.

seg_ptr
    is a pointer to the specified component in the file, or null (if there is an error). (Output)

bc
    is the bit count of the component. (Output)

code
    is a storage system status code. (Output) It can be:
    error_table_$noentry
        if the component requested did not exist and create_sw is off.

Entry: msf_manager_$msf_get_ptr

This entry point returns a pointer to a specified component in a multisegment file. The component can be created if it does not exist. If the file is a single segment file, and the requested component is not component 0, the single segment is converted to a multisegment file. This change does not affect a previously returned pointer to component 0. If the file does not exist, it is created as a "multi-segment file" with a single component. This entry point never creates a single segment file. (See also the msf_manager_$get_ptr entrypoint.)

USAGE

```
declare msf_manager_$msf_get_ptr entry (ptr, fixed bin, bit(1), ptr,
      fixed bin(24), fixed bin(35));

call msf_manager_$msf_get_ptr (fcb_ptr, component, create_sw, seg_ptr,
      bc, code);
```

ARGUMENTS

fcb_ptr
    is a pointer to the file control block. (Input)

component
    is the number of the component desired. (Input)

create_sw
    is the create switch. (Input)
    "1"b create the component if it does not exist.
    "0"b do not create the component if it does not exist.

seg_ptr
    is a pointer to the specified component in the file, or null (if there is an error). (Output)

bc
  is the bit count of the component. (Output)

code
  is a storage system status code. (Output) It can be:
  error_table_$noentry
    if the component requested did not exist and create_sw is off.

Entry: msf_manager_$open

The msf_manager_$open entry point creates a file control block and returns a pointer to it. The file need not exist for a file control block to be created for it.

USAGE

```
declare msf_manager_$open entry (char(*), char(*), ptr, fixed bin(35));

call msf_manager_$open (dir_name, entryname, fcb_ptr, code);
```

ARGUMENTS

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

entryname
  is the entryname of the multisegment file. (Input)

fcb_ptr
  is a pointer to the file control block. (Output)

code
  is a storage system status code. The code error_table_$dirseg is returned when an attempt is made to open a directory. (Output)

NOTES

If the file does not exist, fcb_ptr is nonnull and the code error_table_$noentry is returned. If the file cannot be opened, fcb_ptr is null and the value of code returned indicates the reason for failure.

Name: mvt_

The mvt_ subroutine provides for extremely efficient translation of character strings using translations which are not known at compile time.

USAGE

```
declare mvt_ entry (ptr, ptr, fixed bin(21), char(512) aligned);

call mvt_ (input_string_ptr, output_string_ptr, string_lth,
      translate_table);
```

ARGUMENTS

input_string_ptr
: is a pointer to the unaligned string to be translated. (Input)

output_string_ptr
: is a pointer to the string where the results of the translation will be placed. (Input)

string_lth
: is the length of both the input string and the output string in characters. (Input)

translate_table
: is the translation table which defines the actual translation. See mvt_$make_translation_table for a description of how to create this table. (Input)

Entry: mvt_$make_translation_table

This entry point creates the translation table used by the mvt_ subroutine given the second and third arguments which would be supplied to the PL/I translate builtin function.

USAGE

```
declare mvt_$make_translation_table entry (char(*), char(*), char(512)
      aligned);

call mvt_$make_translation_table (translated_list, untranslated_list,
      translate_table);
```

ARGUMENTS

translated_list
: is the second argument to the PL/I translate builtin and specifies the result of translating any occurrence of the corresponding characters in untranslated_list present in the input string of the mvt_ entry described above. (Input)

untranslated_list
is the third argument to the PL/I translate builtin and specifies the list of characters which will be translated if found in the input string. (Input)

translate_table
is set to the translate table which defines the desired translation. (Output)

NOTES

The table constructed by this subroutine will cause any occurence of the N'th character in untranslated_list present in the input string of mvt_ to be converted into the N'th character in translated_list. See the description of the PL/I translate builtin for more information.

If the PL/I builtin would have been used with only two arguments, use the value of the collate9 builtin for the untranslated_list argument.

Name: nd_handler_

The nd_handler_ subroutine attempts to resolve the name duplication caused when a program tries to create a segment, multisegment file, or link in a directory that already contains an entry by the same name. If the existing entry has additional names, nd_handler_ tries to delete the name needed for the new entry and, if successful, prints a warning message. If the existing entry has only one name, nd_handler_ queries the user whether or not to delete it. A zero status code in either case means that nd_handler_ has succeeded, and the calling program can retry creating the new entry.

USAGE

```
dcl nd_handler_ entry (char (*), char (*), char (*), fixed bin(35));

call nd_handler_ (caller, dn, en, code);
```

ARGUMENTS

caller
is the name of the calling program, used in printed messages. (Input)

dn
is the pathname of the directory involved. (Input)

en
is the name of the entry that the calling program wants to create. (Input)

code
is a standard status code. (Output) It can be:
0
if the old entryname has been removed.
error_table_$action_not_performed
if the user answered "no" to a query.
other codes
if the old entryname could not be removed for some other reason such as lack of access. An error message is then printed by nd_handler_.

NOTES

This subroutine is usually called after another subroutine call has returned error_table_$namedup. If nd_handler_ returns a zero status code, the other subroutine is called a second time. A warning message of the following kind is printed if the existing entry has multiple names:

```
caller:  Name duplication.  Old name foo removed from >udd>m>Smith>oldseg.
```

If the existing entry has only one name, wording of the query depends on the existing entry's type:

```
caller:  Do you want to delete the old segment <path>?
caller:  Do you want to delete the old multisegment file <path>?
caller:  Do you want to unlink the old link <path>?
              (Target <path2> exists.)
         or:  (Target <path2> does not exist.)
         or:  (Cannot get info for target <path2>.)
         or:  (No target pathname.)
```

The following entry points have the same calling sequence.

Entry: nd_handler_$del

This entry point queries whether or not to delete the existing entry, regardless of whether or not it has additional names.

Entry: nd_handler_$del_force

This entry point deletes the old entry (no query), regardless of whether it has additional names.

Entry: nd_handler_$force

This entry point deletes the existing entry if it has only one name, rather than issue a query.

Name: numeric_to_ascii_

The numeric_to_ascii_ subroutine formats a real decimal floating-point number. Integer, fractional, or exponential format is used depending on the number being formatted. The value returned by this function is a varying character string that can contain an optional minus sign, from 1 to 59 decimal digits, and, in some cases, an exponent field. The caller can control the number of digits placed in the string.

For numbers based in a number system other than base 10, use the numeric_to_ascii_base_ subroutine.

USAGE

```
declare numeric_to_ascii_ entry (float dec(59), fixed bin) returns
      (char (72) varying);

result = numeric_to_ascii_ ((value), precision);
```

ARGUMENTS

value
: is the value to be formatted. (Input) The PL/I compiler converts to float dec(59) if the attributes of value are different. The extra pair of parentheses around value suppresses the warning message about the conversion that would normally be generated.

precision
: controls the number of digits placed in the output string. (Input) If precision is equal to 0, from 1 to 59 digits are placed in the result string depending on the value being formatted. If precision is less than 0, the decimal value is truncated to the specified number of digits. If precision is greater than 0, the decimal value is rounded to the specified number of digits. In the cases where precision is not 0, no more than the specified number of digits are placed in the output string.

result
: is the character-string representation of value; it contains no blanks. (Output)

NOTES

To convert integers, use the PL/I sequence:

```
result = ltrim (char (value));
```

If precision equals 0, 59 is used for the precision. In the following discussion, P is equal to min (59, precision).

A number in integer format consists of a string of from 1 to P decimal digits without a decimal point. Integer format is used for integers whose absolute value is less than 10**P.

A number in fractional format consists of from 1 to P decimal digits with a decimal point. Trailing zeros in the fractional part are omitted; a number less than 1 has a 0 to the left of the decimal point. Fractional format is used for nonintegers that can be exactly represented in this format.

A number in exponential format appears as:

```
        xey          or          xe-y
```

where x is a number greater than 1 and less than 10 in fractional format and y is a power of 10 such that the numeric value being formatted is x*10**y. Exponential format is used whenever integer or fractional format cannot be used.

Name: numeric_to_ascii_base_

The numeric_to_ascii_base_ subroutine formats a real decimal floating-point number based in any number system from 2 to 16. For numbers in base 10, use the numeric_to_ascii_ subroutine. See numeric_to_ascii_ for details.

USAGE

```
declare numeric_to_ascii_base_ entry (float dec(59), fixed bin,
      fixed bin) returns (char(72) varying);

result = numeric_to_ascii_base_ ((value), precision, base);
```

ARGUMENTS

value
: is the value to be formatted. (Input)

precision
: controls the number of digits placed in the output string. (Input)

base
    is the radix of the number system in which the result is to be represented. (Input) For example, a base of 2 produces a binary representation, a base of 10 produces decimal, and a base of 16 produces hexadecimal. Bases from 2 through 16 are allowed.

result
    is the character-string representation of value; it contains no blanks. (Output)

NOTES

If precision equals 0, 59 is used for the precision. In the following discussion, P is equal to min (59, precision).

A number in integer format consists of a string of from 1 to P decimal digits without a decimal point. Integer format is used for integers whose absolute value is less than 10**P.

A number in fractional format consists of from 1 to P decimal digits with a decimal point. Trailing zeros in the fractional part are omitted; a number less than 1 has a 0 to the left of the decimal point. Fractional format is used for nonintegers that can be exactly represented in this format.

A number in exponential format appears as:

```
xey        or        xe-y
```

where x is a number greater than 1 and less than 10 in fractional format and y is a power of 10 such that the numeric value being formatted is x*10**y. Exponential format is used whenever integer or fractional format cannot be used.

When the result is represented in base 16, the following characters are used to express the result:

```
0 1 2 3 4 5 6 7 8 9 a b c d e f
```

When the result is represented in another base N, the first N characters from this list are used to express the result.

Name: object_info_

The object_info_ subroutine returns structural and identifying information extracted from an object segment. It has three entry points returning progressively larger amounts of information. All three entry points have identical calling sequences, the only distinction being the amount of information returned in the structure described in "Information Structure" below.

Entry: object_info_$brief

This entry point returns only the structural information necessary to locate the object's major sections.

USAGE

```
declare object_info_$brief entry (ptr, fixed bin(24), ptr,
      fixed bin(35));

call object_info_$brief (seg_ptr, bc, info_ptr, code);
```

ARGUMENTS

seg_ptr
: is a pointer to the base of the object segment. (Input)

bc
: is the bit count of the object segment. (Input)

info_ptr
: is a pointer to the info structure in which the object information is returned. See "Information Structure" later in this description. (Input)

code
: is a standard status code. (Output)

Entry: object_info_$display

This entry point returns, in addition to the information returned in the object_info_$brief entry point, all the identifying data required by certain object display commands, such as the print_link_info command.

USAGE

```
declare object_info_$display entry (ptr, fixed bin(24), ptr,
      fixed bin(35));

call object_info_$display (seg_ptr, bc, info_ptr, code);
```

ARGUMENTS

The arguments are the same as for the object_info_$brief entry point above.

Entry: object_info_$long

This entry point returns, in addition to the information supplied by the object_info_$display entry point, the data required by the Multics binder.

USAGE

```
declare object_info_$long entry (ptr, fixed bin(24), ptr,
      fixed bin(35));

call object_info_$long (seg_ptr, bc, info_ptr, code);
```

ARGUMENTS

The arguments are the same as in the object_info_$brief entry point above.

INFORMATION STRUCTURE

The information structure is as follows (as defined in the system include file object_info.incl.pl1):

```
dcl 1 object_info               aligned based,
      2 version_number          fixed bin,
      2 textp                   ptr,
      2 defp                    ptr,
      2 linkp                   ptr,
      2 statp                   ptr,
      2 symbp                   ptr,
      2 bmapp                   ptr,
      2 tlng                    fixed bin(18),
      2 dlng                    fixed bin(18),
      2 llng                    fixed bin(18),
      2 ilng                    fixed bin(18),
      2 slng                    fixed bin(18),
      2 blng                    fixed bin(18),
      2 format,
        3 old_format            bit(1) unaligned,
        3 bound                 bit(1) unaligned,
        3 relocatable           bit(1) unaligned,
        3 procedure             bit(1) unaligned,
        3 standard              bit(1) unaligned,
        3 gate                  bit(1) unaligned,
        3 separate_static       bit(1) unaligned,
        3 links_in_text         bit(1) unaligned,
```

```
      3 perprocess_static      bit(1) unaligned,
      3 pad                    bit(27) unaligned,
    2 entry_bound              fixed bin,
    2 textlinkp                ptr,
```

/*This is the limit of the $brief info structure.*/

```
    2 compiler                 char(8) aligned,
    2 compile_time             fixed bin(71),
    2 userid                   char(32) aligned,
    2 cvers                    aligned,
      3 offset                 bit(18) unaligned,
      3 length                 bit(18) unaligned,
    2 comment                  aligned,
      3 offset                 bit(18) unaligned,
      3 length                 bit(18) unaligned,
    2 source_map               fixed bin,
```

/*This is the limit of the $display info structure.*/

```
    2 rel_text                 ptr,
    2 rel_def                  ptr,
    2 rel_link                 ptr,
    2 rel_static               ptr,
    2 rel_symbol               ptr,
    2 text_boundary            fixed bin,
    2 static_boundary          fixed bin,
    2 default_truncate         fixed bin,
    2 optional_truncate        fixed bin;
```

/*This is the limit of the $long info structure.*/

STRUCTURE ELEMENTS

version_number
: is the version number of the structure (currently this number is 2). This value is input.

textp
: is a pointer to the base of the text section.

defp
: is a pointer to the base of the definition section.

linkp
: is a pointer to the base of the linkage section.

statp
: is a pointer to the base of the static section.

symbp
is a pointer to the base of the symbol section.

bmapp
is a pointer to the break map.

tlng
is the length (in words) of the text section.

dlng
is the length (in words) of the definition section.

llng
is the length (in words) of the linkage section.

ilng
is the length (in words) of the static section.

slng
is the length (in words) of the symbol section.

blng
is the length (in words) of the break map.

old_format
indicates the format of the segment.
"1"b old format.
"0"b new format.

bound
indicates whether the object segment is bound.
"1"b it is a bound object segment.
"0"b it is not a bound object segment.

relocatable
indicates whether the object is relocatable.
"1"b the object is relocatable.
"0"b the object is not relocatable.

procedure
indicates whether the segment is a procedure.
"1"b it is a procedure.
"0"b it is nonexecutable data.

standard
indicates whether the segment is a standard object segment.
"1"b it is a standard object segment.
"0"b it is not a standard object segment.

gate
indicates whether the procedure is generated in the gate format.
"1"b it is in the gate format.
"0"b it is not in the gate format.

separate_static
indicates whether the static section is separate from the linkage section..
"1"b static section is separate from linkage section.
"0"b static section is not separate from linkage section.

links_in_text
indicates whether the object segment contains text-embedded links.
"1"b the object segment contains text-embedded links.
"0"b the object segment does not contain text-embedded links.

perprocess_static
indicates whether the static section should be reinitialized for a run unit.
"1"b static section is used as is.
"0"b static section is per run unit.

pad
is currently unused.

entry_bound
is the entry bound if this is a gate procedure.

textlinkp
is a pointer to the first text-embedded link if links_in_text is equal to "1"b.

This is the limit of the info structure for the object_info_$brief entry point.

compiler
is the name of the compiler that generated this object segment.

compile_time
is the date and time this object was generated.

userid
is the access identifier (in the form Person_id.Project_id.tag) of the user in whose behalf this object was generated.

cvers.offset
is the offset (in words), relative to the base of the symbol section, of the aligned variable length character string that describes the compiler version used.

cvers.length
is the length (in characters) of the compiler version string.

comment.offset
is the offset (in words), relative to the base of the symbol section, of the aligned variable length character string containing some compiler-generated comment.

comment.length
is the length (in characters) of the comment string.

source_map
is the offset (relative to the base of the symbol section) of the source map.

This is the limit of the info structure for the object_info_$display entry point.

rel_text
is a pointer to the object's text section relocation information.

rel_def
is a pointer to the object's definition section relocation information.

rel_link
is a pointer to the object's linkage section relocation information.

rel_static
is a pointer to the object's static section relocation information.

rel_symbol
is a pointer to the object's symbol section relocation information.

text_boundary
partially defines the beginning address of the text section. The text must begin on an integral multiple of some number, e.g., 0 mod 2, 0 mod 64; this is that number.

static_boundary
is analogous to text_boundary for internal static.

default_truncate
is the offset (in words), relative to the base of the symbol section, starting from which the symbol section can be truncated to remove nonessential information (e.g., relocation information).

optional_truncate
is the offset (in words), relative to the base of the symbol section, starting from which the symbol section can be truncated to remove unwanted information (e.g., the compiler symbol tree).

Name: ocu_

The ocu_ subroutine allows generation of standard format object segments and multi-segment files. The information is emitted via calls to ocu_ entrypoints and stored in internal tables until the invocation is closed, at which time the object is created and the sections assembled and linked together properly.

Entry: ocu_$backpatch

It is often necessary in the creation of an object segment to generate a reference to something which has not been emitted yet. This entrypoint allows changes to be made in a word which has already been emitted. Since many sections of the object segment are being synthesized by ocu_ from other information, it is not practical to patch them. (eg. the definition section contains type_pairs, expression_words, init_info, and ACC_strings generated as byproducts of link generation. The offsets of these items are not known until the object is closed.) This entry is primarily for patching sections which were emitted as blocks of words. (ie. text, static, and symbol sections.)

USAGE

```
dcl ocu_$backpatch entry (ptr, char (*), fixed bin (18) unsigned,
     char (*), fixed bin (35);

call ocu_$backpatch (ocu_datap, section, offset, side, new_value);
```

ARGUMENTS

ocu_datap
: is a pointer returned by ocu_$open. This identifies all the data structures needed to create the object segment. (Input)

section
: is a character identifying the section to be patched. (Input)
Valid values for this argument are:

"text"	to patch the text section.
"static"	to patch the static section.
"symbol"	to patch the symbol section.

offset
    is the word offset within the given section of the halfword to be patched. (Input)

side
    is a string indicating what portion of the specified word is to be patched. (Input) Valid values depend on the section being patched and correspond to the valid types of relocation allowed for that section.
- Text section
    "left 15 unsigned"
    "left 15 signed"
    "left 18 unsigned"
    "left 18 signed"
    "right 18 unsigned"
    "right 18 unsigned"
- Static section
    "left 18 unsigned"
    "left 18 signed"
    "right 18 unsigned"
    "right 18 signed"
- Symbol section
    "left 18 unsigned"
    "left 18 signed"
    "right 18 unsigned"
    "right 18 signed"

new_value
    is the new value to be patched into the specified portion of the word. (Input)

Entry: ocu_$close

takes the information provided by previous calls to ocu_ and assembles the final object segment. The relocation information, object_map, linkage header, definition string map, hash table, and header are synthesized at this point.

USAGE

```
dcl ocu_$close entry (ptr, fixed bin (35));

call ocu_$close (ocu_datap, code);
```

ARGUMENTS

ocu_datap
    is a pointer returned by ocu_$open. This identifies all the data structures needed to create the object segment. (Input)

code
is a standard status code. (Output)

Entry: ocu_$create_msf

Creates component 0 of an object MSF. Given an array of pointers to all of the components of a MSF (excepting component 0), generates component 0, copying the external definitions, and building the first reference trap.

USAGE

```
dcl ocu_$create_msf entry (ptr, fixed bin (15) unsigned, ptr,
      fixed bin (35));

call ocu_$create_msf (component_listp, component_count, gen_infop,
      code);
```

ARGUMENTS

component_listp
is a pointer to an array of pointers of dimension (1:component_count-1). (Input) Each pointer points to one component of the MSF. Each of the pointers points to a completed object segment. It is assumed that each of the components already has it's linkage section built as a MSF (ie. containing appropriate partially-snapped links) and that the msf_map is present in the definition section.

component_count
is the number of components in the final MSF not counting component 0. (Input)

gen_infop
is a pointer to the gen_info structure used to set the generator_info in the symbol header of component 0. (Input) The gen_info structure is declared in the include file ocu_dcls.incl.pl1.

```
dcl 01 gen_info                aligned based,
      02 gen_created           fixed bin (71),
      02 generator             char (8),
      02 gen_number            fixed bin,
      02 gen_version           char (512) varying;
```

gen_created
is the clock time that the generator was created.

generator
is the name of the generator (eg. PL/I, binder, etc.).

gen_number
: is the version number of the generator. This value must the version number if the gen_version string.

gen_version
: is a version string giving the name, version, and date of the generator (eg. Multics PL/I Compiler, Release 28e, of February 14, 1985)

code
: is a standard status code. (Output)

Entry: ocu_$emit_definition

Emits a single non-class-3 definition, and threads it into the definition list. Definitions are threaded in the order of the calls to ocu_$emit_definition and ocu_$emit_segname. Successive calls to emit_segname generate multiple segnames in a single block. Calls to emit_segname with intervening calls to emit_definition create a new block.

USAGE

```
dcl ocu_$emit_definition entry (ptr, char (*) varying,
      fixed bin (3), fixed bin (18) unsigned,
      bit (*)) returns (fixed bin (18) unsigned);

def_relp = ocu_$emit_definition (ocu_datap, name, section, offset,
      flags);
```

ARGUMENTS

ocu_datap
: is a pointer returned by ocu_$open. This identifies all the data structures needed to create the object segment. (Input)

name
: is the name of the definition. (Input)

section
: is the section that the definition refers to. (Input) Constants for the sections can be found in definition_dcls.incl.pl1. Valid sections for this subroutine are:

```
SECTION_TEXT   = 0
SECTION_LINK   = 1
SECTION_SYMBOL = 2
SECTION_STATIC = 4
```

offset
: is the offset of the target of the definition within the given section. (Input)

flags
is a bit string representing the flags to be set in the definition. (Input)
Constants definition the values can be found in ocu_dcls.incl.pl1.

```
DEFINITION_FLAGS_IGNORE   = "1000"b
DEFINITION_FLAGS_ENTRY    = "0100"b
DEFINITION_FLAGS_RETAIN   = "0010"b
DEFINITION_FLAGS_INDIRECT = "0001"b
```

def_relp
is an offset to the generated definition relative to the base of the definition section. (Output)

Entry: ocu_$emit_firstref_trap

Adds a firstref trap to the first reference trap block in the linkage section. The links reference by the call_relp and info_relp must have already been emitted. Errors encountered are reported using the sub_err_ subroutine.

USAGE

```
dcl ocu_$emit_firstref_trap entry (ptr, fixed bin (18) unsigned,
     fixed bin (18) unsigned);

call ocu_$emit_firstref_trap (ocu_datap, call_relp, info_relp);
```

ARGUMENTS

ocu_datap
is a pointer returned by ocu_$open. This identifies all the data structures needed to create the object segment. (Input)

call_relp
is the offset relative to the base of the linkage section of a link to be used to call the trap procedure. (Input)

info_relp
is the offset relative to the base of the linkage section of a link to be passed to the trap procedure. If this value is 0, no parameter will be passed to the trap procedure. (Input)

Entry: ocu_$emit_link

Creates a single external link. The expression word, type_pair, segname and offsetname strings, and any trap_words or external initialization information in the definition section are also generated as required. Errors encountered are reported using the sub_err_ subroutine.

USAGE

```
dcl ocu_$emit_link entry (ptr, fixed bin (3), fixed bin (3),
     char (*) var, char (*) var, fixed bin, bit (6), ptr)
     returns (fixed bin (18) unsigned);

link_relp = ocu_$emit_link (ocu_datap, type, class, segname, offsetname,
     expression, modifier, init_infop);
```

ARGUMENTS

ocu_datap
: is a pointer returned by ocu_$open. This identifies all the data structures needed to create the object segment. (Input)

type
: is the type of the link. Constants for the valid link types can be found in definition_dcls.incl.pl1. Valid valued are:

```
LINK_SELF_BASE          = 1
LINK_REFNAME_BASE       = 3
LINK_REFNAME_OFFSETNAME = 4
LINK_SELF_OFFSETNAME    = 5
```

class
: is the class of the link for type 1 (link self base) and type 5 (link self offsetname). This indicates what section of the object segment the expression value is relative to. It is used only if the type is 1 or 5. Constants usable for this value are declared in definition_dcls.incl.pl1. Valid values are:

```
CLASS_TEXT    = 0
CLASS_LINKAGE = 1
CLASS_SYMBOL  = 2
CLASS_STATIC  = 4
CLASS_SYSTEM  = 5
CLASS_HEAP    = 6
```

segname
: is the segname of the link target. This field is only used if the type of the link is type 3 (link-refname-base) or type 4 (link-refname-offsetname). This is the refname that will be used to search for the segment when the link is snapped. (Input)

offsetname
is the name of the definition to be searched for when the link is snapped. This field is only used if the link type is type 4 (link-refname-offsetname) or type 5 (link-self-offsetname). (Input)

expression
is a word offset to be added to the offset derived from the section and offsetname values. .(Input)

modifier
is the modifier of the link. This is the modifier that will be present in the pointer representing the snapped link. Generally a null modifier (""b) is used. (Input)

init_infop
is a pointer to the initialization info, or to a trap_pair. (Input)
If the link is a type 5, class 5 link (a *system or external link) or a type 5, class 6 link (a *heap link), this points to an initialization info block which will be placed into the definition section. This can point to any type of standard initialization info (INIT_NO_INIT, INIT_COPY_INFO, INIT_DEFINE_AREA, INIT_LIST_TEMPLATE, or INIT_DEFERRED if the object segment being created is an MSF component.) If the link is not a *system or *heap link, a non-null value will be assumed to point to a trap-pair representing a trap-before-link. Since trap-before-links are generally obsolete, this should only be non-null when supplying initialization_info for *system or *heap links.

link_relp
is the offset of this link relative to the base of the linkage section. Note that the link offset returned is the location of the link assuming there is no linkage-resident static section. When the object is closed (via a call to ocu_$close) all link references will be relocated to account for the presence of a static section. If you plan to use this returned link offset for purposes other than to store in one of the other object sections, you will have to adjust for the static section manually.

Entry: ocu_$emit_partial_link

Emits an MSF partially snapped link. A partially snapped link uses no information in the definition section, and is snapped before entry by a first reference trap. This entrypoint should ONLY be called when generating a MSF component. Errors are reported using the sub_err_ subroutine.

USAGE

```
dcl ocu_$emit_partial_link entry (ptr, fixed bin (15) unsigned,
      fixed bin (3), fixed bin (18) unsigned, bit (6))
      returns (fixed bin (18) unsigned);

link_relp = ocu_$emit_link (ocu_datap, component, section, offset,
      modifier);
```

ARGUMENTS

ocu_datap
: is a pointer returned by ocu_$open. This identifies all the data structures needed to create the object segment. (Input)

component
: is the component number of the target component within the MSF. Generally this will be in the range 1 to the maximum component number. (Input)

section
: is the target section of the link within the target MSF component. (Input) Constants for these values can be found in definition_dcls.incl.pl1. Valid values are:

```
SECTION_TEXT    = 0
SECTION_LINKAGE = 1
SECTION_SYMBOL  = 2
SECTION_STATIC  = 4
```

offset
: is the offset of the pointer. This value is relative to the base of the section specified by the section parameter. (Input)

modifier
: is the modifier of the link. This will also be the modifier of the pointer generated by snapping the link. The null modifier (""b) is generally used. (Input)

link_relp
: is the offset of the generated link relative to the base of the linkage section. Note that this value is calculated as if there were no static section resident in the linkage section. When the object is closed (via a call to ocu_$close) all linkage references are relocated to adjust for the presence of a static section. If the caller wishes to use this value for other purposes that to include in another call to ocu_, it will have to be adjusted for the presence of the static section manually. (Output)

Entry: ocu_$emit_segname

Emits a single class-3 (segname) definition, and threads it into the definition list. The definitions are chained in the order of calls to ocu_$emit_definition and ocu_$emit_segname. Sequential calls to emit_segname generate multiple segnames in a single block. A call to emit_segname after calls to emit_definition starts a new block. It is invalid to call emit_definition without calling emit_segname at least once.

USAGE

```
dcl ocu_$emit_segname entry (ptr, char (*) varying, bit (*))
     returns (fixed bin (18) unsigned);

def_relp = ocu_$emit_segname (ocu_datap, name, flags);
```

ARGUMENTS

ocu_datap
: is a pointer returned by ocu_$open. This identifies all the data structures needed to create the object segment. (Input)

name
: is the name of this segname definition. (Input)

flags
: is a bit string representing the flags to be set in the definition. (Input) Constants definition the values can be found in ocu_dcls.incl.pl1.

```
DEFINITION_FLAGS_IGNORE   = "1000"b
DEFINITION_FLAGS_ENTRY    = "0100"b
DEFINITION_FLAGS_RETAIN   = "0010"b
DEFINITION_FLAGS_INDIRECT = "0001"b
```

def_relp
: is an offset to the generated definition relative to the base of the definition section. (Output)

Entry: ocu_$emit_static

Emits a block of words which are appended to the static section. Since there is no relocation info for the static section (and it is forced to be absolute if it is contained in the linkage section), no relocation information is required. Note that even if the static section is to be contained in the linkage section, references to the static section should be made with static relocation info and not attempt to adjust the offsets for the presence of the linkage header. when the new object is closed, all static references will be mapped into the appropriate linkage references. Error encountered are reported using the sub_err_ subroutine.

USAGE

```
dcl ocu_$emit_static entry (ptr, ptr, fixed bin (18) unsigned,
      returns (fixed bin (18) unsigned);

static_relp = ocu_$emit_static (ocu_datap, staticp, word_count);
```

ARGUMENTS

ocu_datap
: is a pointer returned by ocu_$open. This identifies all the data structures needed to create the object segment. (Input)

staticp
: is a pointer to an array or words of dimension (word_count) to be appended to the static section. (Input)

word_count
: is the number of words to be appended to the static section. (Input)

static_relp
: is the offset of the block or words relative to the base of the static section

Entry: ocu_$emit_symbol

Emits a block of symbol words and appends them to the symbol section. Errors encountered are reported using the sub_err_ subroutine.

USAGE

```
dcl ocu_$emit_symbol entry (ptr, ptr, ptr, fixed bin (18) unsigned)
      returns (fixed bin (18) unsigned);

symbol_relp = ocu_$emit_symbol (ocu_datap, symbolp, relocationp,
      word_count);
```

ARGUMENTS

ocu_datap
: is a pointer returned by ocu_$open. This identifies all the data structures needed to create the object segment. (Input)

symbolp
: is a pointer to an array symbol section words of dimension (word_count) to be appended to the symbol section. (Input)

relocationp
: is a pointer to a character string of length (2*word_count) representing the relocation information for the accompanying block of words. (Input)
The relocation characters are taken from the set of standard characters used by language translators (see the Multics Programmers Reference Manual). The relocation string is required even if the object to be generated is not relocatables since the relocation information is used to locate static and linkage references which will have to be relocated if the static section is linkage resident.

word_count
: is the number of symbol words to be emitted. (Input)

symbol_relp
: is the offset of this block of words relative to the base of the symbol section. (Output)

Entry: ocu_$emit_text

emits a block of text words, appending them to the end of the text section and returning the offset within the text section. Errors encountered are reported using the sub_err_ subroutine.

USAGE

```
dcl ocu_$emit_text entry (ptr, ptr, ptr, fixed bin (18) unsigned)
     returns (fixed bin (18) unsigned);

text_relp = ocu_$emit_text (ocu_datap, textp, relocationp, word_count);
```

ARGUMENTS

ocu_datap
: is a pointer returned by ocu_$open. This identifies all the data structures needed to create the object segment. (Input)

textp
: is a pointer to an array of text words of dimension (word_count) to be appended to the text section. (Input)

relocationp
: is a pointer to a character string of length (2*word_count) representing the relocation information associated with the text array. (Input)
The characters used are the standard character relocation codes used by the translators. (see the Multics Programmers Reference Manual). This string is required regardless of whether the output object is to be relocatable since it is used to relocate linkage and static references if the static section is not separate.

word_count
    is the number of words of text to be emitted. (Input)

text_relp
    is an offset to this block of words relative to the base of the section. (Output)

Entry: ocu_$emit_msf_map

Emits the msf_map in the definition section of the new object. This entrypoint should ONLY be called if the object segment being generated is an MSF component. Errors encountered are reported using calls to the sub_err_ subroutine.

USAGE

```
dcl ocu_$emit_msf_map (ptr, fixed bin (15) unsigned,
     fixed bin (15) unsigned);

call ocu_$emit_msf_map (ocu_datap, component_count, my_component);
```

ARGUMENTS

ocu_datap
    is a pointer returned by ocu_$open. This identifies all the data structures needed to create the object segment. (Input)

component_count
    is the number of components in the MSF, including component 0. (Input)

my_component
    is the number of the component being generated in the range 0 to component_count - 1. (Input)

Entry: ocu_$open

Allocates and initializes the data structures used to create the object segment and returns a pointer used to locate the structures.

USAGE

```
dcl ocu_$open entry (char (*), char (*), bit (*), ptr,
     fixed bin (35));

call ocu_$open (dir_name, entry_name, flags, ocu_datap, code);
```

ARGUMENTS

dir_name
    is the name of the directory in which the final object will be created. (Input)

entry_name
is the entry name of the output object segment. (Input)

flags
is a bit string indicating various options to be used in the creation of the object segment. (Input)
The following values may be used to derive the desired flag value. (found in ocu_dcls.incl.pl1)

OPEN_FLAGS_BOUND = "100000"b
The object being created will have the format of a bound object (ie. one containing multiple translator produced objects) and is formatted according to the standards for bound objects. This format is not enforced by ocu_ and it is the responsibility of the caller to set up the object properly.

OPEN_FLAGS_RELOCATABLE = "010000"b
The object being created has relocation information and can be used as input to the binder or linkage editor. This flag is used by ocu_ to determine whether or not to add the relocation information to the linkage section of the object segment when the object is closed.

OPEN_FLAGS_PROCEDURE = "001000"b
The object contains executable code.

OPEN_FLAGS_SEPARATE_STATIC = "000100"b
The object segment is to contain a static section rather than have the static section included in the linkage section. This flag is examined by ocu_ when closing the object segment to determine relocation of static and linkage section references and to generate the sections properly.

OPEN_FLAGS_PERPROCESS_STATIC = "000010"b
The static section of this object segment is not to be duplicated when called from within a run unit.

OPEN_FLAGS_NO_HASHTABLE = "000001"b
Do not create a definition section hash table for this object segment. This is primarily used when creating either MSF components (which are never searched) or objects with very few entrypoints.

ocu_datap
is a pointer to the ocu data structures used by the other calls. (Output)

code
is a standard status code. (Output)

Entry: ocu_$release

Releases table storage used by ocu_ when an invocation is aborted.

USAGE

```
dcl ocu_$release entry (ptr);

call ocu_$release (ocu_datap);
```

ARGUMENTS

ocu_datap
: is a pointer returned by ocu_$open. This identifies all the data structures to be released. (Input)

Name: parse_file_

The parse_file_ subroutine provides a facility for parsing ASCII text into symbols and break characters. It is recommended for occasionally used text-scanning applications. In applications where speed or frequent use are important, in-line PL/I code is recommended (to do parsing) instead.

A restriction of the subroutine is that the text to be parsed must be an aligned character string.

The initialization entry points, parse_file_$parse_file_init_name and parse_file_$parse_file_init_ptr, save a pointer to the text to be scanned and a character count in internal static storage. Thus, only one text can be parsed at one time.

Entry: parse_file_

This entry point scans the text file and returns the next break character or symbol. Blanks, newline characters, and comments enclosed by /* and */, however, are skipped.

USAGE

```
declare parse_file_ entry (fixed bin, fixed bin, fixed bin(1),
      fixed bin(1));

call parse_file_ (ci, cc, break, eof);
```

ARGUMENTS

ci
: is an index to the first character of the symbol or break character. (Output). (The first character of the text is considered to be character 1.)

cc
: is the number of characters in the symbol. (Output)

break
: is set to 1 if the returned item is a break character; otherwise, it is 0. (Output)

eof
: is set to 1 if the end of text has been reached; otherwise, it is 0. (Output)

Entry: parse_file_$parse_file_cur_line

This entry point returns to the caller the current line of text being scanned. This entry is useful in printing diagnostic error messages.

USAGE

```
declare parse_file_$parse_file_cur_line entry (fixed bin, fixed bin);

call parse_file_$parse_file_cur_line (ci, cc);
```

ARGUMENTS

ci
: is an index to the first character of the line. (Output). (The first character of the text is considered to be character 1.)

cc
: is the number of characters in the line. (Output)

Entry: parse_file_$parse_file_init_name

This entry point initializes the subroutine given a directory and an entry point name. It gets a pointer to the desired segment and saves it for subsequent calls in internal static.

USAGE

```
declare parse_file_$parse_file_init_name entry (char(*), char(*), ptr,
      fixed bin(35));

call parse_file_$parse_file_init_name (dir_name, entryname, ptr, code);
```

ARGUMENTS

dir_name
    is the directory name portion of the pathname of the segment to be parsed. (Input)

entryname
    is the entryname of the segment to be parsed. (Input)

ptr
    is a pointer to the segment. (Output)

code
    is a standard status code. (Output). It is zero if the segment is initiated. If nonzero, the segment cannot be initiated. It can return any code from hcs_$initiate except error_table_$segknown.

Entry: parse_file_$parse_file_init_ptr

This entry point initializes the parse_file_ subroutine with a supplied pointer and character count. It is used in cases where a pointer to the segment to be parsed is already available.

USAGE

```
declare parse_file_$parse_file_init_ptr entry (ptr, fixed bin);

call parse_file_$parse_file_init_ptr (ptr, cc);
```

ARGUMENTS

ptr
    is a pointer to a segment or an aligned character string. (Input)

cc
    is the character count of the ASCII text to be scanned. (Input)

Entry: parse_file_$parse_file_ptr

This entry point is identical to the parse_file_ entry point except that a pointer (with bit offset) to the break character or the symbol is returned instead of a character index.

USAGE

```
declare parse_file_$parse_file_ptr entry (ptr, fixed bin, fixed bin(1),
     fixed bin(1));
```

```
call parse_file_$parse_file_ptr (ptr, cc, break, eof);
```

ARGUMENTS

ptr
is a pointer to the symbol or the break character. (Output)

cc
is the number of characters in the symbol. (Output)

break
is set to 1 if the returned item is a break character; otherwise, it is 0. (Output)

eof
is set to 1 if the end of text has been reached; otherwise, it is 0. (Output)

Entry: parse_file_$parse_file_line_no

This entry point returns to the caller the current line number of text being scanned. This entry is useful in printing diagnostic error messages.

USAGE

```
declare parse_file_$parse_file_line_no entry (fixed bin);
```

```
call parse_file_$parse_file_line_no (cl);
```

ARGUMENTS

cl
is the number of the current line. (Output)

Entry: parse_file_$parse_file_set_break

This entry point is used to define break characters. Normally, all nonalphanumeric characters are break characters (including blank and newline).

USAGE

```
declare parse_file_$parse_file_set_break entry (char(*));
```

```
call parse_file_$parse_file_set_break (cs)
```

ARGUMENTS

cs
: is a control string. (Input). Each character found in cs is made a break character.

Entry: parse_file_$parse_file_unset_break

This entry point renders break characters as normal alphanumeric characters. It is not possible to unset blank, newline, or comment delimiters, however. These are always treated as break characters.

USAGE

```
declareparse_file_$parse_file_unset_break entry (char(*));

call parse_file_$parse_file_unset_break (cs);
```

ARGUMENTS

cs
: is a control string, each character of which is made a nonbreaking character. (Input)

EXAMPLES

Suppose the file zilch in the directory dir_name contains the following text:

```
name: foo; /*foo program*/
pathname: >bar;
linkage;
end;
fini;
```

The following calls could be made to initialize the parsing of zilch:

```
call parse_file_$parse_file_init_name (dir_name, zilch, ptr, code);
call parse_file_$parse_file_unset_break (">_");
declare atom char (cc) unaligned based (p);
```

Subsequent calls to the parse_file_$parse_file_ptr entry point would then yield the following:

atom	*break*	*eof*
name	0	0
:	1	0
foo	0	0
;	1	0
pathname	0	0
:	1	0
>bar	0	0
;	1	0
linkage	0	0
;	1	0
end	0	0
;	1	0
fini	0	0
;	1	0
-	-	-

Name: parse_io_channel_name_

The parse_io_channel_name_ subroutine parses a character string that is intended to be an IOM channel number.

USAGE

```
dcl parse_io_channel_name_ entry (char (*), fixed bin (3), fixed bin
     (8), fixed bin (35));

call parse_io_channel_name_ (arg, iom, channel, code);
```

ARGUMENTS

arg
: is the character string to be parsed. It must be of the format:

```
tagnumber
```

where tag is an IOM tag (a through d) and number is a decimal channel number from 0 to 63.

iom
: is the IOM to which the channel is connected. (Output)

channel
is the channel number. (Output)

code
is 0 if arg is a valid representation of a channel; otherwise, error_table_$bad_channel. (Output)

Name: pascal_util_

This subroutine provides interfaces for establishing and removing an on unit for the current procedure and for setting breakall mode on or off for a given Pascal text file.

Entry: pascal_util_$establish_on_unit

This entrypoint establishes an on unit for the current procedure (main or other). An on unit is used to establish a handler for an unusual occurrence (e.g., quit, program_interrupt, overflow, pascal_error). See the *Multics Programmer's Reference Manual* for a complete description of on units and conditions.

If an on unit already exists for the given condition, it is replaced by this one. An on unit can be removed with the pascal_util_$remove_on_unit procedure; on units are automatically removed when the procedure exits.

USAGE

```
$IMPORT
        'pascal_util_ (pascal)' :  establish_on_unit $

PROCEDURE establish_on_unit
        (condition_name :  PACKED ARRAY [a..b :  integer] of CHAR;
        PROCEDURE condition_handler) ; EXTERNAL :
```

ARGUMENTS

condition_name
names the condition for which an on unit is established. Any leading spaces are removed, as are all characters after, including the first space encountered, if any.

condition_handler
procedure to be called if this condition occurs. This procedure must be exported or imported (it cannot be internal).

EXAMPLES

```
establish_on_unit ('program_interrupt', abort_current_request_execution) ;

establish_on_unit ('cleanup', clean_up_environment) ;
```

Entry: pascal_util_$remove_on_unit

This entrypoint removes an on unit in the the current procedure (main or other). An on unit is used to establish a handler for an unusual occurrence (e.g., quit, program_interrupt, overflow, pascal_error). See the *Multics Programmer's Reference Manual* for a complete description of on units and conditions.

It is not an error if no on unit exists in the current procedure for the given condition. An on unit can be established using the pascal_util_$establish_on_unit procedure.

USAGE

```
$IMPORT
        'pascal_util_ (pascal)' :  remove_on_unit $

PROCEDURE remove_on_unit
        (condition_name :  PACKED ARRAY [a..b :  integer] of CHAR)
 ;
        EXTERNAL ;
```

ARGUMENTS

condition_name
: names the condition for which an on unit is removed. Any leading spaces are removed, as are all characters after, including the first space encountered, if any.

EXAMPLES

```
remove_on_unit ('program_interrupt') ;
```

Entry: pascal_util_$breakall_on

This entrypoint sets the given Pascal text file to breakall mode. Some screen applications may need to input characters as they are entered. Since Pascal text input is normally buffered line by line, use this procedure to tell Pascal I/O to use character-by-character input if your application so requires. It also sets Pascal I/O to unbuffered mode for the specified file and attempts to set the terminal to breakall mode. (No error is signaled if the terminal is already in this mode, or if this mode is not accepted).

USAGE

```
$IMPORT
        'pascal_util_ (pascal) :  breakall_on $

PROCEDURE breakall_on
        (VAR text_file :  text) ; EXTERNAL ;
```

ARGUMENTS

text_file
the text file involved. It must be attached, but may or may not be open.

NOTES

If your terminal was switched to breakall mode from ^breakall, it will be returned to ^breakall when the procedure where the file is declared exits, or when pascal_util_$breakall_off is called for this file.

EXAMPLES

```
breakall_on (input) ;
```

Entry: pascal_util_$breakall_off

This entrypoint resets the given Pascal text file to ^breakall mode if it was put in breakall mode by a previous call to pascal_util_$breakall_on (otherwise it has no effect). It also resets Pascal I/O to buffered mode for this text file, and resets the terminal to breakall mode if it was switched from ^breakall mode to breakall by a previous call to pascal_util_$breakall_on.

USAGE

```
$IMPORT
        'pascal_util_ (pascal) :  breakall_off $

PROCEDURE breakall_off
        (VAR text_file :  text) ; EXTERNAL ;
```

ARGUMENTS

text_file
    the text file involved. It must be attached, but may or may not be open.

EXAMPLES

```
breakall_off (input) ;
```

Name: pathname_

The pathname_ subroutine contains entry points for constructing pathnames and archive component pathnames given a directory name, entry name, and optionally, an archive component name.

When a directory name and an entry name are combined to form a pathname, the result may be longer than 168 characters. If truncating the pathname doesn't matter, e.g. in an error message in a call to com_err_ for another, more important error, then use the pathname_ or pathname_$component entry points. These entry points create an invalid pathname with the characters "PATHNAME TOO LONG" to let the reader know truncation occured. If truncating the pathname matters, use the pathname_$component_check entry point.

Entry: pathname_

The pathname_ entry point, given a directory name and an entry name, returns the pathname of the entry.

USAGE

```
declare pathname_ entry (char (*), char (*)) returns (char (168));

path = pathname_ (dirname, entryname);
```

ARGUMENTS

path
    is the pathname of the entry in the given directory. (Output)

dirname
    is the pathname of the containing directory. (Input)

entryname
    is the entryname of the entry. (Input)

NOTES

If the resulting pathname is longer than 168 characters, then the last 20 characters of the result are set to " <PATHNAME TOO LONG>".

EXAMPLES

```
dirname        entryname      path
-------        ---------      ----
>              a              >a
>a>b           c              >a>b>c
```

This page intentionally left blank.

Entry: pathname_$component

This entry point, given a directory name, an entry name, and optionally, an archive component name, constructs a pathname or an archive component pathname.

USAGE

```
declare pathname_$component entry (char (*), char (*), char (*))
      returns (char (194));

path = pathname_$component (dirname, entryname, component_name);
```

ARGUMENTS

path
: is the pathname of the entry in the given directory, or is an archive component pathname. (Output)

dirname
: is the pathname of the containing directory. (Input)

entryname
: is the entryname of the entry. (Input)

component_name
: is the name of an archive component, or is null. (Input)

NOTES

If component_name is not null, the archive suffix on the entryname is optional, and is assumed if not specified. If component_name is not null and entryname ends with the archive suffix, the suffix is omitted from the returned pathname.

If component_name is null and the resulting pathname is longer than 168 characters, then the last 20 characters of the pathname are set to " <PATHNAME TOO LONG>". If component_name is not null and the resulting archive component pathname is longer than 194 characters, then the last 20 characters of the dirname>entryname portion of the archive pathname are changed to " <PATHNAME TOO LONG>" and the component_name remains in the pathname.

EXAMPLES

```
dirname        entryname      component_name   path
-------        ---------      --------------   ----
>              a              ""               >a
>a>b           c              ""               >a>b>c
>a>b           c.archive      ""               >a>b>c.archive
>              a.archive      b                >a::b
>a>b           c              d                >a>b>c::d
>a>b           c.archive      d                >a>b>c::d
```

Entry: pathname_$component_check

This entry point is the same as pathname_$component except a status code indicates truncation instead of an invalid pathname containing "PATHNAME TOO LONG".

USAGE

```
declare pathname_$component_check entry (char (*), char (*), char (*),
      char (*), fixed binary (35));

call pathname_$component_check (dirname, entryname, component_name,
      path, code);
```

ARGUMENTS

dirname
    is the pathname of the containing directory. (Input)

entryname
    is the entryname of the entry. (Input)

component_name
    is the name of an archive component, or is null. (Input)

path
    is the pathname of the entry in the given directory, or is an archive component pathname. (Output)

code
    is a standard status code. (Output) It can be:
    error_table_$pathlong
        the pathname was truncated.

NOTES

If component_name is not null, the archive suffix on the entryname is optional, and is assumed if not specified. If component_name is not null and entryname ends with the archive suffix, the suffix is omitted from the returned pathname.

Name: phcs_$read_disk_label

This entry point is used to read the label of a storage system disk drive. The label is described by the structure "label," in the include file fs_vol_label.incl.pl1.

USAGE

```
dcl phcs_$read_disk_label entry (bit (36) aligned, pointer,
    fixed bin (35));

call phcs_$read_disk_label (pvid, label_ptr, code);
```

ARGUMENTS

pvid
: is the physical volume id of the disk whose label is to be read. (Input). The physical volume id is used instead of the volume name because this is a ring zero interface, and volume names are not accessible by ring zero; hence, all ring zero interfaces that reference physical volumes use the pvid. A pvname can be converted to a pvid by calling the subroutine mdc_$find_volname or can be returned by a previous call to find_partition_.

label_ptr
: is a pointer to the user-supplied buffer in which to read the label. (Input). The label is 1024 words long and is described in fs_vol_label.incl.pl1.

code
: is a nonstandard status code. (Output). It can be:

    0
    : indicates that the label was successfully read.

    error_table_$pvid_not_found
    : indicates that the specified physical volume is not presently mounted.

    an integer between 1 and 10
    : indicates that a physical disk error occurred while trying to read the label. Error messages for physical disk errors are declared in the include file fsdisk_errors.incl.pl1, in the array fsdisk_error_message.

Name: pl1_io_

The pl1_io_ subroutine is a collection of utility functions for extracting information about PL/I files that is not available within the language itself.

Entry: pl1_io_$error_code

This function returns the last nonzero status code encountered by PL/I I/O while performing file operations. This is a standard Multics status code and describes the most recent error more specifically than the PL/I condition which is raised after an error.

USAGE

```
declare pl1_io_$error_code entry (file) returns (fixed bin(35));

code = pl1_io_$error_code (file_variable);
```

ARGUMENTS

file_variable
    is a PL/I file value. (Input)

code
    is the last nonzero status code associated with the file. (Output)

NOTES

The specific values returned by this function are subject to change. See "Handling Unusual Occurrences" in the Programmer's Reference Manual.

Entry: pl1_io_$get_iocb_ptr

This function returns the I/O control block pointer for the Multics I/O System switch associated with an open PL/I file. This pointer may be used to perform control and modes operations upon the switch associated with that file.

USAGE

```
declare pl1_io_$get_iocb_ptr entry (file) returns (ptr);

iocb_ptr = pl1_io_$get_iocb_ptr (file_variable);
```

ARGUMENTS

file_variable
is a PL/I file value. (Input)

iocb_ptr
is a pointer to the I/O control block for the file. (Output)

NOTES

Performing explicit operations via the Multics I/O System upon switches in use by PL/I I/O is potentially dangerous unless care is taken that certain conventions are observed. No calls should be made that affect the data in the PL/I data set being accessed, the positioning of the data set, or the status or interpretation of any I/O operations that may be in progress. In general, this limits such calls to those which obtain status information.

Name: prepare_mc_restart_

The prepare_mc_restart_ subroutine checks machine conditions for restartability, and makes modifications to the machine conditions (to accomplish user modifications to process execution) before a condition handler returns.

The prepare_mc_restart_ subroutine should be called by a condition handler, which was invoked as a result of a hardware-detected condition, if the handler wishes the process to:

1. retry the faulting instruction.
2. skip the faulting instruction and continue.
3. execute some other instruction instead of the faulting instruction and continue.
4. resume execution at some other location in the same program.

When a condition handler is invoked for a hardware-detected condition, it is passed a pointer to the machine-conditions data at the time of the fault. If the handler returns, the system attempts to restore these machine conditions and restart the process at the point of interruption encoded in the machine-conditions data. After certain conditions, however, the hardware is unable to restart the processor. In other cases, an attempt to restart always causes the same condition to occur again, because the system software has already exhausted all available recovery possibilities (e.g., disk read errors).

Entry: prepare_mc_restart_$replace

This entry point is called to modify machine-conditions data so that the process executes a specified machine instruction, instead of the faulting instruction, and then continues normally.

USAGE

```
declare prepare_mc_restart_$replace entry (ptr, bit(36), fixed bin(35));

call prepare_mc_restart_$replace (mc_ptr, new_ins, code);
```

ARGUMENTS

mc_ptr
is a pointer to the machine conditions. (Input)

new_ins
is the desired substitute machine instruction. (Input)

code
is a standard status code. If it is nonzero on return, the machine conditions cannot be restarted. See "Notes" below. (Output)

Entry: prepare_mc_restart_$retry

This entry point is called to prepare the machine conditions for retry at the point of the hardware-detected condition. For example, this operation is appropriate for a linkage error signal, resulting from the absence of a segment, that the condition handler has been able to locate.

USAGE

```
declare prepare_mc_restart_$retry entry (ptr, fixed bin(35));

call prepare_mc_restart_$retry (mc_ptr, code);
```

ARGUMENTS

mc_ptr
is a pointer to the machine conditions. (Input)

code
is a standard status code. If it is nonzero on return, the machine conditions cannot be restarted. See "Notes" below. (Output)

Entry: prepare_mc_restart_$tra

This entry point is called to modify machine conditions data so that the process resumes execution, taking its next instruction from a specified location. The instruction transferred to must be in the same segment that caused the fault.

USAGE

```
declare prepare_mc_restart_$tra entry (ptr, ptr, fixed bin(35));

call prepare_mc_restart_$tra (mc_ptr, newp, code);
```

ARGUMENTS

mc_ptr
is a pointer to the machine conditions. (Input)

newp
is used in replacing the instruction counter in the machine conditions. (Input)

code
is a standard status code. If it is nonzero on return, the machine conditions cannot be restarted. See "Notes" below. (Output)

NOTES

For all entry points in the prepare_mc_restart_ subroutine, a pointer to the hardware machine conditions is required. The format of the machine conditions is described in the Programmer's Reference Manual.

For all entry points in the prepare_mc_restart_ subroutine, the following codes can be returned:

error_table_$badarg
an invalid mc_ptr was provided.
error_table_$no_restart
the machine conditions cannot be restarted.
error_table_$bad_ptr
the restart location is not accessible.
error_table_$useless_restart
the same error will occur again if restart is attempted.

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.

The print_cobol_error_ entry point displays the error information through the user_output I/O switch.

USAGE IN COBOL

```
call "print_cobol_error_".
```

Entry: print_cobol_error_$switch

This entry point outputs the error information to a specified I/O switch.

USAGE IN COBOL

```
01 switch-name pic x(32).

call "print_cobol_error_$switch" using switch-name.
```

ARGUMENTS

switch-name
: is the name of an I/O switch that is open for output. (Input) 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.

Name: print_data_

Entry: print_data_$print_data_

This entry point formats and prints the output of a PL/I put data statement. The output switch for printing may be specified, as well as various formatting options.

USAGE

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

call print_data_ (put_data_string, print_data_info_ptr, code);
```

ARGUMENTS

put_data_string
: is the output of a PL/I put data statement. Usually obtained as follows: put data (xxx) string (put_data_string), where xxx is a structure whose values are to be formatted and printed. (Input)

print_data_info_ptr
: is a pointer to a structure which describes the formatting options to be used for printing the input. See Notes below. (Input)

code
: is a system status code. (Output)

Entry: print_data_$rs

This entry point formats the output of a PL/I put data statement. The result is returned in a string. Various formatting options may be specified.

USAGE

```
declare print_data_$rs entry (char (*) varying, ptr, char (*) varying,
     fixed bin (35));

call print_data_$rs (put_data_string, print_data_info_ptr,
     return_string, code);
```

ARGUMENTS

put_data_string
: is the output of a PL/I put data statement. Usually obtained as follows: put data (xxx) string (put_data_string), where xxx is a structure whose values are to be formatted and printed. (Input)

print_data_info_ptr
: is a pointer to a structure which describes the formatting options to be used for printing the input. See Notes below. (Input)

return_string
: is a string in which the output is returned. (Input/Output)

code
is a system status code. (Output)

NOTES

The include file pointed to by print_data_info_ptr is declared in print_data_info.incl.pl1 as follows:

```
dcl print_data_info_version_1 fixed bin options (constant) init (1)
                                          internal static;

dcl      print_data_info_ptr          ptr;
dcl      1 print_data_info            based (print_data_info_ptr),
           2 version                  fixed bin,
           2 indentation              fixed bin,
           2 value_column             fixed bin,
           2 output_switch            ptr,
           2 flags,
             3 octal                  bit (1) unal,
             3 hex                    bit (1) unal,
             3 pad                    bit (34) unaligned,
           2 intervals                char (256) varying;
```

STRUCTURE ELEMENTS

version
is the version of this structure. It should be set to print_data_info_version_1.

indentation
is the number of spaces by which structure level names are indented.

value_column
is the column in which the printing of values begins. The structure names are indented, but the values all begin in the same column, so this value should allow a reasonable amount of space for structure names so they don't overlap the values column.

output_switch
is the output switch to use. This is ignored for the rs entry. If it is null then user_output is used.

octal
specifies that bit string values should be converted to octal. This is incompatible with the hex flag. The bit string value must be an integral multiple of 3 bits long in order to be converted, otherwise it is not converted.

hex
specifies that bit string values should be converted to hexadecimal. This is incompatible with the octal flag. The bit string value must be an integral multiple of 4 bits long in order to be converted, otherwise it is not converted.

intervals
is not currently supported and must be set to the null string ("").

Name: qedx_

The qedx subroutine provides a subroutine interface to the Multics qedx Editor for use by subsystems wishing to edit arbitrary strings of ASCII text.

USAGE

```
dcl  qedx_ entry (ptr, fixed bin (35));

call qedx_ (qedx_info_ptr, code);
```

This page intentionally left blank.

ARGUMENTS

qedx_info_ptr
: is a pointer to the qedx_info structure which defines the buffers initially available in qedx_ along with other options. See "The qedx_info structure" below. (Input)

code
: is a standard system status code. See "List of status codes" below. (Output)

NOTES

The caller of qedx_ does not need to print an error message when a non-zero status code is returned by the subroutine. Any appropriate error messages will have already been printed by qedx_ itself. The returned code is only intended to inform the caller of conditions requiring further attention.

LIST OF STATUS CODES

0
: editing completed successfully.

error_table_$unimplemented_version
: qedx_ does not recognize the version of the qedx_info structure supplied by the caller.

error_table_$fatal_error
: an error occured during initialization of qedx_ which prevented the user from performing any editing. The caller of qedx_ should abort its execution.

error_table_$recoverable_error
: one of several non-fatal conditions were detected upon exit from qedx_. The exact condition is reflected to the caller in the qedx_info structure (see below). The caller of qedx_ must decide how to proceed after each of the possible conditions (e.g.: the program may decide not to update the permanent copy of the data being edited if the user exited via quit-force (qf)).

NOTES ON INITIAL BUFFERS

The qedx_info structure defines the initial environment to be presented to the user by qedx_. This environment includes an initial set of buffers along with their contents and default pathnames. The contents of these buffers can be read or written from the storage system from regions supplied by the caller (e.g.: the message in send_mail), or by using a caller supplied procedure (e.g.: to read/write abbreviation definitions). The caller can also request that the initial contents of one or more of these buffers be executed as qedx requests before reading the first request line from the user.

qedx_ always creates a buffer named "0" which it makes the current buffer before executing any requests. If the initial buffers marked for execution do not use the buffer (b) request to change the default buffer, buffer "0" will remain the current buffer when the first request line is read from the terminal.

Each initial buffer must have a default pathname. As part of initialization, qedx_ will read the contents of the object specified by this default pathname into the buffer. If the buffer is read and written from the storage system, the default pathname must identify an existing segment or archive component. If the buffer is read and written from a caller supplied region, the default pathname may be omitted but is normally used as comment to describe the contents of the buffer (e.g.: "<send_mail message>") as the data is read directly from the caller's region. If the buffer is read and written by a caller supplied procedure, the default pathname must identify an existing object (e.g.: abbreviation definition) as defined by that procedure.

For each initial buffer, the caller can specify whether or not the default pathname of the buffer is locked. If the default pathname is locked, use of the read (r) and write (w) requests with a pathname will never change the default pathname of the buffer nor cause qedx_ to consider the default pathname untrustworthy. (See "Notes on default pathnames" in the description of the qedx command in the Commands manual).

With a locked default pathname, use of the read and write requests without a pathname will always read/write the original segment, region, or whatever (when using the caller's I/O module) specified by the caller of qedx_. In this case, use of the read request with a pathname will simply insert the contents of a segment into the buffer and use of the write request with a pathname will simply make a copy of the buffer in a segment for later use.

Locking the default pathname is useful in cases where it would be difficult (if not impossible) for the user to reconstruct the default pathname. For example, in send_mail, buffer "0" contains the message being created. The default pathname in this case identifies the region supplied by send_mail and there is no mechanism by which the user can explicitly specify this default by a pathname. Therefore, send_mail locks the default pathname to insure that the write (w) request without a pathname will always update send_mail's copy of the message.

For each initial buffer which is being read and written from a caller supplied region, the caller can request that qedx_ automatically write the contents of the buffer into the region upon exit. If the user exits qedx_ via the quit-force (qf) request, however, the automatic write will be suppressed. If, when writing a buffer to the caller's region, the buffer is too long to fit in that region, qedx_ will issue a warning to the user and the buffer will be marked as truncated. While still in qedx_, the user can make any necessary changes to the buffer to shorten it sufficiently to fit within the caller's region. If, on exit from qedx_, there are truncated buffers, the user will be asked for permission to exit and actually truncate those buffers. Once again, this query is suppressed if the quit-force request is used.

The qedx_info Structure

The qedx_info structure and the named constants referenced below are defined in the include file qedx_info.incl.pl1:

```
dcl 1 qedx_info               aligned based (qedx_info_ptr),
      2 header,
        3 version             char (8),
        3 editor_name         char (72) unaligned,
        3 buffer_io           entry (pointer, fixed binary(35)),
        3 flags,
          4 no_rw_path        bit(1) unaligned,
          4 query_if_modified bit(1) unaligned,
          4 caller_does_io    bit(1) unaligned,
          4 quit_forced       bit(1) unaligned,
          4 buffers_truncated bit(1) unaligned,
          4 pad               bit(29) unaligned,
        3 n_buffers           fixed binary,
      2 buffers (qedx_info_n_buffers refer (qedx_info.n_buffers)),
        3 buffer_name         char (16) unaligned,
        3 buffer_pathname     char (256) unaligned,
        3 region_ptr          pointer,
        3 region_max_lth      fixed binary(21),
        3 region_initial_lth  fixed binary(21),
        3 region_final_lth    fixed binary(21),
        3 flags,
          4 read_write_region bit(1) unaligned,
          4 locked_pathname   bit(1) unaligned,
          4 execute_buffer    bit(1) unaligned,
          4 default_read_ok   bit(1) unaligned,
          4 default_write_ok  bit(1) unaligned,
          4 auto_write        bit(1) unaligned,
          4 truncated         bit(1) unaligned,
          4 pad               bit(29) unaligned;
```

STRUCTURE ELEMENTS

version
identifies the version of the qedx_info structure supplied by the caller. It must have the value of the named constant QEDX_INFO_VERSION_1. (Input)

editor_name
is the name to be used by qedx_ in error messages and queries (e.g.: "send_mail (qedx)"). (Input)

buffer_io
is only used if flags.caller_does_io is set and is the procedure to be invoked by qedx_ to read/write buffers. See "Notes on buffer I/O" below. (Input)

flags.no_rw_path
specifies whether any read (r) or write (w) request within qedx_ can ever be given an explicit pathname. (Input)

flags.query_if_modified
specifies whether qedx_ should query when the quit (q) request is issued and there are buffers which have been modified since they were last written. Initial buffers with the buffers.auto_write flag set are not considered as modified as they are always written before exit. (Input)

flags.caller_does_io
specifies whether qedx_ should call the buffer_io procedure above or perform I/O itself when reading/writing buffers. (Input)

flags.quit_forced
is set by qedx_ to "1"b to indicate that the user either used the quit-force (qf) request or answered "yes" to the modified buffers query in order to exit; it is set to "0"b to indicate that the user used the quit (q) request and there were no modified buffers present. (Output)

flags.buffers_truncated
is set by qedx_ to "1"b to indicate that the final contents of one or more initial buffers were truncated on exit from qedx_. The buffers which were truncated are marked by the buffers.truncated flag. (Output)

n_buffers
is the number of initial buffers defined below. (Input)

buffers
defines the initial buffers available within this invocation of qedx_. See "Notes on Initial Buffers" above.

buffers.buffer_name
is the name of this buffer. (Input)

buffers.buffer_pathname
is the initial default pathname for this buffer. (Input)

buffers.region_ptr
is a pointer to the region where qedx_ will read and write this buffer if buffers.read_write_region is set. (Input)

buffers.region_max_lth
is the maximum number of characters which can be written into the above region if buffers.read_write_region is set. (Input)

buffers.region_initial_lth
is the number of characters present in the caller's region on entry to qedx_ if buffers.read_write_region is set. qedx_ will automatically read the specified characters into the buffer. (Input)

buffers.region_final_lth
is set by qedx_ to the number of characters written into the caller's region upon exit from qedx_ if buffers.read_write_region is set. This value will be larger than buffers.region_max_lth if buffers.truncated is set by qedx_. (Output)

buffers.read_write_region
specifies that qedx_ will use the caller's region to read/write the contents of this buffer until the user changes the default pathname. Use of this flag is incompatible with flags.caller_does_io. (Input)

buffers.locked_pathname
specifies that the default pathname of this buffer is locked and can not be changed by read (r) or write (w) requests. (Input)

buffers.execute_buffer
specifies that the contents of this buffer should be executed as qedx requests before reading requests from the user. (Input)

buffers.default_read_ok
specifies that the read (r) request can be given without a pathname to read the current contents of the caller's region. This flag is ignored if flags.read_write_region is not set or the default pathname is not the caller's region. (Input)

buffers.default_write_ok
specifies that the write (w) request can be given without a pathname to write the buffer to the caller's region. This flag is ignored if flags.read_write_region is not set or the default pathname is not the caller's region. (Input)

buffers.auto_write
: specifies that the contents of this buffer will be written to the caller's region on exit from qedx_ unless the user uses the quit-force (qf) request or answers "yes" to the query to exit with modified buffers. (Input)

buffers.truncated
: is set by qedx_ to "1"b if the entire contents of the buffer could not be written to the caller's region on exit from qedx_. (Output)

NOTES ON BUFFER I/O

If flags.caller_does_io is set, qedx_ will invoke the caller supplied buffer_io procedure in order to read and write the contents of any buffer. qedx_ determines the pathname to which the buffer is to be read or written; the interpretation of this pathname is the responsibility of the caller's buffer_io procedure (e.g.: the procedure can use the pathname as the name of an abbreviation whose definition is to be read/written).

For a read (r) request, qedx_ supplies an I/O region into which the buffer_io procedure should place the text copied from the object designated by the pathname; qedx_ will then insert this text into its proper place in the buffer. For a write (w) request, qedx_ copies the text from the buffer into an I/O region; the buffer_io procedure should then place this text into the object designated by the pathname.

The buffer_io Procedure

qedx_ invokes the buffer_io procedure as follows --

```
declare buffer_io entry (ptr, bit(1) aligned);

call buffer_io (qedx_buffer_io_info_ptr, success);
```

where:

qedx_buffer_io_info_ptr
: is a pointer to the qedx_buffer_io_info structure describing the read/write operation to be undertaken. (Input)

success
: is set by the buffer_io procedure to "1"b if the operation was successful and to "0"b if it failed. (Output)

Note: It is the responsibilty of the buffer_io procedure to print any appropriate error messages if the operation does not succeed.

The qedx_buffer_io_info Structure

The qedx_buffer_io_info structure and the named constants referenced below are defined in the include file qedx_buffer_io_info.incl.pl1:

```
dcl 1 qedx_buffer_io_info       aligned based (qbii_ptr),
      2 version                 char (8),
      2 editor_name             char (72),
      2 pathname                char (256) unaligned,
      2 buffer_ptr              pointer,
      2 buffer_max_lth          fixed binary (21),
      2 buffer_lth              fixed binary (21),
      2 direction               fixed binary,
      2 flags,
        3 default_pathname      bit (1) unaligned,
        3 pad                   bit (35) unaligned;
```

STRUCTURE ELEMENTS

version
: identifies the version of the qedx_buffer_io_info structure supplied by qedx_. This version of the structure is given by the named constant QEDX_BUFFER_IO_INFO_VERSION_1. (Output)

editor_name
: is the name of the editor to be used by the buffer_io procedure in any error messages and queries. (Input)

pathname
: is the pathname to be read/written as determined by qedx_. (Input)

buffer_ptr
: is a pointer to the I/O buffer allocated by qedx_. When reading from the pathname, the buffer_io procedure must place the text into this buffer; when writing to the pathname, the buffer_io procedure must take the text from this buffer. (Input)

buffer_max_lth
: is the maximum size of the I/O buffer. This value is only used when reading from the pathname and specifies a limit on the amount of text which can be returned by the buffer_io procedure. (Input)

buffer_lth
: is the length of the text read/written from the pathname. When reading from the pathname, the buffer_io procedure must set this value to the number of characters read from the pathname and placed in the I/O buffer. (Output) When writing to the pathname, this value is set by qedx_ to the number of characters to be written into the pathname. (Input)

direction
specifies the operation to be undertaken. If it has the value of the named constant QEDX_READ_FILE, the text is to be read from the pathname and placed into the I/O buffer. If it has the value of the named constant QEDX_WRITE_FILE, the text is to be written from the I/O buffer into the pathname. (Input)

flags.default_pathname
is "1"b if the pathname supplied above by qedx_ is the default pathname of the buffer being read/written. (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_$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 Von Neumann method for generating an exponentially 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);
```

ARGUMENTS

seed
  is the optional seed (see the random_$uniform entry point). (Input/Output)

random_no
  is the random number that is generated. (Output)

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);
```

ARGUMENTS

seed
is the optional seed (see the random_$uniform entry point). (Input/Output)

array (N)
is the array of generated random numbers where N is greater than or equal to array_size. (Output)

array_size
is the number of values returned in the array. (Input)

Entry: random_$get_seed

The random_$get_seed entry point is used to obtain the current value of the internal seed (see "Notes" below).

USAGE

```
declare random_$get_seed entry (fixed bin(35));

call random_$get_seed (seed_value);
```

ARGUMENTS

seed_value
is the current value of the internal seed. (Output)

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);
```

ARGUMENTS

seed
: is the optional seed (see the random_$uniform entry point). (Input/Output)

random_no
: is the random number that is generated. (Output)

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);
```

ARGUMENTS

seed
: is the optional seed (see the random_$uniform entry point). (Input/Output)

random_ant
: is the random number that is generated. (Output)

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);
```

ARGUMENTS

seed
: is the optional seed (see the random_$uniform entry point). (Input/Output)

ant_array (N)
: is the array of generated random numbers where N is greater than or equal to array_size. (Output)

array_size
: is the number of values returned in ant_array. (Input)

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);
```

ARGUMENTS

seed
is the optional seed (see the random_$uniform entry point). (Input/Output)

array (N)
is an array of the generated random numbers where N is greater than or equal to array_size. (Output)

array_size
specifies the number of random variables to be returned in array. (Input)

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" below).

USAGE

```
declare random_$set_seed entry (fixed bin(35));

call random_$set_seed (seed_value);
```

ARGUMENTS

seed_value
is the value to which the internal seed is set. (Input) This value must be a nonzero positive integer.

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);
```

ARGUMENTS

seed
is the optional seed (see "Notes"). (Input/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.

random_no
is the random number that is generated. (Output)

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);
```

ARGUMENTS

seed
is the optional seed (see the random_$uniform entry point). (Input/Output)

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);
```

ARGUMENTS

seed
: is the optional seed (see the random_$uniform entry point). (Input/Output)

ant_array
: is the array of generated random numbers where N is greater than or equal to array_size. (Output)

array_size
: is the number of values returned in ant_array. (Input)

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);
```

ARGUMENTS

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.

array (N)
: is an array of the generated random numbers where N is greater than or equal to array_size. (Output)

array_size
: specifies the number of random variables to be returned in array. (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 4084114312. 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 zeros in positions m+a through n.)

5. Exclusive-or register B into register A and set the sign bit of register A to zero. (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: rcp_

The rcp_ subroutine is used to manage resources in the user's process. There are entrypoints to attach, detach, list, and get the status of resources owned by this process or by the system.

Attaching a resource is a two step process, consisting of a call to rcp_$attach, and 1 or more calls to rcp_$check_attach. The example which follows will demonstrate this process, as well as showing how to release the resource back to the system.

```
/**** This example will show how to attach a tape drive to this process,
handling any errors that rcp_ may return.  The procedure calling rcp_
should use the following include files: rcp_resource_types.incl.pl1,
rcp_device_info_structs.incl.pl1, event_wait_channel.incl.pl1, and
event_wait_info.incl.pl1. */

/* We are going to attach a tape drive, requesting to have a specific volume
mounted on it. */

     rcp_id = "0"b;
     event_channel = 0;
     tape_info_ptr = null ();
     on cleanup call CLEANUP_ATTACH();

     allocate tape_info set (tape_info_ptr);       /* allocate device_info */
     tape_info.version_num = tape_info_version_3;  /* fill in structure */
     tape_info.system_flag = "0"b;                 /* not a system process */
     tape_info.device_name = "";         /* not asking for specific device */
     tape_info.model = 0;                          /* same */
     tape_info.write_flag = "1"b;                  /* we are going to write */
     tape_info.speed = 125;                        /* speed of drive */
     tape_info.density = "00001"b;                 /* 6250 bpi */
     tape_info.volume_name = "tape01";             /* the tape we want */
/* Now we need an event channel for rcp_ to tell us when the attachment has
been completed. */
     call ipc_$create_ev_chn (event_channel, code);
     if code ^= 0 then goto ERROR;

/* Now that we have the structure initialized, begin the attachment. */

     call rcp_$attach (DEVICE_TYPE (TAPE_DRIVE_TYPEX), tape_info_ptr,
                       event_channel, "", rcp_id, code);
     if code ^= 0 then goto ERROR;

/* Now we have to wait for the drive to be allocated to us, and the tape to be
mounted. */

     event_wait_channel.channel_id (1) = event_channel;
```

```
ATTACH_LOOP:
     rcp_comment = "";
     call rcp_$check_attach (rcp_id, tape_info_ptr, rcp_comment, ioi_id,
                             ws_max, to_max, rcp_state, code);
     if rcp_comment ^= "" then
       {the program should print the rcp_comment on the user's terminal.};
     goto ATTACH_STATE (rcp_state);

/**** The attachment has been completed. */
ATTACH_STATE (0):
     goto ATTACH_COMPLETE;

/**** The resource has been attached, but we cannot use it yet. This can happen
      in the case of a tape or disk volume having to be mounted, etc. In this
      case, we block on an event channel and rcp_ will tell us when the
      resource is ready for use. */
ATTACH_STATE (1):
     call ipc_$block (addr (event_wait_channel), addr (event_wait_info), code);
     if code ^= 0 then goto ERROR;
     else goto ATTACH_LOOP;

/**** There is no appropriate resource available. */
ATTACH_STATE (2):
     code = error_table_$resource_unavailable;
     goto ERROR;

/**** An error has been encountered while processing the attachment. */
ATTACH_STATE (3):
     goto ERROR;

ERROR:
     call CLEANUP_ATTACH();
     {the appropriate message should be printed based upon the value of code,
      and this procedure should return.}

ATTACH_COMPLETE:

/* At this point, we can begin using the device through ioi_. */
     ...
     ...
/* Now that we have finished using the device, we must detach it. */

     call rcp_$detach (rcp_id, "0"b, error_count, "", code);
     if code ^= 0 then goto ERROR;
     free tape_info;
     call ipc_$delete_ev_chn (event_channel, code);
     return;
```

```
CLEANUP_ATTACH: procedure;

     if rcp_id ^= ""b then call rcp_$detach (rcp_id, "0"b, (0), "", (0));
     if tape_info_ptr ^= null () then free tape_info;
     if event_channel ^= 0 then call ipc_$delete_ev_chn (event_channel, (0));

end CLEANUP_ATTACH;
```

Entry: rcp_$attach

This entry point initiates the attachment of a device. The device that RCP will attempt to attach depends on the values found in the user supplied device_info structure. RCP will first see if an appropriate device is assigned to this process. If there is an appropriate assigned, unattached device, then RCP will use that device for this attachment. If there is not, RCP will attempt to assign an appropriate, available and accessible device to the process and use that device for the attachment. If no device can be found that meets the requirements, then the attachment is aborted. For tape and disk drives, if the specified volume is not mounted, RCP will mount the volume on the device.

This entry point functions in cooperation with the rcp_$check_attach entry point. A call to rcp_$attach initiates the attachment but does not complete it. The caller still cannot successfully call IOI to perform I/O on the device being attached. The attachment will not be completed and the caller will not know the characteristics of the device that was attached until this data is returned from rcp_$check_attach.

USAGE

```
declare rcp_$attach entry (char(*), ptr, fixed bin(71), char(*),
      bit(36) aligned, fixed bin(35));

call rcp_$attach (device_type, device_info_ptr, event_id, comment,
      rcp_id, code);
```

ARGUMENTS

device_type
: is a string that identifies the type of device to attach. It must be one of the constants defined in rcp_resource_types.incl.pl1. (Input)

device_info_ptr
: points to a structure supplied by the caller containing information about the device to be attached. (See below for information about the device_info structure.) (Input)

event_id
: is an event channel ID supplied by the caller. This channel will be used by RCP to notify the user of the progress of the attachment in some cases. For more information, see the example above. (Input)

comment
: is a message to be displayed to the operator upon completion of device assignment and prior to any volume mounting that may be required. (Input)

rcp_id
: is an internally generated unique id for this rcp attachment. (Output) It is passed to other rcp_ entrypoints that manipulate the attachment.

error_code
: is a standard system status code. (Output) Possible codes returned are:

  error_table_$bad_volid
  : for a tape or disk device, a volume must be specified and it was not.

  error_table_$resource_attached
  : the requested device is already attached to the requesting process.

  error_table_$no_operation
  : this is a T&D operation and the privileged attach entry point rcp_priv_$attach must be used instead.

  error_table_$resource_unknown
  : the requested device is not known to the system.

  error_table_$resource_unavailable
  : the requested device was accessible but not available.

  error_table_$resource_bad_access
  : the requested device was unaccessible.

ACCESS REQUIRED

RW access to the Access Control Segment (ACS) associated with the resource is required in order to attach the resource. If RCPRM is not enabled at your site, then the only resource controlled by RCP is the device.and access control is not provided for tape and disk volumes. The ACS is located in >sc1>rcp>RESOURCE_NAME.acs.

If RCPRM is enabled, then access to both devices and volumes is controlled by ACS segments. For reading, the user must have R effective access, and for writing, RW effective access. This access may be derived from an ACS segment associated with the resource, or based on the owner of the resource, as determined by RCPRM. Refer to the Reference Manual for further details.

NOTES

The device_info structure is a general purpose header structure used for the various of types of resources. It is declared is rcp_device_info_structs.incl.pl1, along with the structures for tapes, disks, and printers. Each structure uses device_info as its header. The include file decribes the structures for each resource in more detail.

```
dcl     1 device_info based (device_info_ptr) aligned,
          2 version_num       fixed bin,
          2 usage_time        fixed bin,
          2 wait_time         fixed bin,
          2 system_flag       bit(1),
          2 device_name       char (8),
          2 model             fixed bin,
          2 qualifiers(4)     fixed bin(35);
```

STRUCTURE ELEMENTS

version_num
: is the version number of this structure. This will be different for each resource type. (Input)

usage_time
: number of minutes device will/may be used. (Reserved for future use.)

wait_time
: number of minutes user will/must wait for assignment completion. (Reserved for future use.)

system_flag
: if this is "1"b, the user wants to be considered a system process for this assignment. (Input)

device_name
: the name of the device. (Input to rcp_$attach/Output from rcp_$check_attach)

model
: the model number of the device. (Output from rcp_$check_attach)

qualifiers
: this element will contain different information for each resource type. (Input to rcp_$attach/Output from rcp_$check_attach)

Entry: rcp_$check_attach

This entry point establishes completion of the attach process begun by the rcp_$attach entry point, causes IOI to set the workspace and timeout limits for the device, promotes the device to the caller's validation level, and returns info needed by the user to perform I/O on this device. It should be noted that an attachment is not complete until this entry point is called.

USAGE

```
declare rcp_$check_attach entry (bit(36) aligned, ptr, char(*),
     fixed bin, fixed bin(19), fixed bin(71), fixed bin,
     fixed bin(35));

call rcp_$check_attach (rcp_id, device_info_ptr, comment, ioi_index,
     workspace_max, timeout_max, statex, code);
```

ARGUMENTS

rcp_id
: is the unique identified for this attachment returned by rcp_$attach. (Input)

device_info_ptr
: is a pointer to the device_info structure that was supplied to rcp_$attach when this attachment was begun. (Input) This entrypoint will update the information in this structure to reflect the characteristics of the actual device that was acquired.

comment
: the comment associated with this attachment. This argument is always a null string. (Output)

ioi_index
: is an index used for communication with IOI. (Output)

workspace_max
: is the size of IOI workspace in words. (Output)

timeout_max
: is the amount of time IOI will wait for another operation to begin, after an operation completes, before it unwires the IOI workspace. If the next operation begins before this time out, then the workspace remains wired. Otherwise, it gets unwired automatically and the next operation is delayed while IOI rewires the workspace pages into memory.

statex
: is the current state of the attachment. (Output) Its possible values are:
  0 the attachment is complete
  1 the attachment is nearly complete
  2 the resource is unavailable
  3 an error occurred while attaching the resource

code
is a standard system status code. (Output) Possible returned codes are:

error_table_$bad_arg
the rcp_id supplied is invalid.

error_table_$invalid_state
the attachment of this device is in the wrong state to be completed now.

ACCESS REQUIRED

Only the process that began the attachment with rcp_$attach can complete it with rcp_$check_attach, but no access is required for this entrypoint, as all access checking is performed by rcp_$attach.

Entry: rcp_$detach

This entry point detaches an IOI device attachment. Depending on the disposition, the device will also be unassigned.

USAGE

```
declare rcp_$detach entry (bit(36) aligned, bit(*), fixed bin, char(*),
     fixed bin(35));

call rcp_$detach (rcp_id, disposition, error_count, comment, code);
```

ARGUMENTS

rcp_id
unique rcp identification number that identifies the attachment of the device. (Input)

disposition
specifies whether the device should be unassigned or not. (Input) Any volume associated with the device is always unassigned. This argument's possible values are:
"1"b leave the device assigned to this process
"0"b if the device was assigned to this process by the rcp_$attach call that initiated this attachment, then unassign the device; otherwise leave it assigned to this process.

error_count
user ring error count for the attachment indicating the number of errors during the attachment. This count is logged in the system log message for this detachment. (Input)

comment
: a comment to be displayed to the operator upon detachment of the device. (Input)

code
: is a storage system status code. (Output) Possible codes returned are:

  error_table_$bad_arg
  : indicates a possible invalid or incorrect rcp_id.

  error_table_$bad_processid
  : the device was not attached to this process or the rcp_id (which reflects the process id which made the attachment) is invalid or incorrect.

ACCESS REQUIRED

Only the process which attached the device can detach it using this entry point.

Entry: rcp_$get_status

This entry point will find a resource given its name or UID, and return all the information about it depending on the user's access to the resource.

USAGE

```
declare rcp_$get_status entry (ptr, char (*), fixed bin (35));

call rcp_$get_status (resource_desc_ptr, registry_dir, code);
```

ARGUMENTS

resource_desc_ptr
: is a pointer to the resource_descriptions structure, which is defined in resource_control_desc.incl.pl1. (Input)

registry_dir
: the absolute pathname of the directory containing the RCP registries. (Input) This is usually >system_control_1>rcp.

code
: is a standard system status code. (Output) Possible codes returned are:

  error_table_$action_not_performed
  : the action was not performed and the user does not have enough access to find out why.

  error_table_$bad_resource_spec
  : there was erroneous data in the resource_descriptions structure supplied.

error_table_$resource_awaiting_clear
the resource is awaiting clear and is unavailable for status.

error_table_$not_abs_path
the registry directory pathname supplied is not an absolute pathname.

error_table_$resource_locked
the resource is locked and unavailable.

error_table_$resource_unknown
the requested resource is unknown to the system.

error_table_$unimplemented_version
the version of the resource_descriptions structure supplied is not supported.

error_table_$resource_bad_access
the user does not have enough access to get resource's status.

ACCESS REQUIRED

Read effective access is required to get the status of an RCP object.

NOTES

This entrypoint is only useful when the site has RCPRM enabled.

Entry: rcp_$list_resources

This entry point returns a list of resources owned by a specific user, by the system, or unowned resources. The selection of information to be returned is determined by the userid argument.

USAGE

```
declare rcp_$list_resources entry (char (*), char (*), char (*), ptr,
      fixed bin (35), ptr, fixed bin (35));

call rcp_$list_resources (resource_type, registry_dir, userid,
      user_area_ptr, n_resources, return_ptr, code);
```

ARGUMENTS

resource_type
the resource type, i.e., "tape_vol". (Input)

registry_dir
the absolute pathname of the directory where the RCP registries are located. (Input) This is usually >system_control_1>rcp.

userid
contains the selection criteria for information to be returned. (Input) Its possible values are:

Person.Project
return information about the resources owned by the specified user.

system
return information about the resources owned by the system.

free
return information about the resources in the free pool.

user_area_ptr
pointer to the area where the resource_list structure should be allocated. See "Notes" below for description of the resource_list structure. (Input).

n_resources
number of resources in the resource_list structure returned to the user. (Output)

return_ptr
is a pointer to the allocated structure in the user-supplied area. (Output)

code
is a standard system status code. (Output) Possible codes returned are:

error_table_$insufficient_access
the user does not have enough access to find out the desired information. See "Access Required" below.

error_table_$bad_name
the userid supplied was Person.* and this is not allowed.

error_table_$smallarg
the user-supplied area is too small for the information to be returned.

ACCESS REQUIRED

R effective access is required to list the existence of a resource. This computation takes into account ONLY the AIM range of the resource since R raw mode is not necessary to list the existence of a resource, but read_allowed_ is required.

NOTES

This entrypoint is only useful when the site has RCPRM enabled.

The resource_list structure is defined in resource_list.incl.pl1 and is declared as follows:

```
dcl     1  resource_list aligned based (resource_list_ptr),
        2  forward_ptr pointer initial (null),
        2  max_entries fixed bin,
        2  n_resources fixed bin initial (0),
        2  resource_name
           (Max_entries refer (resource_list.max_entries)) char (32);
```

STRUCTURE ELEMENTS

forward_ptr
    points to the next block, null if there is no next block.

max_entries
    number of elements in the resource_name array.

n_resources
    number of valid resource names in this block.

resource_names
    array of resource names that meet the specified criteria.

Name: read_allowed_

The read_allowed_ function determines whether a subject of specified authorization has access (with respect to the access isolation mechanism) to read an object of specified access class. For information on access classes, see the Programmer's Reference Manual.

USAGE

```
declare read_allowed_ entry (bit(72) aligned, bit(72) aligned) returns
      (bit(1) aligned);

returned_bit = read_allowed_ (authorization, access_class);
```

ARGUMENTS

authorization
    is the authorization of the subject. (Input)

access_class
    is the access class of the object. (Input)

returned_bit
indicates whether the subject is allowed to read the object. (Output)
"1"b read is allowed.
"0"b read is not allowed.

Name: read_password_

The read_password_ subroutine reads a single line from the users' terminal (actually from the user_input I/O switch). It attempts to hide the input line by turning the printing mechanism off before reading and turning it back on afterwards. If the printing mechanism cannot be turned off, then a mask consisting of several layers of printing designed to "black out" the page is printed. One of the layers of printing is pseudo-randomly generated so that it will be different each time the subroutine is called, thus making it difficult to analyze the layers of overprinting. The mask is 12 characters long.

USAGE

```
declare read_password_ entry (char(*), char(*));

call read_password_ (prompt, password);
```

This page intentionally left blank.

ARGUMENTS

prompt
is a message to be printed before the password is read. It can be any length. A newline character is always printed after the prompting message. (Input)

password
is the password that the user typed. It can be up to 120 characters long. (Output)

NOTES

The password is processed as follows: Tab characters are translated to blanks. Leading blanks are removed. Characters after any embedded blanks are removed. If the resulting password is all blank, a single asterisk ("*") is returned, otherwise the password is returned.

Entry: read_password_$switch

This entry is similar to read_password_, but it allows the caller to specify the I/O switches to be used to print the prompt and read the password.

USAGE

```
declare read_password_$switch entry (ptr, ptr, char(*), char(*),
      fixed bin(35));

call read_password_$switch (output_switch, input_switch, prompt,
      password, code);
```

ARGUMENTS

output_switch
is a pointer to the I/O switch on which the prompt, and if necessary the password mask, is printed. (Input)

input_switch
is a pointer to the I/O switch from which the password is read. (Input)

prompt
is a message to be printed before the password is read. It can be any length. A newline character is always printed after the prompting message. (Input)

password
is the password that the user typed. It can be up to 120 characters long. (Output)

code
: is a standard system status code which is non-zero only if a password could not be read. (Output)

NOTES

The password is processed as follows: Tab characters are translated to blanks. Leading blanks are removed. Characters after any embedded blanks are removed. If the resulting password is all blank, a single asterisk ("*") is returned; otherwise the password is returned.

Name: read_write_allowed_

The read_write_allowed_ function determines whether a subject of specified authorization can read, append, modify, and destroy data in an object of specified access class. For information on access class, see the *Multic Programmer's Reference Manual*, Order No. AG91.

USAGE

```
declare read_write_allowed_ entry (bit(72) aligned, bit(72) aligned)
     returns (bit(1) aligned);

returned_bit = read_write_allowed_ (authorization, access_class);
```

ARGUMENTS

authorization
: is the authorization of the subject. (Input)

access_class
: is the access class of the object. (Input)

returned_bit
: indicates whether the subject is allowed to both read and write the object. (Output)
"1"b read and write are allowed.
"0"b read and write are not allowed.

Name: rehash_

This subroutine rehashes (reformats into a different size) a hash table of the form that is maintained by the hash_ subroutine. In most cases, hash_ calls rehash_ automatically when a table becomes too full. For hash tables that are embedded in larger data bases, the data base maintainer must monitor the density of the hash table and call rehash_ when necessary to maintain the optimal table size. See the description of the hash_ subroutine for more information.

USAGE

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

call rehash_ (table_ptr, size, code);
```

ARGUMENTS

table_ptr
    is a pointer to the table to be rehashed. (Input)

size
    is the new size of the hash table. (Input). See the description of hash_$opt_size.

code
    is a standard status code. (Output). It can be:
    0
        table rehashed successfully.
    error_table_$invalid_elsize
        size is too large.
    error_table_$full_hashtbl
        size is not large enough to hold all the entries in the current hash table.

Name: release_area_

The release_area_ subroutine cleans up an area after it is no longer needed. If the area is a segment acquired via the define_area_ subroutine, the segment is released to the free pool via the temporary segment manager. If the area was not acquired (only initialized) via the define_area_ subroutine then the area itself is reinitialized to the empty state. In certain cases when the area is defined by the system or when the area is extended in ring 0, the temporary segment manager is not used and the area segments are actually created and deleted. Segments acquired to extend the area are released to the free pool of temporary segments or deleted if they are not obtained from the temporary segment manager.

USAGE

```
declare release_area_ entry (ptr);

call release_area_ (area_ptr);
```

ARGUMENTS

area_ptr
points to the area to be released. (Input/Output)

NOTES

The release_area_ subroutine sets area_ptr to null after copying it to a local variable.

Name: release_temp_segment_

The release_temp_segment_ subroutine is used to return a temporary segment (acquired with the get_temp_segment_ or the get_temp_segments_ subroutine) to the free pool of temporary segments associated with the 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_segment_ entry (char(*), ptr, fixed bin(35));

call release_temp_segment_ (program_name, temp_seg_ptr, code);
```

ARGUMENTS

program_name
is the name of the program releasing the temporary segment. (Input)

temp_seg_ptr
is a pointer to the temporary segment being released. (Input/Output)

code
is a standard status code. (Output)

NOTES

A nonzero status code is returned if the segment being released was not assigned to the given program. See the description of the get_temp_segment_ or the get_temp_segments_ subroutine for a description of how to acquire a temporary segment.

The pointer in the temp_seg_ptr variable above is set to the null value after the segment is successfully returned to the free pool. This fact can be used by callers to determine if a given temporary segment has been released.

A null input value for the temp_seg_ptr variable is not treated as an error. No action is performed.

Name: release_temp_segments_

The release_temp_segments_ subroutine is used to return temporary segments (acquired with the get_temp_segment_ or 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);
```

ARGUMENTS

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

ptrs
is an array of pointers to the temporary segments being released. (Input/Output)

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_ or the get_temp_segment_ 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.

Null input values in the ptrs array are not treated as errors. No action is performed for them.

Name: request_id_

Given a Multics standard clock value, this entry point returns a char(19) formatted date (expressed in GMT) in the form "^yc^my^dm^Hd^MH^99.999999UM", e.g. 830718105806.808512 (yymmddHHMMSS.SSSSSS) This is a request id as used by the absentee facility, I/O daemons, and other queue-driven facilities.

USAGE

```
declare request_id_ entry (fixed bin(71)) returns(char(19));

result = request_id_ (clock);
```

ARGUMENTS

clock
: is the clock value to be formatted. (Input)

result
: is the resultant character string. (Output)

Name: requote_string_

The requote_string_ subroutine doubles all quotes within a character string and returns the result enclosed in quotes.

USAGE

```
declare requote_string_ entry (char(*)) returns(char(*));

requoted_string = requote_string_ (string);
```

ARGUMENTS

string
: is the string to be requoted. (Input)

requoted_string
: is the string with all quotes doubled and enclosed in quotes. (Output)

EXAMPLES

```
"""a""" = requote_string_ ("a")

"""a""""b""" = requote_string_ ("a""b")
```

Name: resource_control_

The resource_control_ subroutine provides an interface to the Multics resource control facility. Entry points in this subroutine allow programs to reserve or cancel I/O devices and volumes.

See the Programmer's Reference Manual for a description of the Multics resource control facility.

Entry: resource_control_$cancel_id_string

This entry point cancels the reservation of a resource or group of resources.

USAGE

```
declare resource_control_$cancel_id_string entry (char(*), char(*),
      bit(1) aligned, fixed bin (35));

call resource_control_$cancel_id_string (reservation_id, group_id,
      system, code);
```

ARGUMENTS

reservation_id
: is the character string representation of the reservation identifier to be cancelled. (Input)

group_id
: is the group ID of the user to whom the reservation belongs. (Input). This is only valid if system = "1"b.

system
: specifies, if "1"b, that a privileged cancellation is to be performed (see "Notes" below). (Input)

code
: is a standard status code. (Output)

ACCESS REQUIRED

Execute access to the rcp_sys_ gate is necessary to perform a privileged cancellation.

NOTES

If system = "1"b, then the reservation group is forcibly canceled whether or not it belongs to the current process.

Entry: resource_control_$reserve

This entry point reserves a resource or group of resources for use by a process.

USAGE

```
declare resource_control_$reserve entry (pointer, pointer,
      bit (1) aligned, bit (72) aligned, fixed bin (35)));

call resource_control_$reserve (descriptions_ptr, reservation_desc_ptr,
      authorization, system, code);
```

ARGUMENTS

descriptions_ptr
: is a pointer to the structure containing a description of the resources to be reserved (see "Resource Description" below). (Input)

reservation_desc_ptr
: is a pointer to the structure containing reservation information for the resources to be saved (see "Reservation Description" below). (Input)

authorization
: checks the user's authorization to use the devices or volumes and is only valid if system = "1"b. (Input)

system
: specifies, if "1"b, that the calling process wishes to perform a privileged reservation (see "Notes" below). (Input)

code
: is a standard status code. (Output)

RESOURCE DESCRIPTION

The descriptions_ptr argument points to the following structure (this structure is declared in the include file resource_control_desc.incl.pl1):

```
dcl 1 resource_descriptions    based (resource_desc_ptr) aligned,
      2 version_no             fixed bin,
      2 n_items                fixed bin,
      2 item (Resource_count refer (resource_descriptions.n_items)) aligned,
        3 type                 char (32),
        3 name                 char (32),
        3 uid                  bit (36),
        3 potential_attributes bit (72),
        3 attributes            (2) bit (72),
        3 desired_attributes    (4) bit (72),
        3 potential_aim_range   (2) bit (72),
        3 aim_range             (2) bit (72),
        3 owner                char (32),
        3 acs_path             char (168),
        3 location             char (168),
        3 comment              char (168),
        3 charge_type          char (32),
        3 rew                  bit (3) unaligned,
        3 (usage_lock,
           release_lock,
           awaiting_clear,
           user_alloc)         bit (1) unaligned,
        3 pad2                 bit (29) unaligned,
        3 given                aligned,
         (4 (name,
             uid,
             potential_attributes,
             desired_attributes,
             potential_aim_range,
             aim_range,
             owner,
             acs_path,
             location,
             comment,
             charge_type,
             usage_lock,
             release_lock,
             user_alloc)       bit (1),
          4 pad1               bit (22)) unaligned,
        3 state                bit (36) aligned,
        3 status_code          fixed bin (35);
```

STRUCTURE ELEMENTS

version_no
: is the current version number of the structure. (Input). It should be set to "resource_desc_version_1".

n_items
: specifies the number of resources described by this structure. (Input). A consistent combination of the following elements must be supplied for each resource described.

type
: specifies the type of resource desired (e.g., tape, disk_drive). (Input). It must be supplied (see "Notes" below).

name
: is a specific resource name. (Input/Output). If flags.name_given = "1"b, the named resource is chosen. If flags.name_given = "0"b, a resource is chosen depending on criteria specified by other elements of the structure, and the name of the resource chosen is returned in this element (see "Notes" below).

uid
: is the unique identifier of a specific resource. (Input/Output). If flags.uid_given = "1"b, the specified resource is chosen. If flags.uid_given = "0"b, a resource is chosen depending on criteria specified by other elements of the structure, and the unique identifier of the resource chosen is returned in this element.

potential_attributes
: specifies the potential attributes of the resource chosen. (Output)

attributes
: contains, if flags.attr_given = "1"b, the specification of attributes that the resource chosen must possess. (Input/Output). If flags.attr_given = "0"b, the resource to be chosen need not possess any particular attributes. The attributes of the resource chosen are returned in these elements (see "Notes" below).

desired_attributes
: specifies the desired attributes of the resource chosen. (Input)

potential_aim_bounds
: are a pair of AIM access classes, specifying the minimum and maximum process authorization that can be permitted to acquire this resource. (Output)

aim_bounds
: are a pair of AIM access classes, specifying the minimum and maximum process authorization that can be permitted to read and write this resource. (Input/Output). If flags.aim_bounds_given = "1"b, this element is input; otherwise, it is output.

owner
is the owner of the resource. (Input/Output). If flags.owner = "1"b, this element is input; otherwise, this element is output (see "Notes" and "Access Required" below).

acs_path
is the pathname of the Access Control Segment (ACS) for this resource (see "Access Required" below). (Input)

location
contains a character string description of the location of this resource. (Output)

comment
contains a character string comment that is associated with this resource. (Input)

charge_type
is the accounting identifier for this resource. (Input)

rew
is the effective access of the user to this resource. (Output)

usage_lock
specifies, if "1"b, that this resource cannot be used by any user, regardless of the state of the resource. (Input)

release_lock
specifies, if "1"b, that the owner of the resource is not allowed to release the resource. (Input). Unless system = "1"b, this element is ignored (see "Notes" below).

awaiting_clear
specifies that the resource is awaiting manual clear. (Output)

user_alloc
specifies, if "1"b, that the user has not allocated the resource to any use. (Input)

pad2
is unused and must be zero. (Input)

name
is "1"b if item.name has been supplied by the caller. (Input)

uid
is "1"b if item.uid has been supplied by the caller. (Input)

potential_attr
is "1"b if item.potential_attributes has been supplied by the caller. (Input)

desired_attr
is "1"b if item.desired_attributes has been supplied by the caller. (Input)

potential_aim_bounds
is "1"b if item.potential_aim_bounds has been supplied by the caller. (Input)

aim_bounds
is "1"b if item.aim_bounds has been supplied by the caller. (Input)

owner
is "1"b if item.owner has been supplied by the caller. (Input)

acs_path
is "1"b if item.acs_path has been supplied by the caller. (Input)

location
is "1"b if item.location has been supplied by the caller. (Input)

comment
is "1"b if item.comment has been supplied by the caller. (Input)

charge_type
is "1"b if item.charge_type_given has been supplied by the caller. (Input)

usage_lock
is "1"b if item.usage_lock has been supplied by the caller. (Input)

release_lock
is "1"b if item.release_lock has been supplied by the caller. (Input)

user_alloc
is "1"b if item.user_alloc_given has been supplied by the caller. (Input)

pad1
is unused and must be zero. (Input)

state
is for the use of resource_control_ and should not be used by the user. (Output)

status_code
is a standard status code. (Output). If the subroutine argument code is nonzero, one or more items in the structure have a nonzero status_code specifying in more detail why the attempt to manipulate the described resource is refused.

ACCESS REQUIRED

The user must have at least sm permission to the directory in which the ACS is specified to reside.

Unless otherwise stated, the user must have re access to the rcp_sys_ gate to specify system = "1"b in the calling sequence for any entry point of the resource_control_ subroutine.

NOTES

A list of defined resource types can be obtained via the list_resource_types command.

Suitable values for the attributes element can be constructed using the cv_rcp_attributes_$from_string subroutine.

RESERVATION DESCRIPTION

The reservation_desc_ptr argument points to the following structure (declared in the include file resource_control_desc.incl.pl1):

```
dcl 1 reservation_description aligned based,
      2 version_no fixed bin,
      2 reserved_for char (32),
      2 reserved_by char (32),
      2 reservation_id fixed bin (71),
      2 group_starting_time fixed bin (71),
      2 asap_duration fixed bin (71),
      2 flags aligned,
       (3 auto_expire bit (1),
        3 asap bit (1),
        3 rel bit (1),
        3 sec bit (1)) unaligned,
      2 n_items fixed bin,
      2 reservation_group (Resource_count refer
        (reservation_description.n_items)),
        3 starting_time fixed bin (71),
        3 duration fixed bin (71);
```

STRUCTURE ELEMENTS

version_no
: is the current version number of this structure. (Input). It should be set to "resource_control_version_1".

reserved_for
: specifies the User_id of the process for whom this reservation is made. (Input). The use of an asterisk (*) for a component name is permitted. If this element is blanks, the User_id of the current process is used.

reserved_by
: is the User_id of the process that is charged for this reservation (see "Notes" below). (Input). This element is ignored for an unprivileged reservation, and the current User_id is used.

reservation_id
: is an identifier for this reservation group. (Input/Output). It is currently returned as an absolute clock time.

n_items
: is the number of items being reserved. (Input)

The rest of the items in this structure are currently ignored and should be set to zero.

ACCESS REQUIRED

Execute access to the rcp_sys_ gate is necessary to perform a privileged reservation.

NOTES

If system = "1"b, reservation_description.reserved_by is used to specify the User_id of the process to be charged for this reservation.

The reservation_description structure is strongly dependent on the resource_descriptions structure; that is, for each resource described in resource_descriptions, there must be a corresponding entry of the same index in reservation_description.

Name: resource_info_

The resource_info_ subroutine returns selected information about RCP resource types defined on the system.

See the Programmer's Reference Manual for a description of the Multics resource control facility.

Entry: resource_info_$canonicalize_name

This entry point applies the proper canonicalization to a resource name of a a given resource type. Each resource type can have a canonicalization routine, defined by the System Administrator in the Resource Type Master File (RTMF). This routine puts a resource name into standard form by stripping leading zeros, truncating overlong names, or applying other site-defined conventions.

USAGE

```
declare resource_info_$canonicalize_name entry (char (*), char (*),
     char (*), fixed bin(35));

call resource_info_$canonicalize_name (resource_type, resource_name,
     canonicalized_name, code);
```

ARGUMENTS

resource_type
: is the name of a defined resource type. (Input)

resource_name
: is the string to be canonicalized. (Input)

canonicalized_name
: is the canonicalized representation of resource_name. (Output)

code
: is a standard status code. (Output)

Entry: resource_info_$defaults

This entry point fills a resource_descriptions structure with the default registration parameters defined in the RTDT.

USAGE

```
dcl resource_info_$defaults entry (char (*), char (*), pointer, fixed bin
     fixed bin(35));

call resource_info_$defaults (name, subtype, resource_desc_ptr,
     resource_no, code);
```

ARGUMENTS

name
: is the name of a defined resource type. (Input)

subtype
: is the name of a subtype of the resource type, defined in the RTDT. (Input). If subtype is the null string, the master defaults for the resource type are used.

resource_desc_ptr
: is the pointer to the entire resource_descriptions structure. (Input)

resource_no
: specifies the resource description structure as defined by resource_descriptions item (resource_no). If resource_no is 0, all items are used. (Input)

code
    is a standard status code. (Output)

Entry: resource_info_$get_type

Given the name of a resource type, this entry point indicates whether the resource type named is a device or a volume.

USAGE

```
declare resource_info_$get_type entry (char (*), bit (1), fixed
      bin(35));

call resource_info_$get_type (name, is_volume, code);
```

ARGUMENTS

name
    is the name of a defined resource type (see "Notes" below). (Input)

is_volume
    is "1"b if the resource type given specifies a class of volumes. (Output). If "0"b, the resource type given specifies a class of devices.

code
    is a standard status code. (Output)

NOTES

A list of defined resource types can be obtained via the list_resource_types command.

Entry: resource_info_$limits

This entry point returns information about quantity and time limits for a given resource type.

USAGE

```
declare resource_info_$limits entry (char (*), fixed bin, fixed bin,
      fixed bin, fixed bin(35));

call resource_info_$limits (name, max_quantity, default_time, max_time,
      code);
```

ARGUMENTS

name
    is the name of a defined resource type. (Input)

max_quantity
is the maximum number of this type of resource that a process can assign at one time. (Output)

default_time
is the default reservation time, in minutes, for this type of resource. (Output)

max_time
is the maximum allowed reservation time, in minutes, for this type of resource. (Output)

code
is a standard status code. (Output)

NOTES

The information returned by this entry point is from the Resource Type Description Table (RTDT). These are not the limits currently enforced by RCP.

Entry: resource_info_$lock_on_release

This entry point returns a value specifying whether resources of a given type are to be locked for manual clearing at release time.

USAGE

```
dcl resource_info_$lock_on_release entry (char(*), bit(1) aligned,
     fixed bin(35));

call resource_info_$lock_on_release (name, lock_sw, code);
```

ARGUMENTS

name
is the name of a defined resource type. (Input)

lock_sw
specifies whether the resource is locked at release time. (Output)
"1"b lock the resource
"0"b do not lock the resource

code
is a standard status code. (Output)

Entry: resource_info_$mates

This entry provides information about the resource type(s) with which the given resource type can be mounted.

USAGE

```
declare resource_info_$mates entry (char (*), fixed bin, char (*)
     dimension (*), fixed bin(35));

call resource_info_$mates (name, n_mates, mates, code);
```

ARGUMENTS

name
: is the name of a defined resource type. (Input)

n_mates
: is the number of mates returned. (Output)

mates
: contains the name(s) of the resource type(s) that can may be mounted with this resource (see "Notes" below). (Output)

code
: is a standard status code. (Output)

NOTES

If the number of elements in mates is too small to hold all the mates for the given resource type, code is set to error_table_$smallarg, and mates is set to the null string. However, n_mates still contains the number of mates associated with the given resource type.

Name: reversion_

This procedure causes the handler currently established for the given condition in the calling block activation to be disestablished. If no handler for the given condition is established in the calling block activation, no action is taken. A description of the condition mechanism is given in the *Multics Programmer's Reference* manual in the entitled "The Multics Condition Mechanism".

USAGE

```
declare reversion_ entry (char(*));

call reversion_ (name);
```

ARGUMENTS

name
: is the name of the condition for which the handler is to be disestablished. (Input)

NOTES

The condition name unclaimed_signal is an obsolete special condition name and should not be used.

A call to reversion_ must be used only to revert a handler established by a call to condition_. reversion_ must not be used to revert a handler established by a PL/I on statement.

In a PL/I program, when a call to reversion_ appears within the scope of a begin block or internal procedure of a procedure, the no_quick_blocks option must be specified in the procedure statement of that procedure. The no_quick_blocks option is a nonstandard feature of the Multics PL/I language and, therefore, programs using it may not be transferable to other systems.

This page intentionally left blank.

Name: ring0_get_

The ring0_get_ subroutine returns the name and pointer information about hardcore segments.

Entry: ring0_get_$definition

This entry point is used to ascertain the offset of a symbol in a hardcore segment in the running Multics supervisor.

USAGE

```
declare ring0_get_$definition entry (ptr, char(*), char(*),
      fixed bin(18), fixed bin, fixed bin(35));

call ring0_get_$definition (seg_ptr, component_name, sym_name, offset,
      type, code);
```

ARGUMENTS

seg_ptr
: is a pointer to the base of the segment in which it is desired to obtain a symbol offset. (Input/Output). If supplied as null, the segment that bears the name component_name in the SLT is used, and seg_ptr is returned as output as a pointer to the base of this segment.

component_name
: is the name of the segment or segment bound component in which the symbol, sym_name, is to be found. (Input). If sym_name is an unambiguous reference in the segment defined by seg_ptr, this parameter can be given as a null string. If seg_ptr is given as null, this parameter must be supplied, and specifies the segment name as well.

sym_name
: is the name of the external symbol in the segment specified by seg_ptr or component_name. (Input). If more than one external symbol of this name appears in this segment, component_name is used to select the correct component.

offset
: is the offset of this definition, if found, into the section of the specified segment as specified by type. (Output)

type
: is the definition type of this definition, detailing in which section of the specified segment this definition resides. (Output)

code
: is a standard status code. (Output). If the the segment specified has no definitions, error_table_$no_defs is returned.

Entry: ring0_get_$definition_given_slt

This entry point is used to ascertain the offset of a symbol in a hardcore segment in other than the running Multics supervisor. Copies of the Segment Loading Table (SLT), SLT name table, and hardcore definitions segment are supplied.

USAGE

```
declare ring0_get_$definition_given_slt entry (ptr, char(*), char(*),
      fixed bin(18), fixed bin, fixed bin(35), ptr, ptr, ptr);

call ring0_get_$definition_given_slt (seg_ptr, component_name, sym_name,
      offset, type, code, slt_ptr, nametbl_ptr, deftbl_ptr):
```

ARGUMENTS

seg_ptr
: is a pointer to the base of the segment in which it is desired to obtain a symbol offset. (Input/Output). If supplied as null, the segment that bears the name component_name in the SLT is used, and seg_ptr is returned as output as a pointer to the base of this segment.

component_name
: is the name of the segment or segment bound component in which the symbol, sym_name, is to be found. (Input). If sym_name is an unambiguous reference in the segment defined by seg_ptr, this parameter can be given as a null string. If seg_ptr is given as null, this parameter must be supplied, and specifies the segment name as well.

sym_name
: is the name of the external symbol in the segment specified by seg_ptr or component_name. (Input). If more than one external symbol of this name appears in this segment, component_name is used to select the correct component.

offset
: is the offset of this definition, if found, into the section of the specified segment as specified by type. (Output)

type
: is the definition type of this definition, detailing in which section of the specified segment this definition resides. (Output)

code
: is a standard status code. (Output). If the the segment specified has no definitions, error_table_$no_defs is returned.

slt_ptr
: is a pointer to the copy of the segment loading table (SLT) to be used. (Input).

nametbl_ptr
is a pointer to the corresponding copy of the SLT name table. (Input)

deftbl_ptr
is a pointer to the corresponding copy of the hardcore definitions segment (definitions_). (Input)

Entry: ring0_get_$name

This entry point returns the primary name and directory name of a ring 0 segment when given a pointer to the segment.

USAGE

```
declare ring0_get_$name entry (char (*), char (*), ptr, fixed bin);

call ring0_get_$name (dir_name, entryname, seg_ptr, code);
```

ARGUMENTS

dir_name
is the pathname of the directory of the segment. (Output). If the segment does not have a pathname (as is the case for most hardcore segments), this is returned as a null string.

entryname
is the primary name of the segment. (Output)

seg_ptr
is a pointer to the ring 0 segment. (Input)

code
is a standard status code. (Output). It is nonzero if, and only if, seg_ptr does not point to a ring 0 segment.

Entry: ring0_get_$name_given_slt

This entry point is analogous to the name entry point except that external SLT and name tables are used, instead of the versions of these tables currently being used by the system.

USAGE

```
declare ring0_get_$name_given_slt entry (char (*), char (*), ptr, fixed
     bin);

call ring0_get_$name_given_slt (dir_name, entryname, seg_ptr, code,
     sltp, namep);
```

ARGUMENTS

dir_name
is the pathname of the directory of the segment. (Output). If the segment does not have a pathname (as is the case for most hardcore segments), this is returned as a null string.

entryname
is the primary name of the segment. (Output)

seg_ptr
is a pointer to the ring 0 segment. (Input)

code
is a standard status code. (Output). It is nonzero if, and only if, seg_ptr does not point to a ring 0 segment.

sltp
is a pointer to an SLT that contains information about the segment. (Input)

namep
is a pointer to a name table (associated with the above SLT) containing the names of segments. (Input)

Entry: ring0_get_$names

This entry point returns all the names and the directory name of a ring 0 segment when given a pointer to the segment.

USAGE

```
declare ring0_get_$names entry (char(*), ptr, ptr, fixed bin);

call ring0_get_$names (dir_name, names_ptr, seg_ptr, code);
```

ARGUMENTS

dir_name
is the pathname of the directory of the segment. (Output)

names_ptr
is a pointer to a structure (described in "Notes" below) containing the names of the segment. (Output)

seg_ptr
is a pointer to the ring 0 segment. (Input)

code
is nonzero if, and only if, seg_ptr does not point to a ring 0 segment. (Output)

NOTES

The following structure is used:

```
dcl 1 segnames        based (namesptr) aligned,
      2 count         fixed bin,
      2 names         (50 refer (segnames.count)),
        3 length      fixed bin,
        3 name        char(32);
```

STRUCTURE ELEMENTS

count
is the number of names.

names
is a substructure containing an array of segment names.

length
is the length of the name in characters.

name
is the space for the name.

Entry: ring0_get_$segptr

This entry point returns a pointer to a specified ring 0 segment. Only the name is used to determine the pointer.

USAGE

```
declare ring0_get_$segptr entry (char(*), char(*), ptr, fixed bin(35));

call ring0_get_$segptr (dir_name, entryname, seg_ptr, code);
```

ARGUMENTS

dir_name
is ignored. (Input)

entryname
is the name of the ring 0 segment for which a pointer is desired. (Input)

seg_ptr
is a pointer to the segment. (Output)

code
is a standard status code. (Output). It is nonzero if, and only if, the entry is not found.

NOTES

If the entry is not found, seg_ptr is returned as a null pointer.

Entry: ring0_get_$segptr_given_slt

This entry point is analogous to the segptr entry point except that external SLT name tables are used, instead of the versions of these tables currently being used by the system.

USAGE

```
declare ring0_get_$segptr_given_slt entry (char(*), char(*), ptr,
      fixed bin(35), ptr, ptr);

call ring0_get_$segptr_given_slt (dir_name, entryname, seg_ptr, code,
      sltp, namep);
```

ARGUMENTS

dir_name
    is ignored. (Input)

entryname
    is the name of the ring 0 segment for which a pointer is desired. (Input)

seg_ptr
    is a pointer to the segment. (Output)

code
    is a standard status code. (Output). It is nonzero if, and only if, the entry is not found.

sltp
    is a pointer to an SLT that contains information about the segment. (Input)

namep
    is a pointer to a name table (associated with the above SLT) containing the names of segments. (Input)

Name: ring_zero_peek_

The ring_zero_peek_ subroutine is used to copy information out of an inner ring segment. The user must have access to either the phcs_ gate or the metering_ring_zero_peek_ gate in order to use any of the entry points in this subroutine. The phcs_ gate allows unrestricted access to all inner ring segments; metering_ring_zero_peek_ allows the user to examine specifically those data bases that are useful for metering the system. The program chooses the appropriate gate depending on the user's access and the segments being examined.

USAGE

```
declare ring_zero_peek_ entry (ptr, ptr, fixed bin(19), fixed bin(35));

call ring_zero_peek_ (ptr0, ptr_user, nwords, code);
```

ARGUMENTS

ptr0
: is a pointer to the data in ring 0 that is to be copied out. (Input)

ptr_user
: is a pointer to the region in the user's address space where the data is to be copied. (Input)

nwords
: is the number of words to be copied. (Input)

code
: is the standard status code that is nonzero if the user did not have access to the requested data. (Output)

Entry: ring_zero_peek_$by_definition

This entry point is used to copy information out of a named segment in the Multics supervisor, starting at a named symbol. It is like ring_zero_peek_$by_name, except that the copying is done from the specified definition, rather than from the base of the segment.

USAGE

```
dcl ring_zero_peek_$by_definition entry (char (*), char (*),
      fixed bin(18), pointer, fixed bin(19), fixed bin(35));

call ring_zero_peek_$by_definition (segment_name, symbol_name, offset,
      ptr_user, word_count, code);
```

ARGUMENTS

segment_name
is the name of the supervisor segment from which words are to be copied. (Input). It cannot be a pathname.

symbol_name
is the name of the external symbol in the specified segment at which copying is to start. (Input)

offset
is the offset from the specified definition at which copying is to start. (Input). It can be specified as zero to cause copying to start at the specified definition.

ptr_user
is a pointer to the area in the outer ring where the data is to be copied. (Input)

word_count
is the number of words to be copied. (Input)

code
is a standard status code. (Output). It is nonzero if the segment cannot be found, if the specified external symbol does not exist or is ambiguous, or if the user does not have sufficient access to copy the requested data.

NOTES

See "Notes" to ring_zero_peek_$by_name entry point.

Entry: ring_zero_peek_$by_name

This entry point is used to copy information out of a named segment in the Multics supervisor. It is like ring_zero_peek_, except that the name of the ring zero segment is provided, rather than a pointer to it.

USAGE

```
dcl ring_zero_peek_$by_name entry (char(*), fixed bin(18), pointer,
      fixed bin(19), fixed bin(35));

call ring_zero_peek_$by_name (segment_name, offset, copy_ptr,
      word_count, code);
```

ARGUMENTS

segment_name
: is the name of the supervisor segment from which data is to be copied. It cannot be a pathname. (Input)

offset
: is the offset from the beginning of the segment at which copying is to start. (Input). It can be specified as zero to cause copying to start from the base of the segment.

copy_ptr
: is a pointer to the area in the outer ring where the data is to be copied. (Input)

word_count
: is the number of words to be copied. (Input)

code
: is a standard status code. (Output). It is nonzero if the segment cannot be found, or if the user does not have sufficient access to copy the requested data from it.

NOTES

This entry point can be used to avoid a call to ring0_get_. For examining segments in the supervisor, this entry point and the by_definition entry point are recommended because they are much simpler to use than ring0_get_, and they are only minimally less efficient. Generally, it is nearly as efficient to use this entry point as it is to save static pointers to inner ring objects.

Entry: ring_zero_peek_$get_max_length

This entry point is used to determine the maximum length of a named ring zero segment.

USAGE

```
dcl ring_zero_peek_$get_max_length entry (char (*), fixed bin(19),
     fixed bin(35));

call ring_zero_peek_$get_max_length (seg_name, max_length, code);
```

ARGUMENTS

seg_name
  is the name of the ring zero segment. (Input)

max_length
  is the maximum length (in words) of the segment. (Output)

code
  is a standard status code. (Output). It is nonzero if the user does not have sufficient access to copy the requested data, or if the segment does not exist.

Entry: ring_zero_peek_$get_max_length_ptr

This entry point is used to determine the maximum length of a specified segment by examining its SDW. The user must have sufficient access to examine the SDW for the segment.

USAGE

```
dcl ring_zero_peek_$get_max_length_ptr entry (pointer, fixed bin(19),
     fixed bin(35));

call ring_zero_peek_$get_max_length_ptr (seg_ptr, max_length, code);
```

ARGUMENTS

seg_ptr
  is a pointer to the segment for which the max length is to be returned. (Input). If the segment is not active at the time of the call, the user must have sufficient access to reference the segment, and this reference causes a segment fault.

max_length
  is the maximum length (in words) of the segment. (Output)

code
  is a standard status code. (Output). It is nonzero if the user does not have sufficient access to copy the requested data, or if the segment does not exist.

Name: run_

The run_ subroutine manages the environment for a run unit and invokes the main program of a run unit. See the documentation of the run command in the Commands manual for an explanation of run units. This entry sets up the run unit environment, invokes the main program, and restores the environment when the run ends.

USAGE

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

call run_ (main_entry, arglist_ptr, run_cs_ptr, code);
```

ARGUMENTS

main_entry
is the entry point to be called as the main program of the run unit. (Input)

arglist_ptr
points to the argument list for the main program. (Input)

run_cs_ptr
(Input) points to the following structure which is declared in run_control_structure.incl.pl1:

```
dcl 1 run_control_structure    aligned based(run_cs_ptr),
      2 version                fixed bin,
      2 flags                  aligned,
        3 ec                   bit(1) unaligned,
        3 pad                  bit(35) unaligned,
      2 reference_name_switch  fixed bin,
      2 time_limit             fixed bin(35);
```

where:

version
is the version number of the structure. It should be set to run_control_structure_version_1.

ec
is "1"b if the main program is exec_com (main_entry must still be set), otherwise ec must be "0"b.

pad
must be "0"b.

reference_name_switch
is set to one of the named constants NEW_REFERENCE_NAMES, COPY_REFERENCE_NAMES or OLD_REFERENCE_NAMES delcared in run_control_structure.incl.pl1.

time_limit
is the interval in cpu seconds after which the program is to be interrupted.

code
is a standard status code. (Output)

Entry: run_$environment_info

This entry enables the symbolic debugging tools to obtain the saved stack header information used by a given stack frame.

USAGE

```
declare run_$environment_info entry (ptr, ptr, fixed bin(35));

call run_$environment_info (stack_frame_ptr, info_ptr, code);
```

ARGUMENTS

stack_frame_ptr
points to an active stack frame on the current stack. (Input)

info_ptr
is a pointer to the env_ptrs structure defined in "Notes" below.

code
is a standard system status code. (Output)

NOTES

The info_ptr points to the following structure, declared in env_ptrs.incl.pl1:

```
dcl 1 env_ptrs              aligned based,
      2 version             fixed bin,
      2 pad                 fixed bin(35),
      2 lot_ptr             ptr,
      2 isot_ptr            ptr,
      2 clr_ptr             ptr,
      2 combined_stat_ptr   ptr,
      2 user_free_ptr       ptr,
      2 sys_link_info_ptr   ptr,
      2 rnt_ptr             ptr,
      2 sct_ptr             ptr;
```

STRUCTURE ELEMENTS

version
    is the version number of this structure; it must be 1.

pad
    is unused.

lot_ptr
    points to the linkage offset table (LOT).

isot_ptr
    points to the internal static offset table (ISOT).

clr_ptr
    points to the area where linkage sections are allocated.

combined_stat_ptr
    points to the area where separate static sections are allocated.

user_free_ptr
    points to the area where user storage is allocated.

sys_link_info_ptr
    points to the control structure for external static variables.

rnt_ptr
    points to the reference name table.

sct_ptr
    points to the static handler array.

Name: runtime_symbol_info_

This subroutine's various entry points return runtime information about program variables (address, type, etc.) for programs compiled with symbol tables (-table). Declarations for the entry points and the structures they return can be found in the include file runtime_symbol_info_.incl.pl1. Most entry points take a pointer (symbol_ptr) to a symbol node, which can be obtained by calling stu_$find_runtime_symbol. Rather than return error codes, these entry points return null pointers or zero fields in their structures if the symbol node does not contain the requested information. Also see the various stu_ entry points for additional information about program variables and text.

WARNING:
these subroutines requires a good understanding of the symbol table structures generated by translators. For example, given a Pascal symbol "foo" declared variable of type packed array [1..10] of char, runtime_symbol_info_ does not return any useful information because this information resides in the symbol node for the TYPE of "foo".

Entry: runtime_symbol_info_$address

This entry point returns information about the location of a symbol at runtime.

USAGE

```
declare runtime_symbol_info_$address entry (ptr, ptr, fixed bin (35));

call runtime_symbol_info_$address (symbol_ptr, info_ptr, code);
```

ARGUMENTS

symbol_ptr
is a pointer to a symbol node. (Input)

info_ptr
is a pointer to a user-allocated structure to be filled in by the call. This structure, called runtime_address_info, is described under "Notes" below.

code
is error_table_$unimplemented_version if runtime_address_info.version has not been set to a valid version for the structure.

NOTES

Information is returned in the following structure, declared in the include file runtime_symbol_info_.incl.pl1:

```
dcl 1 runtime_address_info      aligned based,
      2 version                 char (8),
      2 location                fixed bin (18) unsigned unaligned,
      2 class                   fixed bin (6) unsigned unaligned,
      2 use_digit               fixed bin (1) unsigned unaligned,
      2 units                   fixed bin (2) unsigned unaligned,
      2 offset_is_encoded       bit (1) unaligned,
      2 pad                     bit (8) unaligned,
      2 offset                  fixed bin (35);
```

STRUCTURE ELEMENTS

version
: is the version of the structure, which the caller must set to RUNTIME_ADDRESS_INFO_VERSION_1.

location
: is the offset of the data within the storage class specified by the next field.

class
: is the storage class:

0	No address information is available for this symbol.
1 - 15	See the symbol table documentation in the Multics Reference Manual.

use_digit
: is "1"b to indicate that units are digits if units = 3.

units
: gives the unit of storage:

0	word	36 bits
1	bit	1 bit
2	byte	9 bits
3	half-word/digit	18 / 4.5 bits

offset_is_encoded
: is "1"b if the address is represented as an encoded offset in the next field. Encoded values, described in the symbol table documentation in the Reference Manual, are interpreted by stu_$decode_runtime_value_extended.

offset
: is the offset of the start of the identifier with respect to the address specified by location and class. It is encoded if offset_is_encoded="1"b.

Entry: runtime_symbol_info_$array

This entry point returns information about array storage allocation.

USAGE

```
declare runtime_symbol_info_$array entry (ptr, ptr, fixed bin (35));

call runtime_symbol_info_$array (symbol_ptr, info_ptr, code);
```

ARGUMENTS

symbol_ptr
is a pointer to a symbol node. (Input)

info_ptr
is a pointer to a user-allocated structure to be filled in by the call. This structure, called runtime_array_info, is described under "Notes" below.

code
is error_table_$unimplemented_version if runtime_array_info.version has not been set to a valid version for the structure.

NOTES

Information is returned in the following structure, declared in the include file runtime_symbol_info_.incl.pl1:

```
dcl 1 runtime_array_info            aligned based,
      2 version                     char (8),
      2 access_info                 aligned,
        3 ndims                     fixed bin (6) unsigned unaligned,
        3 use_digit                 fixed bin (1) unsigned unaligned,
        3 array_units               fixed bin (2) unsigned unaligned,
        3 virtual_origin_is_encoded
                                    bit (1) unaligned,
        3 pad                       bit (26) unaligned,
      2 virtual_origin              fixed bin (35),
      2 bounds                      (16) aligned,
        3 flags                     aligned,
          4 lower_is_encoded        bit (1) unaligned,
          4 upper_is_encoded        bit (1) unaligned,
          4 multiplier_is_encoded   bit (1) unaligned,
          4 pad                     bit (33) unaligned,
        3 lower                     fixed bin (35),
        3 upper                     fixed bin (35),
        3 multiplier                fixed bin (35),
        3 subscript_type            fixed bin (35),
        3 subscript_type_addr       ptr;
```

STRUCTURE ELEMENTS

version
    is the version of the structure, currently RUNTIME_ARRAY_INFO_VERSION_1.

ndims
    is the number of dimensions in the array (eg., 2 => N x M array). If this value is zero, the symbol node does not contain array information and the rest of the information in the structure is meaningless.

use_digit
    is "1"b to indicate that units are digits if array_units = 3.

array_units
    gives the unit of storage:

0	word	36 bits
1	bit	1 bit
2	byte	9 bits
3	half-word/digit	18 / 4.5 bits

virtual_origin_is_encoded
    is "1"b if the origin is represented as an encoded value in the next field. Encoded values are interpreted by stu_$decode_runtime_value_extended.

virtual_origin
    is the virtual origin of the array, in units given by array_units. Its value should be subtracted from the base address specified by the address location and class. This value is meaningless for Pascal conformant arrays, for which the origin is equal to low(1) * multiplier(1).

bounds
    gives, for each dimension, information describing the bounds and subscript.

lower_is_encoded
    is "1"b if lower is an encoded value.

upper_is_encoded
    is "1"b if upper is an encoded value.

multiplier_is_encoded
    is "1"b if multiplier is an encoded value.

lower
    is the lower bound of this dimension.

upper
    is the upper bound of this dimension.

multiplier
    is the size of an element in units given by array_units.

subscript_type
for a Pascal array, this is the type of subscript allowed for this dimension.

subscript_type_addr
for a Pascal array, this is a pointer to a Pascal type node describing the type of subscript allowed for this dimension. It is null if there is no type node.

Entry: runtime_symbol_info_$array_dims

This entry point returns the number of dimensions of an array. It returns null if the symbol has no dimensions.

USAGE

```
declare runtime_symbol_info_$array_dims entry (pointer) returns (fixed
      bin);

n_dims = runtime_symbol_info_$array_dims (symbol_ptr);
```

ARGUMENTS

symbol_ptr
is a pointer to a symbol node. (Input)

Entry: runtime_symbol_info_$brother

This entry point, given a pointer to a symbol node for an aggregate component, returns a pointer to the next component at the same level or null if this is the last component at this level of the aggregate. Given a pointer to a formal parameter, it returns a pointer to the node for the next parameter, or null if there is no next parameter. Given a pointer to any other symbol node whose level is <= 1 (non_aggregate or top-level structure) and which has a name, returns a pointer to the next element on the list of symbol nodes ordered alphabetically by size. It returns null if there is no next symbol.

USAGE

```
declare runtime_symbol_info_$brother entry (pointer) returns (pointer);

brother_ptr = runtime_symbol_info_$brother (symbol_ptr);
```

ARGUMENTS

symbol_ptr
is a pointer to a symbol node. (Input)

Entry: runtime_symbol_info_$father

This entry point, given a pointer to a symbol node for an aggregate component, returns a pointer to the symbol node for its parent aggregate. Given a pointer to a symbol node whose level is <= 1 and which has a name, returns a pointer to the runtime block node that represents the block in which the identifier is declared. It returns null if father = 0 or if there is no father field.

USAGE

```
declare runtime_symbol_info_$father entry (pointer) returns (pointer);

father_ptr = runtime_symbol_info_$father (symbol_ptr);
```

ARGUMENTS

symbol_ptr
is a pointer to to a symbol node. (Input)

Entry: runtime_symbol_info_$father_type

This entry point, given a pointer to a symbol node for a Pascal enumerated type element, returns a pointer to the symbol node for the parent type. Otherwise, it returns null.

USAGE

```
declare runtime_symbol_info_$father_type entry (pointer) returns
      (pointer);

father_type_ptr = runtime_symbol_info_$father_type (symbol_ptr);
```

ARGUMENTS

symbol_ptr
is a pointer to a symbol node. (Input)

Entry: runtime_symbol_info_$level

This entry point, given a pointer to a symbol node for an aggregate component, returns the level number of the component in the aggregate or zero if the symbol is not an aggregate component. Fields in a Pascal "with" block are at level 0.

USAGE

```
declare runtime_symbol_info_$level entry (pointer) returns (fixed bin);

level_number = runtime_symbol_info_$level (symbol_ptr);
```

ARGUMENTS

symbol_ptr
    is a pointer to a symbol node. (Input)

Entry: runtime_symbol_info_$n_variants

This entry point, given a pointer to a symbol node for a tag field in a Pascal record, returns the number of case variants for the field. It returns 0 if the symbol is not a tag field.

USAGE

```
declare runtime_symbol_info_$n_variants entry (pointer) returns (fixed
      bin);

n_variants = runtime_symbol_info_$n_variants (symbol_ptr);
```

ARGUMENTS

symbol_ptr
    is a pointer to a symbol node. (Input)

Entry: runtime_symbol_info_$name

This entry point, given a pointer to a symbol node, returns a pointer to the symbol's name in packed form (see "Notes" below). It returns null if there is no name.

USAGE

```
declare runtime_symbol_info_$name entry (pointer) returns (pointer);

name_string = runtime_symbol_info_$name (symbol_ptr) -> acc.string;
```

ARGUMENTS

symbol_ptr
    is a pointer to a symbol node. (Input)

NOTES

The variable acc.string is declared in the include file acc.incl.pl1:

```
dcl 1 acc based aligned,
      2 num_chars fixed bin (9) unsigned unaligned,
      2 string char (0 refer (acc.num_chars)) unaligned;
```

Entry: runtime_symbol_info_$next

This entry point, given a pointer to a symbol node, returns a pointer to the symbol node for the next identifier having the same name as the current identifier. It returns null if there is no name or if there are no more identifiers with the same name.

USAGE

```
declare runtime_symbol_info_$next entry (pointer) returns (pointer);

next_symbol_ptr = runtime_symbol_info_$next (symbol_ptr);
```

ARGUMENTS

symbol_ptr
is a pointer to a symbol node. (Input)

Entry: runtime_symbol_info_$son

This entry point, given a pointer to a symbol node for an aggregate, returns a pointer to the symbol node for the aggregate's first component. Given a pointer to a symbol node for a procedure, it returns a pointer to the symbol node for the first formal parameter. Given a pointer to a symbol node for an enumerated type, it returns a pointer to the symbol node for the first element of the type. Otherwise, it returns null.

USAGE

```
declare runtime_symbol_info_$son entry (pointer) returns (pointer);

son_ptr = runtime_symbol_info_$son (symbol_ptr);
```

ARGUMENTS

symbol_ptr
is a pointer to a symbol node. (Input)

Entry: runtime_symbol_info_$successor

This entry point, given a pointer to a symbol node for a Pascal enumerated type element, returns a pointer to the symbol node for the next element in the set of enumerated values for the type, or null if there is no next element or no successor field.

USAGE

```
declare runtime_symbol_info_$successor entry (pointer) returns
      (pointer);

successor_ptr = runtime_symbol_info_$successor (symbol_ptr);
```

ARGUMENTS

symbol_ptr
    is a pointer to a symbol node. (Input)

Entry: runtime_symbol_info_$type

This entry point returns information about the data type of a symbol.

USAGE

```
declare runtime_symbol_info_$type entry (pointer, pointer);

call runtime_symbol_info_$type (symbol_ptr, info_ptr);
```

ARGUMENTS

symbol_ptr
    is a pointer to a symbol node. (Input)

info_ptr
    is a pointer to a user-allocated structure to be filled in by the call. This structure, called runtime_type_info, is described under "Notes" below.

NOTES

Information is returned in the following structure, declared in the include file runtime_symbol_info_.incl.pl1:

```
dcl 1 runtime_type_info         aligned based,
      2 version                 char (8),
      2 flags,
        3 aligned               bit (1) unaligned,
        3 packed                bit (1) unaligned,
        3 size_is_encoded       bit (1) unaligned,
        3 pad                   bit (25) unaligned,
      2 scale                   fixed bin (7) unaligned,
      2 (type, base_type)       fixed bin (18) unsigned unaligned,
      2 (type_addr, base_type_addr)
                                ptr,
      2 size                    fixed bin (35);
```

STRUCTURE ELEMENTS

version
: is the version of the structure, which the caller must set to RUNTIME_TYPE_INFO_VERSION_1.

aligned
: is "1"b if the value is aligned. The meaning of alignment depends on the type. Refer to the Reference Manual's section on the runtime symbol table.

packed
: is "1"b if the value is packed. Refer to the Reference Manual.

size_is_encoded
: is "1"b if size is represented as an encoded value. Encoded values are interpreted by stu_$decode_runtime_value_extended.

scale
: is the scale factor for arithmetic values. Refer to the Reference Manual.

type
: is the type of the symbol. The defined types are declared in the include file std_descriptor_types.incl.pl1.

base_type
: when not equal to 0, is used in Pascal type description nodes in the following cases: For subranges, it is either integer, Pascal char, or Pascal enumerated type instance. For arrays, sets, and record files, it is the type of the elements. For typed pointers, it is the type of the referenced variable. For function procedure types, it is the type of the return value. For other procedure types, it is null.

type_addr
for Pascal user-defined and enumerated type variables, constants, record fields, procedure types, subscript types and base types, this is a pointer to a symbol node for the type that the symbol belongs to. Otherwise, it is null.

base_type_addr
is a pointer to a symbol node describing base_type, when base_type itself is neither 0 nor a simple type. Otherwise, it is null.

size
is the arithmetic precision, string size, or area size of the value. Refer to the Reference Manual.

Entry: runtime_symbol_info_$variant

This entry point, given a pointer to a symbol node for a Pascal record field with case variants, returns information describing the variants. If the symbol is not a Pascal symbol, number_of_variants is returned as 0 and the rest of the information is invalid.

USAGE

```
declare runtime_symbol_info_$variant entry (ptr, ptr, fixed bin (35));

call runtime_symbol_info_$variant (symbol_ptr, info_ptr, code);
```

ARGUMENTS

symbol_ptr
is a pointer to a symbol node. (Input)

info_ptr
is a pointer to a user-allocated structure to be fille din by the call. This structure, called runtime_variant_info, is described under "Notes" below.

code
is error_table_$unimplemented_version if runtime_variant_info.version has not been set to a valid version for the structure.

NOTES

Information is returned in the following structure, declared in the include file runtime_symbol_info_.incl.pl1:

```
dcl 1 runtime_variant_info    aligned based,
      2 version               char (8),
      2 number_of_variants    fixed bin,
      2 first_value_in_set    fixed bin (35),
      2 case                  (n_variants),
        3 set_addr            ptr,
        3 brother_addr        ptr;
```

STRUCTURE ELEMENTS

version
: is the version of the structure, which the caller must set to RUNTIME_VARIANT_INFO_VERSION_1.

number_of_variants
: is the number of variants if the symbol node is for a Pascal record tag field. Otherwise, it is null.

first_value_in_set
: is the lowest value used to select a variant.

case
: contains information for a particular variant.

set_addr
: is a pointer to a bit string that specifies the cases of the variant. The bit string represents a set (one bit per set element) whose base type is the type of the symbol node pointed to by symbol_ptr. The first bit corresponds to first_value_in_set.

brother_addr
: is a pointer to the first field of the variant part.

Name: sct_manager_

The sct_manager_ subroutine manipulates the System Condition Table (SCT), which is used to provide static handlers for certain conditions. It has entries to set a handler, get a pointer to a handler, and call a handler if one exists.

Entry: sct_manager_$call_handler

This entry point calls a handler if it exists. If none exists, the "continue" bit is set on to pass this information to the caller.

USAGE

```
declare sct_manager_$call_handler entry (ptr, char(*), ptr, ptr, bit (1)
      aligned);

call sct_manager_$call_handler (mcptr, cname, null(), null(), continue);
```

ARGUMENTS

mcptr
: is a pointer to the machine conditions for the condition to be handled. The fault code within the scu data determines the handler to use. (Input)

cname
: is the name of the condition being signalled. It is passed to the condition handler, if there is one. (Input)

continue
: is set to "1"b if there is no handler, otherwise it is set by the handler. (Output)

The third and fourth arguments are ignored; they must be null. They are declared for compatibility with the standard condition handler mechanism.

Entry: sct_manager_$get

This entry point returns a pointer to the handler for the given index, or null if it does not exist.

USAGE

```
declare sct_manager_$get entry (fixed bin, ptr, fixed bin (35));

call sct_manager_$get (fcode, hptr, code);
```

ARGUMENTS

fcode
: is a fixed binary index into the SCT table. Appropriate values can be selected from static_handlers.incl.pl1, which gives symbolic names for all indices currently defined. (Input)

hptr
: is a pointer to the static handler, if it exists. (Output)

code
: is a standard status code. (Output)

Entry: sct_manager_$set

This entry point sets the handler for the given index to the one given in the call.

USAGE

```
declare sct_manager_$set entry (fixed bin, ptr, fixed bin (35));

call sct_manager_$set (fcode, hptr, code);
```

ARGUMENTS

fcode
: is a fixed binary index into the SCT table. Appropriate values can be selected from static_handlers.incl.pl1, which gives symbolic names for all indices currently defined. (Input)

hptr
: is a pointer to the static handler, if it exists. (Input)

code
: is a standard status code. (Output)

NOTES

The System Condition Table is a based array of 127 packed pointers, pointed to by the sct_pointer in the stack_header of the stack for the ring in which sct_manager_ is executing. The pointers point to the entry to call, and a null value is used for the environment portion of the entry. A static handler has the same calling sequence as any other condition handler. SCT indices are assigned by hardcore systems programmers. Since sct_manager_$call_handler uses machine conditions to locate the handler, conditions without machine conditions (e.g., software conditions such as PL/I support) cannot have static handlers. Ring 0, rather than the user, ensures that there is a proper fault code in the conditions.

Name: search_paths_

Entry: search_paths_$find_dir

The search_paths_$find_dir entry point, when given a search list and an entry name, returns the absolute pathname of a directory in which the entry name can be found. The directories in the search list are searched in order for the entry name.

USAGE

```
declare search_paths_$find_dir entry (char (*), ptr, char (*), char (*),
      char (*), fixed binary (35));

call search_paths_$find_dir (sl_name, search_seg_ptr, entryname,
      ref_path, dir_name, code);
```

ARGUMENTS

sl_name
is the search list name. (Input)

search_seg_ptr
is a pointer to the search segment. If this pointer is null, then the process search segment is used. (Input)

entryname
is the entryname to search for. (Input)

ref_path
is the directory name used for the "-referencing_dir" search path. If ref_path is null, then the "-referencing_dir" search path is skipped. (Input)

dir_name
is the directory in which the entryname was found. (Output)

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

error_table_$no_search_list
the search list was not in the search segment.

error_table_$noentry
the entryname was not found in a directory in the search list.

Entry: search_paths_$find_all

The search_paths_$find_all entry point, when given a search list and an entry name, returns the absolute pathnames of directories in which the entry name can be found. The directories in the search list are searched in order for the entry name.

USAGE

```
declare search_paths_$find_all entry (char (*), ptr, char (*), char (*),
      ptr, fixed binary, ptr, fixed binary(35));

call search_paths_$find_all (sl_name, search_seg_ptr, entryname,
      ref_path, sl_info_area_ptr, sl_info_version, sl_info_ptr, code);
```

ARGUMENTS

sl_name
: is the search list name. (Input)

search_seg_ptr
: is a pointer to the search segment. If this pointer is null, then the process search segment is used. (Input)

entryname
: is the entry name to search for. (Input)

ref_path
: is the directory name used for the "-referencing_dir" search path. If ref_path is null, then the "-referencing_dir" search path is skipped. (Input)

sl_info_area_ptr
: is a pointer to an area in which sl_info can be allocated. (Input)

sl_info_version
: is the version of the sl_info structure required. (Input)

sl_info_ptr
: is a pointer to the sl_info structure containing the directories which contain the entry name. (See search_paths_$get). (Output)

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

  error_table_$no_search_list
  : the search list was not in the search segment.

  error_table_$noentry
  : the entryname was not found in a directory in the search list.

Entry: search_paths_$get

The search_paths_$get entry point returns the search paths in a search list.

USAGE

```
declare search_paths_$get entry (char (*), bit(36), char (*), ptr, ptr,
     fixed binary, ptr, fixed binary(35));

call search_paths_$get (sl_name, sl_control, ref_path, search_seg_ptr,
     sl_info_area_ptr, sl_info_version, sl_info_ptr, code);
```

ARGUMENTS

sl_name
: is the search list name. (Input)

sl_control
: is an expansion control mask. See the sl_control_s structure in "Notes" below. (Input)

ref_path
: is the directory name used for the "-referencing_dir" search path. If ref_path is null, then the "-referencing_dir" search path is skipped. (Input)

search_seg_ptr
: is a pointer to the search segment. If this pointer is null, then the process search segment is used. (Input)

sl_info_area_ptr
: is a pointer to an area in which sl_info can be allocated. (Input)

sl_info_version
: is the version of the sl_info structure required. (Input)

sl_info_ptr
: is a pointer to the sl_info structure containing the search paths in the search list. (Output) (See "Notes" below).

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

  error_table_$no_search_list
  : the search list was not in the search segment.

NOTES

The sl_control argument is defined by the sl_control_s structure contained in sl_control_s.incl.pl1. Expanding the "-referencing_dir" keyword substitutes the ref_path argument for the keyword.

```
dcl 1 sl_info              aligned based (sl_info_p),
      2 version            fixed binary,
      2 num_paths          fixed binary,
      2 change_index_p     pointer,
      2 change_index       fixed binary (71),
      2 pad1               (6) bit (36),
      2 paths              (sl_info_num_paths refer
                            (sl_info.num_paths)),
        3 type             fixed binary,
        3 code             fixed binary (35),
        3 pad2             bit (36),
        3 pathname         char (168) unaligned;
```

STRUCTURE ELEMENTS

version
: is the version of the sl_info structure (sl_info_version_1) which is also declared in the include file.

num_paths
: is the number of search paths in this structure.

change_index_p
: is a pointer to the search lists' update count. The update count is a fixed binary (71) integer, and is incremented each time the search list is modified. The caller can determine if the search list has been modified by comparing change_index in this structure with the value pointed to by change_index_p.

change_index
: is the current value of the search lists' update count.

pad1
: is unused.

path.type
: specifies the type of the search path. Keywords in sl_info.incl.pl1 define the possible values.

path.code
: is a standard status code for this search path.

path.pad2
is unused.

path.pathname
is the search path.

Entry: search_paths_$set

The search_paths_$set entry point sets the search paths of a search list.

USAGE

```
declare search_paths_$set entry (char(*), ptr, ptr, fixed binary(35));

call search_paths_$set (sl_name, search_seg_ptr, sl_info_ptr, code);
```

ARGUMENTS

sl_name
is the search list name. (Input)

search_seg_ptr
is a pointer to the search segment. If this pointer is null, then the process search segment is used. (Input)

sl_info_ptr
is a pointer to an sl_info structure (see search_paths_$get) containing the search paths for the search list. If null, then the search list is set to its default. (Input)

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

error_table_$action_not_performed
the search list was not changed. (See "Notes" below).

error_table_$new_search_list
a new search list was created. This is only a warning.

error_table_$no_search_list_default
the search list has no default.

NOTES

If the error_table_$action_not_performed status code is returned, then some search path may be invalid. A non-zero code for a search path in the sl_info structure indicates that the search path was invalid.

Entry: search_paths_$list

The search_paths_$list entry point returns a linked list of the search list names that are in a search segment.

USAGE

```
declare search_paths_$list entry (ptr, ptr, fixed binary, ptr,
      fixed binary(35));

call search_paths_$list (search_seg_ptr, sl_list_area_ptr,
      sl_list_version, sl_list_ptr, code);
```

ARGUMENTS

search_seg_ptr
: is a pointer to the search segment. If this pointer is null, then the process search segment is used. (Input)

sl_list_area_ptr
: is a pointer to an area in which a linked list of sl_list structures can be allocated. (Input)

sl_list_version
: is the version of the sl_list structure required. (Input)

sl_list_ptr
: is a pointer to a linked list of sl_list structures containing the names of the search lists in the search segment. (Output) (See "Notes" below).

code
: is a standard status code. (Output)

NOTES

The sl_list structure is contained in the include file sl_list.incl.pl1:

```
dcl 1 sl_list              based,
      2 version            fixed binary,
      2 link               pointer,
      2 name_count         fixed binary,
      2 pad                (3) bit (36),
      2 names              (sl_list_name_count refer (sl_list.name_count))
                            char (32);
```

ARGUMENTS

version
is the version of the sl_list structure (sl_list_version_2) which is also declared in the include file.

link
is a pointer to the next sl_list structure in the linked list, or null if this is the last structure in the linked list.

name_count
is the number of synonyms this search list has.

pad
must be 0.

names
is an array of the names of this search list.

Entry: search_paths_$delete_list

The search_paths_$delete_list entry point deletes a search list from a search segment.

USAGE

```
declare search_paths_$delete_list entry (char (*), ptr, fixed bin(35));

call search_paths_$delete_list (sl_name, search_seg_ptr, code);
```

ARGUMENTS

sl_name
is the search list name. (Input)

search_seg_ptr
is a pointer to the search segment. If this pointer is null, then the process search segment is used. (Input)

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

error_table_$no_search_list
the search list was not in the search segment.

Entry: search_paths_$init_search_seg

The search_paths_$init_search_seg entry point initializes a search segment.

USAGE

```
declare search_paths_$init_search_seg entry (ptr, fixed bin(35));

call search_paths_$init_search_seg (search_seg_ptr, code);
```

ARGUMENTS

search_seg_ptr
is a pointer to the search segment. If this pointer is null, then the process search segment is used. (Input)

code
is a standard status code. (Output)

Name: send_as_request_

The send_as_request_ subroutine contains entry points that send messages to the system Answer Service Request server.

Entry: send_as_request_$block

sends an as_request, and blocks to await the system's reply.

USAGE

```
declare send_as_request_$block entry (ptr, fixed bin, bit(72) aligned,
      bit(72) aligned, fixed bin(35));

call send_as_request_$block (as_request_ptr, as_request_len,
      as_request_id, as_request_reply, code);
```

ARGUMENTS

as_request_ptr
is a pointer to standard as_request structure. (Input) as_request_structures begin with a header declared in as_request_header.incl.pl1. Declarations for most as request info structures are found in as_requests.incl.pl1. It is not recommended that any application code send as_requests. Subroutine interfaces are available for all the supported as_request facilities.

as_request_len
is the length of the standard as_request structure, in words. (Input)

as_request_id
is the unique identifier assigned to the request. (Output)

as_request_reply
is the event message returned by the system in reply to the request. (Output)

code
is a standard system status code. (Output)

Entry: send_as_request_$no_block

This entry point sends an as request message to the system as request server, and does not block to await a reply.

USAGE

```
declare send_as_request_$no_block entry (ptr, fixed bin, bit(72)
      aligned, fixed bin(35));

call send_as_request_$no_block (as_request_ptr, as_request_len,
      as_request_id, code);
```

ARGUMENTS

as_request_ptr
is a pointer to standard as_request structure. (Input) as_request_structures begin with a header declared in as_request_header.incl.pl1. Declarations for most as request info structures are found in as_request.incl.pl1. It is not recommended that any application code send as_requests. Subroutine interfaces are available for all the supported as_request facilities.

as_request_len
is the length of the standard as_request structure, in words. (Input)

as_request_id
is the unique identifier assigned to the request. (Output)

code
is a standard system status code. (Output)

Name: send_mail_

The send_mail_ subroutine sends an interactive message or mail to a specified user.

USAGE

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

call send_mail_ (destination, message, info_ptr, code);
```

ARGUMENTS

destination
    is a Person_id.Project_id destination. (Input)

message
    is the text of the message to be sent. (Input)

info_ptr
    points to the structure described in "Notes" below. (Input)

This page intentionally left blank.

code
is a standard status code. (Output) It can 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_$wakeup_denied
the sending process has insufficient access to send a wakeup.
error_table_$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.

Entry: send_mail_$access_class

This entry is identical to send_mail_, except that the caller may specify the access class of the message. (In send_mail_, the access class of the message is always equal to the authorization of the calling process.) This entry is of use only if a site is using AIM. For information on access classes, see the Programmer's Reference Manual.

USAGE

```
declare send_mail_$access_class entry (char(*), char(*), ptr,
      bit(72) aligned, fixed bin(35));

call send_mail_$access_class (destination, message, info_ptr,
      access_class, code);
```

ARGUMENTS

destination
is a Person_id.Project_id destination. (Input)

message
is the text of the message to be sent. (Input)

info_ptr
points to the structure described in "Notes" below. (Input)

access_class
is the access class of the message. (Input)

code
is a standard status code. (Output)

Entry: send_mail_$path

This entry point sends an interactive message or mail to a specified mailbox.

USAGE

```
declare send_mail_$path entry (char (*), char(*), char(*), ptr,
      fixed bin (35));

call send_mail_$path (dir_name, entryname, message, info_ptr, code);
```

ARGUMENTS

dir_name
: is the directory name of a mailbox. (Input)

entryname
: is the entryname of a mailbox. (Input) The .mbx suffix is added if it is not supplied.

message
: is the text of the message to be sent. (Input)

info_ptr
: points to the structure described in "Notes" below. (Input)

code
: is a standard status code. (Output)

Entry: send_mail_$path_access_class

This entry point sends a message to a specified mailbox, allowing the user to specify the access class of the message.

USAGE

```
declare send_mail_$path_access_class entry (char(*), char(*), char(*),
      ptr, bit(72) aligned, fixed bin (35));

call send_mail_$path_access_class (dir_name, entryname, message,
      info_ptr, access_class, code);
```

ARGUMENTS

dir_name
is the directory name of a mailbox. (Input)

entryname
is the entryname of a mailbox. (Input) The .mbx suffix is added if it is not supplied.

message
is the text of the message to be sent. (Input)

info_ptr
points to the structure described in "Notes" below. (Input)

access_class
is the access class of the message. (Input)

code
is a standard status code. (Output)

NOTES

Normally the message is written into the mailbox of the receiver at the access_class specified by the sender. However, if send_mail_info.wakeup is "1"b and the receiver is accepting messages, and further, if the authorization of the receiver is greater than or equal to the access_class of the message, the access_class of the message is automatically upgraded to the authorization of the receiver. This allows the receiver to delete the message once he has read it.

INFO STRUCTURE

The info_ptr pointer points to the following structure (found in the include file, send_mail_info.incl.pl1):

```
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 notify          bit(1) unal,
        3 acknowledge     bit(1) unal,
        3 mbz             bit(30) unal;
```

STRUCTURE ELEMENTS

version
identifies the version of the structure being used. Currently this number must be 2.

sent_from
gives 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

notify
indicates that this message is a mail notification. After sending a piece of mail, a second message should be sent which has the bit ON so that the receiving process (if any) will print the "You have mail." notification.
"1"b this is a mail notification
"0"b this is NOT a mail notification, but either mail or an interactive message depending on the "wakeup" bit above.

acknowledge
indicates whether an acknowledgement is requested when the message is read.
"1"b yes
"0"b no

mbz1, mbz
are not used and must be set to "0"b.

Name: send_message_

This subroutine sends an interactive message to be received by the message facility. If the recipient is not currently accepting messages, the message is added to that user's mailbox.

USAGE

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

call send_message_ (person, project, message, code);
```

ARGUMENTS

person
: is the registered Person_id of the recipient. (Input)

project
: is a Project_id on which the recipient is registered. (Input)

message
: is the text of the message to be sent. (Input)

code
: is a standard status code. (Output) It can be one of the following:
  error_table_$messages_deferred
  : recipient is deferring messages. The message is added to his mailbox, but the recipient will not receive it immediately.

  error_table_$messages_off
  : recipient is not logged in or not accepting messages. The message is added to his mailbox, but the recipient will not receive it immediately.

  error_table_$noentry
  : no mailbox for the specified recipient.

  error_table_$no_append
  : insufficient access to add a message to the recipient's mailbox.

  error_table_$wakeup_denied
  : insufficient access to send the message interactively; the message is added to the recipient's mailbox.

  error_table_$no_info
  : access class restrictions prevent the sender from finding out whether the message was sent.

NOTES

The pathname of the mailbox sent to is:

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

Entry: send_message_$acknowledge

This entry point sends a message to be acknowledged when read by the recipient.

USAGE

```
declare send_message_$acknowledge entry (char(*), char(*), char(*),
      fixed bin(35));

call send_message_$acknowledge (person, project, message, code);
```

ARGUMENTS

person
    is the registered Person_id of the recipient. (Input)

project
    is a Project_id on which the recipient is registered. (Input)

message
    is the text of the message to be sent. (Input)

code
    is a standard status code. (Output)

NOTES

The acknowledgement is an interactive message of the form:

```
Acknowledged
```

if the message is printed immediately by the recipient, or:

```
Acknowledge message of <time>
```

if printed later.

Entry: send_message_$express

This entry point sends a message only if the recipient is logged in and accepting interactive messages by means of the accept_messages command. If the message cannot be sent interactively, it is not added to the recipient's mailbox.

USAGE

```
declare send_message_$express entry (char(*), char(*), char(*),
      fixed bin(35));

call send_message_$express (person, project, message, code);
```

ARGUMENTS

person
: is the registered Person_id of the recipient. (Input)

project
: is a Project_id on which the recipient is registered. (Input)

message
: is the text of the message to be sent. (Input)

code
: is a standard status code. (Output)

Entry: send_message_$notify_mail

This entry point sends a mail notification if the recipient is currently accepting interactive messages. The mail notification needs to be decoded by the message facility (accept_messages), which prints either:

```
You have mail from Person_id.Project_id
```

or for a mailbox other than the default one:

```
You have mail from Person_id.Project_id in <path>
```

USAGE

```
declare send_message_$notify_mail entry (char (*), char (*),
     fixed bin(35));

call send_message_$notify_mail (person, project, code);
```

ARGUMENTS

person
: is the registered Person_id of the recipient. (Input)

project
: is a Project_id on which the recipient is registered. (Input)

code
: is a standard status code. (Output)

Name: set_bit_offset_

This function returns a pointer to the specified bit in the segment referenced by the input pointer.

USAGE

```
declare set_bit_offset_ entry (ptr, fixed bin(24)) returns (ptr)
     reducible;

new_pointer_value = set_bit_offset_ (pointer_value, bit_offset);
```

ARGUMENTS

pointer_value
: identifies the segment in which the desired bit resides. (Input)

bit_offset
: is the offset (relative to the base of the segment) of the desired bit. (Input)

new_pointer_value
: is the result of this operation. (Output)

NOTES

The first bit in a segment has a bit offset of zero.

If the value of bit_offset is negative or greater than 9,437,183 (the offset of the last bit in a 256K word segment), the resulting value of the call is not defined.

Name: set_char_offset_

This function returns a pointer to the specified character in the segment referenced by the input pointer.

USAGE

```
dcl set_char_offset_ entry (ptr, fixed bin (21)) returns (ptr)
     reducible;

new_pointer_value = set_char_offset_ (pointer_value, char_offset);
```

ARGUMENTS

pointer_value
: identifies the segment in which the desired character resides. (Input)

char_offset
: is the offset (relative to the base of the segment) of the desired character. (Input)

new_pointer_value
: is the result of this operation. (Output)

NOTES

The first character in a segment has a character offset of zero.

If the value of char_offset is negative or greater than 1,048,575 (the offset of the last character in a 256K word segment), the resulting value of the call is not defined.

Name: set_ext_variable_

Allows the caller to look up an external variable by name. If the name is not found, the variable is added to the list of external variables.

USAGE

```
dcl set_ext_variable_ entry (char(*), ptr, ptr, bit(1) aligned, ptr,
      fixed bin(35));

call set_ext_variable_ (ext_name, init_info_ptr, sb_ptr, found_sw,
      node_ptr, code);
```

ARGUMENTS

ext_name
: is the name of the external variable. (Input)

init_info_ptr
: is a pointer to the initialization info (see "Notes on init_info Structure" below). (Input)

sb_ptr
: is a pointer to the base of the stack of the caller. (Input)

found_sw
: is set to indicate whether the variable was found or not. (Output)

node_ptr
is a pointer to the external variable node (see "Notes on variable_node Structure" below). (Output)

code
is an error code. (Output)

NOTES

When a new external variable is allocated (not found), it must be initialized.

```
dcl 1 init_info                 aligned based
      2 size                    fixed bin(19),
      2 type                    fixed bin,
      2 init_template
    (init_size refer
    (init_info.size))           fixed bin(35);
```

STRUCTURE ELEMENTS

size
is the initialization template size in words.

type
is the type of initialization to be performed.
0 no init
3 init from template
4 init area to empty

init_template
is the initialization template to be used when type = 3.

*

Entry: set_ext_variable_$locate

This entry point locates the specified external variable and returns a pointer to the structure describing the variable.

USAGE

```
dcl set_ext_variable_$locate entry (char(*), ptr, ptr, fixed bin(35));

call set_ext_variable_$locate (ext_name, sb_ptr, node_ptr, code);
```

ARGUMENTS

ext_name
is the name of the external variable. (Input)

sb_ptr
is a pointer to the base of the stack of the caller. (Input)

node_pointer
is a pointer to the variable_node describing the specified variable. This structure is defined in the system_link_names.incl.pl1 include file. (see "Notes" above) (Output)

code
is an error code. (Output)

Entry: set_ext_variable_$pointer

allows the caller to create a system external variable using list_init_ pointer intialization.

USAGE

```
declare set_ext_variable_$pointer entry (char(*), ptr, ptr,
     ptr bit(1) aligned, ptr, fixed bin(35));

call set_ext_variable_$pointer (ext_name, init_info_ptr, sb_ptr,
     seg_ptr, found_sw, node_ptr, code);
```

ARGUMENTS

ext_name
is the name of the external variable. (Input)

init_info_ptr
is a pointer to the initialization info (see "Notes on init_info Structure"). (Input)

sb_ptr
is a pointer to the base of the stack of the caller. (Input)

seg_ptr
is a pointer to the segment containing the object to be initialized. (Input).

found_sw
is set to indicate whether the variable was found or not. (Output)

node_ptr
is a pointer to the external variable node. (see "Notes on variable_node Structure") (Output)

code
is an error code. (Output)

Entry: set_ext_variable_$star_heap

allows the caller to look up heap variables by name. If the name is not found, the variable is created and added to the list of heap variables.

USAGE

```
declare set_ext_variable_$star_heap entry (char(*), ptr, ptr,
      ptr bit(1) aligned, ptr, fixed bin(35));

call set_ext_variable_$star_heap (ext_name, init_info_ptr, sb_ptr,
        seg_ptr, found_sw, node_ptr, code);
```

ARGUMENTS

ext_name
: is the name of the external variable. (Input)

init_info_ptr
: is a pointer to the initialization info (see "Notes on init_info Structure"). (Input)

sb_ptr
: is a pointer to the base of the stack of the caller. (Input)

seg_ptr
: is a pointer to the segment containing the object to be initialized. (Input).

found_sw
: is set to indicate whether the variable was found or not. (Output)

node_ptr
: is a pointer to the external variable node. (see "Notes on variable_node Structure") (Output)

code
: is an error code. (Output)

NOTES ON INIT_INFO STRUCTURE

When a new external variable is allocated (not found), it must be initialized. The following structure, described in system_link_init_info.incl.pl1, is pointed to by init_info_ptr:

```
dcl 1 init_info               aligned based,
      2 size                  fixed bin(19),
      2 type                  fixed bin,
      2 init_template
    (init_size refer
    (init_info.size))         fixed bin(35);
```

STRUCTURE ELEMENTS

size
: is the initialization template size, in words.

type
: is the type of initialization to be performed.
0 no init
1 invalid
2 invalid
3 init from template
4 init area to empty ()
5 list_template intialization (see "Notes on list_template Initialization Structure").

init_template
: is the initialization template to be used when type = 3.

NOTES ON LIST_TEMPLATE INITIALIZATION STRUCTURE

When the initialization type is 5 or a list_template initialization is being performed the init_info structure is not used. The structure used is the list_init_info structure which has the following definition in system_link_init_info.incl.pl1 :

```
dcl     1 list_init_info        aligned based,
          2 size                fixed bin (35),
          2 type                fixed bin,
          2 pad                 bit (18) unaligned,
          2 list_size           fixed bin (18)
                                unsigned unaligned,
          2 template            (0 refer
                                (list_init_info.list_size))
                                bit (36);
```

STRUCTURE ELEMENTS

size
: is the size of the variable in words.

type
: is the type of initialization to be performed. 5 list_template

list_size
: is the number of list_template_entries that make up the template.

template
: takes the form of a list_template_entry structure as defined in system_link_init_info.incl.pl1. This structure is passed on to list_init_ and decoded into data which is copied to the variable. See the description of list_init_ in the Privileged Subroutines Manual for a more complete description.

NOTES ON VARIABLE_NODE STRUCTURE

Great care should be taken when using the node_ptr. The variable_node structure should never be modified. Modifications to the variable_node will have unpredictable results.

A pointer to the following structure is returned by the entry points in this subroutine. It is declared in system_link_names.incl.pl1.

```
dcl 1 variable_node         aligned based,
      2 forward_thread      ptr unaligned,
      2 vbl_size            fixed bin(23) unaligned,
      2 init_type           fixed bin(11) unaligned,
      2 time_allocated      fixed bin(71),
      2 vbl_ptr             ptr,
      2 init_ptr            ptr,
      2 name_size           fixed bin(21) aligned,
      2 name                char (nchars refer
                            (variable_node.name_size)),
      2 seg_ptr             ptr;
```

STRUCTURE ELEMENTS

forward_thread
: is used by the linker to thread this variable to the next.

vbl_size
: is the size, in words, of this variable.

init_type
is the type of initialization that is performed:
0 none
1 invalid
2 invalid
3 initialize from template
4 initialize to an empty area
5 initialize using a list template (see "Notes on list_template Initialization Structure").

time_allocated
is the clock reading at the time this variable was allocated.

vbl_ptr
is a pointer to the variable's storage.

init_ptr
is a pointer to the initialization template.

name_size
is the number of characters in the variable name.

name
is the name of the variable.

seg_ptr
is a pointer to the segment containing the variables initialization information.

*

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 in an inconsistent state, 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 place the lock identifier 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 sets the lock word to "0"b 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(35));

call set_lock_$lock (lock_word, wait_time, code);
```

ARGUMENTS

lock_word
: is the word to be locked. (Input/Output)

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. (Input) A value of -1 indicates no time limit.

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

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.

error_table_$no_w_permission
: indicates that calling process does not have proper write permission to lock_word.

NOTES

The most efficient method for locking a lock is:

```
dcl static_lock_id bit(36) aligned internal static init (""b);
dcl get_lock_id_ entry returns (bit(36) aligned);
dcl stacq builtin;

if static_lock_id = ""b then
   static_lock_id = get_lock_id_();

if stacq (lock_word, static_lock_id, ""b)
then code = 0;
else call set_lock_$lock (lock_word, wait_time, code)
```

*

The code fragment above will take significantly less time to lock a lock than would calls to set_lock_$lock. If it is not already locked by another process, the stacq locking builtin is much faster than the subroutine call to set_lock_$lock. Therefore, in applications where execution speed is a primary concern use of the code above is recommended. In such cases, the subroutine call overhead can be avoided. In cases where the lock is already locked, however, the invalid lock detector and waiting facilities of set_lock_$lock are needed.

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 (35));

call set_lock_$unlock (lock_word, code);
```

ARGUMENTS

lock_word
  is the lock word to be reset. (Input/Output)

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

NOTES

The most efficient method for unlocking the lock is:

```
if stacq (lock_word, ""b, static_lock_id)
then code = 0;
else call set_lock_$unlock (lock_word, code);
```

The code fragment above will take significantly less time to lock a unlock than would calls to set_lock_$unlock. If it is not already unlocked by another process, the stacq unlocking builtin is much faster than the subroutine call to set_lock_$unlock. Therefore, in applications where execution speed is a primary concern use of the code above is recommended. In such cases, the subroutine call overhead can be avoided. In cases where the lock is already unlocked, however, the invalid lock detector and waiting facilities of set_lock_$unlock are needed.

Name: shcs_$set_force_write_limit

The shcs_$set_force_write_limit entry point sets the write limit of the calling process. This limit specifies the maximum number of pages that can be queued for I/O at the same time by calls to hcs_$force_write. The default for this limit is 1.

USAGE

```
declare shcs_$set_force_write_limit entry (fixed bin, fixed bin (35));

call shcs_$set_force_write_limit (npages, code);
```

ARGUMENTS

npages
: is the maximum number of pages that are allowed to be queued for I/O at the same time. (Input)

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

Name: signal_

The signal_ subroutine signals the occurrence of a given condition. A description of the condition mechanism and the way in which a handler is invoked by the signal_ subroutine is given in the Programmer's Reference Manual.

USAGE

```
declare signal_ entry options (variable);

call signal_ (name, mc_ptr, info_ptr, wc_ptr);
```

ARGUMENTS

name
: is the name (declared as a nonvarying character string) of the condition to be signalled. (Input)

mc_ptr
: is a pointer (declared as an aligned pointer) to the machine conditions at the time the condition was raised. This argument is used by system programs only in order to signal hardware faults. In user programs, this argument should be null if a third argument is supplied. This argument is optional. (Input)

info_ptr
is a pointer (declared as an aligned pointer) to information relating to the condition being raised. The structure of the information is dependent upon the condition being signalled; however, conditions raised with the same name should provide the information in the same structure. All structures must begin with a standard header. The format for the header as well as the structures provided with system conditions are described in the Programmer's Reference Manual. This argument is intended for use in signalling conditions other than hardware faults. This argument is optional. (Input)

wc_ptr
is a pointer (declared as an aligned pointer) to the machine conditions at the time a lower ring was entered to process a fault. This argument is used only by the system and only in the case where a condition that occurred in a lower ring is being signalled in the outer ring and when the lower ring has been entered to process a fault occurring in the outer ring. This argument is optional.

NOTES

If the signal_ subroutine returns to its caller, indicating that the handler has returned to it, the calling procedure should retry the operation that caused the condition to be signalled.

The PL/I signal statement differs from the signal_ subroutine in that the above parameters cannot be provided in the signal statement. Also, for PL/I-defined conditions, a call to the signal_ subroutine is not equivalent to a PL/I signal statement since information about these conditions is kept internally.

Name: sort_items_

The sort_items_ subroutine provides a generalized, yet highly efficient, sorting facility. Entry points are provided for sorting fixed binary (35) numbers, float binary (63) numbers, fixed-length character strings, varying character strings, and fixed-length bit strings. A generalized entry point is provided for sorting other data types (including data structures and data aggregates) and for sorting data into a user-defined order.

The procedure implements the HEAPSORT algorithm of J. W. J. Williams with the optimization suggested by R. W. Floyd. HEAPSORT does not maintain input order on duplicate keys.

The subroutine takes a vector of unaligned pointers to the data items to be sorted and rearranges the elements of this vector to point to the data items in correct order. Only the pointers are moved or copied into temporary storage; the data items remain where they were when sort_items_ was invoked.[1]

Entry: sort_items_$bit

This entry point sorts a group of fixed-length unaligned bit strings into bit string order by reordering a pointer array whose elements point to the bit strings in the group. Bit string ordering guarantees that, if each ordered bit string were converted to a binary natural number, the binary value would be less than or equal to the value of its successors.

USAGE

```
declare sort_items_$bit entry (ptr, fixed bin (24));

call sort_items_$bit (v_ptr, length);
```

ARGUMENTS

v_ptr
: points to the structure containing an array of unaligned pointers to the fixed-length unaligned bit strings to be sorted. (Input). The structure is declared as follows, where n is the value of v.n:

```
dcl 1 v aligned,
      2 n fixed bin (18),
      2 vector (n) ptr unaligned;
```

length
: is the number of bits in each string. (Input)

---

[1] Donald Knuth, "The Art of Computer Programming", vol. 3, pp 143–149, 149 (problem 18), 618 (answer to problem 18); 1973, Addison-Wesley Publishing Company.

Entry: sort_items_$char

This entry point sorts a group of fixed-length unaligned character strings into ASCII collating sequence by reordering a pointer array whose elements point to the character strings in the group.

USAGE

```
declare sort_items_$char entry (ptr, fixed bin (24));

call sort_items_$char (v_ptr, string_lth);
```

ARGUMENTS

v_ptr
points to the structure containing an array of unaligned pointers to the fixed-length character strings to be sorted. (Input). The structure is declared as follows, where n is the value of v.n:

```
dcl 1 v aligned,
      2 n fixed bin (18),
      2 vector (n) ptr unaligned;
```

string_lth
is the length of each character string. (Input)

Entry: sort_items_$fixed_bin

This entry point sorts a group of aligned fixed binary (35,0) numbers into numerical order by reordering a pointer array whose elements point to the numbers in the group.

USAGE

```
declare sort_items_$fixed_bin entry (ptr);

call sort_items_$fixed_bin (v_ptr);
```

ARGUMENTS

v_ptr
points to a structure containing an array of unaligned pointers to the aligned fixed binary (35,0) numbers to be sorted. (Input). The structure is declared as follows, where n is the value of v.n:

```
dcl 1 v aligned,
      2 n fixed bin (18),
      2 vector (n) ptr unaligned;
```

Entry: sort_items_$float_bin

This entry point sorts a group of aligned float binary (63) numbers into numerical order by reordering a pointer array whose elements point to the numbers in the group.

USAGE

```
declare sort_items_$float_bin entry (ptr);

call sort_items_$float_bin (v_ptr);
```

ARGUMENTS

v_ptr
: points to the above structure containing an array of unaligned pointers to the aligned float binary (63) numbers to be sorted. (Input)

Entry: sort_items_$general

This entry point sorts a group of arbitrary data elements, structures, or other aggregates into a user-defined order by reordering a pointer array whose elements point to the data items in the group. The structure of data items, the information field or fields within each item by which items are sorted, and the data ordering principle are all decoupled from the sorting algorithm by calling a user-supplied function to order pairs of data items. The function is called with pointers to a pair of items. It must compare the items and return a value that indicates whether the first item of the pair is less than, equal to, or greater than the second item. The sorting algorithm reorders the elements of the pointer array based upon the results of the item comparisons.

USAGE

```
declare sort_items_$general entry (ptr, entry);

call sort_items_$general (v_ptr, function);
```

ARGUMENTS

v_ptr
: points to the structure containing an array of unaligned pointers to the data items to be sorted. (Input). The structure is declared as follows, where n is the value of v.n:

```
dcl 1 v aligned,
      2 n fixed bin (18),
      2 vector (n) ptr unaligned;
```

function
    is a user-supplied ordering function. (Input). Its calling sequence is shown under "Notes" below.

NOTES

The sort_items_$general entry point calls a user-supplied function to compare pairs of data items. This function must know the structure of the data items being compared, the field or fields within each item that are to be compared, and the ordering principle to be used in performing the comparisons. The function returns a relationship code as its value. The calling sequence of the function is as follows:

```
declare function entry (ptr unaligned, ptr unaligned)
     returns (fixed bin(1));

value = function (ptr_first_item, ptr_second_item);
```

where:

ptr_first_item
    is an unaligned pointer to the first data item. (Input)

ptr_second_item
    is an unaligned pointer to a data item to be compared with the first data item. (Input)

value
    is the value of the first data item compared to the second data item. (Output). It can be:

-1	the first data item is less than the second.
0	the first data item is equal to the second.
+1	the first data item is greater than the second.

EXAMPLE

A simple example of a user-supplied ordering function is shown below. It compares pairs of fixed binary (35,0) numbers. If this function is passed to the sort_items_$general entry point, it performs the same function as a call to the sort_items_$fixed_bin entry point, but with less efficiency because of the overhead involved in calling the function.

```
function: procedure (p1, p2) returns (fixed bin(1));

    declare (p1, p2) ptr unaligned,
              datum fixed bin(35,0) based;

              if p1 -> datum < p2 -> datum then
                   return (-1);
              else if p1 -> datum = p2 -> datum then
                   return ( 0);
              else
                   return (+1);

              end function;
```

Entry: sort_items_$varying_char

This entry point sorts a group of varying character strings into ASCII collating sequence by reordering a pointer array whose elements point to the character strings in the group.

USAGE

```
declare sort_items_$varying_char entry (ptr);

call sort_items_$varying_char (v_ptr);
```

ARGUMENTS

v_ptr
: points to the structure containing an array of unaligned pointers to the varying character strings to be sorted. (Input). The structure is declared as follows, where n is the value of v.n:

```
dcl 1 v aligned,
      2 n fixed bin (18),
      2 vector (n) ptr unaligned;
```

Name: sort_items_indirect_

The sort_items_indirect_ subroutine is a variation of the sort_items_$general entry point. It provides a facility for sorting a group of data items, based upon the value of an information field that is logically associated with each item but resides at a varying offset from the beginning of each item. A name in the name list associated with the status block returned by the hcs_$status_ entry point is an example of such an information field.

The sort_items_indirect_ subroutine provides high performance entry points for sorting data items by the value of a single fixed binary (35) field, float binary (63) field, fixed-length bit string field, fixed-length character string field, or adjustable length character string field associated with each item. A generalized entry point is provided for sorting other types of information fields, for sorting aggregate information fields, or for sorting items into a user-defined order.

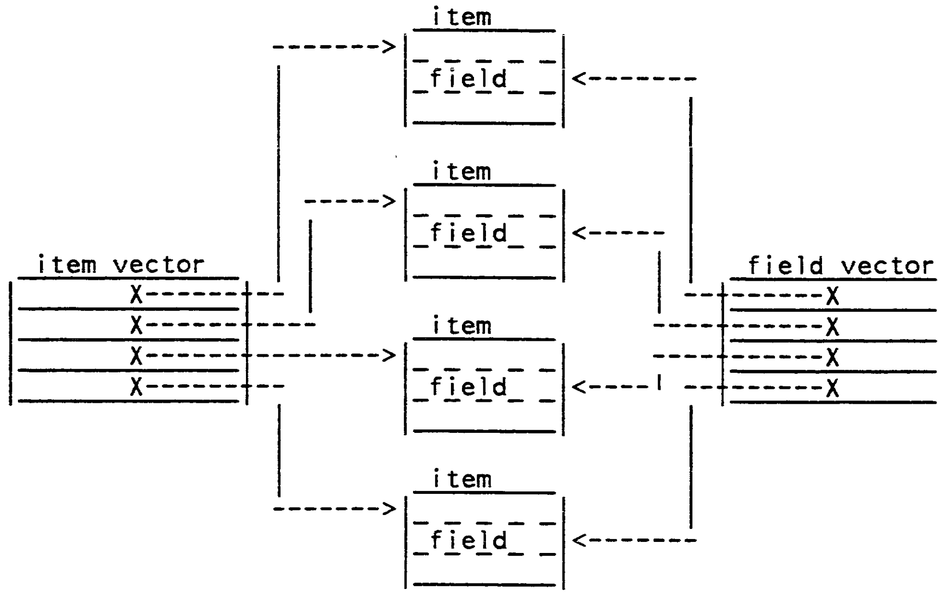
To use the sort_items_indirect_ subroutine, for some entries the caller must create three arrays: a vector of pointers to the data items being sorted (the item vector), a vector of pointers to the single information field within each item on which the sort is based (the field vector), and an array of indices into these two vectors. For other entries, only two arrays are required: a vector of pointers to the data items being sorted and an array of indices into the vectors. This index array is initialized sequentially with integers by sort_items_indirect_, which then reorders these indices to index the pointer vectors to the data items in correct order. Only indices are moved or copied into temporary storage. Vector elements and data items remain where they were when sort_items_indirect_ was invoked.

This procedure differs from that used in the sort_items_ subroutine in that an array of indices into the vector is sorted rather than the vector itself. This allows the caller to create two vectors of pointers: one containing pointers to the data items to be sorted and one containing pointers to the particular data field within each item on which the item is to be sorted. There is a one-to-one correspondence between the elements of the data items vector and the elements of the data field within each item vector. This correspondence is maintained across the reordering of the index array. Thus, the index array provides indices into the sorted list of data fields and also into the sorted list of data items containing these fields.

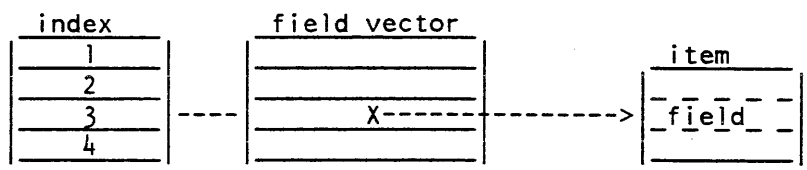
NOTES

To use the sort_items_indirect_$adj_char entry point, one additional array must be created: an array of lengths of the adjustable length character string information fields on which the sort is based.

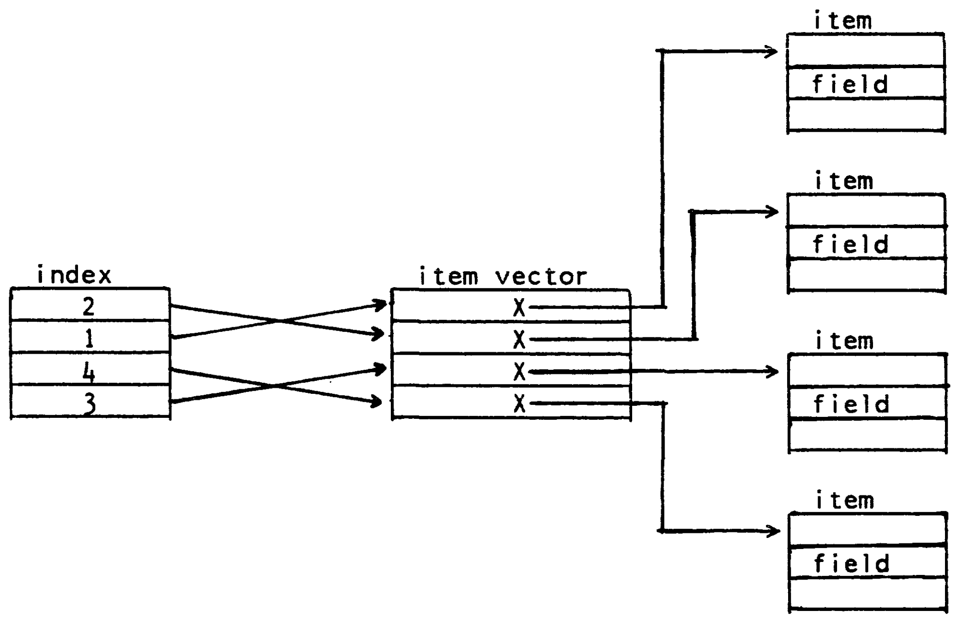
For the sake of simplicity, the sort information field is shown as part of the items being sorted in each of the diagrams below. A more general application might show each item containing a locator variable that addresses the sort field(s) associated with that item. The one-to-one correspondence between elements of the item vector and elements of the field vector is shown below.



The array of indices can be used to reference elements of both vectors. The field vector and index array are passed to the sort_items_indirect_ subroutine, which references the sorting field in each item through elements of these two arrays, as shown below.



The sort_items_indirect_ subroutine reorders the index values so that values selected sequentially from the index array reference pointer to the elements of a sorted list of information fields. Because the sorting process involves only the interchange of index values, there is still a correspondence between the elements of the item vector and the elements of the field vector after the sort is complete.



If the information field upon which the sort is based is located at a known offset from the beginning of each item, then the calling program can avoid creating the index array and the item vector by using the sort_items_ subroutine. (This subroutine cannot process adjustable length fields.) The field vector is passed to the sort_items_ subroutine, and then the elements of the item vector are computed by applying the appropriate offset to the corresponding field vector elements.

The procedure implements the HEAPSORT algorithm of J. W. J. Williams with the optimization suggested by R. W. Floyd.[1]

Entry: sort_items_indirect_$adj_char

This entry point sorts a group of information fields, which are unaligned adjustable length character strings, into ASCII collating sequence order by reordering an index array. The elements in this index array are indices into an array of unaligned pointers to the character strings in the group.

USAGE

```
declare sort_items_indirect_$adj_char (ptr, ptr, ptr);

call sort_items_indirect_$adj_char (v_ptr, i_ptr, l_ptr);
```

---

[1] Donald Knuth, "The Art of Computer Programming", vol. 3, pp 143-149, 159 (problem 18), 618 (answer to problem 18); 1973, Addison-Wesley Publishing Company.

ARGUMENTS

v_ptr
points to a structure containing an array of unaligned pointers to the aligned fixed binary (35,0) numbers to be sorted. (Input). The structure pointed to by v_ptr is to be declared as follows, where n is the number of elements to be sorted.

```
dcl 1 v aligned,
      2 n fixed bin (18),
      2 vector (n) ptr unaligned;
```

i_ptr
points to a structure containing an ordered array of fixed binary (18) indices into the unaligned pointer array. (Input). The structure pointed to by i_ptr is to be declared as follows, where n is the number of elements to be sorted. Since sort_items_indirect_ sets the i.n and i.array elements, the user needs not set them prior to calling the subroutine.

```
dcl 1 i aligned,
      2 n fixed bin (18),
      2 array (n) fixed bin (18);
```

l_ptr
points to a structure containing an array of lengths of the unaligned adjustable length character strings to be sorted. (Input). The structure is declared as follows, where n is the number of elements to be sorted.

```
dcl 1 l aligned,
      2 n fixed bin (18),
      2 vector (n) fixed bin (21);
```

Entry: sort_items_indirect_$bit

This entry point sorts a group of information fields, which are fixed-length unaligned bit strings into bit string order by reordering an index array. The elements of this index array are indices into an array of pointers to the bit strings in the group. Bit string ordering guarantees that, if each ordered bit string is converted to a binary natural number, the binary value is less than or equal to the value of each of its successors.

USAGE

```
declare sort_items_indirect_$bit entry (ptr, ptr, fixed bin (24));

call sort_items_indirect_$bit (v_ptr, i_ptr, length);
```

ARGUMENTS

v_ptr
points to the above structure v containing an array of unaligned pointers to the fixed-length unaligned bit strings to be sorted. (Input)

i_ptr
points to the above structure i containing an ordered array of fixed binary (18) indices into the unaligned pointer array. (Input)

length
is the number of bits in each string. (Input)

Entry: sort_items_indirect_$char

This entry point sorts fixed-length unaligned character strings into ASCII collating sequence by reordering an index array whose elements are indices into a pointer array that points to the strings. All the strings must be the same length.

USAGE

```
declare sort_items_indirect_$char entry (ptr, ptr, fixed bin (21));

call sort_items_indirect_$char (v_ptr, i_ptr, string_lth);
```

ARGUMENTS

v_ptr
points to the above structure v containing an array of unaligned pointers to the fixed-length unaligned character string to be sorted. (Input)

i_ptr
points to the above structure i of fixed binary (18) indices into the unaligned pointer array. (Input)

string_lth
indicates the length of each character string. (Input)

Entry: sort_items_indirect_$fixed_bin

This entry point sorts a group of information fields, which are aligned fixed binary (35,0) numbers, into numerical order by reordering an index array. The elements of this index array are indices into an array of unaligned pointers to the numbers in the group.

USAGE

```
declare sort_items_indirect_$fixed_bin entry (ptr, ptr);

call sort_items_indirect_$fixed_bin (v_ptr, i_ptr);
```

ARGUMENTS

v_ptr
: points to the above structure v containing an array of unaligned pointers to the unaligned adjustable length character strings to be sorted. (Input)

i_ptr
: points to the above structure i containing an ordered array of fixed binary (18) indices into the unaligned pointer array. (Input)

Entry: sort_items_indirect_$float_bin

This entry point sorts a group of information fields, which are aligned float binary (63,0) numbers, into numerical order by reordering an index array. The elements of this index array are indices into an array of unaligned pointers to the numbers in the group.

USAGE

```
declare sort_items_indirect_$float_bin entry (ptr, ptr);

call sort_items_indirect_$float_bin (v_ptr, i_ptr);
```

ARGUMENTS

v_ptr
: points to the above structure v containing an array of unaligned pointers to the aligned float binary (63,0) numbers to be sorted. (Input)

i_ptr
: points to the above structure i containing an ordered array of fixed binary (18) indices into the unaligned pointer array. (Input)

Entry: sort_items_indirect_$general

This entry point sorts a group of information fields (which are arbitrary data elements, structures, or other aggregates) into a user-defined order. It does this by reordering an array of indices into a pointer array. The elements of this index array point to the sort information field within the data items of the group. The structure and data type of the information field and the data ordering principle are decoupled from the sorting algorithm by calling a user-supplied function to order pairs of information fields. The function is called with pointers to a pair of fields. It must compare the fields and return a value that indicates whether the first field of the pair is less than, equal to, or greater than the second field. The sorting algorithm reorders the elements of the index array based upon the results of the information field comparisons.

USAGE

```
declare sort_items_indirect_$general entry (ptr, ptr, entry);

call sort_items_indirect_$general (v_ptr, i_ptr, function):
```

ARGUMENTS

v_ptr
: points to the above structure v containing an array of unaligned pointers to the information fields to be sorted. (Input)

i_ptr
: points to the above structure i containing an ordered array of fixed bin (18) indices into the unaligned pointer array. (Input)

function
: is a user-supplied ordering function. (See "Notes" below.) (Input)

NOTES

The sort_items_indirect_$general entry point calls a user-supplied function to compare pairs of data items. This function must know the structure and data type of the information fields, and it must know the ordering principle to be used to compare a pair of information fields. The function returns a relationship code as its value. The calling sequence of the function is as follows:

```
declare function entry (ptr unaligned, ptr unaligned)
     returns (fixed!bin(1));

value = function (ptr_1st_field, ptr_2nd_field);
```

where:

ptr_1st_field
: is an unaligned pointer to the first information field. (Input)

ptr_2nd_field
: is an unaligned pointer to an information field to be compared with the first information field. (Input)

value
: is the value of the first information field compared to the second information field. (Output). It can be:
: -1 first information field is less than the second.
: 0 first information field is equal to the second.
: +1 first information field is greater than the second.

A simple example of a user-supplied ordering function is shown in the sort_items_ subroutine.

Entry: sort_items_indirect_$varying_char

This entry point sorts a group of information fields, which are varying unaligned character strings, into ASCII collating sequence by reordering an index array. The elements of this index array are indices into an array of pointers to the character strings in the group.

USAGE

```
declare sort_items_indirect_$varying_char entry (ptr, ptr);

call sort_items_indirect_$varying_char (v_ptr, i_ptr);
```

ARGUMENTS

v_ptr
: points to the above structure v containing an array of unaligned pointers to the varying fixed-length character strings to be sorted. (Input)

i_ptr
: points to the above structure i containing an ordered array of fixed binary (18) indices into the unaligned pointer array. (Input)

Name: sort_seg_

The sort_seg_ subroutine provides entry points for sorting segments and character strings. It is the subroutine interface used by the sort_seg command, and provides all of the facilities of this command at a subroutine level.

OVERVIEW OF SORTING

Segments and strings are sorted by dividing the input characters into sort units. Each sort unit is composed of N sort strings, where N is the blocking factor. Sort strings are identified in the input by a delimiter, which can be specified in terms of a fixed number of characters per sort string, or in terms of delimiting characters which match a string or qedx regular expression. When delimiting characters are used, the characters themselves are not treated as part of the sort string. They simply end one sort string and begin the next.

Sort units are sorted by comparing specific sort fields in one unit with those of another. At least one sort field must be given. However, it may identify the entire sort unit, in which case the sort units themselves are compared to perform the sorting.

Sort fields within a sort unit are located by specifying their starting and ending points. These points can be specified in terms of a character index within the sort unit (or an index and a length), or in terms of field start and end characters which match a caller-supplied string or regular expression.

By default, all sort fields are treated as ASCII character data, but fields can be sorted as integer or numeric fields with sort_seg_ converting the sort fields before sorting. Fields can be sorted in ascending or descending ASCII collating sequence. Optionally, uppercase letters in the sort data can be treated as lowercase letters for sorting purposes, providing a case-insensitive sorting capability.

After sorting the units, special action can be taken if duplicate sort units (or units containing duplicate sort fields) are found in the sorted output. Such duplicate sort units may be removed from the sorted output, or may be chosen for output in place of the normal sort results.

For further information about sorting, refer to the description of the sort_seg command in the Commands manual. It discusses sort strings, sort units, and sort fields in more detail and includes some examples.

Entry: sort_seg_$seg

This entry point sorts an entire segment. The sorted output can either replace the original segment or be written into a new segment.

USAGE

```
declare sort_seg_$seg entry (char (*), ptr, char (*), char (*), char (*),
     char (*), fixed bin(21), fixed bin(21), fixed bin(35));

call sort_seg_$seg (caller, ss_info_ptr, in_dir, in_ent, out_dir,
     out_ent, out_len, undelim_char_index, code);
```

ARGUMENTS

caller
: specifies the name of the calling procedure. Temporary segments used for sort work space are obtained in the caller's name, and the user may be asked questions in the caller's name when errors occur. (Input)

ss_info_ptr
: points to the ss_info structure described in "Info Structure" below. (Input)

in_dir
: is the pathname of the directory containing the segment to be sorted. (Input)

in_ent
: is the entryname of the segment to be sorted. (Input)

out_dir
: is the pathname of the directory in which the sorted results are to be placed. (Input)

out_ent
: is the entryname of the segment in which the sorted results are placed. The same segment can be identified by in_dir/in_ent and out_dir/out_ent, in which case the input segment is replaced by the sorted results. (Input)

out_len
: is the the length in characters of the sorted results. This is useful if the caller wants to examine or print the sorted results. The caller need not truncate or set the bit count for the output segment; sort_seg_ performs these functions. (Output)

undelim_char_index
: if characters are found following the last sort string delimiter in the input segment, then this is the character index of the first such character in the sorted output results. Such undelimited characters always appear at the end of the sorted output. It is 0 if no such undelimited characters are found in the input segment. (Output)

code
is a system status code. If code is nonzero, then sort_seg_ will already have printed an error message via sub_err_. (Output)

ACCESS REQUIREMENTS

To use the sort_seg_$seg interface, the user must have read access to the segment being sorted, and rw access to the output segment. If the user lacks rw access to the output segment, sort_seg_$seg will ask if access should be temporarily set to allow sorting.

INFO STRUCTURE

The ss_info_ptr argument to sort_seg_ points to a structure which defines the type of sorting to be performed, the sort field specifications, and so forth. The caller must set all structure elements before calling sort_seg_ entry points. This info structure is declared in sort_seg_info.incl.pl1:

```
dcl  1 ss_info               aligned based (ss_info_ptr),
       2 header,
         3 version           char (8),
         3 block_size        fixed bin,
         3 field_count       fixed bin,
         3 duplicate_mode    fixed bin,
         3 mbz1 (3)          fixed bin,
         3 delim,
           4 type            fixed bin,
           4 number          fixed bin,
           4 string          char(256) varying,
       2 field (ss_field_count refer (ss_info.field_count)),
         3 from              like ss_info.delim,
         3 to                like ss_info.delim,
         3 modes,
          (4 descending      bit(1),
           4 non_case_sensitive bit(1),
           4 numeric         bit(1),
           4 integer         bit(1),
           4 mbz2            bit(32)) unal;
```

STRUCTURE ELEMENTS

version
is the version number of this structure. The current version is 1. This version is represented by the character string value stored in the SS_info_version_1 constant. Use this constant in setting the version number.

block_size
specifies the number of sort strings to be used in forming each sort unit. The input is divided into sort strings according to the specifications given in ss_info.delim. The block_size value must be 1 or larger.

field_count
specifies the number of sort fields defined for each sort unit. At least 1 field must be defined. It can define all or part of the sort unit to be the sort field.

duplicate_mode
specifies how duplicate sort units, or units containing duplicate sort fields, are to be treated as part of the sorting process. Values which may be assigned to this element are defined by the following named constants:

SS_duplicate
retains duplicate sort units in the sorted results.

SS_unique
deletes duplicate sort units from the sorted results.

SS_only_duplicates
only duplicated sort units appear in the sorted results. One unit from each set of duplicate sort units is placed in the results, in sorted order. This is a means of identifying and returning only the duplicates.

SS_only_duplicate_keys
only sort units which have duplicate sort fields appear in the sorted results. All units from each set of sort units having duplicate fields are placed in the results, in sorted order. This provides a means of identifying and returning sort units which have the same sort fields.

SS_unique_keys
deletes sort units having duplicate sort fields from the sorted results. For each set of sort units having duplicate sort fields, only the first appears in the sorted results, along with nonduplicate sort units.

SS_only_unique
only sort units which are unique appear in the sorted results. Whenever a set of duplicate units are found, they are removed entirely from the output.

SS_only_unique_keys
only sort units which have unique sort fields appear in the sorted results. All units having duplicate sort fields are removed entirely from the output.

mbz1
must be set to 0.

delim.type
specifies the type of delimiter to be used in dividing the input into sort strings. Allowable values are:

SS_length
specifies that the input is to be divided into fixed length sort strings. delim.number specifies the length of each sort string.

SS_string
specifies that the input is to be divided into sort strings by the character string contained in delim.string. The first sort string consists of characters from the beginning of the input up to the first instance of this delimiter string. Subsequent sort strings consist of the characters following the delimiter string for the previous unit, up to the next instance of the delimiter string in the input. Note that the delimiter string itself does not appear in any sort string.

SS_reg_exp
specifies that the input is to be divided into sort strings by character strings which match the qedx regular expression contained in delim.string. Division occurs as for SS_string, except that regular expression matching is used instead of simple string comparision to find the delimiter strings. The delimiter strings matching the qedx regular expression do not appear in any sort string.

delim.number
specifies the length of each sort string, as described under the SS_length case of delim.type above.

delim.string
specifies the delimiter string or delimiter regular expression, as specified under the SS_string and SS_reg_exp cases of delim.type above.

field
is an array of structures which defines the sort fields within each sort unit used to compare sort units. Each field is specified in terms of a starting location, ending location, and comparison modes. At least 1 field must be specified. To define the entire sort unit as a field, specify the following:

```
field(1).from.type = SS_index;
field(1).from.number = 1;
field(1).to.type = SS_length;
field(1).to.number = -1;
```

field.from.type
specifies the type of starting locator used to define the sort field. Allowable values are:

SS_index
: specifies that the character index given in field.from.number is the first character of the sort field. If the sort unit is shorter than this character index, then the unit is sorted as if the field consisted of space characters.

SS_string
: specifies that field.from.string contains a character string which identifies the start of the sort field. The field begins with the first character following the first occurrence of this string in the sort unit. If the string does not appear in the sort unit, then the unit is sorted as if the field consisted of space characters.

SS_reg_exp
: specifies that field.from.string contains a qedx regular expression. The sort field begins with the first character following the first string of the sort unit which matches this regular expression. If no match is found in the sort unit, then the unit is sorted as if the field consisted of space characters.

field.from.number
: specifies the character index within the sort unit of the first character of the sort field, as described under the SS_index case of field.from.type above.

field.from.string
: specifies the field start string or regular expression, as described under the SS_string and SS_reg_exp cases of field.from.type above.

field.to.type
: specifies the type of ending locator used to define the sort field. Allowable values are:

SS_length
: specifies that the sort field will have a fixed length. field.to.number specifies the number of characters in the sort field. If the sort unit is too short to hold a field of this length, then the unit is sorted as if the field were extended on the right with space characters to the fixed field length. If a length of -1 is specified, then the sort field extends to the end of the sort unit.

SS_index
: specifies that the character index given in field.to.number is the last character of the sort field. If the sort unit is shorter than this character index, then the unit is sorted as if the field were extended on the right with space characters to the specified character position. If the field starting location falls after the ending character index, then the unit is sorted as if the field consisted of space characters.

SS_string
: specifies that field.to.string contains a character string which identifies the end of the sort field. The field ends with the first character preceding the first occurrence of this string following the field starting location in the sort unit. If the string does not appear in the sort unit following the field starting location, then the unit is sorted as if the field contained space characters.

SS_reg_exp
: specifies that field.to.string contains a qedx regular expression. The sort field ends with the first character preceding the first string of the sort unit following the field starting location which matches this regular expression. If no match is found, then the unit is sorted as if the field consisted of space characters.

field.to.number
: specifies the character index within the sort unit of the last character of the sort field, as described under the SS_index case of field.to.type above; or specifies the character length of the sort field, as described under the SS_length case of field.to.type above.

field.to.string
: specifies the field end string or regular expression, as described under the SS_string and SS_reg_exp cases of field.to.type above.

field.modes.descending
: if "1"b, causes the field to be sorted in descending ASCII collating sequence. Otherwise, the field is sorted in ascending sequence.

field.modes.non_case_sensitive
: if "1"b, causes the field to be translated to lowercase when field comparisons are performed. The actual sort unit remains unchanged. Otherwise, field comparisons are performed without translating the field to lowercase.

field.modes.numeric
: if "1"b, causes the field to be converted to a numeric value (float decimal(59)) before field comparisons are performed.

field.modes.integer
: if "1"b, causes the field to be converted to an integer value (fixed bin(71,0)) before field comparisons are performed. If neither numeric nor integer are "1"b, field comparisons are performed as ASCII character strings. The character string representation must be acceptable to the PL/I or Fortran language conversion rules. The actual sort field remains unchanged in the sorted results.

field.modes.mbz2
: must be set to "0"b.

NOTES

A special named constant, SS_unset, can be assigned to duplicate_mode, delim.type, field.from.type or field.to.type to indicate that the type has not yet been set. This value should not be assigned when sort_seg_ is invoked. It can be used by the caller when filling in the structure, based upon control arguments supplied by the user.

Entry: sort_seg_$string

This entry point sorts the contents of a character string. The sorted output can either replace the original string or be written into another string.

USAGE

```
declare sort_seg_$string entry (char (*), ptr, char (*), char (*),
      fixed bin(21), fixed bin(21), fixed bin(35));

call sort_seg_$string (caller, ss_info_ptr, in_string, out_string,
      out_len, undelim_char_index, code);
```

ARGUMENTS

caller
: specifies the name of the calling procedure. Temporary segments used for sort work space will be obtained in the caller's name, and the user may be asked questions in the caller's name when errors occur. (Input)

ss_info_ptr
: points to the ss_info structure described under the sort_seg_$seg entry point. (Input)

in_string
: is the string to be sorted. (Input)

out_string
: is the string in which the sorted results are placed. The same string may be given for both in_string and out_string, in which case the sorted results overwrite the in_string. The out_string may also overlap part of the storage for in_string. When the overlapping is partial or complete, the in_string is copied into a temporary segment prior to being sorted. (Output)

out_len
: is the length in characters of the sorted results. (Output)

undelim_char_index
: if characters are found following the last sort string delimiter in the input string, then this is the character index of the first such character in the sorted output results. Such undelimited characters always appear at the end of the sorted output. It is 0 if no such undelimited characters are found in the input string. (Output)

code
: is a system status code. If code is nonzero, then sort_seg_ will already have printed an error message via sub_err_. (Output)

Name: spg_util_

The spg_util_ subroutine collects metering information from the Multics supervisor and subtracts it from the previous sample taken. It is normally called by the system_performance_graph command. To use this subroutine, access to either the phcs_ or the metering_gate_ gate is required.

USAGE

```
declare spg_util_$spg_util_ (float, float, float, float, float, float,
      float, float, float, char(110), fixed bin, fixed bin)

call spg_util_$spg_util_ (pzi, pnmpi, pmpi, pint, ptc, ppf, psf,
      puse_rz, px, string, length, chsw)
```

ARGUMENTS

pzi
: is the percentage of zero idle time. (Output)

pnmpi
: is the percentage of nonmultiprogramming idle time. (Output)

pmpi
: is the percentage of multiprogramming idle time. (Output)

pint
: is the percentage of time in interrupts. (Output)

ptc
: is the percentage of time in the traffic controller. (Output)

ppf
: is the percentage of time in page control. (Output)

psf
  is the percentage of time in segment control. (Output)

puse_rz
  is the percentage of time executing nonsupervisor code spent in ring zero. (Output)

px
  is no longer used. A value of 0.0 is returned. (Output)

string
  if the variable chsw is nonzero, string contains upon output a character string that describes a new configuration or a new setting of the scheduler tuning parameters. (Output)

length
  is the length of the character string "string". (Output)

chsw
  is a switch that, if zero, indicates normal output; if nonzero, it indicates that string and length are valid and should be output. (Output)

Entry: spg_util_$reset

The effect of this call is to reset the internal initialization switch of the subroutine.

USAGE

```
declare spg_util_$reset entry;

call spg_util_$reset;
```

ARGUMENTS

There are no arguments.

Name: spg_ring_0_info_

The spg_ring_0_info_ subroutine returns information about the virtual CPU time spent in the three main gates into ring zero. The three gates are hcs_, phcs_, and hphcs_. To use this subroutine, access to either the phcs_ or the metering_gate_ gate is required.

USAGE

```
declare spg_ring_0_info_ entry (fixed bin (71));

call spg_ring_0_info (time_rz);
```

ARGUMENTS

time_rz
    is the cumulative time, in microseconds, spent in ring zero. (Output)

Name: ssu_

The ssu_ subroutine provides a set of standard functions for use by application writers in developing their own subsystems. Use of ssu_ functions will enable the application builder to provide subsystems which are consistent in terms of user interface and system response. For detailed instructions on creating subsystems, see "Interactive Subsystem Programming Environment" in Section 4 of the Programmer's Reference Manual.

Entry: ssu_$abort_line

This entry is used to print an error message and abort the execution of the current request line. Additional information on interactive subsystem error handling is contained in the Programmer's Reference Manual.

USAGE

```
declare ssu_$abort_line entry () options (variable);

call ssu_$abort_line (sci_ptr, status_code, ioa_string, optional_args);
```

ARGUMENTS

sci_ptr
: is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input) It must be an aligned (unpacked) pointer.

status_code
: is the status code for printing a message from an error table. (Input) If it is zero, no error table message is printed. It can be any datatype which can be converted to fixed bin(35). This argument is optional.

ioa_string
: is an ioa_ control string used to generate the user message portion of the message to be printed. (Input, optional) It can be a varying or nonvarying character string. If it is not present, no user message is printed.

optional_args
: are arguments to be substituted into the ioa_ control string. (Input, optional) They can be of any type required by the control string.

NOTES

This is a replaceable procedure. See the Programmer's Reference Manual for information on the use of replaceable procedures within interactive subsystems.

In a standalone invocation, a call to this entry point is translated into a call to com_err_ or active_fnc_err_ as appropriate. See the Programmer's Reference Manual for information on the use of standalone subsystem invocations.

The format of the message is as follows:

```
     subsystem_name (request_name): Code message User message
or:
     system_name: Code message User message
```

The second form (without the request name) is used when the call is made when no request is currently being executed, such as when it is called by the subsystem command procedure, rather than a request procedure.

The "Code message" is the error message associated with the status code. If the code argument is omitted or if its value is zero, the "Code message" is omitted, and only the "User message" is printed. The "User message" is formed by the appropriate substitutions in the ioa_ control string.

Entry: ssu_$abort_subsystem

This entry is used to abort the current invocation of the subsystem, and optionally print an error message. Additional information on interactive subsystem error handling is contained in the Programmer's Reference Manual.

USAGE

```
declare ssu_$abort_subsystem entry () options (variable);

call ssu_$abort_subsystem (sci_ptr, status_code, ioa_string,
      optional_args);
```

ARGUMENTS

sci_ptr
: is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input) It must be an aligned (unpacked) pointer.

status_code
: is the status code for printing a message from an error table. (Input, optional) If it is zero or omitted, no error table message is printed. It can be any datatype which can be converted to fixed bin(35).

ioa_string
: is an ioa_ control string which will be used to generate the user message portion of the message to be printed. (Input, optional) It can be a varying or nonvarying character string. If it is not present, no user message is printed.

optional_args
: are arguments to be substituted into the ioa_ control string. (Input, optional) They can be of any type required by the control string.

NOTES

This is a replaceable procedure. See the Programmer's Reference Manual for information on the use of replaceable procedures within interactive subsystems.

In a standalone invocation, a call to this entrypoint is translated into a call to com_err_ or active_fnc_err_ as appropriate. See the Programmer's Reference Manual for information on the use of standalone subsystem invocations.

The format of the message is as follows:

```
     subsystem_name (request_name): Code message User message
or:
     subsystem_name: Code message User message
```

The second form (without the request name) is used when the call is made when no request is currently being executed, such as when it is called by the subsystem command procedure, rather than a request procedure.

The "Code message" is the error message associated with the status code. If the code argument is omitted or if its value is zero, the "Code message" is omitted, and only the "User message" is printed. The "User message" is formed by the appropriate substitutions in the ioa_ control string.

Entry: ssu_$add_info_dir

This entry adds a new directory at the specified location in the list of info directories being searched by this subsystem invocation. Additional information on interactive subsystem self-documentation facilities is contained in the Programmer's Reference Manual.

USAGE

```
declare ssu_$add_info_dir entry (ptr, char(*), fixed bin, fixed
      bin(35));

call ssu_$add_info_dir (sci_ptr, info_dir, position, code);
```

ARGUMENTS

sci_ptr
: is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

info_dir
: is the pathname of the directory to be added to the list of info directories for this invocation. (Input)

position
: is the position in the list where the info dir is to be added. (Input) It can be any positive integer. The new directory will become the Nth entry in the list of info directories (i.e., it is added before the directory already present in position N). To add a directory at the end, position should be specified as a large number, such as 100000, which will guarantee its being added after the last info directory.

code
is a storage system status code. (Output) If it is zero, the directory was valid and was added; otherwise, it indicates the nature of difficulty associated with the directory.

NOTES

This entry point validates the existence of the specified info directory, and refuses to add it, returning a nonzero status code, unless it is valid. The user must have "s" access to the directory in order to add it as an info directory.

Entry: ssu_$add_request_table

This entry adds a new request table at the specified location in the list of request tables being searched by this subsystem invocation. Addition information on the use of interactive subsystem request tables is provided in the Programmer's Reference Manual.

USAGE

```
declare ssu_$add_request_table entry (ptr, ptr, fixed bin, fixed
      bin(35));

call ssu_$add_request_table (sci_ptr, request_table_ptr, position,
      code);
```

ARGUMENTS

sci_ptr
is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

request_table_ptr
is a pointer to a valid subsystem request table, to be added to the list searched for this invocation. (Input)

position
is the position in the list where the request table is to be added. (Input) It can be any positive integer. The new request table will be added as the Nth entry in the request tables list (i.e., it is added after the request table already present in position N. To add a request table at the end, position should be specified as a large number, such as 100000, which will guarantee its being added after the last request table.

code
is a status code. (Output) If it is zero, the table was valid and was added; or ssu_et_$invalid_request_table if it was not.

NOTES

This entry point validates the existence and validity of the specified request table, and refuses to add it, returning a nonzero status code, unless it is valid.

Entry: ssu_$apply_request_util

This entry is a utility procedure for implementing subsystem "apply" requests. The apply request is defined to create a Multics command line out of some or all of its request arguments, concatenate the pathname of a segment containing the specified object in the subsystem, and call the command processor. It can be used, for instance, to allow a user to edit a text file with the editor of her choice, with a request like "apply ted -pathname", which would be passed to the command processor as:

```
ted -pathname PATHNAME_OF_TEMP_SEG
```

If the apply request can take arguments which are meaningful to the subsystem, they must all come before the first argument which is to become part of the command line. It is recommended that the syntax of the apply request be designed so that the first argument which is not a control argument is taken as the beginning of the command line; its index will be passed as first_command_arg below. The temp_seg_ptr should point to a segment on which the bitcount can be set, or which is already set. It is recommended that the segment used for this purpose be obtained by calling ssu_$get_temp_segment.

This entry returns no error code; rather, since it is only useful in implementing the apply request, it simply calls ssu_$abort_line if it encounters any serious errors, and prints a more informative message than could otherwise have been described by an error code.

USAGE

```
declare ssu_$apply_request_util entry (ptr, fixed bin, ptr,
      fixed bin(21), fixed bin(21));

call ssu_$apply_request_util (sci_ptr, first_command_arg, temp_seg_ptr,
      input_lth, output_lth);
```

ARGUMENTS

sci_ptr
: is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

first_command_arg
: is the index of the first request argument which is to become part of the command line. (Input) If the subsystem apply request accepts no subsystem arguments, this should be 1; otherwise, it should be the index of the first non-control argument to the apply request.

temp_seg_ptr
is a pointer to the segment containing the data to be manipulated by the command line. (Input) Its pathname will be determined, and concatenated onto the end of the command line.

input_lth
is the length, in characters, of the data in the segment, or -1. (Input) If it is non-negative, the bitcount of the segment is set to nine times input_lth before the command line is executed; otherwise, the bitcount is not altered. If the bitcount is set to correspond to input_lth, it will be restored to its previous value after the command line has been executed and after output_lth has been set to reflect its value.

output_lth
is the length, in characters, (derived from the bitcount) of the data in the segment, after the command line has been executed. (Output)

Entry: ssu_$arg_count

This entry is used to determine how many arguments a subsystem request received.

USAGE

```
declare ssu_$arg_count entry (ptr, fixed bin);

call ssu_$arg_count (sci_ptr, arg_count);
```

ARGUMENTS

sci_ptr
is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

arg_count
is the number of arguments the request received. (Output)

NOTES

This entry point should only be used by requests which can not be invoked as an active request. If called by an active request, this entrypoint will abort the request line with the message:

```
This request can not be used as an active function.
```

This is a replaceable procedure. See the Programmer's Reference Manual for information on the use of replaceable procedures within interactive subsystems.

In a standalone procedure, a call to this entry point is translated into a call to cu_$arg_count. See the Programmer's Reference Manual for information on the use of standalone subsystem invocations.

Entry: ssu_$arg_list_ptr

This entry can be used by a subsystem request to get a pointer to its request argument list. This argument list is identical to that supplied to a Multics command by the command processor: a series of nonvarying character arguments followed by a varying character string argument if the request is invoked as an active request.

The argument list can be manipulated with calls to cu_$arg_count_rel, cu_$arg_ptr_rel, and cu_$af_return_arg_rel, which are equivalent to ssu_$arg_count, ssu_$arg_ptr, and ssu_$return_arg for this application.

USAGE

```
declare ssu_$arg_list_ptr entry (ptr, ptr);

call ssu_$arg_list_ptr (sci_ptr, arg_list_ptr);
```

ARGUMENTS

sci_ptr
: is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

arg_list_ptr
: is a pointer to the request argument list. (Output)

NOTES

This is a replaceable procedure. See the Programmer's Reference Manual for information on the use of replaceable procedures within interactive subsystems.

Entry: ssu_$arg_ptr

This entry is used by the procedure implementing a subsystem request to access its arguments.

USAGE

```
declare ssu_$arg_ptr entry (ptr, fixed bin, ptr, fixed bin(21));

call ssu_$arg_ptr (sci_ptr, arg_index, arg_ptr, arg_lth);
```

ARGUMENTS

sci_ptr
is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

arg_index
is the index of the argument to be accessed. It must be between one and the number of arguments supplied to the request. (Input)

arg_ptr
is a pointer to the selected argument, as a character string. (Output)

arg_lth
is the length of the selected argument, as a character string. (Output)

NOTES

If asked for an argument whose index exceeds the request's argument count, this entry point will abort the request line with the message:

```
Expected argument missing.
```

This is a replaceable procedure. See the Programmer's Reference Manual for information on the use of replaceable procedures within interactive subsystems.

In a standalone invocation, a call to this entry point is translated into a call to cu_$arg_ptr or cu_$af_arg_ptr. See the Programmer's Reference Manual for information on the use of standalone subsystem invocations.

Entry: ssu_$create_invocation

This entry is used to create an invocation of a subsystem. The subsystem invocation must later be destroyed by a call to ssu_$destroy_invocation. Additional information on interactive subsystems is provided in the Programmer's Reference Manual.

USAGE

```
declare ssu_$create_invocation entry (char(*), char(*), ptr, ptr,
      char(*), ptr, fixed bin(35));

call ssu_$create_invocation (subsystem_name, version_string, info_ptr,
      request_table_ptr, info_directory, sci_ptr, code);
```

ARGUMENTS

subsystem_name
: is the name of the subsystem. (Input) This name is used in error messages, in the output of the "." request (if any), as the default prompt, and as the default exec_com suffix.

version_string
: is the name of the current version of the subsystem, such as "4.3j". (Input) It is used in the output of the "." request.

info_ptr
: is a pointer to the invocation info structure specific to this subsystem. (Input) It points to a data structure containing all the information which must be passed between the command procedure and the request procedures for this invocation.

request_table_ptr
: is a pointer to the request table used for this subsystem, or null. (Input) If it is null, there are no request tables for the subsystem invocation, and if any are desired, they must be added by calls to ssu_$add_request_table. At least one request table is required for processing of any requests.

info_directory
: is the name of a directory in which the help and list_help requests (if any) will look for info files on the subsystem. (Input) If this is the null string, and no info directories are later added by calling ssu_$add_info_directory, the help and list_help requests will not operate.

sci_ptr
: is a pointer to the subsystem control structure created for this invocation. (Output)

code
: is a status code; if it is nonzero, the subsystem invocation could not be created, and processing should not continue. (Output)

Entry: ssu_$delete_info_dir

This entry is used to delete a directory from the list of info directories being searched. The specified info directory must be in the list. In order to avoid confusion about pathnames, the comparison is done by filesystem unique ID, if that can be determined, otherwise by literal pathname. If the directory is not present, a nonzero status code is returned.

USAGE

```
declare ssu_$delete_info_dir entry (ptr, char(*), fixed bin(35));

call ssu_$delete_info_dir (sci_ptr, info_dir, code);
```

ARGUMENTS

sci_ptr
is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

info_dir
is the name of the directory to be deleted. (Input)

code
is a status code. (Output) It is zero if the specified info directory was found in the list, or error_table_$noentry if not.

Entry: ssu_$delete_request_table

This entry is used to delete a request table from the list of tables being searched. The specified request table must be in the list. If it is not present, a nonzero status code is returned.

USAGE

```
declare ssu_$delete_request_table entry (ptr, ptr, fixed bin(35));

call ssu_$delete_request_table (sci_ptr, request_table_ptr, code);
```

ARGUMENTS

sci_ptr
is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

request_table_ptr
is a pointer to the request table to be deleted. (Input) It is not checked for validity, or whether it points to a valid request table, so that invalid request table pointers can be removed from the list.

code
is a status code. It is zero if the specified request table pointer is found in the list, or ssu_et_$request_table_not_found if not. (Output)

Entry: ssu_$destroy_invocation

This entry is used to destroy a subsystem invocation created by a previous call to ssu_$create_invocation or ssu_$standalone_invocation.

USAGE

```
declare ssu_$destroy_invocation entry (ptr);

call ssu_$destroy_invocation (sci_ptr);
```

ARGUMENTS

sci_ptr
: is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

NOTES

If sci_ptr is null on input, the call is ignored. This entry point sets sci_ptr to null after destroying the invocation.

Entry: ssu_$evaluate_active_string

This entry is used to interpret a single active request string. It is the subsystem equivalent to cu_$evaluate_active_string.

USAGE

```
declare ssu_$evaluate_active_string entry (ptr, ptr, char(*), fixed bin,
      char(*) varying, fixed bin(35));

call ssu_$evaluate_active_string (sci_ptr, rp_options_ptr,
      active_string, string_type, return_value, code);
```

ARGUMENTS

sci_ptr
: is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

rp_options_ptr
: if null, specifies that the request processor options in effect for this subsystem are used. Otherwise, it locates an rp_options structure defining the options to be used to evaluate the active string.

active_string
: is the active string to be evaluated. (Input) It should not include the outermost brackets.

string_type
: specifies the type of active string to be evaluated. (Input) It must be one of the following values defined in the include file cp_active_string_types.incl.pl1:

NORMAL_ACTIVE_STRING
: the active string return value should be rescanned for all command processor constructs. ([...])

TOKENS_ONLY_ACTIVE_STRING
: the active string return value should be rescanned only for whitespace and quotes. (|[...])

ATOMIC_ACTIVE_STRING
: the active string return value should not be rescanned. (||[...])

return_value
: is the result of the evaluation of the active string. (Output)

code
: is a standard status code. If the standard evaluate_active_string procedure is being used, it will have one of the following values; if a user supplied procedure is in use, the list can be different. (Output)

0
: indicates that the active string was successfully evaluated.

error_table_$command_line_overflow
: indicates that the return value of the active string was too large to fit in the supplied return_value argument; as much as would fit is returned, however.

ssu_et_$request_line_aborted
: indicates that evaluation of the active string was terminated by a call to ssu_$abort_line. This usually indicates that an error was encountered by one of the active requests; however, the error message has already been printed, so no message should be printed by the caller of ssu_$evaluate_active_string.

ssu_et_$subsystem_aborted
: indicates that evaluation of the active string was terminated by a call to ssu_$abort_subsystem; this generally indicates that the subsystem should be terminated, and no further processing be done. In any case, no error message should be printed.

anything else
: indicates a serious error condition occurred while trying to evaluate the active string.

NOTES

This is a replaceable procedure. See the Programmer's Reference Manual for information on the use of replaceable procedures within interactive subsystems.

In a standalone invocation, a call to this entrypoint is translated into a call to cu_$evaluate_active_string. See the Programmer's Reference Manual for information on the use of standalone subsystem invocations.

Entry: ssu_$execute_line

This entry is used to intepret a single request line.

USAGE

```
declare ssu_$execute_line entry (ptr, ptr, fixed bin(21), fixed
      bin(35));

call ssu_$execute_line (sci_ptr, request_line_ptr, request_line_lth,
      code);
```

ARGUMENTS

sci_ptr
: is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

request_line_ptr
: is a pointer to the request line to be executed. (Input)

request_line_lth
: is the length of the request line, in characters. (Input)

code
: is a standard status code. (Output) If the default execute_line procedure is being used, it can have one of the following values; these can be different if a user procedure is being used for this function.

  0
  : the request line was executed successfully, and returned normally.

  ssu_et_$null_request_line
  : the request line was "blank", in the command_processor_ sense, i.e., contained no requests, iteration or brackets, but was merely a mixture of whitespace and semi-colons.

  ssu_et_$request_line_aborted
  : the request line was terminated during its execution by a call to ssu_$abort_line. This usually indicates that an error was encountered by one of the requests; however, the error message has already been printed, so no message should be printed by the caller of ssu_$execute_line.

  ssu_et_$subsystem_aborted
  : the request line was terminated normally by a call to ssu_$abort_subsystem; this generally indicates that the subsystem should be terminated, and no further processing be done. In any case, no error message should be printed.

  ssu_$program_interrupt
  : the request line was terminated by the user interrupting its execution and using the program interrupt command. The caller of this entry point need not print a message.

  anything else
  : indicates a serious error condition occurred while trying to execute the request line.

NOTES

This is a replaceable procedure. See the Programmer's Reference Manual for information on the use of replaceable procedures within interactive subsystems.

In a standalone invocation, a call to this entry point is translated into a call to cu_$cp. See the Programmer's Reference Manual for information on the use of standalone subsystem invocations.

Entry: ssu_$execute_start_up

This entry point executes the current subsystem's start_up exec_com.

USAGE

```
declare ssu_$execute_start_up entry () options (variable);

call ssu_$execute_start_up (sci_ptr, code, optional_ec_args);
```

ARGUMENTS

sci_ptr
: is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

code
: is a standard system status code. (Output) It can have one of the following values:
  0
  : the start_up exec_com was executed successfully.
  error_table_$noentry
  : there is no start_up exec_com for this subsystem.
  ssu_et_$exec_com_aborted
  : execution of the exec_com was abnormally terminated.
  ssu_et_$subsystem_aborted
  : execution of the exec_com was abnormally terminated by a call to ssu_$abort_subsystem.

optional_ec_args
: are optional arguments to be passed to the start_up exec_com. (Input) These arguments must be either nonvarying unaligned or varying aligned character strings.

NOTES

The subsystem's start_up exec_com is a segment named "start_up.ec_suffix" where ec_suffix is the subsystem's exec_com suffix. See ssu_$set_ec_suffix for a description of how to change the suffix.

This entrypoint searches for the start_up exec_com first in the the user's home directory, then in the user's project directory >udd>Project_id, and last in >site The first exec_com found, if any, is used.

Entry: ssu_$execute_string

This entry is used to execute a request string, usually expressed as an in-line constant or character string variable. It is provided only as a utility function which allows the execution of character strings as strings, rather than by pointer and length. It is implemented by a call to ssu_$execute_line; therefore, if ssu_$execute_line is changed, ssu_$execute_string will change in exactly the same way.

USAGE

```
declare ssu_$execute_string entry (ptr, char(*), fixed bin(35));

call ssu_$execute_string (sci_ptr, request_string, code);
```

ARGUMENTS

sci_ptr
: is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

request_string
: is a character string containing the requests to be executed. (Input)

code
: is a status code. (Output) It can have any of the values specified for ssu_$execute_line. The list of error codes is only for the default procedure, and can be different if a user procedure is substituted.

Entry: ssu_$get_area

This entry is used to obtain an area for use by the subsystem invocation. It calls the define_area_ subroutine to obtain an area in a temporary segment. The difference between using this entry and calling define_area_ directly is that areas acquired by calling ssu_$get_area are released when the subsystem invocation is destroyed, regardless of whether the user program had freed them earlier. Areas acquired by calling ssu_$get_area should be released by calling ssu_$release_area.

USAGE

```
declare ssu_$get_area entry (ptr, ptr, char(*), ptr);

call ssu_$get_area (sci_ptr, area_info_ptr, comment, area_ptr);
```

ARGUMENTS

sci_ptr
is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

area_info_ptr
is a pointer to an area_info structure (defined in area_info.incl.pl1) describing the area, or null. (Input) If the area_info_ptr is null, an area with default characteristics is defined; see below for details. Only the area_control flags in the area_info are used by ssu_$get_area; the area is always put in a temporary segment. The area pointer is returned as the final argument to ssu_$get_area; it is not written into the area_info structure.

comment
is a comment identifying the use to which the area will be put. (Input) It is used in constructing the owner name for the call to define_area_, in the form:

```
subsys_name.N (comment)
```

where subsys_name is the name of the subsystem, N is the invocation level for this invocation, and the comment (if any) follows, in parentheses. This is done to make it easier to identify the segment names listed by list_temp_segments.

area_ptr
is a pointer to the area. (Output) It will always be valid; if for some reason the area cannot be acquired, the current request line (or subsystem invocation, if there is no request line) is aborted with an appropriate message. No errors are ever reflected back to the caller of ssu_$get_area.

NOTES

If the area_info_ptr supplied to ssu_$get_area is null, an area with default characteristics is created. The area is extensible, initially one segment long, and is zero_on_free (but not zero_on_alloc). All other area_control flags are off.

If the subsystem is in "debug mode" (see description of ssu_$set_debug_mode), all areas (both user-specified and default) are created with the dont_free attribute.

Entry: ssu_$get_debug_mode

This entry is used to get the current state of debug mode. Debug mode controls several features intended only as an aid to debugging. A description of interactive subsystem debug mode is provided in the Programmer's Reference Manual.

USAGE

```
declare ssu_$get_debug_mode entry (ptr) returns (bit(1) aligned);

debug_mode = ssu_$get_debug_mode (sci_ptr);
```

ARGUMENTS

sci_ptr
: is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

debug_mode
: is the current debug mode, either on or off. (Output)

Entry: ssu_$get_default_procedure

This entry is used to get the default value for a replaceable procedure value. The value returned is the procedure which is called to perform the specified function if no calls to ssu_$set_procedure are ever made for that procedure.

USAGE

```
declare ssu_$get_default_procedure entry (ptr, char(*), entry, fixed
      bin(35));

call ssu_$get_default_procedure (sci_ptr, procedure_name,
      procedure_value, code);
```

ARGUMENTS

sci_ptr
: is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

procedure_name
: is the name of the procedure the value of which is to be returned. (Input)

procedure_value
: is the default value of the specified replaceable procedure value. (Output)

code
is a status code. (Output) It can have the following values:
0
success.
error_table_$noentry
procedure_name is not an acceptable value.

NOTES

See the Programmer's Reference Manual for the currently defined list of replaceable procedure names.

Entry: ssu_$get_default_rp_options

This entrypoint returns the default request processor options for the current subsystem.

USAGE

```
declare ssu_$get_default_rp_options entry (ptr, char(8), ptr,
      fixed bin(35));

call ssu_$get_default_rp_options (sci_ptr, version_wanted,
      rp_options_ptr, code);
```

ARGUMENTS

sci_ptr
is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

version_wanted
specifies which version of the rp_options structure is to be returned by this procedure. (Input) This argument must have the value of the named constant RP_OPIONS_VERSION_1 defined in the system include file ssu_rp_options.incl.pl1.

rp_options_ptr
is a pointer to an rp_options structure previously allocated by the caller which is to be filled in by this entrypoint. (Input) This structure is declared in the system include file ssu_rp_options.incl.pl1.

code
is a standard system status code. (Output) It can have one of the following values:
0
the structure was successfully filled in.
error_table_$unimplemented_version
the caller requested an unrecognized version of the rp_options structure.

NOTES

See the information on the subsystem request language in the Programmer's Reference Manual for a description of the rp_options structure and the request processor options mechanism.

This is a replaceable procedure. See the Programmer's Reference Manual for information on the use of replaceable procedures within interactive subsystems.

Entry: ssu_$get_ec_search_list

This entry returns the name of the search list currently being used to find subsystem exec_com files. By default, no search list is used; exec_coms must be specified by full pathname, and this value is a null string. See also the description of ssu_$set_ec_search_list.

USAGE

```
declare ssu_$get_ec_search_list entry (ptr) returns (char(32));

search_list_name = ssu_$get_ec_search_list (sci_ptr);
```

ARGUMENTS

sci_ptr
: is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

search_list_name
: is the name of the current exec_com search list or a null string if no search list is used by this subsystem invocation. (Return)

Entry: ssu_$get_ec_subsystem_ptr

This entry returns the pointer currently used to implement the "referencing_dir" rule in the search list for subsystem exec_coms. By default, this pointer is null, meaning that the referencing_dir rule has no effect, even if the exec_com search list name is non-null. See also the description of ssu_$set_ec_subsystem_ptr.

USAGE

```
declare ssu_$get_ec_subsystem_ptr entry (ptr) returns (ptr);

subsystem_ptr = ssu_$get_ec_subsystem_ptr (sci_ptr);
```

ARGUMENTS

sci_ptr
is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

subsystem_ptr
is a pointer to some segment in the directory being used as the subsystem's exec_com referencing_dir or null if no such directory is being used. (Output)

Entry: ssu_$get_ec_suffix

This entry returns the suffix currently being used for subsystem exec_com files. By default, this string is the subsystem name. See also the description of ssu_$set_ec_suffix.

USAGE

```
declare ssu_$get_ec_suffix entry (ptr) returns (char(32));

suffix_string = ssu_$get_ec_suffix (sci_ptr);
```

ARGUMENTS

sci_ptr
is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

suffix_string
is the current exec_com suffix for this subsystem invocation. (Output)

Entry: ssu_$get_info_ptr

This entry is used to get the info_ptr for this subsystem invocation. Normally, this value is otherwise available, either as a request parameter, or as a variable in the command procedure of the subsystem. This entry is only useful in subroutines which are passed only the sci_ptr and not the info_ptr as parameters, such as user supplied abort procedures. Additional information on the use of sci_ptr and info_ptr is provided in the description of interactive subsystems in the Programmer's Reference Manual.

USAGE

```
declare ssu_$get_info_ptr entry (ptr) returns (ptr);

info_ptr = ssu_$get_info_ptr (sci_ptr);
```

ARGUMENTS

sci_ptr
: is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

info_ptr
: is the info_ptr for this subsystem invocation. (Output)

Entry: ssu_$get_invocation_count

This entry is used to determine the invocation index of the current subsystem invocation, and also determine how many invocations of the subsystem are currently active.

USAGE

```
declare ssu_$get_invocation_count entry (ptr, fixed bin, fixed bin);

call ssu_$get_invocation_count (sci_ptr, this_level, max_level);
```

ARGUMENTS

sci_ptr
: is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

this_level
: is the invocation index for this subsystem invocation. (Output)

max_level
: is the invocation index for the highest numbered subsystem invocation presently active. (Output)

Entry: ssu_$get_level_n_sci_ptr

This entry is used to examine the state of other invocations of the subsystem by returning the info_ptr and sci_ptr for the other invocation. If the level index specifies an invocation which does not exist, the info_ptr and sci_ptr are returned as null. Additional information on the use of info_ptr and sci_ptr is provided in the description of interactive subsystems in the Programmer's Reference Manual.

USAGE

```
declare ssu_$get_level_n_sci_ptr entry (ptr, fixed bin, ptr, ptr);

call ssu_$get_level_n_sci_ptr (sci_ptr,level_index, other_sci_ptr,
     other_info_ptr);
```

ARGUMENTS

sci_ptr
: is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

level_index
: is the index (invocation number, recursion level) of the other invocation of the subsystem for which information is desired.

other_sci_ptr
: is the sci_ptr for the specified invocation of the subsystem. (Output)

other_info_ptr
: is the info_ptr for the specified invocation of the subsystem. (Output)

Entry: ssu_$get_prev_sci_ptr

This entry is used to examine the state of other invocations of the subsystem by returning the info_ptr and sci_ptr for the immediately previous invocation. If there is no previous invocation of the subsystem, the sci_ptr and info_ptr are returned as null. Additional information on the use of the info_ptr and sci_ptr is provided the description of interactive subsystems in the Programmer's Reference Manual.

USAGE

```
declare ssu_$get_prev_sci_ptr entry (ptr, ptr, ptr);

call ssu_$get_prev_sci_ptr (sci_ptr, previous_sci_ptr,
     previous_info_ptr);
```

ARGUMENTS

sci_ptr
: is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

previous_sci_ptr
: is the sci_ptr for previous invocation of the subsystem. (Output)

previous_info_ptr
: is the info_ptr for previous invocation of the subsystem. (Output)

Entry: ssu_$get_procedure

This entry is used to get the current value for a replaceable procedure value in the specified subsystem invocation. The value returned is the procedure which is called to perform the specified function.

USAGE

```
declare ssu_$get_procedure entry (ptr, char(*), entry, fixed bin(35));

call ssu_$get_procedure (sci_ptr, procedure_name, procedure_value,
      code);
```

ARGUMENTS

sci_ptr
: is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

procedure_name
: is the name of the procedure for which the value is to be returned. (Input)

procedure_value
: is the current value of the specified replaceable procedure value. (Output)

code
: is a status code. (Output) It can have the following values:
  0
  : success.
  error_table_$noentry
  : procedure_name is not an acceptable value.

NOTES

See the Programmer's Reference Manual for the currently defined list of replaceable procedure names.

Entry: ssu_$get_prompt

This entry is used to get the string currently being used as a prompt. See the description of request loops for interactive subsystems in the Programmer's Reference Manual for additional information on prompts.

USAGE

```
declare ssu_$get_prompt entry (ptr) returns (char(64) varying);

prompt_string = ssu_$get_prompt (sci_ptr);
```

ARGUMENTS

sci_ptr
: is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

prompt_string
: is the current prompt string. (Output)

Entry: ssu_$get_prompt_mode

This entry is used to get the current state of prompting in the subsystem. See the description of request loops for interactive subsystems in the Programmer's Reference Manual for additional information on prompts.

USAGE

```
declare ssu_$get_prompt_mode entry (ptr) returns (bit(36) aligned);

prompt_mode = ssu_$get_prompt_mode (sci_ptr);
```

ARGUMENTS

sci_ptr
: is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

prompt_mode
: is the current prompt mode. (Output) The individual bits are interpreted as follows:

bit 1 ON	suppress all prompts.
bit 2 ON	prompt after blank request lines.
bit 3 ON	suppress prompts if there is type ahead.

There are named constants in the ssu_prompt_modes.incl.pl1 include file that may be used to examine this return value.

Entry: ssu_$get_ready_mode

This entry is used to determine the current state of ready processing. See the description of request loops for interactive subsystems in the Programmer's Reference Manual for additional information on ready processing.

USAGE

```
declare ssu_$get_ready_mode entry (ptr) returns (bit(1) aligned));

enable_sw = ssu_$get_ready_mode (sci_ptr);
```

ARGUMENTS

sci_ptr
    is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

enable_sw
    is a bit indicating whether or not ready processing is enabled. (Output) If it is "1"b, ready processing is being performed; if it is "0"b, ready processing is not being performed.

Entry: ssu_$get_request_name

This entry is used to determine the primary name (from the request table) of the subsystem request currently being executed. If it is the null string, no request is currently being executed.

USAGE

```
declare ssu_$get_request_name entry (ptr) returns (char(32));

request_name = ssu_$get_request_name (sci_ptr);
```

ARGUMENTS

sci_ptr
    is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

request_name
    is the name of the request currently being executed, or the null string. (Output)

Entry: ssu_$get_request_processor_options

This entrypoint returns the request processor options presently in effect for the current subsystem.

USAGE

```
declare ssu_$get_request_processor_options entry (ptr, char(8), ptr,
      fixed bin(35));

call ssu_$get_request_processor_options (sci_ptr, version_wanted,
      rp_options_ptr, code);
```

ARGUMENTS

sci_ptr
: is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

version_wanted
: specifies which version of the rp_options structure is to be returned by this procedure. (Input) This argument must have the value of the named constant RP_OPIONS_VERSION_1 defined in the system include file ssu_rp_options.incl.pl1.

rp_options_ptr
: is a pointer to an rp_options structure previously allocated by the caller which is to be filled in by this entry point. (Input) This structure is declared in the system include file ssu_rp_options.incl.pl1.

code
: is a standard system status code. (Output) It can have one of the following values:

  0
  : the structure was successfully filled in.

  error_table_$unimplemented_version
  : the caller requested an unrecognized version of the rp_options structure.

NOTES

See the information on the subsystem request language in the Programmer's Reference Manual for a description of the rp_options structure and the request processor options mechanism.

This is a replaceable procedure. See the Programmer's Reference Manual for information on the use of replaceable procedures within interactive subsystems.

Entry: ssu_$get_subsystem_and_request_name

This entry is used to acquire a string identifying the subsystem and the current request, suitable for use in printing error messages or asking questions. If no request is currently being executed, the name returned is the name of the subsystem; otherwise, it has the following format:

```
subsystem_name (request_name)
```

USAGE

```
declare ssu_$get_subsystem_and_request_name entry (ptr) returns
      (char (72) varying);

name = ssu_$get_subsystem_and_request_name (sci_ptr);
```

ARGUMENTS

sci_ptr
: is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

name
: is the name of the subsystem and request currently being executed, as described above. (Output)

NOTES

This is a replaceable procedure. See the *Programmer's Reference Manual,* Order No. AG91, for information on the use of replaceable procedures within interactive subsystems.

Entry: ssu_$get_subsystem_name

This entry is used to determine the name, supplied in the call to ssu_$create_invocation or ssu_$standalone_invocation, of the subsystem owning the specified invocation.

USAGE

```
declare ssu_$get_subsystem_name entry (ptr) returns (char (32));

subsystem_name = ssu_$get_subsystem_name (sci_ptr);
```

ARGUMENTS

sci_ptr
is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

subsystem_name
is the name of the subsystem owning the current invocation. (Output)

Entry: ssu_$get_subsystem_version

This entry is used to determine the version number of the subsystem which was supplied as a parameter in the call to ssu_$create_invocation or ssu_$standalone_invocation.

USAGE

```
declare ssu_$get_subsystem_version entry (ptr) returns (char(32));

subsystem_version = ssu_$get_subsystem_version (sci_ptr);
```

ARGUMENTS

sci_ptr
is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

subsystem_version
is the version number of the subsystem owning the current invocation. (Output)

Entry: ssu_$get_temp_segment

This entry is used to obtain a temporary segment for use by the current subsystem invocation. It calls get_temp_segment_ to acquire the segment. The difference between using this entry and calling get_temp_segment_ directly is that segments acquired by calling ssu_$get_temp_segment are released when the subsystem invocation is destroyed, regardless of whether the user program had freed them earlier. Segments acquired by calling ssu_$get_temp_segment should be released by calling ssu_$release_temp_segment.

USAGE

```
declare ssu_$get_temp_segment entry (ptr, char(*), ptr);

call ssu_$get_temp_segment (sci_ptr, comment, temp_seg_ptr);
```

ARGUMENTS

sci_ptr
: is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

comment
: is a comment identifying the use to which the segment will be put. (Input) It is used in constructing the owner name for the call to get_temp_segment_, in the form:

```
subsystem_name.N (comment)
```

where subsys_name is the name of the subsystem, N is the invocation level for this invocation, and the comment (if any) follows, in parentheses. This is done to make it easier to identify the segment names listed by list_temp_segments.

temp_seg_ptr
: is a pointer to the temporary segment. (Output) It will always be valid. If for some reason a temporary segment cannot be acquired, the current request line (or subsystem invocation, if there is no request line) is aborted with an appropriate message. No errors are ever reflected back to the caller of ssu_$get_temp_segment.

Entry: ssu_$list_info_dirs

This entry is used to obtain a list of the info directories currently in use by this subsystem invocation.

USAGE

```
declare ssu_$list_info_dirs entry (ptr, ptr, fixed bin, ptr, fixed
      bin(35));

call ssu_$list_info_dirs (sci_ptr, area_ptr, info_dirs_list_version,
      idl_ptr, code);
```

ARGUMENTS

sci_ptr
: is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

area_ptr
: is a pointer to an area in which the returned list of info directories can be allocated. (Input)

info_dirs_list_version
is the version of the info_dirs_list structure which the caller expects. The only version of this structure which can currently be requested is INFO_DIRS_LIST_VERSION_1. (Input)

idl_ptr
is a pointer to the info_dirs_list structure described below allocated by this entrypoint. (Output)

code
is a standard system status code. If the version of the info_dirs_list structure requested is not supported, this value will be error_table_$unimplemented_version. (Output)

NOTES

The info_dirs_list structure and the named constant INFO_DIRS_LIST_VERSION_1 are defined in the include file ssu_info_dirs_list.incl.pl1.

The info_dirs_list structure is defined as follows:

```
dcl 1 info_dirs_list   aligned based (idl_ptr),
      2 header,
        3 version           fixed bin,
        3 n_info_dirs       fixed bin,
      2 info_dirs           (0 refer (info_dirs_list.n_info_dirs)),
        3 info_dirname      char(168) unaligned,
        3 uid               bit(36),
        3 flags,
          4 info_dir_valid  bit(1) unaligned,
          4 pad             bit(35) unaligned;
```

STRUCTURE ELEMENTS

version
is the current version of this structure and has the value of the named constant INFO_DIRS_LIST_VERSION_1.

n_info_dirs
is the number of info directories in use by this subsystem invocation.

info_dirs(i).info_dirname
is the absolute pathname of this directory.

info_dirs(i).uid
is the file system unique ID of this directory if it can be determined; otherwise, it is "0"b.

info_dirs(i).info_dir_valid
is "1"b if this info directory is considered valid by the subsystem utilities; otherwise, it is "0"b.

Entry: ssu_$list_request_tables

This entry is used to obtain a list of the request tables currently in use by this subsystem invocation.

USAGE

```
declare ssu_$list_request_tables entry (ptr, ptr, fixed bin, ptr, fixed
      bin(35));

call ssu_$list_request_tables (sci_ptr, area_ptr,
      request_tables_list_version, rtl_ptr, code);
```

ARGUMENTS

sci_ptr
is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

area_ptr
is a pointer to an area in which the returned list of request tables can be allocated. (Input)

request_tables_list_version
is the version of the request_tables_list structure which the caller expects. The only version of this structure which can currently be requested is REQUEST_TABLES_LIST_VERSION_1. (Input)

rdl_ptr
is a pointer to the request_tables_list structure described below allocated by this entrypoint. (Output)

code
is a standard system status code. If the version of the request_tables_list structure requested is not supported, this value will be error_table_$unimplemented_version. (Output)

NOTES

The request_tables_list structure and the named constant REQUEST_TABLES_LIST_VERSION_1 are defined in the include file ssu_request_tables_list.incl.pl1.

```
dcl 1 request_tables_list        aligned based (rtl_ptr),
      2 header,
        3 version                fixed bin,
        3 n_tables               fixed bin,
      2 tables                   (0 refer (request_tables_list.n_tables)),
        3 table_ptr              ptr,
        3 flags,
          4 table_valid          bit(1) unaligned,
          4 pad                  bit(35) unaligned,
        3 pad                    bit(36);
```

STRUCTURE ELEMENTS

version
: is the current version of this structure and has the value of the named constant REQUEST_TABLES_LIST_VERSION_1.

n_tables
: is the number of request tables in use by this subsystem invocation.

tables(i).table_ptr
: is a pointer to this request table.

tables(i).table_valid
: is "1"b if this request table is considered valid by the subsystem utilities; otherwise, it is "0"b.

Entry: ssu_$listen

This entry implements the subsystem listener. The interactive subsystem listener is described in the Programmer's Reference Manual.

USAGE

```
declare ssu_$listen entry (ptr, ptr, fixed bin(35));

call ssu_$listen (sci_ptr, iocb_ptr, code);
```

ARGUMENTS

sci_ptr
: is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

iocb_ptr
: is a pointer to the IOCB for the I/O switch from which input lines are to be read. If it is null, the default I/O switch, user_input, is used. (Input)

code
: is a status code. It can never be zero for this entrypoint. If it is ssu_et_$subsystem_aborted, it indicates that the subsystem was exited normally, by a call to ssu_$subsystem_abort, and is not an error condition. Any other code indicates an error condition which prevented the listener from operating. This list of codes is only valid if the default listen procedure is being used; it can be different if a user listen procedure is in use. (Output)

NOTES

This is a replaceable procedure. See the Programmer's Reference Manual for information on the use of replaceable procedure within interactive subsystems.

Entry: ssu_$print_blast

This entry is used to print a "blast" message announcing a new version of the subsystem, if the user running the subsystem has not yet used the new version more than a certain number of times. This "threshold" value is provided to give the user several opportunities to find out what has been changed. See the description of interactive subsystem usage monitoring in the Programmer's Reference Manual for additional information.

USAGE

```
declare ssu_$print_blast entry (ptr, ptr, fixed bin, char(*) varying,
      fixed bin(35));

call ssu_$print_blast (sci_ptr, ref_ptr, threshold, blast_message,
      code);
```

ARGUMENTS

sci_ptr
is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

ref_ptr
is a pointer used as a referencing_dir argument in the call to hcs_$make_ptr to find the usage segment. It is usually codeptr of an entrypoint in the calling procedure. (Input)

threshold
is the maximum number of times to print a blast message announcing the new version for any particular user. (Input)

blast_message
is a blast message to be appended to the version announcement message. (Input) This might typically be a brief list of the changes in this version of the subsystem. If the blast_message is a null string, only the subsystem name and version number are printed (if anything is printed at all).

code
is a status code indicating whether the usage was successfully recorded. (Output) In general, the code should be ignored, since there is nothing useful the caller can do about it.

Entry: ssu_$print_message

This entry is used by a request procedure or the subsystem command procedure to print an informational, warning, or nonfatal error message.

USAGE

```
declare ssu_$print_message entry () options (variable);

call ssu_$print_message (sci_ptr, status_code, ioa_string,
     optional_args);
```

ARGUMENTS

sci_ptr
is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

status_code
is the status code for printing a message from an error table. (Input) If it is zero, no error table message is printed. It can be any datatype which can be converted to fixed bin(35).

ioa_string
is an ioa_ control string which is used to format the user message portion of the message to be printed. (Input, optional) It can be a varying or nonvarying character string. If it is not present, no user message is printed.

optional_args
are arguments to be substituted into the ioa_ control string. (Input, optional) They can be of any type required by the control string.

NOTES

This is a replaceable procedure. See the Programmer's Reference Manual for information on the use of replaceable procedures within interactive subsystems.

In a standalone invocation, a call to this entrypoint is translated into a call to com_err_ or active_fnc_err_ as appropriate. See the Programmer's Reference Manual for information on the use of standalone subsystem invocations.

The format of the message is as follows:

```
     subsystem_name (request_name): Code message User message
or:
   subsystem_name: Code message User message
```

The second form (without the request name) is used when the call is made when no request is currently being executed, such as when it is called by the subsystem command procedure, rather than a request procedure.

The "Code message" is the error message associated with the status code. If the code is zero, the "Code message" is omitted, and only the "User message" is printed. The "User message" is formed by the appropriate substitutions in the ioa_ control string.

Entry: ssu_$record_usage

This entry is used to make an entry in the usage segment to record a use of the subsystem, without printing a "blast" message. See the description of interactive subsystem usage monitoring in the Programmer's Reference Manual for additional information.

USAGE

```
declare ssu_$record_usage entry (ptr, ptr, fixed bin(35));

call ssu_$record_usage (sci_ptr, ref_ptr, code);
```

ARGUMENTS

sci_ptr
: is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

ref_ptr
: is a pointer used as a referencing_dir argument in the call to hcs_$make_ptr to find the usage segment. (Input) It is usually codeptr of an entrypoint in the calling procedure.

code
: is a status code indicating whether the usage was successfully recorded. (Output) In general, the code should be ignored, since there is nothing useful the caller can do about it.

Entry: ssu_$release_area

This entry is used to release an area previously obtained by a call to ssu_$get_area. It can only be used to release areas acquired for the specified subsystem invocation. It returns no error code because any errors encountered are simply ignored, and taken care of at the time the subsystem invocation is destroyed.

USAGE

```
declare ssu_$release_area entry (ptr, ptr);

call ssu_$release_area (sci_ptr, area_ptr);
```

ARGUMENTS

sci_ptr
: is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

area_ptr
: is a pointer to the area to be released. (Input)

Entry: ssu_$release_temp_segment

This entry is used to release a temporary segment previously acquired by a call to ssu_$get_temp_segment. It can only be used to release temporary segments acquired for the specified subsystem invocation. It returns no error code because any errors encountered are simply ignored, and taken care of at the time the subsystem invocation is destroyed.

USAGE

```
declare ssu_$release_temp_segment entry (ptr, ptr);

call ssu_$release_temp_segment (sci_ptr, temp_seg_ptr);
```

ARGUMENTS

sci_ptr
: is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

temp_seg_ptr
: is a pointer to the temporary segment to be released. (Input)

Entry: ssu_$reset_procedure

This entrypoint resets a replaceable procedure in the current subsystem to its default value. See the Programmer's Reference Manual for information on the use of replaceable procedures within interactive subsystems.

USAGE

```
declare ssu_$reset_procedure entry (ptr, char(*), fixed bin(35));

call ssu_$reset_procedure (sci_ptr, procedure_name, code);
```

ARGUMENTS

sci_ptr
: is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

procedure_name
: is the name of the replaceable procedure which is to be reset. (Input)

code
: is a standard system status code. (Output) It can have one of the following values:
: 0
: the replaceable procedure was successfully reset.
: error_table_$noentry
: the supplied procedure name is not the name of a replaceable procedure.

NOTES

See the Programmer's Reference Manual for the currently defined list of replaceable procedure names.

Entry: ssu_$reset_request_processor_options

This entrypoint resets the request processor options presently in effect for the current subsystem to their default values.

USAGE

```
declare ssu_$reset_request_processor_options entry (ptr);

call ssu_$reset_request_processor_options (sci_ptr);
```

ARGUMENTS

sci_ptr
is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation (Input)

NOTES

See the information on the subsystem request language in the Programmer's Reference Manual for a description of the request processor options mechanism.

This is a replaceable procedure. See the Programmer's Reference Manual for information on the use of replaceable procedures within interactive subsystems.

Entry: ssu_$return_arg

This entry is used by a subsystem request procedure to determine whether it has been invoked as an active request, and to get a pointer and length for its return value if so. The return string should be declared as:

```
dcl return_value char(rv_lth) varying based (rv_ptr);
```

USAGE

```
declare ssu_$return_arg entry (ptr, fixed bin, bit(1) aligned, ptr,
      fixed bin(21));

call ssu_$return_arg (sci_ptr, arg_count, af_sw, rv_ptr, rv_lth);
```

ARGUMENTS

sci_ptr
is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

arg_count
is the number of arguments supplied to this request, not including the final return value argument, if any. (Output)

af_sw
is a bit indicating whether the request was invoked as a command request or an active request. (Output) It is "1"b if an active request, and "0"b otherwise.

rv_ptr
is a pointer to the return string if af_sw is "1"b; otherwise, it is null. (Output)

rv_lth
is the maximum length of the return string. (Output)

NOTES

This is a replaceable procedure. See the Programmer's Reference Manual for information on the use of replaceable procedures within interactive subsystems.

Entry: ssu_$set_debug_mode

This entry is used to set debug mode for the subsystem. Information on the use of interactive subsystem debugging facilities is provided in the Programmer's Reference Manual.

USAGE

```
declare ssu_$set_debug_mode entry (ptr, bit(1) aligned);

call ssu_$set_debug_mode (sci_ptr, debug_mode);
```

ARGUMENTS

sci_ptr
is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

debug_mode
is the new value for debug mode, either on or off. (Input)

Entry: ssu_$set_ec_search_list

This entry is used to set the name of the search list used to find subsystem exec_com files. By default, no search list is used, and subsystem exec_coms must be specified by full pathname. This is the case if a null string is supplied. See also the descriptions of ssu_$set_ec_suffix and ssu_$set_ec_subsystem_ptr.

USAGE

```
declare ssu_$set_ec_search_list entry (ptr, char(32));

call ssu_$set_ec_search_list (sci_ptr, search_list_name);
```

ARGUMENTS

sci_ptr
: is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

search_list_name
: is the name of the search list to use for finding subsystem exec_coms in this subsystem invocation or a null string if no search list should be used in this subsystem invocation. (Input)

Entry: ssu_$set_ec_subsystem_ptr

This entry sets the directory used to implement the "referencing_dir" rule in the search list for subsystem exec_coms. By default, this pointer is null, meaning that the referencing_dir rule has no effect, even if the search list name is non-null. This value is ignored if there is no exec_com search list set or if the search list does not use the referencing_dir rule. The referencing_dir can be used as a sort of exec_com "subsystem directory" to contain standard subsystem exec_coms for this subsystem. See also the descriptions of ssu_$set_ec_suffix and ssu_$set_ec_search_path.

USAGE

```
declare ssu_$set_ec_subsystem_ptr entry (ptr, ptr);

call ssu_$set_ec_subsystem_ptr (sci_ptr, subsystem_ptr);
```

ARGUMENTS

sci_ptr
    is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

subsystem_ptr
    is a pointer to a segment which resides in the directory that will be used as the exec_com subsystem directory in this subsystem invocation or null if no directory is to be used. (Input) This pointer is only used in calls to hcs_$make_ptr and hcs_$fs_get_path_name.

Entry: ssu_$set_ec_suffix

This entry is used to set the suffix for subsystem exec_com files in this subsystem invocation. By default, this is the name of the subsystem (eg: for the read_mail subsystem, subsystem exec_coms would be named XXX.read_mail by default.) See also the descriptions of ssu_$set_ec_search_path and ssu_$set_ec_subsystem_ptr below.

USAGE

```
declare ssu_$set_ec_suffix entry (ptr, char(32));

call ssu_$set_ec_suffix (sci_ptr, suffix_string);
```

ARGUMENTS

sci_ptr
    is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

suffix_string
    is the string to use as a suffix for subsystem exec_coms in this subsystem invocation. (Input)

Entry: ssu_$set_info_dirs

This entry is used to set the list of info directories searched by this subsystem invocation. Additional information on the use of interactive subsystem self-documentation facilities is provided in the Programmer's Reference Manual.

USAGE

```
declare ssu_$set_info_dirs entry (ptr, ptr, fixed bin(35));

call ssu_$set_info_dirs (sci_ptr, idl_ptr, code);
```

ARGUMENTS

sci_ptr
: is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

idl_ptr
: is a pointer to the info_dirs_list structure which is the new list of info directories for this subsystem invocation; this structure is described in detail in the writeup of ssu_$list_info_dirs, above. The uid and info_dir_valid elements of the caller's structure are set appropriately by this call. (Input)

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

  0
  : indicates that the supplied info directories list was accepted.

  error_table_$unimplemented_version
  : indicates that the supplied version of the info_dirs_list structure is not supported by this entry.

  any other value
  : indicates that one or more directories in the list are not valid; the invalid directories are marked by the info_dir_valid bits in the caller's structure.

NOTES

Each supplied info directory is validated as described above in the description of ssu_$add_info_dir. This entry also fills in the values of the info_dir_valid and uid fields of the supplied info_dirs_list structure.

If all the info directories are valid, the current list is replaced with the newly supplied list and a zero status code is returned. Otherwise, if one of the supplied info directories is invalid, a storage system status code indicating the difficulty with the first invalid directory on the list is returned. If a non-zero code is returned, the invalid directories can be identified by examining the info_dir_valid bits.

Entry: ssu_$set_info_ptr

This entry is used to set the info_ptr for this subsystem invocation.

USAGE

```
declare ssu_$set_info_ptr entry (ptr, ptr);

call ssu_$set_info_ptr (sci_ptr, info_ptr)
```

ARGUMENTS

sci_ptr
is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

info_ptr
is the info_ptr to be used for this subsystem invocation. (Input)

NOTES

This entry can be useful, for instance, when the subsystem info structure does not get set up until it is known that the call to ssu_$create_invocation succeeded; a null pointer can be passed as the info_ptr in the call to ssu_$create_invocation, and then when the subsystem info itself is set up, ssu_$set_info_ptr can be called to set the info pointer to point to it.

Entry: ssu_$set_procedure

This entry is used to set the current value of a replaceable procedure in this subsystem invocation. It is used when a subsystem wishes to use a function other than the standard ssu_ provided interface for the specified function, such as a replacement for the request processor. See the Programmer's Reference Manual for information on the use of replaceable procedures within interactive subsystems.

USAGE

```
declare ssu_$set_procedure entry (ptr, char(*), entry, fixed bin(35));

call ssu_$set_procedure (sci_ptr, procedure_name, procedure_value,
     code);
```

ARGUMENTS

sci_ptr
is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

procedure_name
is the name of the replaceable procedure which is to be set. (Input)

procedure_value
is the new value for the specified procedure. (Input)

code
is a status code. (Output) It can have the following values:
0
success.
error_table_noentry
procedure_name is not a acceptable name.

NOTES

See the Programmer's Reference Manual for the currently defined list of replaceable procedure names.

Entry: ssu_$set_prompt

This entry is used to set the prompt string for the subsystem. See the description of interactive subsystem request loops in the Programmer's Reference Manual for additional information on prompts.

USAGE

```
declare ssu_$set_prompt entry (ptr, char(64) varying);

call ssu_$set_prompt (sci_ptr, prompt_string);
```

ARGUMENTS

sci_ptr
is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

prompt_string
is the string to use as the prompt from now on. (Input)

Entry: ssu_$set_prompt_mode

This entry is used to set the prompting mode for the subsystem. See the description of interactive subsystem request loops in the Programmer's Reference Manual for additional information on prompts.

USAGE

```
declare ssu_$set_prompt_mode entry (ptr, bit(*));

call ssu_$set_prompt_mode (sci_ptr, prompt_mode);
```

ARGUMENTS

sci_ptr
is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

prompt_mode
is the new prompt mode. (Input) The individual bits are interpreted as follows:

bit 1 ON	suppress all prompts.
bit 2 ON	prompt after blank request lines.
bit 3 ON	suppress prompts if there is type ahead.

There are named constants in the ssu_prompt_modes.incl.pl1 include file that can be used to create this value.

Entry: ssu_$set_ready_mode

This entry is used to turn ready message processing in the subsystem listener on or off. See the description of interactive subsystem request loops in the Programmer's Reference Manual for additional information on ready messages.

USAGE

```
declare ssu_$set_ready_mode entry (ptr, bit(1) aligned);

call ssu_$set_ready_mode (sci_ptr, enable_sw);
```

ARGUMENTS

sci_ptr
is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

enable_sw
is a bit indicating whether or not ready processing is to be enabled. (Input) If it is "1"b, ready processing will be performed; if it is "0"b, ready processing will not be performed.

Entry: ssu_$set_request_processor_options

This entrypoint changes the request processor options presently in effect for the current subsystem.

USAGE

```
declare ssu_$set_request_processor_options entry (ptr, ptr,
     fixed bin(35));

call ssu_$set_request_processor_options (sci_ptr, rp_options_ptr, code);
```

ARGUMENTS

sci_ptr
: is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

rp_options_ptr
: is a pointer to the rp_options structure containing the new request processor options for this subsystem. (Input) The rp_options structure is defined in the system include file ssu_rp_options.incl.pl1.

code
: is a standard system status code. (Output) It can have one of the following values:

  0
  : the requested options have been accepted and are now in effect for this subsytem.

  error_table_$unimplemented_version
  : the version of the supplied rp_options structure is not RP_OPTIONS_VERSION_1.

  error_table_$bad_subr_arg
  : a value other than one of the named constants in the system include file cp_character_types.incl.pl1 was used in the new request language.

  error_table_$unbalanced_brackets
  : the new request language defined in the rp_options structure contains a character defined as a begin or end active string and no matching end or begin active string exists in the language.

  error_table_$unbalanced_parentheses
  : the new request language defined in the rp_options structure contains a character defined as a begin or end iteration set and no matching end or begin iteration set exists in the language.

NOTES

See the information on the subsystem request language in the Programmer's Reference Manual for a description of the rp_options structure and the request processor options mechanism.

This is a replaceable procedure. See the Programmer's Reference Manual for information on the use of replaceable procedures within interactive subsystems.

Entry: ssu_$set_request_tables

This entry is used to set the list of request tables searched by this subsystem invocation. See the Programmer's Reference Manual for additional information on the use of interactive subsystem request tables.

USAGE

```
declare ssu_$set_request_tables entry (ptr, ptr, fixed bin(35));

call ssu_$set_request_tables (sci_ptr, rtl_ptr, code);
```

ARGUMENTS

sci_ptr
: is a pointer to the subsystem control structure for this invocation as returned by ssu_$create_invocation. (Input)

rtl_ptr
: is a pointer to the request_tables_list structure which is the new list of request tables for this subsystem invocation; this structure is described in detail in the writeup of ssu_$list_request_tables, above. The table_valid elements of the caller's structure are set appropriately by this call. (Input)

code
: is a standard system status code. (Output)
  - 0
    the supplied request tables list was accepted.
  - error_table_$unimplemented_version
    the supplied version of the request_tables_list structure is not supported by this entry.
  - ssu_et_$invalid_request_table
    one or more of the request tables in the list are not valid; the invalid request tables are marked by the table_valid bits in the caller's structure.

NOTES

Each supplied request table is validated as described above in the description of ssu_$add_request_table. This entry also fills in the values of table_valid field of the supplied request_tables_list structure.

If all the request tables are valid, the current list is replaced with the newly supplied list and a zero status code is returned.

Entry: ssu_$standalone_invocation

This entry creates a "standalone" subsystem invocation for use by Multics commands/active functions which can also be used as subsytem requests. Additional information on the use of Multics commands as subsystem requests is provided in the Programmer's Reference Manual.

USAGE

```
declare ssu_$standalone_invocation entry (ptr, char(*), char(*), ptr,
      entry, fixed bin(35));

call ssu_$standalone_invocation (sci_ptr, command_name, command_version,
      arg_list_ptr, abort_entry, code);
```

ARGUMENTS

sci_ptr
: is a pointer to the subsystem control information created for this standalone invocation. (Output)

command_name
: is the name of the Multics command/active function on whose behalf the standalone invocation is created. (Input) This name is printed in the message produced by calls to ssu_$print_message, ssu_$abort_line, and ssu_$abort_subsystem.

command_version
: is the current version of this command/active function. (Input)

arg_list_ptr
: is a pointer to the command's argument list. (Input) If a null pointer is supplied, the subsystem will use the argument list of the caller of ssu_$standalone_invocation. This argument list can be examined by calls to ssu_$arg_ptr, ssu_$return_arg, and ssu_$arg_count.

abort_entry
: is a procedure of no arguments which will be invoked by ssu_$abort_line and ssu_$abort_subsystem after the error message, if any, has been printed. (Input)

code
: is a system status code. (Output)

NOTES

Calls to ssu_$execute_line and ssu_$evaluate_active_string within a standalone invocation are translated into calls to cu_$cp and cu_$evalaute_active_string, respectively.

Calls to ssu_$print_message, ssu_$abort_line and ssu_$abort_subsystem within a standalone invocation are translated into calls to com_err_ or active_fnc_err_ depending on whether the program which created the invocation was invoked as a Multics command or as a Multics active function. In addition, after the error message is printed, ssu_$abort_line and ssu_$abort_subsystem will invoke the abort_entry supplied when the subsystem invocation was created. This entry will normally perform a non-local goto back to a label in the command's main procedure so that it can clean up and return to its caller.

Name: stu_

The stu_ (symbol table utility) subroutine provides a number of entry points for retrieving information from the runtime symbol table section of an object segment generated by the PL/I, FORTRAN, or COBOL compilers. A runtime symbol table is produced when a program is compiled with the -table control argument or when a runtime symbol table is required to support a feature of the language such as PL/I data-directed or FORTRAN NAMELIST input/output statements. A partial symbol table, containing only a statement map, is produced when a program is compiled with the -brief_table control argument.

Entry: stu_$decode_runtime_value

This entry point is called to decode encoded values (e.g., string length or arithmetic precision) stored in a runtime_symbol node.

USAGE

```
declare stu_$decode_runtime_value entry (fixed bin(35), ptr, ptr, ptr,
      ptr, ptr, fixed bin) returns (fixed bin(35));

value = stu_$decode_runtime_value (v, block_ptr, stack_ptr, link_ptr,
      text_ptr, ref_ptr, code);
```

ARGUMENTS

v
: is an encoded value from a runtime_symbol node, e.g., runtime_symbol.size. (Input)

block_ptr
: points to the runtime_block node that corresponds to the block that contains the declaration of the identifier whose runtime_symbol node contains the encoded value. Normally, the value of block_ptr is obtained from a call to the stu_$find_runtime_symbol entry point described below. (Input)

stack_ptr
: is a pointer to the active stack frame associated with the procedure or begin block that corresponds to the specified runtime_block node. If the specified block node is quick, stack_ptr should point to the stack frame in which the quick block is placing its automatic storage. If the specified block is not active and does not have a current stack frame, stack_ptr can be null. (Input)

link_ptr
: is a pointer to the linkage section of the specified block. If link_ptr is null, the stu_$decode_runtime_value entry point attempts to obtain the linkage pointer, if it is needed, from the linkage offset table (LOT); decoding fails if a pointer to the linkage section is needed and text_ptr, block_ptr, and link_ptr are all null or if the segment has never been executed. (Input)

text_ptr
: is a pointer to the base of the object segment that contains the specified block. If text_ptr is null, the stu_$decode_runtime_value entry point attempts to obtain the text pointer, if it is needed, from the active stack frame or the block_ptr; decoding fails if a pointer to the object segment is needed and stack_ptr, block_ptr, and text_ptr are all null. (Input)

ref_ptr
: is the value of the pointer to be used as locator qualifier if the variable that corresponds to the runtime_symbol node that contains the encoded value is based. The value of ref_ptr can often be determined by means of the stu_$get_implicit_qualifier entry point described below. (Input)

code
: is a status code. (Output) It can be:
0 if the encoded value was successfully decoded.
1 if the value could not be decoded.

value
: is the decoded value if the value of code is 0. (Output)

Entry: stu_$decode_runtime_value_extended

This entry point is called to decode encoded values (e.g., string length or arithmetic precision) stored in a runtime_symbol node.

USAGE

```
declare stu_$decode_runtime_value entry (fixed bin(35), ptr, ptr, ptr,
      ptr, ptr, fixed bin) returns (fixed bin(35));

value = stu_$decode_runtime_value (v, block_ptr, stack_ptr, link_ptr,
      text_ptr, ref_ptr, code);
```

ARGUMENTS

v
: is an encoded value from a runtime_symbol node, e.g., runtime_symbol.size. (Input)

block_ptr
: points to the runtime_block node that corresponds to the block that contains the declaration of the identifier whose runtime_symbol node contains the encoded value. Normally, the value of block_ptr is obtained from a call to the stu_$find_runtime_symbol entry point described below. (Input)

stack_ptr
: is a pointer to the active stack frame associated with the procedure or begin block that corresponds to the specified runtime_block node. If the specified block node is quick, stack_ptr should point to the stack frame in which the quick block is placing its automatic storage. If the specified block is not active and does not have a current stack frame, stack_ptr can be null. (Input)

link_ptr
: is a pointer to the linkage section of the specified block. If link_ptr is null, the stu_$decode_runtime_value entry point attempts to obtain the linkage pointer, if it is needed, from the linkage offset table (LOT); decoding fails if a pointer to the linkage section is needed and text_ptr, block_ptr, and link_ptr are all null or if the segment has never been executed. (Input)

text_ptr
: is a pointer to the base of the object segment that contains the specified block. If text_ptr is null, the stu_$decode_runtime_value entry point attempts to obtain the text pointer, if it is needed, from the active stack frame or the block_ptr; decoding fails if a pointer to the object segment is needed and stack_ptr, block_ptr, and text_ptr are all null. (Input)

ref_ptr
: is the value of the pointer to be used as locator qualifier if the variable that corresponds to the runtime_symbol node that contains the encoded value is based. The value of ref_ptr can often be determined by means of the stu_$get_implicit_qualifier entry point described below. (Input)

code
: is a status code. (Output) It can be:

    0 if the encoded value was successfully decoded.
    1 if the value could not be decoded.

value
: is the decoded value if the value of code is 0. (Output)

Entry: stu_$find_block

This entry point, given a pointer to the symbol table header of an object segment, searches the runtime symbol table of the object segment for the runtime_block node that corresponds to a given procedure block in the object program.

USAGE

```
declare stu_$find_block entry (ptr, char(*) aligned) returns (ptr);

block_ptr = stu_$find_block (header_ptr, name)
```

ARGUMENTS

header_ptr
: points to a symbol table header. (Input)

name
: is the ASCII name of the runtime_block node to be found. The name of a runtime_block node is the same as the first name written on the procedure statement that corresponds to the runtime_block node. (Input)

block_ptr
: is set to point to the runtime_block node if it is found or is null if the block is not found. (Output)

Entry: stu_$find_containing_block

This entry point, given a pointer to the symbol table header of a standard object segment and an offset into the text section, returns a pointer to the runtime_block node corresponding to the smallest procedure or begin block that lexically contains the source line for the instuction pointed to, or null if none could be found.

USAGE

```
declare stu_$find_containing_block entry (ptr, fixed bin(18) unsigned)
      returns (ptr);

bp = stu_$find_containing_block (hp, offset);
```

ARGUMENTS

hp
: is a pointer to the symbol table header. (Input)

offset
: is the offset from the base of the segment of an instruction. (Input)

bp
: is the returned pointer to the runtime_block node, or null.

Entry: stu_$find_header

This entry point, given an ASCII name and/or a pointer to any location in a (possibly bound) object segment, searches the given segment for the symbol table header corresponding to the designated program.

USAGE

```
declare stu_$find_header entry (ptr, char(32) aligned, fixed bin(24))
      returns (ptr);

header_ptr = stu_$find_header (seg_ptr, name, bc);
```

ARGUMENTS

seg_ptr
: points to any location in the object segment. (Input)

name
: is the ASCII name of the program whose symbol header is to be found. If seg_ptr is null, name is treated as a reference name and the segment is determined according to the user's search rules. If the designated segment is bound, name specifies the component. (Input)

bc
: is the bit count of the object segment; if 0, the stu_$find_header entry point determines the bit count itself. (Input)

header_ptr
: points to the symbol table header if it is found or is null if the header is not found. (Output)

NOTES

Since determining the bit count of a segment is relatively expensive, the user should provide the bit count if he has it available (e.g., as a result of a call to hcs_$initiate_count).

Entry: stu_$find_runtime_symbol

This entry point, given a pointer to the runtime_block node that corresponds to a procedure or begin block, searches for the runtime_symbol node that corresponds to a specified identifier name. If the name is not found in the given block, the parent block is searched. This is repeated until the name is found or the root block of the symbol structure is reached, in which case a null pointer is returned.

USAGE

```
declare stu_$find_runtime_symbol entry (ptr, char(*) aligned, ptr,
      fixed bin) returns (ptr);

symbol_ptr = stu_$find_symbol (block_ptr, name, found_ptr, steps);
```

ARGUMENTS

block_ptr
: points to the runtime_block node in which the search is to begin. (Input)

name
: is the ASCII name of the runtime_symbol node to be found. A name can be a fully or partially qualified structure name (e.g., "a.b.c"), in which the runtime_symbol node that corresponds to the lowest level item is located. (Input)

found_ptr
: is set to point to the runtime_block node in which the specified identifier is found. (Output)

steps
: is set to the number of steps that must be taken along the pl1_stack_frame.display_ptr chain to locate the stack_frame associated with the block designated by found_ptr starting at the stack frame for the block designated by block_ptr. (See "Example" below.) If the given identifier is found in the specified block, the value of steps is 0. (Output)

  If the search fails, the value of steps indicates the reason for the failure as follows:
  - -1 block_ptr is null
  - -2 more than 64 structure levels
  - -3 name too long
  - -4 no declaration found
  - -5 symbol reference is ambiguous

symbol_ptr
: is set to point to the runtime_symbol node if it is found or is null if an error occurs. (Output)

Entry: stu_$get_block

Given a pointer to the stack frame, gets a pointer to the runtime_block for the entry that created the frame and to the header for the object segment. This entry point is equivalent to stu_$get_runtime_block except that the location is determined by the information in the stack frame.

USAGE

```
declare stu_$get_block entry (ptr, ptr, ptr);

call stu_$get_block (sp, hp, bp);
```

ARGUMENTS

sp
: points to the stack frame in question. (Input)

hp
: points to the header for the runtime symbol table of the object segment that contains the entry that created the frame. If is set to null if the object segment has no symbol table, or if the object segment cannot be interpreted. (Output)

bp
: points to the runtime_block node for the entry that created the frame. It is set to null if the object segment has no symbol table or could not be interpreted.

Entry: stu_$get_implicit_qualifier

This entry point, given a pointer to the symbol node that corresponds to a PL/I based variable, attempts to return the value of the pointer variable that appeared in the based declaration (e.g., the value of "p" in "dcl a based (p);"). A null pointer is returned if the declaration does not have the proper form or if the value of the pointer cannot be determined.

USAGE

```
declare stu_$get_implicit_qualifier entry (ptr, ptr, ptr, ptr, ptr)
      returns (ptr);

ref_ptr = stu_$get_implicit_qualifier (block_ptr, symbol_ptr, stack_ptr,
      link_ptr, text_ptr);
```

ARGUMENTS

block_ptr
: points to the runtime_block node that corresponds to the procedure or begin block in which the based variable is declared. (Input)

symbol_ptr
: points to the runtime_symbol node that corresponds to the based variable. (Input)

stack_ptr
: is a pointer to the active stack frame associated with the block in which the based variable is declared. If the specified block node is quick, stack_ptr should point to the stack frame in which the quick block is placing its automatic storage. If the specified block is not active and does not have a current stack frame, stack_ptr can be null. (Input)

link_ptr
: is a pointer to the linkage section of the specified block. If link_ptr is null, the stu_$get_implicit_qualifier entry point attempts to obtain the linkage pointer, if it is needed, from the active stack frame; the implicit qualifier cannot be determined if a pointer to the linkage section is needed and stack_ptr and link_ptr are both null. (Input)

text_ptr
: is a pointer to the base of the object segment that contains the specified block. If text_ptr is null, the stu_$get_implicit_qualifier entry point attempts to obtain the text pointer, if it is needed, from the active stack frame; the implicit qualifier cannot be determined if a pointer to the object section is needed and stack_ptr and text_ptr are both null. (Input)

ref_ptr
: is set to the value of the implicit qualifier or is null if the value cannot be determined. (Output)

NOTES

A null pointer is returned for any one of a number of reasons. Some of these are:

The based variable was declared without an implicit qualifier, e.g.,

```
dcl a based;
```

Determining the implicit qualifier involves evaluating an expression, for example, the based variable was declared as:

```
dcl a based(p(i));
```

The based variable was declared with an implicit qualifier, but it is not possible to obtain the address of the qualifier (e.g., it is an authentic pointer, and stack_ptr is null).

Entry: stu_$get_line

This entry point, given a pointer to the symbol header of a standard object segment and an offset in the text section of the object segment, returns information that allows the source line that generated the specified location to be accessed. This entry point can be used with programs that have only a partial runtime symbol table.

USAGE

```
declare stu_$get_line entry (ptr, fixed bin(18), fixed bin,
      fixed bin(18), fixed bin(18), fixed bin, fixed bin);

call stu_$get_line (head_ptr, offset, n_stms, line_no, line_offset,
      line_length, file);
```

ARGUMENTS

head_ptr
is a pointer to the symbol section header of a standard object segment. (Input)

offset
is the offset of an instruction in the text section. (Input)

n_stms
indicates the number of source statements about which information is desired; the string specified by file, line_offset, and line_length is the source for n_stms statements, starting with the statement that contains the given instruction. (Input)

line_no
is set to the line number, in the file in which it is contained, of the statement that contains the specified instruction or is −1 if the given offset does not correspond to a statement in the object program. (Output)

line_offset
is set to the number of characters that precede the first character of the source for the specified statement. (Output)

line_length
is set to the number of characters occupied by the n_stms statements that start with the statement that contains the specified location; the source for these statements is assumed to be entirely contained within a single source file. Let S be the contents of the source file that contains the specified statements considered as a single string; then the source string for the n_stms statements is substr(S,line_offset+1,line_length). (Output)

file
is the number of the source file in which the source for the desired statements is contained. (Output)

Entry: stu_$get_line_no

This entry point, given a pointer to a runtime_block node and an offset in the text segment that corresponds to the block, determines the line number, starting location, and number of words in the source statement that contains the specified location.

USAGE

```
declare stu_$get_line_no entry (ptr, fixed bin(18), fixed bin(18),
      fixed bin(18)) returns (fixed bin(18));

line_no = stu_$get_line_no (block_ptr, offset, start, num);
```

ARGUMENTS

block_ptr
: points to the runtime_block node that corresponds to the block in which the instruction offset exists. (Input)

offset
: is the offset of an instruction in the text segment. (Input)

start
: is set to the offset in the text segment of the first instruction generated for the source line that contains the specified instruction or is -1 if the line is not found. (Output)

num
: is set to the number of words generated for the specified source line. (Output)

line_no
: is set to the line number, in the main source file, of the statement that contains the specified instruction or is -1 if the specified offset does not correspond to a statement in the program. (Output)

NOTES

All line numbers refer to the main source file and not to files accessed by means of the %include statement.

No distinction is made between several statements that occur on the same source line. The start argument is the starting location of the code generated for the first statement on the line and num is the total length of all the statements on the line.

Entry: stu_$get_location

This entry point, given a pointer to a runtime_block node and the line number of a source statement in the block, returns the location in the text segment of the first instruction generated by the specified source line.

USAGE

```
declare stu_$get_location entry (ptr, fixed bin(18)) returns
      (fixed bin(18));

offset = stu_$get_location (block_ptr, line_no);
```

ARGUMENTS

block_ptr
    points to the runtime_block node. (Input)

line_no
    specifies the source line number, which must be in the main source file. (Input)

offset
    is set to the offset in the text segment of the first instruction generated for the given line or is -1 if no instructions are generated for the given line. (Output)

Entry: stu_$get_map_index

This entry point, given a pointer to the symbol header of a standard object segment and an offset into the text section, returns the index of the statement map entry for the source line that generated the instruction at the offset and a pointer to the map entry. This entry can be used with object segments that have only a partial runtime symbol table.

USAGE

```
declare stu_$get_map_index entry (ptr, fixed bin(18) unsigned,
      fixed bin, ptr);

call stu_$get_map_index (header, offset, map_index, map_entry_ptr);
```

ARGUMENTS

header
    is a pointer to the symbol header for the object segment. (Input)

offset
    is the offset of an instruction, relative to the base of the segment. (Input)

map_index
: is the index in the statement map array of the statement map entry for the line corresponding to the instruction, or −1 if no such map entry could be found. (Output)

map_entry_ptr
: is a pointer to the map entry identified by map_index, or null if no such entry could be found. (Output)

NOTES

Even though the map entry index and map entry pointer can be computed from each other, both are supplied to the user for convenience.

Entry: stu_$get_runtime_address

This entry point, given a pointer to a runtime_symbol node and information about the current environment of the block in which the symbol that corresponds to the runtime_symbol node is declared, determines the address of the specified variable.

USAGE

```
declare stu_$get_runtime_address entry (ptr, ptr, ptr, ptr, ptr, ptr,
      ptr) returns (ptr);

add_ptr = stu_$get_runtime_address (block_ptr, symbol_ptr, stack_ptr,
      link_ptr, text_ptr, ref_ptr, subs_ptr);
```

ARGUMENTS

block_ptr
: points to the runtime_block node that corresponds to the block in which the symbol, whose address is to be determined, is declared. (Input)

symbol_ptr
: points to the runtime_symbol node that corresponds to the symbol whose address is to be determined. (Input)

stack_ptr
: is a pointer to the active stack frame associated with the procedure or begin block that corresponds to the specified runtime_block node. If the specified block is quick, stack_ptr should point to the stack frame in which the quick block is placing its automatic storage. If the specified block is not active and does not have a current stack frame, stack_ptr can be null. (Input)

link_ptr
is a pointer to the linkage section of the specified block. If link_ptr is null, the stu_$get_runtime_address entry point attempts to obtain the linkage pointer, if it is needed, from the LOT; the address of the specified symbol cannot be determined if a pointer to the linkage section is needed and text_ptr, block_ptr, and link_ptr are all null or the segment has never been executed. (Input)

text_ptr
is a pointer to the base of the object segment that contains the specified block. If text_ptr is null, the stu_$get_runtime_address entry point attempts to obtain the text pointer, if it is needed, from the active stack frame or the block_ptr; the address of the specified symbol cannot be determined if a pointer to the object segment is needed and stack_ptr, block_ptr, and text_ptr are all null. (Input)

ref_ptr
is the value of the reference pointer to be used if the runtime_symbol node corresponds to a based variable. If ref_ptr is null, the stu_$get_runtime_address entry point calls the stu_$get_implicit_qualifier entry point (described above) to determine the value of the pointer that was used in the declaration of the based variable. (Input)

subs_ptr
points to a vector of single-precision fixed-point binary subscripts. The number of subscripts is assumed to match the number required by the declaration. This argument can be null if the runtime_symbol node does not correspond to an array. (Input)

add_ptr
is set to the full bit address (with full bit offset) of the variable that corresponds to the symbol node or is null if the address cannot be determined. (Output)

Entry: stu_$get_runtime_block

This entry point, given a pointer to an active stack frame and a location within the object segment that created the frame, returns pointers to the symbol table header of the object segment and the runtime_block node that corresponds to the procedure or begin block associated with the stack frame. Null pointers are returned if the stack frame does not belong to a PL/I, FORTRAN, or COBOL program or if the object segment does not have a runtime symbol table.

USAGE

```
declare stu_$get_runtime_block entry (ptr, ptr, ptr, fixed bin(18));

call stu_$get_runtime_block (stack_ptr, header_ptr, block_ptr, loc);
```

ARGUMENTS

stack_ptr
: points to an active stack frame. (Input)

header_ptr
: is set to point to the symbol table header or is null if the object segment does not have a runtime symbol table. (Output)

block_ptr
: is set to point to the runtime_block node that corresponds to the procedure or begin block associated with the stack frame or is null if the object segment does not have a runtime symbol table. (Output)

loc
: is an address within the object segment (e.g., where execution was interrupted); a negative value for loc means no location information is specified. The additional information provided by loc enables the stu_$get_runtime_block entry point to return the runtime_block node that corresponds to the quick PL/I procedure or begin block that is sharing the designated stack frame and was active at the time execution was interrupted. (Input)

Entry: stu_$get_runtime_line_no

This entry point, given a pointer to the symbol header of a standard object segment and an offset in the text section of the object segment, returns information about the line that caused the specified instruction to be generated. Since the symbol header is used to locate the statement map, this entry point can be used with object segments that have only a partial runtime symbol table.

USAGE

```
declare stu_$get_runtime_line_no entry (ptr, fixed bin(18),
      fixed bin(18), fixed bin(18), fixed bin(18));

call stu_$get_runtime_line_no (head_ptr, offset, start, num, line_no);
```

ARGUMENTS

head_ptr
: is a pointer to the symbol section header of a standard object segment. (Input)

offset
: is the offset of an instruction in the text section. (Input)

start
: is set to the offset in the text segment of the first instruction generated for the source line that contains the specified instruction or is -1 if the line is not found. (Output)

num
is set to the number of words in the object code generated for the specified source line. (Output)

line_no
is set to the line number, in the main source file, of the statement that contains the specified instruction or is -1 if the specified offset does not correspond to a statement in the program. (Output)

NOTES

All line numbers refer to the main source file and not to files accessed by means of the %include statement.

No distinction is made between several statements that occur on the same source line. The start argument is the starting location of the code generated for the first statement on the line and num is the total length of all the statements on the line.

Entry: stu_$get_runtime_location

This entry point, given a pointer to the symbol header of a standard object segment and a line number in the main source file, returns the starting location in the text section of the object code generated for the line. This entry point can be used with object segments that have only a partial runtime symbol table.

USAGE

```
declare stu_$get_runtime_location entry (ptr, fixed bin) returns
      (fixed bin(18));

offset = stu_$get_runtime_location (head_ptr, line_no);
```

ARGUMENTS

head_ptr
is a pointer to the symbol section header of a standard object segment. (Input)

line_no
is the line number of a statement in the main source file. (Input)

offset
is set to the location in the text segment where the object code generated for the specified line begins or is -1 if no code is generated for the given line. (Output)

Entry: stu_$get_statement_map

This entry point, given a pointer to the symbol header of a standard object segment, returns information about the statement map of the object segment. This entry point can be used with object segments that have only a partial runtime symbol table.

USAGE

```
declare stu_$get_statement_map entry (ptr, ptr, ptr, fixed bin);

call stu_$get_statement_map (head_ptr, first_ptr, last_ptr, map_size);
```

ARGUMENTS

head_ptr
: is a pointer to the symbol section header of a standard object segment. (Input)

first_ptr
: is set to point to the first entry in the statement map of the object segment or is null if the object segment does not have a statement map. (Output)

last_ptr
: is set to point to the location following the last entry in the statement map of the object segment or is null if the object segment does not have a statement map. (Output)

map_size
: is set to the number of words in an entry in the statement map. (Output)

Entry: stu_$offset_to_pointer

This entry point attempts to convert an offset variable to a pointer value using the area, if any, on which the offset was declared.

USAGE

```
declare stu_$offset_to_pointer entry (ptr, ptr, ptr, ptr, ptr, ptr)
      returns (ptr);

off_ptr = stu_$offset_to_pointer (block_ptr, symbol_ptr, data_ptr,
      stack_ptr, link_ptr, text_ptr);
```

ARGUMENTS

block_ptr
points to the runtime_block node that corresponds to the procedure or begin block in which the offset variable is declared. (Input)

symbol_ptr
points to the runtime_symbol node that corresponds to the offset variable. (Input)

data_ptr
points to the offset value to be converted to a pointer. (Input)

stack_ptr
is a pointer to the active stack frame associated with the block in which the offset variable is declared. If the specified block node is quick, stack_ptr should point to the stack frame in which the quick block is placing its automatic storage. If the specified block is not active and does not have a current stack frame, stack_ptr can be null. (Input)

link_ptr
is a pointer to the linkage section of the specified block. If link_ptr is null, the stu_$offset_to_pointer entry point attempts to obtain the linkage pointer, if it is needed, from the stack frame; conversion fails if a pointer to the linkage section is needed and stack_ptr and link_ptr are both null. (Input)

text_ptr
is a pointer to the base of the object segment that contains the specified block. If text_ptr is null, the stu_$offset_to_pointer entry point attempts to obtain the text pointer, if it is needed, from the active stack frame; conversion fails if a pointer to the text section is needed and stack_ptr and link_ptr are both null. (Input)

off_ptr
is set to the pointer value that corresponds to the offset value; it is null if the conversion fails or if the offset value is itself null. (Output)

Entry: stu_$pointer_to_offset

This entry point attempts to convert a pointer value to an offset variable using the area, if any, on which the offset was declared.

USAGE

```
declare stu_$pointer_to_offset entry (ptr, ptr, ptr, ptr, ptr, ptr)
     returns (offset);

off_val = stu_$pointer_to_offset (block_ptr, symbol_ptr, data_ptr,
     stack_ptr, link_ptr, text_ptr);
```

ARGUMENTS

block_ptr
: points to the runtime_block node that corresponds to the procedure or begin block in which the offset variable is declared. (Input)

symbol_ptr
: points to the runtime_symbol node that corresponds to the offset variable. (Input)

data_ptr
: points at the pointer value to be converted to an offset. This pointer value must be an unpacked pointer value. (Input)

stack_ptr
: is a pointer to the active stack frame associated with the block in which the offset variable is declared. If the specified block node is quick, stack_ptr should point to the stack frame in which the quick block is placing its automatic storage. If the specified block is not active and does not have a current stack frame, stack_ptr can be null. (Input)

link_ptr
: is a pointer to the linkage section of the specified block. If link_ptr is null, the stu_$offset_to_pointer entry point attempts to obtain the linkage pointer, if it is needed, from the stack frame; conversion fails if a pointer to the linkage section is needed and stack_ptr and link_ptr are both null. (Input)

text_ptr
: is a pointer to the base of the object segment that contains the specified block. If text_ptr is null, the stu_$offset_to_pointer entry point attempts to obtain the text pointer, if it is needed, from the active stack frame; conversion fails if a pointer to the text section is needed and stack_ptr and link_ptr are both null. (Input)

off_val
: is set to the offset value that corresponds to the pointer value; it is null if the conversion fails or if the pointer value is itself null. (Output)

Entry: stu_$remote_format

This entry point decodes a remote format specification.

USAGE

```
declare stu_$remote_format entry (fixed bin(35), ptr, ptr, label)
      returns (fixed bin);

code = stu_$remote_format (value, stack_ptr, ref_ptr, format);
```

ARGUMENTS

value
is the remote format value to be decoded. (Input)

stack_ptr
is a pointer to the active stack frame of the block that contains the format being decoded. (Input)

ref_ptr
is the pointer value to be used if the format value being decoded requires pointer qualification. (Input)

format
is set to the format value if decoding is successful. (Output)

code
is a status code. (Output) It can be:
0 if decoding is successful.
1 if decoding is not successful.

EXAMPLES

The use of some of the entry points documented above is illustrated by the following sample program, which is called with:

stack_ptr
a pointer to the stack frame of a PL/I block.

symbol
an ASCII string giving the name of a user symbol in the PL/I program.

subs_ptr
a pointer to an array of binary integers that give subscript values.

The procedure determines the address and size of the specified symbol. If any errors occur, the returned address is null.

```
example:  proc (stack_ptr, symbol, subs_ptr, size) returns (ptr);

declare   stack_ptr ptr,
          symbol    char(*) aligned,
          subs_ptr  ptr,
          size      fixed bin(35);

declare   (header_ptr, block_ptr, symbol_ptr, ref_ptr, sp, blk_ptr,
           stack_ptr, add_ptr) ptr,
          (i, steps) fixed bin,
          code fixed bin(35),
          stu_$get_runtime_block entry (ptr, ptr, ptr, fixed bin(18)),
          stu_$find_runtime_symbol entry (ptr, char(*) aligned, ptr,
           fixed bin) returns(ptr),
          stu_$get_runtime_address entry (ptr, ptr, ptr, ptr, ptr, ptr,
           ptr) returns(ptr),
          stu_$decode_runtime_value entry (fixed bin(35), ptr, ptr, ptr,
           ptr, ptr, fixed bin) returns(fixed bin(35));

%include pl1_stack_frame;
%include runtime_symbol;

          /* determine header and block pointers */

          call stu_$get_runtime_block (stack_ptr, header_ptr,
                    block_ptr, -1);
          if block_ptr = null then return(null);

          /* search for specified symbol */

          symbol_ptr = stu_$find_runtime_symbol (block_ptr, symbol,
                    blk_ptr, steps);
```

```
if symbol_ptr = null then return(null);

/* determine stack frame of block owning symbol */

sp = stack_ptr;
do i = 1 to steps;
     sp = sp -> pl1_stack_frame.display_ptr;
     end;

/* determine address of symbol */

ref_ptr = null;
add_ptr = stu_$get_runtime_address (blk_ptr, symbol_ptr, sp,
            null, null, ref_ptr, subs_ptr);

if add_ptr = null then return(null);

/* determine size */

size = symbol_ptr -> runtime_symbol.size;

if size < 0
then do;
     size = stu_$decode_runtime_value (size, blk_ptr, sp, null,
       null, ref_ptr, code);
     if code ^= 0 then return(null);
     end;

return(add_ptr);
end example;
```

Name: sub_err_

The sub_err_ subroutine is called by other programs that wish to report an unexpected situation without usurping the calling environment's responsibility for the content of and disposition of the error message and the choice of what to do next. The caller specifies an identifying message and may specify a status code. Switches that describe whether and how to continue execution and a pointer to further information may also be passed to this subroutine. The environment that invoked the subroutine caller of sub_err_ may intercept and modify the standard system action taken when this subroutine is called.

General purpose subsystems or subroutines, which can be called in a variety of I/O and error handling environments, should report the errors they detect by calling the sub_err_ subroutine.

USAGE

```
declare sub_err_ entry options (variable);

call sub_err_ (code, name, flags, info_ptr, retval, ctl_string,
      ioa_args);
```

ARGUMENTS

code
: is a standard status code describing the reason for calling the sub_err_ subroutine. (It is normally declared fixed bin(35); but it can be any computational data type. If not fixed bin(35), it will be converted to fixed bin(35)). (Input)

name
: is the name (declared as a nonvarying character string) of the subsystem or module on whose behalf the sub_err_ subroutine is called. (Input)

flags
: describe options associated with the error. The flags argument should be declared as a nonvarying bit string. The following values, located in the include file sub_err_flags.incl.pl1, are permitted: (Input)

```
ACTION_CAN_RESTART       init (""b),
ACTION_CANT_RESTART      init ("1"b),
ACTION_DEFAULT_RESTART   init ("01"b),
ACTION_QUIET_RESTART     init ("001"b)
ACTION_SUPPORT_SIGNAL    init ("0001"b)) bit (36) aligned
                         internal static options (constant);
```

Each bit corresponds to one of the action flags in the standard condition_info_header structure, declared in condition_info_header.incl.pl1. If multiple bits are on in the supplied string, all the specified flags are set. See the Programmer's Reference Manual for definitions of the flags.

info_ptr
is a pointer (declared as an aligned pointer) to optional information specific to the situation. This argument is used as input to initialize info.info_ptr (see "Info Structure" below). The standard system environment does not use this pointer, but it is provided for the convenience of other environments. (Input)

retval
is a return value from the environment to which the error was reported. This argument is used as input to initialize info.retval (see "Info Structure" below). The standard system environment sets this value to zero. Other environments may set the retval argument to other values, which may be used to select recovery strategies. The retval argument should be declared fixed bin(35). (Input/Output)

ctl_string
is an ioa_ format control string (declared as a nonvarying character string) that defines the message associated with the call to the sub_err_ subroutine. Consult the description of the ioa_ subroutine. (Input)

ioa_args
are any arguments required for conversion by the ctl_string argument. (Input)

NOTE

There is an obsolete calling sequence to this subroutine, in which the flags argument is a character string instead of a bit string. In that calling sequence, the legal values are "s" for ACTION_CAN_RESTART, "h" for ACTION_CANT_RESTART, "q" for ACTION_QUIET_RESTART, and "c" for ACTION_DEFAULT_RESTART.

OPERATION

The sub_err_ subroutine proceeds as follows: the structure described below is filled in from the arguments to the sub_err_ subroutine and the signal_ subroutine is called to raise the sub_error_ condition.

When the standard system environment receives a sub_error_ signal, it prints a message of the form:

```
name error by sub_name|location
Status code message. Message from ctl_string.
```

The standard environment then sets retval to zero and returns, if the value ACTION_DEFAULT_RESTART is specified; otherwise it calls the listener. If the start command is invoked, the standard environment returns to sub_err_, which returns to the subroutine caller of the sub_err_ subroutine unless ACTION_CANT_RESTART is specified. If the value ACTION_CANT_RESTART is specified, the sub_err_ subroutine signals the illegal_return condition.

HANDLER OPERATION

All handlers for the any_other condition must either pass the sub_error_ condition on to another handler, or else must handle the condition correctly. Correct handling consists of printing the error message and of respecting the cant_restart, default_restart, and quiet_restart flags, unless the environment deliberately countermands these actions (for example, for debugging purposes).

If an application program wishes to call a subsystem that reports errors by the sub_err_ subroutine and wishes to replace the standard system action for some classes of sub_err_ subroutine calls, the application should establish a handler for the sub_error_ condition by a PL/I on statement. When the handler is activated as a result of a call to the sub_err_ subroutine by some dynamic descendant, the handler should call the find_condition_info_ subroutine to obtain the sub_error_info_ptr that points to the structure described in "Info Structure" below.

The following is an example of how to establish a handler for the sub_error_ condition.

```
1    on sub_error_ begin;

2    %include condition_info;
3    %include condition_info_header;
4    %include sub_error_info;
5    %include format_document_error;

6       condition_info_ptr = addr (local_condition_info);
7       condition_info.version = condition_info_version_1;
8       call find_condition_info_ (null (), condition_info_ptr,
          error_code);
9       if error_code ^= 0
10      then call error_routine (error_code);
11      sub_error_info_ptr = condition_info.info_ptr;

12      if sub_error_info.name ^= "format_document_"
13      then do;
14         call continue_to_signal_ (error_code);
15         return;
16      end;
17      format_document_error_ptr = sub_error_info.info_ptr;
18      ...
19   end;
```

In the example above, line 1-4, 6-11 and 19 are the general purpose section of the sub_error_ on unit; lines 5 and 12-18 would change, depending upon which caller of sub_err_ the on unit was designed to handle (ie, format_document_ or some other subroutine like sort_seg_ or sort_). Line 18 of the example above represents code to access the format_document_error structure, analyze the information in this structure describing the error, take appropriate action (either stop execution of format_document_ by doing a nonlocal goto outside of the sub_error_ on unit, or continue execution by allowing the on unit's begin block to end normally).

INFO STRUCTURE

The structure pointed to by sub_error_info_ptr is declared as follows in the sub_error_info.incl.pl1 include file:

```
dcl 1 sub_error_info         aligned based,
      2 header               aligned like condition_info_header,
      2 retval               fixed bin(35),
      2 name                 char(32),
      2 info_ptr             ptr;
```

STRUCTURE ELEMENTS

header
: is a standard header required at the beginning of each information structure provided to an on unit.

retval
: is the return value. The standard environment sets this value to zero.

name
: is the name of the module encountering the condition.

info_ptr
: is a pointer to additional information associated with the condition.

NOTES

The handler should check sub_error_info.name and sub_error_info.code to make sure that this particular call to the sub_err_ subroutine is the one desired and, if not, call the continue_to_signal_ subroutine. If the handler determines that it wishes to intercept this case of the sub_error_ condition, the information structure provides the message as converted, switches, etc. If control returns to the sub_err_ subroutine, any change made to the value of info.retval is returned to the caller of this subroutine.

Name: suffixed_name_

This subroutine handles storage system entrynames. It provides an entry point that creates a properly suffixed name from a user-supplied name that might or might not include a suffix, an entry point that changes the suffix on a user-supplied name that might or might not include the original suffix, and an entry point that finds a segment, a directory, or a multisegment file whose name matches a user-supplied name that might or might not include a suffix. It is intended to be used by commands that deal with segments with a standard suffix, but that do not require the user to supply the suffix in the command arguments.

Entry: suffixed_name_$find

This entry point attempts to find a directory entry whose name matches a user-supplied name that might or might not include a suffix. This directory entry can be a segment, directory, or a multisegment file.

USAGE

```
declare suffixed_name_$find entry (char(*), char(*), char(*), char(32),
      fixed bin(2), fixed bin(5), fixed bin(35));

call suffixed_name_$find (directory, name, suffix, entry, type, mode,
      code);
```

ARGUMENTS

directory
: is the name of the directory in which the entry is to be found. (Input)

name
: is the name that has been supplied by the user, and that might or might not include a suffix. (Input)

suffix
: is the suffix that is supposed to be part of name. It should not contain a leading period. (Input)

entry
: is a version of name that includes a suffix. It is returned even if the directory entry does not exist. (Output)

type
: is a switch indicating the type of directory entry that was found. (Output)
- 0 no entry was found.
- 1 a segment was found.
- 2 a directory was found.
- 3 a multisegment file was found.

mode
is the caller's access mode to the directory entry that was found. See the hcs_$append_branch entry point for a description of mode. The caller's access mode to the multisegment file directory is returned for a multisegment file. (Output)

code
is a standard status code. (Output) It can be one of the following:
error_table_$noentry
no directory entry that matches name was found.
error_table_$no_info
no directory entry that matches name was found, and furthermore, the caller does not have status permission to the directory.
error_table_$incorrect_access
a directory entry that matches name was found, but the caller has null access to this entry, and to the directory containing this entry.
error_table_$entlong
the properly suffixed name that was made is longer than 32 characters.

Entry: suffixed_name_$make

This entry point makes a properly suffixed name out of a name supplied by the user that might or might not include a suffix.

USAGE

```
declare suffixed_name_$make entry (char(*), char(*), char(32),
      fixed bin(35));

call suffixed_name_$make (name, suffix, proper_name, code);
```

ARGUMENTS

name
is the name that has been supplied by the user, and that might or might not include a suffix. (Input)

suffix
is the suffix that is supposed to be part of name. It should not contain a leading period. (Input)

proper_name
is the suffixed version of name. (Output)

code
is a standard status code. (Output) It can be:
error_table_$entlong
the properly suffixed name that was made is longer than proper_name; proper_name contains only a part of the properly suffixed name.

Entry: suffixed_name_$new_suffix

This entry point creates a name with a new suffix by changing the (possibly existing) suffix on a user-supplied name to the new suffix. If there is no suffix on the user-supplied name, then the new suffix is merely appended to the user-supplied name.

USAGE

```
declare suffixed_name_$new_suffix entry (char (*), char (*), char (*),
      char (32), fixed bin(35));

call suffixed_name_$new_suffix (name, suffix, new_suffix, new_name,
      code);
```

ARGUMENTS

name
: is the name that has been supplied by the user, and that might or might not include a suffix. (Input)

suffix
: is the suffix that might or might not already be on name.

new_suffix
: is the new suffix. (Input)

new_name
: is the name that was created. If name ends with .suffix, then .new_suffix replaces .suffix in new_name. Otherwise, new_name is formed by appending .new_suffix to name. (Output)

code
: is a standard status code. (Output) It can be:

  error_table_$entlong
  : meaning that the suffixed new name is longer than new_name and therefore new_name contains only part of the suffixed new name.

NOTES

If error_table_$no_s_permission is encountered during the processing for suffixed_name_$find, it is ignored and is not returned in the status code.

Name: sus_signal_handler_

The sus_signal_handler_ subroutine is the static condition handler for the sus_ condition. The standard process overseers establish this handler by calling sct_manager_$set. For interactive processes, the sus_ condition typically occurs when the process is disconnected from its login terminal channel. For absentee processes, the sus_ condition occurs when the operators suspend the job.

When the user reconnects to the process, sus_signal_handler_ may attempt to execute an exec_com, according to whether reconnect_ec_enable or reconnect_ec_disable was last called before disconnection.

Entry: sus_signal_handler_$reconnect_ec_enable

This entry point enables searching for the segment reconnect.ec when the user reconnects to a disconnected process. As a result, sus_signal_handler_ looks first in the user's home directory, then in his project directory (`>user_dir_dir>Project_name`), and finally in `>system_control_dir`. When the reconnect.ec segment is found, the command "`exec_com >Directory_name>reconnect`" is executed.

USAGE

```
declare sus_signal_handler_$reconnect_ec_enable entry;

call sus_signal_handler_$reconnect_ec_enable ();
```

NOTES

The use of reconnect.ec is enabled automatically by the standard process overseer process_overseer_.

Invocation of the reconnect.ec is not automatically enabled by the project_start_up_ process overseer. Thus, when using project_start_up_, the project administrator may enable the invocation of reconnect.ec at any point in the project_start_up.ec by using the reconnect_ec_enable command.

The current command processor is used to execute the reconnect.ec command. If the user is using the abbrev command processor, any applicable abbreviation will be expanded.

Entry: sus_signal_handler_$reconnect_ec_disable

This entry point reverses the effect of the sus_signal_handler_$reconnect_ec_enable entry. After reconnection to a disconnected process, there is no attempt made to find or invoke the exec_com "reconnect.ec".

USAGE

```
declare sus_signal_handler_$reconnect_ec_disable entry;

call sus_signal_handler_$reconnect_ec_disable ();
```

Name: sweep_disk_

The sweep_disk_ subroutine walks through the subtree below a specified node of the directory hierarchy, calling a user-supplied subroutine once for every entry in every directory in the subtree.

USAGE

```
declare sweep_disk_ entry (char(168) aligned, entry);

call sweep_disk_ (base_path, subroutine);
```

ARGUMENTS

base_path
    is the pathname of the directory that is the base node of the subtree to be scanned. (Input)

subroutine
    is an entry point called for each branch or link in the subtree (see "User-Supplied Subroutines" below). (Input)

USER-SUPPLIED SUBROUTINES

The subroutine is assumed to have the following declaration and call:

```
declare subroutine entry (char(168) aligned, char(32) aligned,
     fixed bin, char(32)!aligned, ptr, ptr);
```

```
call subroutine (path, dir_name, level, entryname, b_ptr, n_ptr);
```

where:

path
: is the pathname of the directory immediately superior to the directory that contains the current entry. (Input)

dir_name
: is the name of the directory that contains the current entry. (Input)

level
: is the number of levels deep from the base_path directory of the subtree. (Input)

entryname
: is the primary name on the current entry. (Input)

b_ptr
: is a pointer to the branch structure returned by hcs_$star_list for the current entry. (Input)

n_ptr
: is a pointer to the names area for the immediately superior directory of the current entry returned by hcs_$star_list. (Input)

Entry: sweep_disk_$dir_list

This entry point operates in the same way as sweep_disk_ but is much less expensive to use and does not return date_time_contents_modified, date_time_used, or bit_count.

USAGE

```
declare sweep_disk_$dir_list entry (char(168) aligned, entry);
```

```
call sweep_disk_$dir_list (base_path, subroutine);
```

The user-supplied subroutine is called in the same way as sweep_disk_, but b_ptr points instead to the branch structure returned by hcs_$star_dir_list. See the hcs_$star_ subroutine.

NOTES

If the base_path argument to the sweep_disk_ subroutine is the root (">"), the directory >process_dir_dir is omitted from the tree walk.

The sweep_disk_ subroutine attempts to force access to the directories in the subtree by adding an ACL term of the form "sma Person.Project.tag" to each directory ACL, and deleting that ACL term when finished processing the directory. If the user does not have sufficient access to add this ACL term for a given directory, the subroutine processes those parts of the subtree under it where the user already has sufficient access to list the directories.

Entry: sweep_disk_$loud

This entry point is used for debugging subsystems that use the sweep_disk_ subroutine. It sets an internal static flag in sweep_disk_ that causes sweep_disk_ to call com_err_ and report any errors encountered in listing directories or setting ACLs. Since sweep_disk_$loud takes no arguments, and should only be used for debugging, it can readily be invoked as a command ("sweep_disk_$loud") to cause sweep_disk_ to exhibit this debugging behavior for the rest of the process. There is no corresponding entry point to turn the switch off. Because this is a static switch, and affects all callers of sweep_disk_, it should not be turned on, except to debug, when it is important to understand the exact nature of any errors encountered. Normally, sweep_disk_ ignores errors and continues as best it can.

USAGE

```
declare sweep_disk_$loud entry ();

call sweep_disk_$loud ();
```

Name: system_info_

The system_info_ subroutine allows the user to obtain information concerning system parameters. All entry points that accept more than one argument count their arguments and only return values for the number of arguments given. Certain arguments, such as the price arrays, must be dimensioned as shown.

Entry: system_info_$abs_chn

This entry point returns the event channel and process ID for the process that is running the absentee user manager.

USAGE

```
declare system_info_$abs_chn entry (fixed bin(71), bit(36) aligned);

call system_info_$abs_chn (ec, p_id);
```

ARGUMENTS

ec
: is the event channel over which signals to absentee_user_manager_ should be sent. (Output)

p_id
: is the process ID of the absentee manager process (currently the initializer). (Output)

Entry: system_info_$abs_prices

This entry point returns the prices for CPU and real time for each absentee queue.

USAGE

```
declare system_info_$abs_prices entry ((4) float bin, (4) float bin);

call system_info_$abs_prices (cpurate, realrate);
```

ARGUMENTS

cpurate
: is the price per CPU hour for absentee queues 1 to 4. (Output)

realrate
: is the memory unit rate for absentee queues 1 to 4. (Output)

*

Entry: system_info_$access_ceiling

This entry point returns the system_high access authorization or class.

USAGE

```
declare system_info_$access_ceiling entry (bit(72) aligned);

call system_info_$access_ceiling (ceil);
```

ARGUMENTS

ceil
    is the access ceiling. (Output)

Entry: system_info_$category_names

This entry point returns the 32-character long names and the eight-character short names for the access categories.

USAGE

```
declare system_info_$category_names entry (dim(18) char(32), dim(18)
      char(8));

call system_info_$category_names (long, short);
```

ARGUMENTS

long
    is an array of the long level names. (Output)

short
    is an array of the short level names. (Output)

Entry: system_info_$default_absentee_queue

This entry point returns the number of the default absentee queue used for submission of absentee jobs by the enter_abs_request, pl1_abs, fortran_abs, etc., commands.

USAGE

```
declare system_info_$default_absentee_queue entry (fixed bin);

call system_info_$default_absentee_queue (default_q);
```

ARGUMENTS

default_q
    is the default absentee queue. (Output)

Entry: system_info_$device_prices

This entry point returns the per-shift prices for system device usage.

USAGE

```
declare system_info_$device_prices entry (fixed bin, ptr);

call system_info_$device_prices (ndev, dev_ptr);
```

ARGUMENTS

ndev
    is the number of devices with prices. (Output)

dev_ptr
    points to an array where device prices are stored. (Input)

NOTES

In the above entry point, the user must provide the following array (in his storage) for device prices:

```
dcl 1 dvt(16)               based (dev_ptr) aligned,
      2 device_id           char(8),
      2 device_price        (0:7) float bin;
```

STRUCTURE ELEMENTS

dvt
    is the user structure. Only the first ndev of the 16 is filled in.

device_id
    is the name of the device.

device_price
    is the per-hour, per-shift price for the device.

Entry: system_info_$installation_id

This entry point returns the 32-character installation identifier that is typed in the header of the how_many_users command when the -long control argument is specified.

USAGE

```
declare system_info_$installation_id entry (char(*));

call system_info_$installation_id (id);
```

ARGUMENTS

id
    is the installation identifier. (Output)

Entry: system_info_$io_prices

This entry point returns the prices for unit processing for each I/O daemon queue.

USAGE

```
declare system_info_$io_prices entry ((4) float bin);

call system_info_$io_prices (rp);
```

ARGUMENTS

rp
    is the price per 1000 lines for each I/O daemon queue. (Output)

Entry: system_info_$last_shutdown

This entry point returns the clock time of the last shutdown or crash and an eight-character string giving the ERF (error report form) number of the last crash (blank if the last shutdown was not a crash).

USAGE

```
declare system_info_$last_shutdown entry (fixed bin(71), char(*));

call system_info_$last_shutdown (time, erfno);
```

ARGUMENTS

time
    is the clock time of the last shutdown. (Output)

erfno
is the ERF number of the last crash, or blank. (Output)

Entry: system_info_$level_names

This entry point returns the 32-character long names and eight-character short names for sensitivity levels.

USAGE

```
declare system_info_$level_names entry (dim(0:7) char(32), dim(0:7)
      char(8));

call system_info_$level_names (long, short);
```

ARGUMENTS

long
is an array of the long level names. (Output)

short
is an array of the short level names. (Output)

Entry: system_info_$max_rs_number

This entry point returns the largest valid rate structure number.

USAGE

```
declare system_info_$max_rs_number entry (fixed bin(17));

call system_info_$max_rs_number (rs_number);
```

ARGUMENTS

rs_number
is the largest valid rate structure number. If it is zero, there are no rate structures defined, other than the default one in installation_parms. (Output)

Entry: system_info_$next_shift_change

This entry point returns the number of the current shift, the time it started, the time it will end, and the number of the next shift.

USAGE

```
declare system_info_$next_shift_change entry (fixed bin, fixed bin(71),
      fixed bin, fixed bin(71);

call system_info_$next_shift_change (now_shift, change_time, new_shift,
      start_time);
```

ARGUMENTS

now_shift
: is the current shift number. (Output)

change_time
: is the time the shift changes. (Output)

new_shift
: is the shift after change_time. (Output)

start_time
: is the time the current shift started. (Output)

Entry: system_info_$next_shutdown

This entry point returns the time of the next scheduled shutdown, the reason for the shutdown, and the time when the system will return, if these data are available.

USAGE

```
declare system_info_$next_shutdown entry (fixed bin(71), char(*),
      fixed bin(71));

call system_info_$next_shutdown (td, rsn, tn);
```

ARGUMENTS

td
: is the time of the next scheduled shutdown. If none is scheduled, this is 0. (Output)

rsn
: is the reason for the next shutdown (a maximum of 32 characters). If it is not known, it is blank. (Output)

tn
is the time the system will return. If it is not known, it is 0. (Output)

Entry: system_info_$prices

This entry point returns the per-shift prices for interactive use.

USAGE

```
declare system_info_$prices entry ((0:7) float bin, (0:7) float bin,
      (0:7) float bin, (0:7) float bin, float bin, float bin);

call system_info_$prices (cpu, log, prc, cor, dsk, reg);
```

ARGUMENTS

cpu
is the CPU-hour rate per shift. (Output)

log
is the connect-hour rate per shift. (Output)

prc
is the process-hour rate per shift. (Output)

cor
is the page-second rate for main memory per shift. (Output)

dsk
is the page-second rate for secondary storage. (Output)

reg
is the registration fee per user per month. (Output)

Entry: system_info_$resource_price

This entry point returns the price of a specified resource.

USAGE

```
declare system_info_$resource_price entry (char(*), float bin, fixed bin
      (35));

call system_info_$resource_price entry (name, price, code);
```

ARGUMENTS

name
is the name of the resource. (Input)

price
is the price of the resource in dollars per unit. (Output)

code
is a standard status code. It will be error_table_$noentry if the resource is not in the price list. (Output)

Entry: system_info_$rs_name

This entry point returns the rate structure name corresponding to a rate structure number.

USAGE

```
declare system_info_$rs_name entry (fixed bin(17), char(*),
      fixed bin(35));

call system_info_$rs_name (rs_number, rs_name, code);
```

ARGUMENTS

rs_number
is the number of a rate structure. (Input)

rs_name
is the name corresponding to rs_number. (The name can be up to 32 characters long.) (Output)

code
is zero if no error occurred, or error_table_$noentry if rs_number is not the number of a defined rate structure. (Output)

Entry: system_info_$rs_number

This entry point returns the rate structure number corresponding to a rate structure name.

USAGE

```
declare system_info_$rs_number entry (char(*), fixed bin(17),
      fixed bin(35));

call system_info_$rs_number (rs_name, rs_number, code);
```

ARGUMENTS

rs_name
: is the name of a rate structure. (Input)

rs_number
: is the number corresponding to rs_name. (Output)

code
: is zero if no error occurred, or error_table_$noentry if rs_name is not the name of a rate structure. (Output)

Entry: system_info_$shift_table

This entry point returns the local shift definition table of the system.

USAGE

```
declare system_info_$shift_table entry ((336) fixed bin);

call system_info_$shift_table (stt);
```

ARGUMENTS

stt
: is a table of shifts, indexed by half-hour within the week e.g., stt(1) gives the shift for 0000-0030 Mondays. (Output)

Entry: system_info_$sysid

This entry point returns the eight-character system identifier that is typed in the header of the who command and at dial-up time.

USAGE

```
declare system_info_$sysid entry (char(*));

call system_info_$sysid (sys);
```

ARGUMENTS

sys
: is the system identifier that identifies the current system. (Output) Normally this is the Multics Release number (eg, MR10.1).

Entry: system_info_$timeup

This entry point returns the time at which the system was last started up.

USAGE

```
declare system_info_$timeup entry (fixed bin(71));

call system_info_$timeup (tu);
```

ARGUMENTS

tu
: is when the system came up. (Output)

Entry: system_info_$titles

This entry point returns several character strings that more formally identify the installation.

USAGE

```
declare system_info_$titles entry (char(*), char(*), char(*), char(*));

call system_info_$titles (c, d, cc, dd);
```

ARGUMENTS

c
: is the company or institution name (a maximum of 64 characters). (Output)

d
: is the department or division name (a maximum of 64 characters). (Output)

cc
: is the company name, double spaced (a maximum of 120 characters). (Output)

dd
: is the department name, double spaced (a maximum of 120 characters). (Output)

Entry: system_info_$trusted_path

This entry point returns bit flags indicating which trusted path facilities are required by the site. At present, only one, "login" is implemented under the control of the "trusted_path_login" installation parameter, to disable the use of logout -hold and new_proc -authorization.

USAGE

```
declare system_info_$trusted_path entry () returns (bit (36) aligned;

string (trusted_path) = system_info_$trusted_path ();

declare 1 trusted_path_flags aligned
              2 login bit(1) unaligned,
              2 pad bit(35) unaligned;
```

ARGUMENTS

trusted_path_login
    indicates the state of the "trusted_path_login" installation parameter.

Entry: system_info_$users

This entry point returns the current and maximum number of load units and users.

USAGE

```
declare system_info_$users entry (fixed bin, fixed bin, fixed bin,
     fixed bin);

call system_info_$users (mn, nn, mu, nu);
```

ARGUMENTS

mn
    is the maximum number of users. (Output)

nn
    is the current number of users. (Output)

mu
    is the maximum number of load units (times 10). (Output)

nu
    is the current number of load units (times 10). (Output)

Entry: system_info_$version_id

This entry point returns the eight-character version identifier that is written on the hardcore system tape currently running. This might be set to "37-19.3", which is an internal version number. This information is different from the information that is obtained with the system_info_$sysid entry point.

USAGE

```
declare system_info$_version_id entry (char(*));

call system_info_$version_id (vers);
```

ARGUMENTS

vers
    is the version identifier that identifies the current version of the system. (Output)

Name: teco_get_macro_

The teco_get_macro_ subroutine is called by teco to search for an external macro.

By default the following directories are searched:

1. working directory
2. home directory
3. >system_library_tools

USAGE

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

call teco_get_macro_ (mname, mptr, mlen, code);
```

ARGUMENTS

name
    is the name of the macro to be found. (Input)

mptr
    is a pointer to the macro. (Output)

mlen
    is the length of the macro. (Output)

code
    is a standard Multics status code. (Output)

Name: term_

The term_ subroutine terminates the reference names of a segment or multisegment file (MSF) 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 file

USAGE

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

call term_ (dir_path, entryname, code);
```

ARGUMENTS

dir_path
    is the pathname of the containing directory. (Input)

entryname
    is the entryname of the segment or MSF. (Input)

code
    is a standard status code. (Output)

Entry: term_$refname

This 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(*), fixed bin(35));

call term_$refname (ref_name, code);
```

ARGUMENTS

ref_name
    is the reference name of the segment or MSF. (Input)

code
    is a standard status code. (Output)

Entry: term_$seg_ptr

This entry point performs the same function as the term_ entry point given a pointer to the segment. If the segment pointed to is a component of an object MSF, all the components are terminated.

USAGE

```
declare term_$seg_ptr entry (ptr, fixed bin(35));

call term_$seg_ptr (seg_ptr, code);
```

ARGUMENTS

seg_ptr
is a pointer to the segment. (Input)

code
is a standard status code. (Output)

Entry: term_$single_refname

This entry point allows termination of a single reference name. The segment or MSF is not made unknown unless the specified reference name was the only reference name initiated for the file.

USAGE

```
declare term_$single_refname entry (char(*), fixed bin(35));

call term_$single_refname (ref_name, code);
```

ARGUMENTS

ref_name
is a reference name of the file. (Input)

code
is a standard status code. (Output)

Entry: term_$unsnap

This entry point unsnaps links to the segment or MSF but does not terminate any reference names or make the unknown.

USAGE

```
declare term_$unsnap entry (ptr, fixed bin(35));

call term_$unsnap (seg_ptr, code);
```

ARGUMENTS

seg_ptr
    is a pointer to the file. (Input)

code
    is a standard status code. (Output)

NOTES

The term_ subroutine performs the same operation as certain hcs_ entry points; however, the term_ entry points also unsnap links and deal with object MSFs correctly. 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. The term_ subroutine also deals with terminating portions of object MSFs by terminating all the components to prevent them from becoming inconsistent.

Name: terminate_file_

This subroutine performs common operations that are often necessary after a program has finished using a segment. It optionally sets the bit count, truncates the segment, ensures that bits in the last word of the segment after the bit count are zero, and terminates a null reference name from the segment. It may also ensure that all modified pages of the segment are no longer in main memory. It can also be instructed to delete the segment.

USAGE

```
declare terminate_file_ entry (pointer, fixed bin(24), bit(*),
      fixed bin(35));

call terminate_file_ (seg_ptr, bit_count, switches, code);
```

ARGUMENTS

seg_ptr
: is a pointer to the segment. (Input/Output) If null on input, no action is taken. It is set to null after the segment is terminated.

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

switches
: control the action of this subroutine. (Input) See the "Notes" section below.

code
: is a standard status code. (Output)

NOTES

The bits in the switches bit string mean the following:

```
dcl 1 terminate_file_switches based,
      2 truncate              bit (1) unaligned,
      2 set_bc                bit (1) unaligned,
      2 terminate             bit (1) unaligned,
      2 force_write           bit (1) unaligned;
      2 delete                bit (1) unaligned;
```

STRUCTURE ELEMENTS

truncate
: (Input)
: "1"b truncate the segment to the word containing the bit count and ensure that the bits following the bit count in the last word of the segment are zero.
: "0"b don't truncate the segment.

set_bc
(Input)
"1"b set the bit count of the segment to bit_count.
"0"b don't set the bit count.

terminate
(Input)
"1"b terminate a null reference name on the segment.
"0"b don't terminate the segment.

force_write
(Input)
"1"b ensure that modified pages of the segment are no longer in main memory.
"0"b allow modified pages to remain in main memory.

delete
(Input)
"1"b instructs the program to delete the program.
"0"b don't delete the segment.

If a request is made to delete the segment, any other options selected are performed first in case it is not possible to delete the segment.

NOTES

The terminate_file_switches structure is declared in terminate_file.incl.pl1. The named constants in the "List of named constants" section are also declared with one or more of the above bits on.

LIST OF NAMED CONSTANTS

TERM_FILE_TRUNC
truncate the segment to bit_count bits

TERM_FILE_BC
set the bit count to bit_count

TERM_FILE_TERM
terminate a null reference name on the segment

TERM_FILE_TRUNC_BC
truncate the segment to the bit_count bits and set the bit count to bit_count

TERM_FILE_TRUNC_BC_TERM
truncate the segment to the bit_count bits, set the bit count to bit_count, and terminate a null reference name on the segment

| TERM_FILE_FORCE_WRITE
ensure that modified pages of the segment are no longer in main memory

| TERM_FILE_DELETE
delete the segment

This subroutine should never be called from a cleanup handler with the truncate or set_bc switches on. In a cleanup handler, seg_ptr may contain an invalid segment number.

The force_write switch should only be used when data integrity is absolutely essential. The use of the force_write switch may introduce a substantial real time delay in execution, since this subroutine does not return until all modified pages are no longer in main memory. However, use of this switch protects data against unrecoverable main memory failures.

EXAMPLES

The following calls illustrate the two ways to set the switches to set the bit count and terminate a segment. Using the named constants:

```
call terminate_file_ (seg_pointer, bit_count,
     TERM_FILE_BC | TERM_FILE_TERM, code);
```

Using a structure:

```
dcl 1 tfs aligned like terminate_file_switches
string (tfs) = ""b;
tfs.set_bc = "1"b;
tfs.terminate = "1"b;
call terminate_file_ (p, bc, string (tfs), code);
```

Name: terminate_process_

This procedure causes the process in which it is called to be terminated. The arguments determine the exact nature of the termination.

USAGE

```
declare terminate_process_ entry (char (*), ptr);

call terminate_process_ (action, info_ptr);
```

ARGUMENTS

action
: specifies one of four general actions to be taken upon process termination. (Input) The permissible values are logout, new_proc, fatal_error, or init_error (see "Notes").

info_ptr
: points to more specific information about the action to be taken at termination. (Input) The structure pointed to by info_ptr depends upon action (see "Notes").

NOTES

If action is logout then the user's process is logged out. The info_ptr points to:

```
dcl 1 logout_info    aligned,
      2 version      fixed bin,
      2 hold         bit(1) unaligned,
      2 brief        bit(1) unaligned,
      2 pad          bit(34) unaligned,
```

STRUCTURE ELEMENTS

version
: must be 0.

hold
: must be "1"b if the terminal associated with this process is not to be hung up, so that another user may log in.

brief
: must be "1"b if the logout message is to be suppressed.

pad
: must be "0"b.

If action is new_proc, then the user's current process is logged out and a new process is created. The info_ptr points to:

```
dcl 1 new_proc_info              aligned,
      2 version                  fixed bin,
      2 authorization_option     bit(1) unaligned,
      2 pad                      bit(35) unaligned,
      2 new_authorization        bit(72) aligned;
```

STRUCTURE ELEMENTS

version
    must be 1.

authorization_option
    must be 1 if new_authorization is to be used.

pad
    must be 0.

new_authorization
    is the authorization of the new process.

If action is fatal_error, then the user's current process is terminated due to an unrecoverable error. A fatal error message is printed on the terminal and a new process is created. The info_ptr points to:

```
dcl 1 fatal_error_info      aligned,
      2 version             fixed bin,
      2 status_code         fixed bin(35);
```

STRUCTURE ELEMENTS

version
    must be 0.

status_code
    is a standard system status code (in error_table_) indicating the nature of the fatal error, the corresponding error message will be printed on the user's console.

If action is init_error, then the user's process is logged out and a message indicating that his process could not be initialized is printed. The info_ptr points to:

```
dcl 1 init_error_info      aligned,
      2 version            fixed bin,
      2 status_code        fixed bin(35);
```

STRUCTURE ELEMENTS

version
    must be 0.

status_code
    is a standard Multics code indicating the nature of the error.

Name: timed_io_$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. This entry point has the same function as the iox_$get_chars entry point except that it returns error_table_$timeout if it cannot complete its operation within the time specified.

USAGE

```
declare timed_io_$get_chars entry (ptr, fixed bin(71), ptr, fixed
      bin(21), fixed bin(21), fixed bin(35));

call timed_io_$get_chars (iocb_ptr, timeout, buff_ptr, buff_len,
      chars_read, code);
```

ARGUMENTS

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

timeout
: is the number of microseconds to wait before returning the code error_table_$timeout.

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

buff_len
: is the number of bytes to be read where buff_len >= 0. (Input)

chars_read
: is the number of bytes actually read. (Output)

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

NOTES

See also the get_chars_timeout control order of the tty_ I/O module.

Entry: timed_io_get_charsget_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. This entry point has the same function as the iox_$get_line entry point except that it returns error_table_$timeout if it cannot complete its operation within the time specified.

USAGE

```
declare timed_io_$get_line entry (ptr, fixed bin(71), ptr, fixed
      bin(21), fixed bin(21), fixed bin(35));

call timed_io_$get_line (iocb_ptr, timeout, buff_ptr, buff_len,
      chars_read, code);
```

ARGUMENTS

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

timeout
: is the number of microseconds to wait before returning the code error_table_$timeout.

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

buff_len
: is the number of bytes to be read where buff_len >= 0. (Input)

chars_read
: is the number of bytes actually read. (Output)

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

NOTES

See also the get_line_timeout control order of the tty_ I/O module.

Entry: timed_io_get_charsput_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. This entry point has the same function as the iox_$put_chars entry point except that it returns error_table_$timeout if it cannot complete its operation within the time specified.

USAGE

```
declare timed_io_$put_chars entry (ptr, fixed bin(71), ptr, fixed
      bin(21), fixed bin(21), fixed bin(35));

call timed_io_$put_chars (iocb_ptr, timeout, buff_ptr, buff_len,
      chars_written, code);
```

ARGUMENTS

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

timeout
: is the number of microseconds to wait before returning the code error_table_$timeout.

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

buff_len
: is the number of bytes to be read where buff_len >= 0. (Input)

chars_written
: is the number of bytes actually written. (Output)

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

NOTES

See also the put_chars_timeout control order of the tty_ I/O module.

Name: timer_manager_

The timer_manager_ subroutine allows many CPU usage timers and real-time timers to be used simultaneously by a process. The caller can specify for each timer whether a wakeup is to be issued or a specified procedure is to be called when the timer goes off. If a procedure is to be called, the calling procedure can specify a data pointer to pass to that procedure.

The timer_manager_ subroutine fulfills a specialized need of certain sophisticated programs. A user should be familiar with interprocess communication in Multics and the pitfalls of writing programs that can run asynchronously within a process. For example, if a program does run asynchronously within a process and it does input or output with the tty_ I/O module, then the program should issue the "start" control order of tty_ before it returns. This is necessary because a wakeup from tty_ may be intercepted by the asynchronous program. Most pitfalls can be avoided by using only the timer_manager_$sleep entry point.

For most uses of the timer_manager_ subroutine, a cleanup condition handler, which resets all the timers that might be set by a software subsystem, should be set up. If the subsystem is aborted and released, any timers set up by the subsystem can be reset instead of going off at undesired times.

To be used, the timer_manager_ subroutine must be established as the condition handler for the alrm and cput conditions. This is done automatically by the standard Multics environment.

GENERIC ARGUMENTS

At least one of the following arguments is called in all of the timer_manager_ entry points. For convenience, these common arguments are described below rather than in each entry point description.

channel
: is the name of the event channel (fixed binary(71)) over which a wakeup is desired. Two or more timers can be running simultaneously, all of which may, if desired, issue a wakeup on the same event channel.

routine
: is a procedure entry point that is called when the timer goes off. The entry value must be valid when the routine is invoked, i.e., if the routine is an internal procedure, the procedure that created the entry value must still be on the stack. The routine is called as follows:

```
declare routine entry (ptr, char (*), ptr, ptr);

call routine (mc_ptr, name, wc_ptr, data_ptr);
```

ARGUMENTS

mc_ptr
: is a pointer to a structure containing the machine conditions at the time of the process interrupt. (Input)

name
: is the condition name: alrm for a real-time timer and cput for a CPU timer. (Input)

wc_ptr
: is a pointer to crawlout machine conditions. (Input) This pointer will invariably be null, and is only provided for compatibility with other condition handlers.

data_ptr
: is a copy of the pointer passed to the timer_manager_ entry point which established the timer. (Input)

(See the signal_ subroutine for a full description of the mc_ptr and name arguments.) Two or more timers can be running simultaneously, all of which may, if desired, call the same routine.

time
: is the time (fixed binary(71)) at which the wakeup or call is desired.

flags
: is a 2-bit string (bit(2)) that determines how time is to be interpreted. The high-order bit indicates whether it is an absolute or a relative time. The low-order bit indicates whether it is in units of seconds or microseconds. Absolute real time is time since January 1, 1901, 0000 hours Greenwich mean time, i.e., the time returned by the clock_ subroutine. Absolute CPU time is total virtual time used by the the process, i.e., the time returned by the cpu_time_and_paging_ subroutine. Relative time begins when the timer_manager_ subroutine is called.

"11"b means relative seconds
"10"b means relative microseconds
"01"b means absolute seconds
"00"b means absolute microseconds

data_ptr
: is a pointer to a data structure which is to be associated with this particular timer. This is useful for those applications which use timers to manage various related processes, using the same program to manipulate different data. Since earlier versions of timer_manager_ did not provide this service, data_ptr is an optional argument to all entry points which use it.

Entry: timer_manager_$alarm_call

This entry point sets up a real-time timer that calls the routine specified when the timer goes off.

USAGE

```
dcl timer_manager_$alarm_call entry (fixed bin(71), bit(2), entry, ptr);

call timer_manager_$alarm_call (time, flags, routine, data_ptr);
```

Entry: timer_manager_$alarm_call_inhibit

This entry point sets up a real-time timer that calls the handler routine specified when the timer goes off. The call is made with all interrupts inhibited (i.e., all interprocess signal (IPS) are masked off). When the handler routine returns, interrupts are reenabled. If the handler routine does not return, interrupts are not reenabled and the user process may malfunction.

USAGE

```
dcl timer_manager_$alarm_call_inhibit entry (fixed bin(71), bit(2),
      entry, ptr);

call timer_manager_$alarm_call_inhibit (time, flags, routine, data_ptr);
```

Entry: timer_manager_$alarm_wakeup

This entry point sets up a real-time timer that issues a wakeup on the event channel specified when the timer goes off. The event message passed is the string "alarm___" (three underscores). (See the ipc_ subroutine for a discussion of event channels.)

USAGE

```
declare timer_manager_$alarm_wakeup entry (fixed bin(71), bit(2),
      fixed bin(71));

call timer_manager_$alarm_wakeup (time, flags, channel);
```

Entry: timer_manager_$cpu_call

This entry point sets up a CPU timer that calls the routine specified when the timer goes off.

USAGE

```
dcl timer_manager_$cpu_call entry (fixed bin(71), bit(2), entry, ptr);

call timer_manager_$cpu_call (time, flags, routine, data_ptr);
```

Entry: timer_manager_$cpu_call_inhibit

This entry point sets up a CPU timer that calls the handler routine specified when the timer goes off. The call is made with all interrupts inhibited (i.e., all IPS are masked off). When the handler routine returns, interrupts are reenabled. If the handler routine does not return, interrupts are not reenabled and the user process may malfunction.

USAGE

```
dcl timer_manager_$cpu_call_inhibit entry (fixed bin(71), bit(2), entry,
      ptr);

call timer_manager_$cpu_call_inhibit (time, flags, routine, data_ptr);
```

Entry: timer_manager_$cpu_wakeup

This entry point sets up a CPU timer that issues a wakeup on the event channel specified when the timer goes off. The event message passed is the string "cpu_time".

USAGE

```
declare timer_manager_$cpu_wakeup entry (fixed bin(71), bit(2),
      fixed bin(71));

call timer_manager_$cpu_wakeup (time, flags, channel);
```

Entry: timer_manager_$reset_alarm_call

This entry point turns off all real-time timers that call the routine specified when they go off.

USAGE

```
dcl timer_manager_$reset_alarm_call entry (entry);

call timer_manager_$reset_alarm_call (routine);
```

or:

```
dcl timer_manager_$reset_alarm_call entry (entry, ptr);

call timer_manager_$reset_alarm_call (routine, data_ptr);
```

NOTES

If the data_ptr is provided, all real-time timers which are to call the given routine with that value of data_ptr are cancelled. Otherwise, all real-time timers which are to call that routine with any value of data_ptr are cancelled.

Entry: timer_manager_$reset_alarm_wakeup

This entry point turns off all real-time timers that issue a wakeup on the event channel specified when they go off.

USAGE

```
declare timer_manager_$reset_alarm_wakeup entry (fixed bin(71));

call timer_manager_$reset_alarm_wakeup (channel);
```

Entry: timer_manager_$reset_cpu_call

This entry point turns off all CPU timers that call the routine specified when they go off.

USAGE

```
declare timer_manager_$reset_cpu_call entry (entry);

call timer_manager_$reset_cpu_call (routine);

or:

declare timer_manager_$reset_cpu_call entry (entry, ptr);

call timer_manager_$reset_cpu_call (routine, data_ptr);
```

NOTES

If the data_ptr is provided, all CPU timers which are to call the given routine with that value of data_ptr are cancelled. Otherwise, all CPU timers which are to call the routine with any value of data_ptr are cancelled.

Entry: timer_manager_$reset_cpu_wakeup

This entry point turns off all CPU timers that issue a wakeup on the event channel specified when they go off.

USAGE

```
declare timer_manager_$reset_cpu_wakeup entry (fixed bin(71));

call timer_manager_$reset_cpu_wakeup (channel);
```

Entry: timer_manager_$sleep

This entry point causes the process to go blocked for a period of real time. Other timers that are active continue to be processed whenever they go off; however, this routine does not return until the real time has been passed.

USAGE

```
dcl timer_manager_$sleep entry (fixed bin(71), bit(2));

call timer_manager_$sleep (time, flags);
```

The time is always real time; however, it can be relative or absolute, seconds or microseconds, as explained above in "Generic Arguments."

Name: transaction_manager_

Entry points in transaction_manager_ begin and end transactions on behalf of users, return information about transactions, and recover transactions after system failure.

See the section entitled "Multics Data Management" in the *Multics Programmer's Reference Manual*, Order No. AG91, for a complete description of transactions and their use.

Entry: transaction_manager_$abandon_txn

This entry point relinquishes control of the current transaction, causing it to be adjusted (aborted unless a commit was already in progress) by the DM daemon (Data_Management.Daemon). The caller is immediately given a new TDT entry and can begin another transaction.

USAGE

```
declare transaction_manager_$abandon_txn entry (bit (36) aligned, fixed
      bin(35));

call transaction_manager_$abandon_txn (txn_id, code);
```

ARGUMENTS

txn_id
: is the identifier of the current transaction, or "0"b to default to the current transaction. (Input) If txn_id is neither "0"b nor the transaction identifier of the current transaction, dm_error_$transaction_not_current is returned. This argument can be used as a check to be sure which transaction is being abandoned.

code
: is a standard system status code. (Output) It can also be:

dm_error_$no_current_transaction
: No current transaction is defined for this process.

dm_error_$not_own_transaction
: A process can only abandon its own transaction.

dm_error_$transaction_suspended
: The current transaction is suspended and therefore cannot be abandoned.

Entry: transaction_manager_$abort_txn

This entry point aborts the current transaction, returning all modified DM files to the state they were in before the transaction began.

USAGE

```
declare transaction_manager_$abort_txn entry (bit(36) aligned, fixed
     bin(35));

call transaction_manager_$abort_txn (txn_id, code);
```

ARGUMENTS

txn_id
: is the identifier of the current transaction, or "0"b to default to the current transaction. (Input) If txn_id is neither "0"b nor the transaction identifier of the current transaction, dm_error_$transaction_not_current is returned. This argument can be used as a check to be sure which transaction is being aborted.

code
: is a standard system status code. (Output) It can also be:

  dm_error_$no_current_transaction
  : No current transaction is defined for this process.

  dm_error_$not_own_transaction
  : A process can only abort its own transaction.

  dm_error_$transaction_suspended
  : The current transaction is suspended and therefore cannot be aborted.

  dm_error_$unfinished_commit
  : The transaction was left in the middle of a commit operation. It is possible to call $commit_txn to complete the commit, or call either $abandon_txn or $kill_txn.

NOTES

If the transaction has already been abandoned, this entry point causes the DM daemon to abort it immediately.

This entry point will retry abort of a transaction that was left in an error state by a previous abort or rollback. It will not attempt abort of a transaction left in error by any other operation.

Entry: transaction_manager_$begin_txn

This entry point begins a transaction on behalf of the caller, by generating a unique transaction identifier and recording it in a TDT entry as the current transaction for the process. Other information, such as owner name, begin time, and transaction state (in-progress) are also recorded. The transaction id is passed to the before journal manager to begin the transaction.

USAGE

```
declare transaction_manager_$begin_txn (fixed bin(17), bit(36), bit(36)
      aligned, fixed bin(35));

call transaction_manager_$begin_txn (begin_mode,
      before_journal_opening_id, txn_id, code);
```

ARGUMENTS

begin_mode
: determines which of several protocols to use. (Input) The only mode currently available is normal mode.

  TM_NORMAL_MODE
  : requires locks to accompany all gets and puts, and requires all updates to be journalized.

before_journal_opening_id
: is the opening identifier of the before journal to be used by this transaction. (Input) If zero, a before journal is assigned by default to this transaction.

txn_id
: is the identifier of the newly created transaction. (Output) It is generated by transaction_manager_$begin_txn and is guaranteed to be unique across all Multics systems. Transaction identifiers are not reusable.

code
: is a standard system status code. (Output) It can also be:

  dm_error_$invalid_mode
  : The specified begin_mode is not currently supported.

  dm_error_$no_begins
  : Transactions are not allowed to be begun because DM daemon has disallowed beginning new transactions, for example when preparing to do a systemwide DMS shutdown.

  dm_error_$transaction_suspended
  : A transaction cannot be begun because a suspended one already exists.

dm_error_$transaction_in_progress
A transaction cannot be begun because one is already active.

Entry: transaction_manager_$commit_txn

This entry point commits the current transaction. Any modifications made to DM files since the transaction began become permanent and visible to other transactions, as if all the changes were made in the same instant.

USAGE

```
declare transaction_manager_$commit_txn entry (bit(36) aligned, fixed
      bin(35));

call transaction_manager_$commit_txn (txn_id, code);
```

ARGUMENTS

txn_id
is the identifier of the current transaction, or "0"b to default to the current transaction. (Input) If txn_id is neither "0"b nor the transaction identifier of the current transaction, dm_error_$transaction_not_current is returned. This argument can be used as a check to be sure which transaction is being committed.

code
is a standard system status code. (Output) It can also be:

dm_error_$no_current_transaction
No current transaction is defined for this process.

dm_error_$not_own_transaction
A process can only commit its own transaction.

dm_error_$transaction_suspended
The current transaction is suspended and therefore cannot be committed.

dm_error_$unfinished_abort
The transaction was left in the middle of an abort operation. It is possible to call $abort_txn to complete the abort, or call either $abandon_txn or $kill_txn.

dm_error_$unfinished_rollback
The transaction was left in the middle of a rollback operation. It is possible to call $rollback_txn to complete the rollback, call $abort_txn to abort the transaction, or call either $abandon_txn or $kill_txn.

NOTES

This entry point will retry commit of a transaction that was left in an error state by a previous commit. It will not, however, attempt to commit a transaction left in error by any other operation.

Entry: transaction_manager_$get_current_ids

This entry point returns the identifier of the current transaction, the most recent checkpoint number, and the number of times this transaction has been rolled back.

USAGE

```
declare transaction_manager_$get_current_ids entry (bit(36)aligned,
      fixed bin, fixed bin, fixed bin (35));

call transaction_manager_$get_current_ids (txn_id, checkpoint_id,
      rollback_count, code);
```

ARGUMENTS

txn_id
    is the identifier of the current transaction. (Output)

checkpoint_id
    is the number of the most recent checkpoint. This value is currently always zero. (Output)

rollback_count
    is the number of times this transaction has been rolled back. (Output)

code
    is a standard system status code. (Output) It can also be:

    dm_error_$no_current_transaction
        there is no transaction for the user process.

    dm_error_$transaction_suspended
        the current transaction is suspended. The returned information is still valid.

Entry: transaction_manager_$get_current_txn_id

This entry point returns the identifier of the current transaction, and tells whether the transaction is suspended or in error. See "Notes" below for a table of transaction identifiers and error codes returned.

USAGE

```
declare transaction_manager_$get_current_txn_id entry (bit(36) aligned,
      fixed bin(35));

call transaction_manager_$get_current_txn_id (txn_id, code);
```

ARGUMENTS

txn_id
  is the identifier of the current transaction. (Output)

code
  is one of the codes listed below. (Output)

NOTES

The txn_id and code values returned depend on the status of the current transaction:

```
                         txn_id      code
                         ------      ----
1. Txn in progress.      valid id    0
2. No current txn.       0           dm_error_$no_current_transaction
3. Txn suspended.        valid id    dm_error_$transaction_suspended
4. Txn in error.         valid id    dm_error_$unfinished_abort
                                 or: dm_error_$unfinished_commit
                                 or: dm_error_$unfinished_rollback
```

Entry: transaction_manager_$get_state_description

This entry point generates a character string description of a numeric state returned by transaction_manager_$get_txn_info or transaction_manager_$get_txn_info_index.

USAGE

```
declare transaction_manager_$get_state_description entry (fixed bin)
      returns (char(*));

state_description = transaction_manager_$get_state_description (state);
```

Entry: transaction_manager_$get_tdt_size

This entry point returns the number of entries allocated in the TDT. This number can be used as an upper bound for looping through all TDT entries, for example:

```
do i = 1 to transaction_manager_$get_tdt_size();
   call transaction_manager_$get_txn_info_index
         (i,txn_info_ptr, code);
end;
```

USAGE

```
dcl transaction_manager_$get_tdt_size entry returns (fixed bin); number
      = txn_$get_tdt_size ();
```

ARGUMENTS

There are no arguments.

Entry: transaction_manager_$get_tdt_index

This entry point returns the index of the TDT entry occupied by a specified transaction.

USAGE

```
declare transaction_manager_$get_tdt_index entry (bit(36) aligned, fixed
      bin(35)) returns (fixed bin);

txn_index = transaction_manager_$get_tdt_index (txn_id, code);
```

ARGUMENTS

txn_id
is the identifier of a transaction. (Input) If it is "0"b, the current transaction is used.

code
is a standard system status code. (Output) It can also be:

dm_error_$no_current_transaction
with txn_id = "0"b, no current transaction is defined for this process.

dm_error_$transaction_not_found
No transaction exists with the specified transaction identifier.

ACCESS REQUIRED

The caller requires re access to dm_admin_gate_ to obtain the index of another user's transaction.

Entry: transaction_manager_$get_txn_info

This entry point returns a structure containing all the information in the TDT about a transaction.

USAGE

```
declare transaction_manager_$get_txn_info entry (bit(36) aligned, ptr,
      fixed bin(35));

call transaction_manager_$get_txn_info (txn_id, txn_info_ptr, code);
```

ARGUMENTS

txn_id
: is the identifier of a transaction, or "0"b to default to the current transaction. (Input)

txn_info_ptr
: is a pointer to the txn_info structure, declared in dm_tm_txn_info.incl.pl1. (Input)

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

ACCESS REQUIRED

The caller requires re access to dm_admin_gate_ to obtain information about another user's transaction.

STRUCTURE

This structure, declared in dm_tm_txn_info.incl.pl1, returns information about a transaction.

```
dcl 1 txn_info              aligned based (txn_info_ptr),
      2 version             char (8),
      2 txn_id              bit (36) aligned,
      2 txn_index           fixed bin,
      2 mode                fixed bin,
      2 state               fixed bin,
      2 error_code          fixed bin (35),
      2 checkpoint_id       fixed bin,
      2 rollback_count      fixed bin,
      2 owner_process_id    bit (36),
      2 owner_name          char (32),
      2 date_time_created   fixed bin (71),
      2 flags,
       3 (dead_process_sw,
          suspended_sw,
          error_sw,
          abandoned_sw,
          kill_sw)          bit (1) unaligned,
       3 mbz                bit (31) unaligned,
      2 journal_info        aligned,
       3 bj_uid             bit (36),
       3 bj_oid             bit (36),
       3 last_completed_operation
                            char (4),
       3 first_bj_rec_id    bit (36),
       3 last_bj_rec_id     bit (36),
       3 n_rec_written      fixed bin (35),
       3 n_bytes_written    fixed bin (35);
```

STRUCTURE ELEMENTS

version
    is the version of the structure, currently TXN_INFO_VERSION_5.

txn_id
    is the identifier of the transaction.

txn_index
    is the index of the TDT entry for the transaction.

mode
    is the begin_mode according to which the transaction was begun. See transaction_manager_$begin_txn for a list of modes.

state
: is one of the states declared in the include file dm_tm_states.incl.pl1. It is either TM_IN_PROGRESS_STATE for an in-progress transaction, one of several intermediate states corresponding to calls made by the transaction manager (usually when the owner process has died in the middle of a call to transaction_manager_), or one of several error states corresponding to error codes returned by transaction_manager_.

error_code
: is 0 or an error code returned by the last call made by the transaction manager.

checkpoint_id
: is the identifier of the checkpoint that has most recently been rolled back to, or 0 for the start of the transaction.

rollback_count
: is the number of times that the transaction has been rolled back, either by a rollback operation or as part of an unfinished abort.

owner_process_id
: is the identifier of the process that began the transaction. This process may or may not still be running.

owner_name
: is the Person.Project identifier of the process that began the transaction.

date_time_created
: is the date-time that the transaction was begun.

dead_process_sw
: is "1"b if the process that began the transaction is no longer running.

suspended_sw
: is "1"b if the transaction is currently suspended.

error_sw
: is "1"b if the transaction manager received an error code from one of its calls (error_code ^= 0) and the transaction has not been adjusted since.

abandoned_sw
: is "1"b if the transaction was abandoned by the owner via a call to $abandon_txn.

kill_sw
: is "1"b if the owner called $kill_txn and the transaction is therefore waiting to be killed.

bj_uid
: is the UID of the before journal chosen when the transaction was begun.

bj_oid
is the per-process opening identifier of the before journal used by the transaction.

last_completed_operation
is the name of the last completed before journal operation.

first_bj_rec_id
is the identifier of the first mark for this transaction.

last_bj_rec_id
is the identifier of the last mark for this transaction.

n_rec_written
is the number of marks that were written for this transaction.

n_bytes_written
is the total number of bytes written to the journal.

Entry: transaction_manager_$get_txn_info_index

This entry point returns the same information as transaction_manager_$get_txn_info but accepts the index of a TDT entry rather than a transaction identifier. The transaction command, for example, calls this entry point with numbers 1 through transaction_manager_$get_tdt_size() to print information for the entire TDT.

USAGE

```
declare transaction_manager_$get_txn_info_index entry (fixed bin, ptr,
      fixed bin(35));

call transaction_manager_$get_txn_info_index (txn_index, txn_info_ptr,
      code);
```

ARGUMENTS

txn_index
is the index of a TDT entry. (Input)

txn_info_ptr
is a pointer to the txn_info structure, declared in dm_tm_txn_info.incl.pl1. (Input)

code
is a standard system status code. (Output)

ACCESS REQUIRED

The caller requires re access to dm_admin_gate_ to obtain information about another user's transaction.

Entry: transaction_manager_$handle_conditions

This entry point, intended to be called by "any_other" handlers in user programs, temporarily suspends the current transaction during an interruption caused by a signalled condition. When invoked, it suspends the current transaction, allows the condition to propagate, and resumes the transaction when control returns.

USAGE

```
declare transaction_manager_$handle_conditions entry ();

call transaction_manager_$handle_conditions ();
```

ARGUMENTS

There are no arguments.

Entry: transaction_manager_$kill_txn

This entry point is intended to be called by the owner of a transaction when the owner cannot end the transaction normally and does not want the daemon to try to abort it for reasons of efficiency. Killing a transaction can destroy the consistency of the databases changed during the transaction, and is therefore appropriate only if consistency is no longer an issue (for example, if the databases are to be deleted). As with $abandon_txn, calling this entry point frees the user to begin a new transaction.

USAGE

```
declare transaction_manager_$kill_txn entry (bit(36) aligned, fixed
      bin(35));

call transaction_manager_$kill_txn (txn_id, code);
```

ARGUMENTS

txn_id
: is the identifier of the transaction to be killed. (Input) If it is "0"b, the current transaction is used.

code
: is a standard system status code. (Output) It can also be:

  dm_error_$no_current_transaction
  : With txn_id="0"b, no current transaction is defined for this process.

  dm_error_$transaction_suspended
  : With txn_id="0"b, the current transaction is suspended and therefore cannot be killed.

ACCESS REQUIRED

The caller requires re access to dm_admin_gate_.

Entry: transaction_manager_$resume_txn

This entry point reactivates a suspended transaction, once again allowing data operations on protected files.

USAGE

```
declare transaction_manager_$resume_txn entry (fixed bin(35));

call transaction_manager_$resume_txn (code);
```

ARGUMENTS

code
: is a standard system status code. (Output) It can also be:

  dm_error_$no_current_transaction
  : No current transaction is defined for this process.

  dm_error_$no_suspended_transaction
  : The current transaction is not suspended.

Entry: transaction_manager_$rollback_txn

This entry point rolls the current transaction back to its beginning, by replacing all modifications to protected files caused by the transaction, with the before images preserved in the appropriate before journal. The transaction remains current for the user process.

USAGE

```
declare transaction_manager_$rollback_txn entry (bit(36) aligned, fixed
      bin, fixed bin(35));

call transaction_manager_$rollback_txn (txn_id, checkpoint_number,
      code);
```

ARGUMENTS

txn_id
: is the identifier of the current transaction, or "0"b to default to the current transaction. (Input) If txn_id is neither "0"b nor the transaction identifier of the current transaction, dm_error_$transaction_not_current is returned. This argument can be used as a check to be sure which transaction is being rolled back.

checkpoint_number
must currently be 0. (Input)

code
is a standard system status code. (Output) It can also be:

dm_error_$no_current_transaction
No current transaction is defined for this process.

dm_error_$not_own_transaction
A process can only roll back its own transaction.

dm_error_$transaction_suspended
The current transaction is suspended and therefore cannot be rolled back.

dm_error_$unfinished_abort
The transaction was left in the middle of an abort operation. It is possible to call $abort_txn to complete the abort, or call either $abandon_txn or $kill_txn.

dm_error_$unfinished_commit
The transaction was left in the middle of a commit operation. It is possible to call $commit_txn to complete the commit, or call either $abandon_txn or $kill_txn.

NOTES

This entry point will retry rollback of a transaction that was left in an error state by a previous rollback. It will not attempt to rollback a transaction left in error by any other operation.

Entry: transaction_manager_$suspend_txn

This entry point puts the current transaction into a suspended state wherein it is temporarily unusable. Data operations to protected files are not allowed while the transaction is suspended, that is, until $resume_txn is called. Since the suspended transaction has not been completed, no new transaction can be begun.

USAGE

```
declare transaction_manager_$suspend_txn entry (fixed bin(35));

call transaction_manager_$suspend_txn (code);
```

ARGUMENTS

code
is a standard system status code. (Output) It can also be:

dm_error_$no_current_transaction
No current transaction is defined for this process.

dm_error_$transactions_suspended
The current transaction is already suspended.

NOTES

Suspension has the following effects:

1. The current transaction is temporarily unusable. As a result, the entry point $get_current_txn_id returns "0"b and the error code dm_error_$transaction_suspended.

2. No data operations on protected files are allowed while the transaction is suspended.

3. Both $begin_txn and $commit_txn return dm_error_$transaction_suspended.

4. Both $abort_txn and $adjust_tdt_entry (called by DMS) work on suspended transactions.

Entry: transaction_manager_$user_shutdown

This entry point shuts down DMS in the calling process. All TDT entries belonging to the caller's Person.Project are adjusted before DMS is turned off. If the calling process is not currently using DMS, the entry does a return.

Information about the adjusted TDT entries is returned in the structure tm_shutdown_info, declared in dm_tm_shutdown_info.incl.pl1 (see below).

USAGE

```
dcl transaction_manager_$user_shutdown entry (ptr, ptr, fixed bin(35));

call transaction_manager_$user_shutdown (area_ptr, tm_shutdown_info_ptr,
      code);
```

ARGUMENTS

area_ptr
is a pointer to an area in which to allocate the shutdown_info structure. (Input)

tm_shutdown_info_ptr
is the returned pointer to tm_shutdown_info, found in the dm_tm_shutdown_info.incl.pl1 include file. (Output)

code
is a standard system status code. (Output)

STRUCTURE

The shutdown_info structure contains information about adjusted TDT entries belonging to the calling process and is declared in dm_tm_shutdown.incl.pl1.

```
dcl 1 tm_shutdown_info aligned based (tm_shutdown_info_ptr),
      2 version          char (8),
      2 count            fixed bin,
      2 transaction      (tm_shutdown_alloc_count refer
                         (tm_shutdown_info.count)),
       3 txn_id          bit (36) aligned,
       3 op_completed    fixed bin,
       3 state           fixed bin,
       3 error_code      fixed bin (35);
```

STRUCTURE ELEMENTS

version
is the version of the structure, currently TM_SHUTDOWN_INFO_VERSION_1.

count
is the number of transactions that were adjusted.

txn_id
is the identifier of a transaction that was adjusted.

op_completed
is equal to one of the constants ABORTED, FINISHED_ABORT, FINISHED_COMMIT, or ABANDONED declared in the same include file.

state
is the state after adjusting; 0 = a successful adjustment.

error_code
is the error code returned by adjust; 0 = a successful adjustment.

Name: translate_aim_attributes_

This subroutine translates the AIM attributes in an authorization or access class from one system's defintion to another system's definition if possible.

USAGE

```
declare translate_aim_attributes_ entry (ptr, bit(72) aligned, ptr,
      bit(72) aligned, fixed bin(35));

call translate_aim_attributes_ (source_aim_attributes_ptr,
      source_authorization, target_aim_attributes_ptr,
      target_aim_authorization, code);
```

ARGUMENTS

source_aim_attributes_ptr
: is a pointer to the aim_attributes structure defining the AIM attributes of the source system. (Input) This structure is declared in aim_attributes.incl.pl1.

source_aim_authorization
: is the access class or authorization expressed to be translated to the equivalent value on the target system. (Input)

target_aim_attributes_ptr
: is a pointer to the aim_attributes structure defining the AIM attributes of the target system. (Input)

target_aim_authorization
: is set to the access class or authorization on the target system which is equivalent to the value given on the source system. (Output)

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

  0
  : the authorization or access class was successfully translated.

  error_table_$unimplemented_version
  : one of the aim_attributes structures supplied by the caller was of a version not supported by this procedure.

  error_table_$ai_no_common_max
  : there is no set of AIM attributes in common between the two systems.

  error_table_$ai_outside_common_range
  : the source access class or authorization is not less than or equal to the common access class ceiling between the two systems.

NOTES

See the description of the get_system_aim_attributes_ subroutine for a definition of the aim_attributes structure.

The translation of AIM attributes can only be performed for an authorization or access class that is less than or equal to the common access ceiling between the two systems. See the Programmers' Reference Manual for a definition of common access ceiling.

Name: translate_bytes_to_hex9_

This entry point translates a bit string to a character string containing the hexadecimal representation of the bits. Each 9-bit byte of the input is translated into two hex digits by using the low-order (rightmost) 8 bits in each byte.

USAGE

```
declare translate_bytes_to_hex9_ entry (bit (*), char (*));

call translate_bytes_to_hex9_ (bit_string, hex_string);
```

ARGUMENTS

bit_string
: is the bit string to be translated. This argument must start on a byte boundary and should be a multiple of 9 bit long. Any extra bits, not part of a complete byte, are ignored. (Input)

hex_string
: is the output character string containing hexadecimal digits obtained by translating the low-order (rightmost) 8 bits of each 9-bit byte of the input string into 2 hex digits. If the output string argument is longer than necessary, then it is filled with ASCII "0" characters. (Output)

NOTES

This subroutine uses the hardware mvt instruction with a desc4a descriptor for the input string and a desc9a descriptor for the output string to do the translation.

Name: translator_info_

The translator_info_ subroutine contains utility routines needed by the various system translators. They are centralized here to avoid repetitions in each of the individual translators.

Entry: translator_info_$get_source_info

This entry point returns the information about a specified source segment that is needed for the standard object segment: storage system location, date-time last modified, unique ID.

USAGE

```
declare translator_info_$get_source_info entry (ptr, char(*), char(*),
      fixed bin(71), bit(36) aligned, fixed bin(35));

call translator_info_$get_source_info entry (source_ptr, dir_name,
      entryname, date_time_mod, unique_id, code);
```

ARGUMENTS

source_ptr
: is a pointer to the source segment about which information is desired. (Input)

dir_name
: is the pathname of the directory in which the source segment is located. (Output)

entryname
: is the primary name of the source segment. (Output)

date_time_mod
: is the date-time-modified of the source segment as obtained from the storage system. (Output)

unique_id
: is the unique ID of the source segment as obtained from the storage system. (Output)

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

NOTES

Because the interface to this procedure is a pointer to the source segment, the presence of a nonzero status code probably indicates that the storage system entry for the source segment has been altered since the segment was initiated, i.e., the segment has been deleted, or this process no longer has access to the segment.

The entryname returned by this procedure is the primary name on the source segment. It is not necessarily the same name as that by which the translator initiated it.

Entry: translator_info_$component_get_source_info

This entry point returns the information about a specified source segment that is needed for the standard object segment: storage system location, date-time last modified, unique ID. Although there is an argument called component_name, this entry point does not currently handle archive components.

USAGE

```
declare translator_info_$component_get_source_info entry (ptr, char(*),
     char(*), char(*), fixed bin(71), bit(36) aligned, fixed bin(35));

call translator_info_$component_get_source_info (source_ptr, dir_name,
     entry_name, component_name, date_time_mod, unique_id, code);
```

ARGUMENTS

source_ptr
    is a pointer to the source segment about which information is desired. (Input)

dir_name
    is a pathname of the directory in which the source segment is located. (Output)

entry_name
    is the primary name of the source segment. (Output)

component_name
    is currently always null. (Output)

date_time_mod
    is the date_time modified of the source segment as obtained from the storage system. (Output)

unique_id
    is the unique ID of the source segment as obtained from the storage system. (Output)

This page intentionally left blank.

code
    is a storage system status code. (Output)

STATUS CODE

A status code of zero indicates that all information has been returned normally.

A nonzero status code returned by this entry is a storage system status code. Because the interface to this procedure is a pointer to the source segment, the presence of a nonzero status code probably indicates that the storage system entry for the source segment has been altered since the segment was initiated, i.e., the segment has been deleted, or this process no longer has access to the segment.

NOTES

The entryname returned by this procedure is the primary name on the source segment. It is not necessarily the same name as that by which the translator initiated it.

Name: translator_temp_

This subroutine provides an inexpensive temporary storage management facility for translators in the Tools Library. It uses the get_temp_segment_ subroutine to obtain temporary segments in the user's process directory. Each segment begins with a header that defines the amount of free space remaining in the segment. An entry is provided for allocating space in temporary segments, but once allocated, the space can never be freed.

Entry: translator_temp_$allocate

This entry point can be called to allocate a block of space within a temporary segment.

USAGE

```
declare translator_temp_$allocate entry (ptr, fixed bin) returns (ptr);

Pspace = translator_temp_$allocate (Psegment, Nwords);
```

ARGUMENTS

Psegment
is a pointer to the temporary segment in which space is to be allocated. (Input/Output). Psegment must be passed by reference rather than by value, because the allocation routine may change its value if there is insufficient space in the current temporary segment to perform the allocation.

Nwords
is the number of words to be allocated. (Input). It must not be greater than sys_info_$max_seg_size-32.

Pspace
is a pointer to the space that was allocated. (Output). If Nwords > sys_info$max_seg_size-32, then Pspace is a null pointer on return.

NOTES

As an alternative to calling translator_temp_$allocate, a procedure that must perform many allocations can include translator_temp_alloc.incl.pl1. This include segment contains the program definition of an "allocate" function that can be called like the $allocate entry point above. The allocate function is a quick internal PL1 procedure that adds about 60 words to the external procedure and that shares its stack frame. Use of the allocate internal procedure can significantly reduce the cost of performing many allocations.

Entry: translator_temp_$get_next_segment

This entry point can be called by a program activation to obtain additional temporary segments.

USAGE

```
declare translator_temp_$get_next_segment entry (ptr, ptr,
      fixed bin(35));

call translator_temp_$get_next_segment (Psegment, Pnew_segment, code);
```

ARGUMENTS

Psegment
is a pointer to one of the temporary segments that the program has previously obtained during its current activation. (Input)

Pnew_segment
is a pointer to the new temporary segment. (Output)

code
is a status code. (Output)

Entry: translator_temp_$get_segment

This entry point should be called by each program activation to obtain the first temporary segment to be used during that activation. Before the activation ends, the program should release the temporary segment for use by other programs. (See the translator_temp_$release_all_segments entry point below.)

USAGE

```
declare translator_temp_$get_segment entry (char(*) aligned, ptr,
      fixed bin (35));

call translator_temp_$get_segment (program_id, Psegment, code);
```

ARGUMENTS

program_id
is the name of the program that is using the temporary segment. (Input). This name is printed out by the list_temp_segments command.

Psegment
is a pointer to the temporary segment that was created. (Output)

code
is a status code. (Output)

Entry: translator_temp_$release_all_segments

This entry point releases all of the temporary segments used by a program activation for use by other programs. It truncates these segments to conserve space in the process directory. It should be called by each program activation that uses temporary segments before the activation is terminated.

USAGE

```
declare translator_temp_$release_all_segments entry (ptr,
      fixed bin(35));

call translator_temp_$release_all_segments (Psegment, code);
```

ARGUMENTS

Psegment
is a pointer to any one of the temporary segments. (Input)

code
is a status code. (Output)

Entry: translator_temp_$release_segment

This entry point releases one of the temporary segments used by a program activation. It truncates the temporary segment to conserve space in the process directory.

USAGE

```
declare translator_temp_$release_segment entry (ptr, fixed bin(35));

call translator_temp_$release_segment (Psegment, code);
```

ARGUMENTS

Psegment
is a pointer to the temporary segment to be released. (Input)

code
is a status code. (Output)

Name: tssi_

The tssi_ (translator storage system interface) subroutine simplifies the way the language translators use the storage system. The tssi_$get_segment and tssi_$get_file entry points prepare a segment or multisegment file for use as output from the translator, creating it if necessary, truncating it, and setting the access control list (ACL) to rw for the current user. The tssi_$finish_segment and tssi_$finish_file entry points set the bit counts of segments or multisegment files, make them unknown, and put the proper ACL on them. The tssi_$clean_up_segment and tssi_$clean_up_file entry points are used by cleanup procedures in the translator (on segments and multisegment files respectively).

Entry: tssi_$get_segment

This entry point returns a pointer to a specified segment. The ACL on the segment is rw for the current user. If an ACL must be replaced to do this, aclinfo_ptr is returned pointing to information to be used in resetting the ACL.

USAGE

```
declare tssi_$get_segment entry (char(*), char(*), ptr, ptr,
      fixed bin(35));

call tssi_$get_segment (dir_name, entryname, seg_ptr, aclinfo_ptr,
      code);
```

ARGUMENTS

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

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

seg_ptr
: is a pointer to the segment, or is null if an error is encountered. (Output)

aclinfo_ptr
: is a pointer to ACL information (if any) needed by the tssi_$finish_segment entry point. (Output)

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

Entry: tssi_$get_file

This entry point is the multisegment file version of the tssi_$get_segment entry point. It returns a pointer to the specified file. Additional components, if necessary, can be accessed using the msf_manager_$get_ptr entry point (see the description of the msf_manager_ subroutine), with the original segment considered as component 0.

USAGE

```
declare tssi_$get_file entry (char(*), char(*), ptr, ptr, ptr,
      fixed bin(35));

call tssi_$get_file (dir_name, entryname, seg_ptr, aclinfo_ptr, fcb_ptr,
      code);
```

ARGUMENTS

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

entryname
is the entryname of the multisegment file. (Input)

seg_ptr
is a pointer to component 0 of the file. (Output)

aclinfo_ptr
is a pointer to ACL information (if any) needed by the tssi_$finish_file entry point. (Output)

fcb_ptr
is a pointer to the file control block needed by the msf_manager_ subroutine. (Output)

code
is a storage system status code. (Output)

Entry: tssi_$finish_segment

This entry point sets the bit count on the segment after the translator is finished with it. It also terminates the segment. If the segment existed before the call to tssi_$get_segment, the ACL is reset to the way it was before the tssi_$get_segment entry point was called. If no ACL existed for the current user, the mode is set to "mode" for the current user. If the segment was created, and the "mode" parameter contains the "e" mode, all entries on the segment's ACL (as derived from the containing directory's Initial ACL) receive the "e" bit, as well as the other modes specified. The current user, if not specified on the Initial ACL, receives an ACL term of "mode" on the segment. Otherwise, the segment's Initial ACL is restored, and, if the current user does not have an ACL term, the segment receives an ACL term of "mode" for the user.

USAGE

```
declare tssi_$finish_segment entry (ptr, fixed bin(24), bit(36) aligned,
      ptr, fixed bin(35));

call tssi_$finish_segment (seg_ptr, bc, mode, aclinfo_ptr, code);
```

ARGUMENTS

seg_ptr
is a pointer to the segment. (Input)

bc
is the bit count of the segment. (Input)

mode
is the access mode to be put on the segment. (Input) It can be one of the following named constants (declared in access_mode_values.incl.pl1):

"110"b RE_ACCESS
"101"b RW_ACCESS

aclinfo_ptr
is a pointer to the saved ACL information returned by the tssi_$get_segment entry point. (Input)

code
is a storage system status code. (Output)

Entry: tssi_$finish_file

This entry point is the same as the tssi_$finish_segment entry point, except that it works on multisegment files, and closes the file, freeing the file control block.

USAGE

```
declare tssi_$finish_file entry (ptr, fixed bin, fixed bin(24), bit(36)
      aligned, ptr, fixed bin(35));

call tssi_$finish_file (fcb_ptr, component, bc, mode, aclinfo_ptr,
      code);
```

ARGUMENTS

fcb_ptr
is a pointer to the file control block returned by the tssi_$get_file entry point. (Input)

component
is the highest-numbered component in the file. (Input)

bc
is the bit count of the highest-numbered component. (Input)

mode
is the access mode to be put on the multisegment file. (Input) It can be one of the following named constants (declared in access_mode_values.incl.pl1):
"110"b RE_ACCESS
"101"b RW_ACCESS

aclinfo_ptr
is a pointer to the saved ACL information returned by the tssi_$get_file entry point. (Input)

code
is a storage system status code. (Output)

Entry: tssi_$clean_up_segment

Programs that use the tssi_ subroutine must establish a cleanup procedure that calls this entry point. (For a discussion of cleanup procedures see the Programmer's Reference Manual.) If more than one call is made to the tssi_$get_segment entry point, the cleanup procedure must make the appropriate call to the tssi_$clean_up_segment entry point for each aclinfo_ptr.

The purpose of this call is to free the storage that the tssi_$get_segment entry point allocated to save the old ACLs of the segments being translated. It is to be used in case the translation is aborted (e.g., by a quit signal).

USAGE

```
declare tssi_$clean_up_segment entry (ptr);

call tssi_$clean_up_segment (aclinfo_ptr);
```

ARGUMENTS

aclinfo_ptr
is a pointer to the saved ACL information returned by the tssi_$get_segment entry point. (Input)

Entry: tssi_$clean_up_file

This entry point is the cleanup entry point for multisegment files. In addition to freeing ACLs, it closes the file, freeing the file control block.

USAGE

```
declare tssi_$clean_up_file entry (ptr, ptr);

call tssi_$clean_up_file (fcb_ptr, aclinfo_ptr);
```

ARGUMENTS

fcb_ptr
is a pointer to the file control block returned by the tssi_$get_file entry point. (Input)

aclinfo_ptr
is a pointer to the saved ACL information returned by the tssi_$get_segment entry point. (Input)

Name: total_cpu_time_

The total_cpu_time_ subroutine returns the total CPU time used by the calling process since it was created. The time includes time spent handling page faults, segment faults, and bound faults for the calling process as well as time spent handling any system interrupt that occurred while the calling process was executing.

USAGE

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

time = total_cpu_time_();
```

ARGUMENTS

time
is the total CPU time, in microseconds, used by the calling process. (Output)

Name: ttt_info_

The ttt_info_ subroutine extracts information from the terminal type table (TTT).

Entry: ttt_info_$additional_info

This entry point returns additional information for a specified terminal type to be used by I/O modules other than tty_.

USAGE

```
declare ttt_info_$additional_info entry (char(*), char(*) varying,
      fixed bin(35));

call ttt_info_$additional_info (tt_name, add_info, code);
```

ARGUMENTS

tt_name
is the terminal type name. (Input)

add_info
is the additional information string. (Output). If no additional information is defined for the terminal type, a null string is returned. Maximum length is 512 characters.

code
is a standard status code. (Output)

Entry: ttt_info_$decode_answerback

This entry point decodes a specified answerback string into a terminal type name and terminal identifier.

USAGE

```
declare ttt_info_$decode_answerback entry (char(*), fixed bin, char(*),
      char(*), fixed bin(35));

call ttt_info_$decode_answerback (ansb, line_type, tt_name, id, code);
```

ARGUMENTS

ansb
is the answerback string. (Input)

line_type
is a line type number with which the decoded terminal type must be compatible. (Input). A nonpositive line type number is ignored. For further description, see the tty_ I/O module.

tt_name
is the terminal type name decoded from the answerback. (Output). Its length should be at least 32 characters. If no terminal type is indicated, a null string is returned.

id
is the terminal identifier decoded from the answerback. (Output). Its length should be at least four characters. If no id is indicated, a null string is returned.

code
is a standard status code. (Output)

Entry: ttt_info_$decode_type

This entry point obtains the terminal type name that corresponds to a specified terminal type code number.

USAGE

```
declare ttt_info_$decode_type entry (fixed bin, char(*), fixed bin(35));

call ttt_info_$decode_type (type_code, tt_name, code);
```

ARGUMENTS

type_code
: is the terminal type code number. (Input)

tt_name
: is the corresponding terminal type name. (Output)

code
: is a standard status code. (Output)

Entry: ttt_info_$dialup_flags

This entry point returns the values of two flags for a specified terminal type.

USAGE

```
declare ttt_info_$dialup_flags entry (char(*), bit(1), bit(1),
     fixed bin(35));

call ttt_info_$dialup_flags (tt_name, ppm_flag, cpo_flag, code);
```

ARGUMENTS

tt_name
: is the terminal type name. (Input)

ppm_flag
: indicates whether a preaccess message should be printed when an unrecognizable login line is received from a terminal of the specified type (Output):
"1"b yes
"0"b no

cpo_flag
indicates whether "conditional printer off" is defined for the terminal type; i.e., if the answerback indicates whether a terminal is equipped with the printer off feature (Output):
"1"b yes
"0"b no

code
is a standard status code. (Output)

Entry: ttt_info_$encode_type

This entry point obtains a code number that corresponds to a specified terminal type name.

USAGE

```
declare ttt_info_$encode_type entry (char(*), fixed bin, fixed bin(35));

call ttt_info_$encode_type (tt_name, type_code, code);
```

ARGUMENTS

tt_name
is the terminal type name. (Input)

type_code
is the corresponding terminal type code number. (Output)

code
is a standard status code. (Output)

Entry: ttt_info_$function_key_data

This entry point returns a collection of information describing the function keys of a specified terminal type.

USAGE

```
dcl ttt_info_$function_key_data entry (char(*), ptr, ptr,
      fixed bin (35));

call ttt_info_$function_key_data (tt_name, areap, function_key_data_ptr,
      code);
```

ARGUMENTS

tt_name
: is the terminal type name. (Input)

areap
: points to an area where the function_key_data info structure can be allocated. (Input). If null, the system free area is used. If the area is not large enough, the area condition is signaled.

function_key_data_ptr
: points to the function_key_data structure allocated by this entry point. (Output). The structure is described below.

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

DATA STRUCTURE

The data structure allocated by this routine is declared in the include file function_key_data.incl.pl1.

```
dcl 1 function_key_data aligned based (function_key_data_ptr),
      2 version fixed bin,
      2 highest fixed bin,
      2 sequence,
        3 seq_ptr pointer,
        3 seq_len fixed bin (21),
      2 cursor_motion_keys,
        3 home (0:3) like key_info,
        3 left (0:3) like key_info,
        3 up (0:3) like key_info,
        3 right (0:3) like key_info,
        3 down (0:3) like key_info,
      2 function_keys (0:function_key_data_highest refer
          (function_key_data.highest), 0:3) like key_info;

dcl (KEY_PLAIN init (0),
     KEY_SHIFT init (1),
     KEY_CTRL init (2),
     KEY_CTRL_AND_SHIFT init (3))
      fixed bin internal static options (constant);

dcl 1 key_info unaligned based (key_info_ptr),
      2 sequence_index fixed bin (12) unsigned unaligned,
      2 sequence_length fixed bin (6) unsigned unaligned;
```

STRUCTURE ELEMENTS

version
is the version of this structure. It should be set to function_key_data_version_1.

highest
is the number of the highest function key defined.

sequence
defines the character string holding the concatenation of all the sequences. The sequence for a given key is defined as a substring of this string.

seq_ptr
is the address of the string.

seq_len
is its length.

cursor_motion_keys
defines some miscellaneous keys whose names connote motion of the cursor. Note that the meaning of these keys is defined only by the application, which may or may not choose to take advantage of the mnemonic value of these key legends.

home
defines the sequences for the HOME key, used by itself, with SHIFT, with CONTROL, and with SHIFT and CONTROL. An absent sequence has a sequence length of zero.

left
defines the left arrow key in the same way as HOME is defined.

up
defines the up-arrow key.

right
defines the right-arrow key.

down
defines the down-arrow key.

function_keys
defines the sequences for the function keys of the terminal. If the terminal has no function key labelled "0", all sequences for 0 have zero length.

key_info
defines a given sequence.

sequence_index
    is the index of the beginning of the sequence in the string of all sequences.

sequence_length
    is the length of the sequence. If zero, the sequence is not present.

NOTES

Mnemonic values are defined for the subscripts for various key combinations: KEY_PLAIN, KEY_SHIFT, KEY_CTRL, and KEY_CTRL_AND_SHIFT. For example, the sequence for the left-arrow key with SHIFT is:

```
substr (function_key_seqs,
   function_key_data.left(KEY_SHIFT).sequence_offset,
   function_key_data.left(KEY_SHIFT).sequence_length)
```

Entry: ttt_info_$initial_string

This entry point returns a string that can be used to initialize terminals of a specified terminal type. The string must be transmitted to the terminal in raw output (rawo) mode. The initial string is most commonly used to set tabs on terminals that support tabs set by software.

USAGE

```
declare ttt_info_$initial_string entry (char(*), char(*) varying,
      fixed bin(35));

      call ttt_info_$initial_string (tt_name, istr_info, code);
```

ARGUMENTS

tt_name
    is the terminal type name. (Input)

istr_info
    is the initial string. (Output). If no initial string is defined for the terminal type, a null string is returned. Maximum length is 512 characters.

code
    is a standard status code. (Output)

Entry: ttt_info_$modes

This entry point returns the default modes for a specified terminal type.

USAGE

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

call ttt_info_$modes (tt_name, modes, code);
```

ARGUMENTS

tt_name
    is the terminal type name. (Input)

modes
    is the default modes string for the terminal type. (Output). If its length is less than 256 characters, the entire modes string is not necessarily returned.

code
    is a standard status code. (Output)

Entry: ttt_info_$preaccess_type

This entry point returns the terminal type name associated with a specified preaccess request.

USAGE

```
declare ttt_info_$preaccess_type entry (char(*), char(*),
      fixed bin(35));

call ttt_info_$preaccess_type (request, tt_name, code));
```

ARGUMENTS

request
    is one of the following three preaccess requests: MAP, 963, or 029. (Input)

tt_name
    is the name of the associated terminal type. (Output). Its length should be at least 32 characters.

code
    is a standard status code. (Output)

Entry: ttt_info_$terminal_data

This entry point returns a collection of information that describes a specified terminal type.

USAGE

```
declare ttt_info_$terminal_data entry (char(*), fixed bin, fixed bin,
     ptr, fixed bin(35));

call ttt_info_$terminal_data (tt_name, line_type, baud, ttd_ptr, code);
```

ARGUMENTS

tt_name
: is the terminal type name. (Input)

line_type
: is a line type number against which the compatibility of the terminal type is verified. (Input). If nonpositive, the line type number is ignored. For further description, see the tty_ I/O module.

baud
: is a baud rate used to select the appropriate delay table. (Input)

ttd_ptr
: is a pointer to a structure in which information is returned. (Input). (See "Notes" below.)

code
: is a standard status code. (Output). If the terminal type is incompatible with the line type, a value of error_table_$incompatible_term_type is returned.

NOTES

The ttd_ptr argument should point to the following structure (terminal_type_data.incl.pl1):

```
dcl 1 terminal_type_data            aligned,
      2 version                     fixed bin,
      2 old_type                    fixed bin,
      2 name                        char (32) unaligned,
      2 tables,
        3 input_tr_ptr              ptr,
        3 output_tr_ptr             ptr,
        3 input_cv_ptr              ptr,
        3 output_cv_ptr             ptr,
        3 special_ptr               ptr,
        3 delay_ptr                 ptr,
      2 editing_chars               unaligned,
        3 erase char (1)            unaligned,
        3 kill char (1)             unaligned,
      2 framing_chars               unaligned,
        3 frame_begin               char (1) unaligned,
        3 frame_end                 char (1) unaligned,
      2 flags,                      unaligned,
        3 keyboard_locking          bit(1),
        3 input_timeout             bit(1),
        3 output_block_acknowledge  bit(1),
        3 mbz                       bit(15),
      2 line_delimiter              char (1) unaligned,
      2 mbz                         bit(9) unaligned,
      2 flow_control_chars          unaligned,
        3 input_suspend             char (1),
        3 input_resume              char (1),
        3 output_suspend_etb        char (1),
        3 output_resume_ack         char (1),
      2 output_buffer_size          fixed bin;
```

STRUCTURE ELEMENTS

version
    is the version number of the above structure. (Input). It must be 1 or 2.

old_type
    is the old terminal type number that corresponds to the terminal type name. (Output). (The old terminal type number is provided only for compatibility with the obsolete set_type and info tty_ order requests.) A value of -1 indicates that no corresponding old type exists.

name
    is the terminal type name. (Output)

input_tr_ptr
: is a pointer to a structure containing the input translation table. (Output). This structure is identical to the info structure for the set_input_translation order of the tty_ I/O module.

output_tr_ptr
: is a pointer to a structure containing the output translation table. (Output). This structure is identical to the info structure for the set_output_translation order of the tty_ I/O module.

input_cv_ptr
: is a pointer to a structure containing the input conversion table. (Output). This structure is identical to the info structure for the set_input_conversion order of the tty_ I/O module.

output_cv_ptr
: is a pointer to a structure containing the output conversion table. (Output). This structure is identical to the info structure for the set_output_conversion order of the tty_ I/O module.

special_ptr
: is a pointer to a structure containing the special characters table. (Output). This structure is identical to the info structure for the set_special order of the tty_ I/O module.

delay_ptr
: is a pointer to a structure containing the delay table. (Output). This structure is identical to the info structure for the set_delay order of the tty_ I/O module.

erase
: is the erase character. (Output)

kill
: is the kill character. (Output)

frame_begin
: is the frame-begin character. (Output)

frame_end
: is the frame-end character. (Output)

keyboard_locking
: indicates whether the terminal type requires keyboard locking and unlocking. (Output)
"1"b yes
"0"b no

input_timeout
: is "1"b if the timeout option was specified on an input_resume statement in the TTF. (Output)

output_block_acknowledge
is "1"b if output_end_of_block and output_acknowledge statements were specified in the TTF. (Output)

mbz
must be "0"b.

line_delimiter
is the line delimiter character. (Output)

The remaining elements are not present if version (above) is 1.

flow_control_chars
identifies the flow control characters.

input_suspend
is the character sent to the terminal to suspend input, or sent by the terminal to indicate that it is suspending input. (Output)

input_resume
is the character sent to the terminal to resume input. (Output)

output_suspend_etb
is the character sent by the terminal to suspend output if output_block_acknowledge is "0"b; otherwise, it is the character to be appended to each output block. (Output)

output_resume_ack
is the character sent by the terminal to resume output if output_block_acknowledge is "0"b; otherwise, it is the character used to acknowledge an output block. (Output)

output_buffer_size
is the size, in characters, of the terminal's buffer, for use with a block acknowledgement protocol. (Output). It is 0 unless output_block_acknowledge is "1"b.

Entry: ttt_info_$video_info

This entry point is used to obtain a copy of the video sequences table for a particular terminal type.

USAGE

```
declare ttt_info_$video_info entry (char (*), fixed bin, ptr, ptr,
     fixed bin(35));

call ttt_info_$video_info (terminal_type, baud_rate, areap,
     tty_vtbl_ptr, code);
```

ARGUMENTS

terminal_type
: is the name of the terminal type for which the video table is required. (Input)

baud_rate
: is the current baud rate of the terminal. (Input). This can be set to 0 if it is unknown.

area
: is a pointer to an area where the video table may be allocated. (Input). If null, the system free area is used.

tty_vtbl_ptr
: is a pointer to the video table, if present. (Output)

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

NOTES

The format of a video table is given in the include file tty_video_tables.incl.pl1.

```
dcl 1 tty_video_table          aligned based (ttyvtblp),
      2 version                fixed bin,
      2 screen_height          fixed bin,
      2 screen_line_length     fixed bin,
      2 scroll_count           fixed bin,
      2 flags                  unaligned,
        3 overstrike_available bit (1) unal,
        3 automatic_crlf       bit (1) unal,
        3 simulate_eol         bit (1) unal,
        3 pad                  bit (33) unaligned,
      2 video_chars_len        fixed binary (21)
      2 pad                    (2) bin (36)
      2 nseq                   fixed bin,
      2 sequences              (N_VIDEO_SEQUENCES refer (tty_video_table.nseq))
                                like tty_video_seq aligned,
      2 video_chars            char (tty_video_table_video_chars_len refer
                                (tty_video_table.video_chars_len)) unal;
```

STRUCTURE ELEMENTS

version
is the version of this structure. It must be tty_video_tables_version_1, also declared in this include file.

screen_height
is the number of lines on this terminal.

screen_line_length
is the number of character positions (columns) in each line.

scroll_count
is the number of lines scrolled upward when a scroll command is sent to the terminal (if the terminal is capable of scrolling). For most terminals this will be 1. A value of 0 indicates that one line is scrolled.

flags
describe characteristics of the terminal.

overstrike_available
is "1"b if the terminal can overstrike (i.e., more than one character can be seen in the same character position).

automatic_crlf
is "1"b if the terminal performs a carriage return and line feed when a character is displayed in the last column.

simulate_eol
is reserved for future expansion.

pad
has an undefined value, and is reserved for future expansion.

video_chars_len
specifies the length of the string containing all video sequences.

pad
is reserved for future expansion.

nseq
is the number of the highest video sequence defined for this terminal. Not all sequences are defined for all terminals, so programs should check this value before indexing the sequence array.

sequences
is an array of video sequences. Each element of the array specifies the character sequence for a video control operation. The indices for specific sequences are defined by constants also declared in this include file. See below.

video_chars
is a string holding concatenations of all video sequences.

The include file defines values for the indices into the array of sequences for the video operations supported. The names of these values are: ABS_POS, CLEAR_SCREEN, CLEAR_TO_EOS, HOME, CLEAR_TO_EOL, CURSOR_UP, CURSOR_RIGHT, CURSOR_DOWN, CURSOR_LEFT, INSERT_CHARS, END_INSERT_CHARS, DELETE_CHARS, INSERT_LINES, DELETE_LINES. The include file also defines N_VIDEO_SEQUENCES, which is the number of the highest index ever defined.

A video sequence is defined by the tty_video_seq structure in the include file tty_video_tables.incl.pl1.

```
dcl 1 tty_video_seq based (ttyvseqp) aligned,
      2 flags                  unaligned,
        3 present              bit (1) unal,
        3 interpret            bit (1) unal,
        3 able_to_repeat       bit (1) unal,
        3 cpad_present         bit (1) unal,
        3 cpad_in_chars        bit (1) unal,
        3 pad                  bit (7) unaligned,
        3 general              bit (6) unaligned,
      2 cpad                   fixed bin (18) unsigned unaligned,
      2 pad                    bit (15) unal,
      2 len                    fixed bin (9) unsigned unaligned,
      2 seq_index              fixed bin (12) unsigned unaligned;
```

STRUCTURE ELEMENTS

present
is "1"b if the operation is supported.

interpret
is "1"b if the sequence contains the encoding of the line, column, or repeat count and must be inspected more closely.

able_to_repeat
is "1"b if the terminal can perform multiple sequences of this operation by receiving a single-character sequence containing the repeat count that is encoded in the sequence.

cpad_present
is "1"b if the terminal requires padding after the operation.

cpad_in_chars
is "1"b if the padding is in characters, or "0"b if the padding is in tenths of milliseconds. If the baud rate is supplied to the ttt_info_$video_info subroutine, then padding is always expressed in characters.

pad
is reserved for future expansion.

general
is reserved for future expansion to define per-sequence information.

cpad
is the padding count in units defined by cpad_in_chars.

pad
is reserved for future expansion.

len
is the length of the string of characters defining this sequence.

seq_index
is the index of the start of the string in tty_video_table.video_chars.

Many terminals allow a repetition count to be supplied with an operation (e.g., to delete multiple lines). Positioning operations require line and column coordinates. These values must be expressed in some encoding. A variety of encodings are supported. Parameters to be transmitted are specified by an encoding character in the video sequence string. An encoding character is a nine-bit byte whose high order bit is set and is defined by the structure tty_numeric_encoding in the include file tty_video_tables.incl.pl1. The encoding scheme is described in the write-up for the video_info table of the Terminal Type file in the Programmer's Reference Manual.

```
dcl 1 tty_numeric_encoding based unaligned,
      2 flags,
        3 must_be_on          bit (1) unal,
        3 express_in_decimal  bit (1) unal,
        3 express_in_octal    bit (1) unal,
        3 offset_is_0         bit (1) unal,
      2 l_c_or_n              fixed bin (2) unsigned unaligned,
      2 num_digits            fixed bin (2) unsigned unaligned,
      2 pad                   bit (1) unaligned
      2 offset                fixed bin (8) unaligned;
```

STRUCTURE ELEMENTS

must_be_on
is "1"b for an encoding character.

express_in_decimal
is "1"b if the value should be expressed as decimal digits.

express_in_octal
is "1"b if the value should be expressed in octal digits. If both flags are off, the value should be sent as a single character.

offset_is_0
if "0"b, the following byte is a fixed bin(8) value to be added to the value before encoding. If "1"b, the offset is 0, and the next byte has no special significance.

l_c_or_n
specifies the type of value to be encoded. Its value can be 0, 1, or 2, and indicates that this encoding character specifies the line number, column number, or repeat count, respectively.

num_digits
specifies the number of digits to be sent. A value of 0 causes all significant digits to be sent, with leading zeroes suppressed.

pad
is reserved for future expansion.

offset
is present only if offset_is_0 is "0"b. It gives an offset to be added to the value before expressing it in octal or decimal.

Name: unique_bits_

The unique_bits_ function 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_ ();
```

ARGUMENTS

bit_string
is the unique value. (Output)

Name: unique_chars_

The unique_chars_ function 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);
```

ARGUMENTS

char_string
is a unique character string. (Output)

bits
is a bit string of up to 70 bits. (Input) See "Notes" below.

NOTES

If the bits argument is less than 70 bits in length, unique_chars_ pads it with zeros on the right to produce a 70-bit string. If the bits argument equals zero, unique_chars_ calls unique_bits_ to obtain a unique bit string.

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.

Entry: unique_chars_$bits

This entrypoint converts a unique character string to its bit string representation.

USAGE

```
declare unique_chars_$bits entry (char(15)) returns (bit(70));

bits = unique_chars_$bits (char_string);
```

ARGUMENTS

bits
is a bit string representation of the unique char_string. (Output)

char_string
is a unique character string to be converted to a bit string. (Input)

Name: unwinder_

The unwinder_ subroutine is used to perform a nonlocal goto on the Multics stack. It is not intended to be called by direct programming (i.e., an explicit call statement in a program) but rather, by the generated code of a translator. For example, it is automatically invoked by a PL/I goto statement involving a nonlocal label variable.

When invoked, the unwinder_ subroutine traces the Multics stack backward until it finds the stack frame associated with its label variable argument or until the stack is exhausted. In each stack frame it passes, it invokes the handler (if any) for the cleanup condition. When it finds the desired stack frame, it passes control to the procedure associated with that frame at the location indicated by the label variable argument. If the desired stack frame cannot be found or if other obscure error conditions arise (e.g., the stack is not threaded correctly), the unwinder_ subroutine signals the unwinder_error condition. If the target is not on the current stack, and there is a stack in a higher ring, that stack is searched after the current one is unwound.

USAGE

```
declare unwinder_ entry (label);

call unwinder_ (tag);
```

ARGUMENTS

tag
: is a nonlocal label variable. (Input)

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.

The user_info_ 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);
```

ARGUMENTS

person_id
is the user's name from the login line (maximum of 22 characters). (Output)

project_id
is the user's project identifier (maximum of 9 characters). (Output)

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);
```

ARGUMENTS

queue
is the number of the absentee queue. (Output)

Entry: user_info_$absentee_request_id

This entry point returns the identifier by which the absentee request is known to the absentee user manager. This is the ID which is used by the absentee request commands enter_abs_request, cancel_abs_request and move_abs_request.

USAGE

```
declare user_info_$absentee_request_id entry (fixed bin(71));

call user_info_$absentee_request_id (request_id);
```

ARGUMENTS

request_id
is the request ID corresponding to this absentee process. (Output) For an interactive or daemon process, the request_id returned is 0.

Entry: user_info_$absentee_restarted

This entry point returns a bit indicating whether this absentee process has been restarted after a system crash.

USAGE

```
declare user_info_$absentee_restarted (bit(1) aligned);

call user_info_$absentee_restarted (restarted_bit);
```

ARGUMENTS

restarted_bit
    is on if the absentee job has been restarted after a system crash.

NOTES

If this absentee process was restarted after a system crash, and the absout_truncation bit is on, truncation will not be performed. See user_info_$absout_truncation.

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);
```

ARGUMENTS

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);
```

ARGUMENTS

path
  is the pathname of the absentee output segment (maximum of 168 characters). (Output)

Entry: user_info_$absout_truncation

This entry point returns a bit indicating whether this absentee process had the -truncate absout file argument requested.

USAGE

```
declare user_info_$absout_truncation (bit(1) aligned);

call user_info_$absout_truncation (truncate_bit);
```

ARGUMENTS

truncate_bit
  is "a"b if the -truncate argument was used for the request that created this absentee process; "0"b if not. See Notes.

NOTES

If the absentee process has been restarted after a system crash, and the truncate bit is set, truncation will not be performed. See user_info_$absentee_restarted.

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. Login attributes are defined in the MAM Project Administrator's manual.

USAGE

```
declare user_info_$attributes entry (char(*) varying);

call user_info_$attributes (attr);
```

ARGUMENTS

attr
is the string containing the names of the user's attributes. (Output)

Entry: user_info_$authorization_range

This entry point returns the range of authorizations at which the calling user may create a process.

USAGE

```
declare user_info_$authorization_range entry ((2) bit (72) aligned);

call user_info_$authorization_range (auth_range);
```

ARGUMENTS

auth_range
represents the range of authorizations at which the user may log in.

Entry: user_info_$homedir

This entry point returns the pathname of the user's initial working directory.

USAGE

```
declare user_info_$homedir entry (char(*));

call user_info_$homedir (hdir);
```

ARGUMENTS

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);
```

ARGUMENTS

mlim
: is the dollar amount the user can spend in the month. (Output)

clim
: is the dollar amount the user can spend (cutoff limit). (Output)

cdate
: is the cutoff date. (Output)

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

shlim
: is an array that shows the dollar amount the user can spend per shift. (Output)

msp
: is the month-to-date spending in dollars. (Output)

csp
: is the spending against the cutoff limit in dollars. (Output)

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);
```

ARGUMENTS

group
    is the name of the load control group. (Output)

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

preempt_time
    is the clock time after which the user becomes standby. (Output)

weight
    is 10 times the user's weight. (Output) Weight is a measure of the load placed on the system by the user; most users have a weight of 1.

Entry: user_info_$login_arg_count

This entry point returns the number of arguments which were provided to the process by the command responsible for the creation of the process. For an absentee process, arguments are given to the enter_abs_request command, using the control argument -arguments. For interactive and daemon processes, arguments are specified on the login command line, also using the control argument -arguments.

USAGE

```
declare user_info_$login_arg_count entry (fixed bin, fixed bin (21),
      fixed bin (21));

call user_info_$login_arg_count (count, max_length, total_length);
```

ARGUMENTS

count
is a number representing the number of arguments supplied by the command which caused the process creation. (Output)

max_length
is the length of the longest login argument. (Output)

total_length
is the total length of all the login arguments. (Output)

Entry: user_info_$login_arg_ptr

This entry point returns a pointer to the character-string login argument specified by the argument number, and also returns the length of the argument-string. See the description of user_info_$login_arg_count for more information about login arguments.

USAGE

```
declare user_info_$login_arg_ptr entry (fixed bin, ptr, fixed bin (21),
     fixed bin (35));

call user_info_$login_arg_ptr (arg_no, arg_ptr, arg_len, code);
```

ARGUMENTS

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

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

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

code
is a standard status code. (Output) If the code error_table_$noarg is returned, the values of arg_ptr and arg_len are undefined.

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);
```

ARGUMENTS

person_id
is the user's name from the login line (maximum of 22 characters). (Output)

project_id
is the user's project identifier (maximum of 9 characters). (Output)

acct
is the user's account identifier (maximum of 32 characters). (Output)

anon
indicates whether a user is an anonymous user. (Output)
1 is anonymous
0 is not anonymous

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

weight
is 10 times the user's weight. (Output) See the user_info_$load_ctl_info entry point.

time_login
is the time the user logged in. (Output) It is expressed as a calendar clock reading in microseconds.

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);
```

ARGUMENTS

logout_channel
    is the event channel over which logouts are to be signalled. (Output)

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);
```

ARGUMENTS

om
    is the name of the user's outer module (maximum of 32 characters). (Output) The outer module is the initial I/O module attached to the user_i/o switch.

Entry: user_info_$process_type

This entry point returns information about the type of the current process.

USAGE

```
declare user_info_$process_type entry (fixed bin (17));

call user_info_$process_type (process_type);
```

ARGUMENTS

process_type
is the type of the user's current process. (Output) It can be:
1 interactive
2 absentee
3 daemon

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);
```

ARGUMENTS

resp
is the name of the user's login responder (maximum of 64 characters). (Output)

Entry: user_info_$rs_name

This entry returns the name of the rate structure that is in effect for the process in which the call is made.

USAGE

```
dcl user_info_$rs_name entry (char (*));

call user_info_$rs_name (rs_name);
```

ARGUMENTS

rs_name
is the name of the rate structure in effect for this process. (Output) (The name may be up to 32 characters long).

Entry: user_info_$rs_number

This entry returns the number of the rate structure that is in effect for the process in which the call is made.

USAGE

```
dcl user_info_$rs_number entry (fixed bin (9));

call user_info_$rs_number (rs_number);
```

ARGUMENTS

rs_number
: is the number of the rate structure in effect for this process. (Output)

Entry: user_info_$service_type

This entry point returns the service type of the terminal on which the user logged in.

USAGE

```
declare user_info_$service_type entry (fixed bin);

call user_info_$service_type (type);
```

ARGUMENTS

type
: is a number representing the service type of the user's terminal. (Output) It can be:
1 login type; interactive command level.
2 FTP type; Advanced Research Projects Agency Network (ARPANET) file transfer protocol

Entry: user_info_$terminal_data

This entry point returns information about the terminal on which the user is logged in.

USAGE

```
declare user_info_$terminal_data entry (char (*), char (*), char (*),
      fixed bin, char (*));

call user_info_$terminal_data (id_code, type, channel, line_type,
      charge_type);
```

ARGUMENTS

id_code
    is the identifier code of the user's terminal (maximum of 4 characters). (Output)

type
    is the type of terminal as it was at login time. (Output)

channel
    is the channel identification (maximum of 32 characters). (Output)

line_type
    is the line type associated with the channel. (Output)

charge_type
    is the name of the device charge associated with the user's login terminal (maximum of 8 characters). (Output) The rate can be found in the array returned by system_info_$device_prices.

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);
```

ARGUMENTS

nproc
    is the number of processes created for this login session. (Output)

old_cpu
    is the CPU time used by previous processes in the login session. (Output)

time_login
    is the time the user logged in. (Output) It is expressed as a calendar clock reading in microseconds.

time_create
    is the time that the current process was created. (Output)

old_mem
    is the memory usage by previous processes in this login session. (Output)

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);
```

ARGUMENTS

person_id
is the user's name from the login line (maximum of 22 characters). (Output)

project_id
is the user's project identifier (maximum of 9 characters). (Output)

acct
is the user's account identifier (maximum of 32 characters). (Output)

Name: valid_decimal_

The valid_decimal_ subroutine tests decimal data for validity.

USAGE

```
declare valid_decimal_ entry (fixed bin, ptr, fixed bin) returns
     (bit(1));

b = valid_decimal_ (dtype, dptr, dprec);
```

ARGUMENTS

dtype
is the data type descriptor of the decimal data. It must be one of the following: 9-12, 29, 30, 35-36, 38-39, 41-46 81-84. (Input)

dptr
is a pointer to the data to be tested for validity. (Input)

dprec
is the precision of the data. (Input)

b
is the value returned by valid_decimal_. It is "1"b if the data is valid, "0"b otherwise. (Output)

NOTES

For decimal data to be valid, it must pass the following tests:

(1) The precision must be > 0 and <= 59;

(2) The data type descriptor must be one handled by valid_decimal_;

(3) If the data is stored as nonoverpunched 9-bit characters, then if it has a sign, then the sign must be either "+" or "-". The digits must all be one of the ASCII characters "0123456789";

(4) If the data is stored as overpunched 9-bit characters, then the sign character must be either octal 173, 175, or in the range 101 to 122. The remaining digits must all be one of the ASCII characters "0123456789";

(5) If the data is stored as 4-bit characters, then if it has a sign, then sign must be in the range "1010"b to "1111"b. All digits must be in the range "0000"b to "1001"b.

Name: value_

The value_ subroutine reads and maintains value segments containing name-value pairs across process boundaries.

To initialize a new value segment, create a segment with suffix "value" and call value_$init_seg with a pointer to its base. The default value segment is initially:

```
[home_dir]>[user_name].value
```

but can be changed by value_$set_path or the value_set_path (vsp) command.

Perprocess values are stored in a temporary value segment in the process directory, and disappear when the process terminates.

Entry: value_$defined

This entry point returns "1"b if a value is defined for name, "0"b otherwise.

USAGE

```
declare value_$defined entry (ptr, bit(36), char(*), fixed bin(35))
     returns (bit (1));

defined_sw = value_$defined (seg_ptr, switches, name, code);
```

ARGUMENTS

seg_ptr
: is a pointer to the base of a value segment. (Input) If seg_ptr is null, the default value segment is used.

switches
: is a bit (36) aligned word of switches: (Input)
  perprocess
  : If bit number one is ON, looks for a perprocess value, as opposed to one stored in the value segment. Either this switch or "permanent" must be on. If both switches are on, a perprocess value is returned if one exists, otherwise a value stored in the value segment is returned.

  permanent
  : If bit number two is ON, looks for a value stored in the value segment.

name
: is a character string with at least one nonblank character. (Input) Trailing blanks are trimmed.

code
: is a standard status code. (Output)

NOTES

The user requires r access on the value segment, except for perprocess values.

Entry: value_$delete

This entry point causes there to be no value defined for name.

USAGE

```
declare value_$delete entry (ptr, bit(36), char(*), fixed bin(35));

call value_$delete (seg_ptr, switches, name, code);
```

ARGUMENTS

seg_ptr
: is a pointer to the base of a value segment. (Input) If seg_ptr is null, the default value segment is used.

switches
: is a bit (36) aligned word of switches: (Input)

  perprocess
  : If bit number one is ON, deletes a perprocess value, as opposed to one stored in the value segment. Either this switch or "permanent" must be on. If both switches are on, the perprocess value is deleted if one exists, otherwise the value in the value segment is deleted.

  permanent
  : If bit number two is ON, deletes a value stored in the value segment.

name
: is a character string with at least one nonblank character. (Input) Trailing blanks are trimmed.

code
: is a standard status code. (Output)

NOTES

The user requires rw access on the value segment, except for perprocess values.

Entry: value_$delete_data

This entry point deletes values set by value_$set_data.

USAGE

```
declare value_$delete_data entry (ptr, bit(36), char(*), fixed bin(35));

call value_$delete_data (seg_ptr, switches, name, code);
```

ARGUMENTS

seg_ptr
is a pointer to the base of a value segment. If seg_ptr is null, the default value segment is used. (Input)

switches
is a bit (36) aligned word of switches: (Input)
perprocess
If bit number one is ON, deletes a perprocess value, as opposed to one stored in the value segment. Either this switch or "permanent" must be on. If both switches are on, the perprocess value is deleted if one exists, otherwise a value stored in the value segment is deleted.
permanent
If bit number two is ON, deletes a value stored in the value segment.

name
is a character string with at least one nonblank character. Trailing blanks are trimmed. (Input)

code
is a standard status code. (Output)

NOTES

The user requires rw access on the value segment, except for perprocess values.

Entry: value_$get

This entry point returns the defined value of a name.

USAGE

```
declare value_$get entry options (variable);

call value_$get (seg_ptr, switches, name, value_arg, code);
```

ARGUMENTS

seg_ptr
is a pointer to the base of a value segment. (Input) If seg_ptr is null, the default value segment is used.

switches
is a bit (36) aligned word of switches: (Input)
perprocess
If bit number one is ON, looks for a perprocess value, as opposed to one stored in the value segment. Either this switch or "permanent" must be on. If both switches are on, a perprocess value is returned if one exists, otherwise a value stored in the value segment is returned.
permanent
If bit number two is ON, looks for a value stored in the value segment.

name
is a fixed-length or varying character string. (Input) If fixed-length, trailing blanks are trimmed. There must be at least one character.

value_arg
is the returned value, having any data type. (Output) If conversion from the internal character string representation cannot be performed, error_table_$bad_conversion is returned. Conversion errors cannot occur if value_arg is a character string, but if it has a maximum length greater than 0 and truncation occurs, the error code error_table_$smallarg is returned.

code
is a standard error code. (Output) It is error_table_$oldnamerr ("Name not found.") if no value is defined.

NOTES

The user requires r access to the value segment, except for perprocess values.

Entry: value_$get_data

This entry point returns, into a caller-supplied buffer, the region of storage that is defined as the value of a name, as set by either value_$set_data or value_$test_and_set_data. Values set by other entry points are not seen by value_$get_data.

USAGE

```
declare value_$get_data entry (ptr, bit(36), char(*), ptr, ptr,
      fixed bin(18), fixed bin(35));

call value_$get_data (seg_ptr, switches, name, area_ptr, data_ptr,
      data_size, code);
```

ARGUMENTS

seg_ptr
: is a pointer to the base of a value segment. (Input) If seg_ptr is null, the default value segment is used.

switches
: is a bit (36) aligned word of switches: (Input)
  perprocess
  : If bit number one is ON, looks for a perprocess value, as opposed to one stored in the value segment. Either this switch or "permanent" must be on. If both switches are on, a perprocess value is returned if one exists, otherwise a value stored in the value segment is returned.

  permanent
  : If bit number two is ON, looks for a value stored in the value segment.

name
: is a character string with at least one nonblank character. (Input) Trailing blanks are trimmed.

area_ptr
: points to an area in which the value can be allocated. (Input)

data_ptr
: points to the value returned. (Output)

data_size
: is the number of words in the value. (Output)

code
: is a standard error code. (Output) It is error_table_$oldnamerr ("Name not found.") if no value is defined.

NOTES

The user requires r access on the value segment, except for perprocess values.

Entry: value_$get_path

This entry point returns the pathname of the current default value segment used by value commands without -pathname.

USAGE

```
declare value_$get_path entry (char (*), fixed bin(35));

call value_$get_path (path, code);
```

ARGUMENTS

path
: is the pathname. (Output)

code
: is a standard status code. (Output)

Entry: value_$init_seg

This entry point initializes a segment to be a value segment.

USAGE

```
declare value_$init_seg entry (ptr, fixed bin, ptr, fixed bin(19),
      fixed bin(35));

call value_$init_seg (seg_ptr, seg_type, remote_area_ptr, seg_size,
      code);
```

ARGUMENTS

seg_ptr
: is a pointer to a segment. (Input)

seg_type
: determines the type of use to which the value segment will be put, and therefore the method of allocating values: (Input)

0 permanent: shareable by multiple processes and therefore locked when modified, with values always stored in the value segment itself.

1 perprocess: for use only by the calling process and therefore never locked, with values optionally stored in an area outside the value segment (see the remote_area_ptr argument below).

remote_area_ptr
: for a perprocess segment only, points to an area outside the value segment in which values are to be allocated. (Input) For example, the value segment can be a region of storage 72 words long consisting only of a header, and remote_area_ptr can point to the user's own area. If remote_area_ptr is null and seg_type is 1, values are allocated in the system free area.

seg_size
: is the number of words available to the value segment, or to the remote area if remote_area_ptr is nonnull. (Input) If seg_size is 0, the available size is an entire segment.

code
: is a standard status code. (Output)

NOTES

The user requires rw access on the value segment.

Entry: value_$list

This entry point returns a list of variable names and their values when given a list of starnames and regular expressions to match and exclude. Only values set by value_$set are returned; see value_$list_data_names to list variables set by value_$set_data.

USAGE

```
declare value_$list entry (ptr, bit(36) aligned, ptr, ptr, ptr,
      fixed bin(35));

call value_$list (seg_ptr, switches, match_info_ptr, area_ptr,
      value_list_info_ptr, code);
```

ARGUMENTS

seg_ptr
: is a pointer to the base of a value segment. (Input) If seg_ptr is null, the default value segment is used.

switches
: is a bit (36) aligned word of switches: (Input)
  perprocess
  : If bit number one is ON, looks for perprocess values, as opposed to those stored in the value segment. Either this switch or "permanent" must be on. If both switches are on, both kinds of values are listed.

  permanent
  : If bit number two is ON, looks for values stored in the value segment.

match_info_ptr
is a pointer to the following user-allocated structure, declared in value_structures.incl.pl1: (Input)

```
dcl 1 match_info     aligned based (match_info_ptr),
      2 version      fixed bin, /* = 1 */
      2 name_count   fixed bin,
      2 max_name_len fixed bin (21),
      2 name_array   (alloc_name_count refer
                      (match_info.name_count)),
        3 exclude_sw bit (1) unaligned,
        3 regexp_sw  bit (1) unaligned,
        3 pad        bit (34) unaligned /* = "0"b */
        3 name char  (alloc_max_name_len refer
                      (match_info.max_name_len)) varying;
```

If a name's regexp_sw is ON, the name is a regular expression to be matched. Otherwise, it is a starname to be matched. If the name's exclude_sw is ON, variables matching the name are excluded from the list built up so far, as for the -exclude control argument to the value_list command. Otherwise, matching variables are added to the list. (See "Examples" below.)

area_ptr
is a pointer to an area in which the output value_list_info structure is to be allocated. (Input)

value_list_info_ptr
is a pointer to the following structure, allocated by value_$list and freed by the caller when done. (Output) It is also declared in the include file value_structures.incl.pl1:

```
dcl 1 value_list_info   aligned based (value_list_info_ptr),
      2 version         fixed bin,      /* = 1 */
      2 pair_count      fixed bin,
      2 chars_len       fixed bin (21),
      2 pairs           (alloc_pair_count refer
                         (value_list_info.pair_count)),
        3 type_switches bit (36),
        3 name_index    fixed bin (21),
        3 name_len      fixed bin (21),
        3 value_index   fixed bin (21),
        3 value_len     fixed bin (21),
      2 chars           char (alloc_chars_len refer
                         (value_list_info.chars_len));
```

For each pair (i), the variable name is:

```
substr (chars, name_index (i), name_len (i))
```

and the value is:

```
substr (chars, value_index (i), value_len (i))
```

code
is a standard status code. (Output)

NOTES

The user requires r access on the value segment, except for perprocess values.

Names are returned in alphabetical order.

The user is responsible for freeing the value_list_info structure when done.

EXAMPLES

In the match_info structure, names are matched in the order given. Those with exclude_sw OFF add to the list (union) and those with exclude_sw ON narrow down the list (intersection). For example, assume the defined variables to be rs_seg_length, rs_area_length, rs_str_ptr, rs_str_len, arg_str_ptr, and arg_str_len, and assume the following entries in match_info:

```
(exclude_sw)        (name)
OFF                 /_len/
ON                  /_length/
OFF                 /seg_length/
```

The first name causes the list of selected variables to be:

```
rs_seg_length, rs_area_length, rs_str_len, arg_str_len
```

The next name (with exclude_sw ON) produces the intersection of this set with the set of names NOT matching /_length/:

```
rs_str_len, arg_str_len
```

The last name produces the union of this set with the set of names matching /seg_length/:

```
rs_str_len, arg_str_len, rs_seg_length
```

which is the set of names returned in value_info.

Entry: value_$list_data_names

This entry point operates exactly the same as value_$list, but returns variables set by value_$set_data instead of value_$set, and does not return the values. Instead, it sets value_list_info.value_len to the number of words in the value.

USAGE

```
declare value_$list_data_names entry (ptr, bit(36) aligned, ptr, ptr,
      ptr, fixed bin(35));

call value_$list_data_names (seg_ptr, switches, match_info_ptr,
      area_ptr, value_list_info_ptr, code);
```

ARGUMENTS

seg_ptr
: is a pointer to the base of a value segment. (Input) If seg_ptr is null, the default value segment is used.

switches
: is a bit (36) aligned word of switches: (Input)

  perprocess
  : If bit number one is ON, looks for perprocess values, as opposed to those stored in the value segment. Either this switch or "permanent" must be on. If both switches are on, both kinds of values are listed.

  permanent
  : If bit number two is ON, looks for values stored in the value segment.

match_info_ptr
: is a pointer to the user-allocated match_info structure (declared in value_structures.incl.pl1, described under the value_$list entry point above. (Input)

  If a name's regexp_sw is ON, the name is a regular expression to be matched. Otherwise, it is a starname to be matched. If the name's exclude_sw is ON, variables matching the name are excluded from the list built up so far, as for the -exclude control argument to the value_list command. Otherwise, matching variables are added to the list. (See "Examples" below.)

area_ptr
: is a pointer to an area in which the output value_list_info structure is to be allocated. (Input)

value_list_info_ptr
: is a pointer to the value_list_info structure (described under the value_$list entry point above), allocated by value_$list_data_names and freed by the caller when done. (Output) It is also declared in the include file value_structures.incl.pl1.

For each pair (i), the variable name is:

```
substr (chars, name_index (i), name_len (i))
```

The first bit in type_switches (i) is ON if the variable is perprocess, the second is ON instead for a variable stored in the value segment.

code
: is a standard status code. (Output)

Entry: value_$set

This entry point defines a value for a name, readable by value_$get.

USAGE

```
declare value_$set entry options (variable);

call value_$set (seg_ptr, switches, name, new_value, old_value, code);
```

ARGUMENTS

seg_ptr
: is a pointer to the base of a value segment. (Input) If seg_ptr is null, the default value segment is used.

switches
: is a bit (36) aligned word of switches: (Input)
  perprocess
  : If bit number one is ON, sets a perprocess value, as opposed to one stored in the value segment. Either this switch or "permanent" must be on. If both switches are on, a perprocess value is set if one already exists, otherwise a value is set in the value segment.

  permanent
  : If bit number two is ON, sets a value in the value segment.

name
: is a fixed-length or varying character string. (Input) If fixed-length, trailing blanks are trimmed. There must be at least one character.

new_value
: is the value to be set, having any data type. (Input) If conversion to the internal character string representation cannot be performed, error_table_$badcall is returned.

old_value
is the current value, having any data type. (Output) If no value is currently defined, the value of this argument is not changed. If conversion from the internal character string representation cannot be performed, error_table_$bad_conversion is returned.

code
is a standard error code. (Output) Having no previous value defined does not cause an error code to be returned.

NOTES

The user requires rw access to the value segment, except for perprocess values.

Entry: value_$set_data

This entry point defines the value for a name to be a specified number of words of data, readable by value_$get_data. Values set by this entry point cannot be seen by value_$get or value_$defined.

USAGE

```
declare value_$set_data entry (ptr, bit(36), char(*), ptr,
      fixed bin(18), ptr, ptr, fixed bin(18), fixed bin(35));

call value_$set_data (seg_ptr, switches, name, new_data_ptr,
      new_data_size, area_ptr, old_data_ptr, old_data_size, code);
```

ARGUMENTS

seg_ptr
is a pointer to the base of a value segment. (Input) If seg_ptr is null, the default value segment is used.

switches
is a bit (36) aligned word of switches: (Input)
perprocess
If bit number one is ON, sets a perprocess value, as opposed to one stored in the value segment. Either this switch or "permanent" must be on. If both switches are on, a perprocess value is set if one already exists, otherwise a value is set in the value segment.
permanent
If bit number two is ON, sets a value in the value segment.

name
is a character string with at least one nonblank character. (Input) Trailing blanks are trimmed.

new_data_ptr
is a pointer to the value to be set. (Input)

new_data_size
is the number of words in the value to be set. (Input)

area_ptr
if nonnull, points to an area in which the old (return) value is to be allocated. (Input) If null, the old value is not returned.

old_data_ptr
is a pointer to the old value. (Output)

old_data_size
is the number of words returned as the old value. (Output)

code
is a standard status code. (Output) Having no previous value defined does not cause an error code to be returned.

NOTES

The user requires rw access on the value segment, except for perprocess values.

Entry: value_$set_path

This entry point sets the default value segment used by the value commands with no −pathname argument.

USAGE

```
declare value_$set_path entry (char(*), bit(1), fixed bin(35));

call value_$set_path entry (path, create_sw, code);
```

ARGUMENTS

path
is the pathname. (Input) The value suffix is assumed.

create_sw
is ON to create a value segment if none exists. (Input)

code
is a standard status code. (Output) If it is error_table_$no_w_permission, the value segment has been set. Any other nonzero code indicates that the value segment was not set.

Entry: value_$test_and_set

This entry point defines a new value for a name, only if the name has a specified current value.

USAGE

```
declare value_$test_and_set entry options (variable);

call value_$test_and_set (seg_ptr, switches, name, new_value, old_value,
      code);
```

ARGUMENTS

seg_ptr
: is a pointer to the base of a value segment. (Input) If seg_ptr is null, the default value segment is used.

switches
: is a bit (36) aligned word of switches: (Input)
  perprocess
  : If bit number one is ON, tests and sets a perprocess value, as opposed to one in the value segment. Either this switch or "permanent" must be on. If both switches are on, the perprocess value is tested if one is defined, otherwise the one in the value segment is tested. The value set is of the same type as the value tested.

  permanent
  : If bit number two is ON, tests and sets the value in the value segment.

name
: is a fixed-length or varying character string. (Input) If fixed-length, trailing blanks are trimmed. There must be at least one character.

new_value
: is the value to be set, having any data type. (Input) If conversion to the internal character string representation cannot be performed, the error code error_table_$badcall is returned.

old_value
: is the caller-supplied value that must equal the value currently defined in order for the new value to be set. (Input)

code
: is a standard status code. (Output) It is error_table_$action_not_performed if old_value does not match the currently defined value.

NOTES

The user requires rw access on the value segment, except for perprocess values.

If the value tested is perprocess, the value set is also perprocess, and vice-versa.

Entry: value_$test_and_set_data

This entry point defines the value for a name to be a specified number of words of data, readable by value_$get_data, only if the first N words of the name's current value have specified contents.

USAGE

```
declare value_$test_and_set_data entry (ptr, bit(36), char(*), ptr,
      fixed bin(18), ptr, fixed bin(18), fixed bin(35));

call value_$test_and_set_data (seg_ptr, switches, name, new_data_ptr,
      new_data_size, old_data_ptr, old_data_size, code);
```

ARGUMENTS

seg_ptr
: is a pointer to the base of a value segment. (Input) If seg_ptr is null, the default value segment is used.

switches
: is a bit (36) aligned word of switches: (Input)
  perprocess
  : If bit number one is ON, tests and sets a perprocess value, as opposed to one in the value segment. Either this switch or "permanent" must be on. If both switches are on, the perprocess value is tested if one is defined, otherwise the one in the value segment is tested. The value set is of the same type as the value tested.

  permanent
  : If bit number two is ON, tests and sets the value in the value segment.

name
: is a character string with at least one nonblank character. (Input) Trailing blanks are trimmed.

new_data_ptr
: is a pointer to the value to be set. (Input) If null, the current value is deleted and no value is defined.

new_data_size
: is the number of words in the value to be set. (Input)

old_data_ptr
is a pointer to some data, whose first old_data_size words must equal the first old_data_size words of the name's current value in order for the new value to be set. (Input)

old_data_size
is the number of words to be compared. (Input) This number can be less than the number of words in the name's current value (used, for example, to compare only the header of a structure), but an error code is returned if it is greater.

code
is a standard status code. (Output) It is error_table_$action_not_performed if the old-value match fails.

NOTES

The user requires rw access on the value segment, except for perprocess values.

If the value tested is perprocess, the value set is also perprocess, and vice-versa.

The value of a name can be conditionally deleted by passing a null new_data_ptr.

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);
```

ARGUMENTS

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

entryname
is the entryname of the file of interest. (Input) If the entry is a link, the information returned pertains to the entry to which it points.

info_ptr
is a pointer to the structure in which information is to be returned. (Input) See "File Information" below.

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

FILE INFORMATION

The info_ptr argument points to one of the following structures which are all declared in the include file vfs_info.incl.pl1.

Unstructured Files

For unstructured files, use the following structure:

```
dcl 1 uns_info              based (addr (info)),
      2 info_version        fixed bin,
      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:

info_version
: identifies the version of the info structure; this must be set to the named constant vfs_version_1.

type
: identifies the file type and the info structure returned:

1 unstructured	3 blocked
2 sequential	4 indexed

bytes
: gives the length of the file, not including the header in bytes.

header_present
: if set, indicates that an optional header is present.

header_id
: contains the identification from the header of the file, if present. Its meaning is defined by the user.

Sequential Files

For sequential files, use the following structure:

```
dcl 1 seq_info              based (addr (info)),
      2 info_version        fixed bin,
      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:

info_version
identifies the version of the info structure; this must be set to the named constant vfs_version_1.

type
identifies the file type and the info structure returned:

1	unstructured	3	blocked
2	sequential	4	indexed

records
is the number of records in the file, including those of zero length.

lock_status
if zero, indicates that the lock of the file 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

version
identifies the version number of the file and its creating program.

Blocked Files

For blocked files, use the following structure:

```
dcl 1 blk_info              based (addr (info)),
      2 info_version        fixed bin,
      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:

info_version
: identifies the version of the info structure; this must be set to the named constant vfs_version_1.

type
: identifies the file type and the info structure returned:

1	unstructured	3	blocked
2	sequential	4	indexed

records
: is the number of records in the file, including those of zero length.

lock_status
: if zero, indicates that the lock of the file 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

version
: identifies the version number of the file and its creating program.

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

max_rec_len
: is the maximum record length (in bytes) associated with the file.

Indexed Files

For indexed files, use the following structure:

```
dcl  1 indx_info              based (addr (info)),
       2 info_version         fixed bin,
       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:

info_version
: identifies the version of the info structure; this must be set to the named constant vfs_version_1.

type
: identifies the file type and the info structure returned:

1	unstructured	3	blocked
2	sequential	4	indexed

records
: is the number of records in the file, including those of zero length.

lock_status
: if zero, indicates that the lock of the file 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

file_version
: identifies the version number of the file.

program_version
: identifies the version number of vfile_ that created the file.

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

non_null_recs
: is a count, not including those of zero length, of the records in the file.

record_bytes
: is the total length of all records in the file in bytes.

free_blocks
: is the number of blocks in the free space of the file list for records.

index_height
: is the height of the index tree (equal to zero if file is empty).

nodes
: is the number of single page nodes in the index.

key_bytes
: is the total length of all keys in the file in bytes.

change_count
: is the number of times the file has been modified.

num_keys
: is the total number of index entries, each associating a key with a record.

dup_keys
: is the number of index entries with nonunique keys, not including the first instance of each key.

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. The following declaration:

```
dcl info aligned like indx_info;
```

will provide the necessary space for the largest of the structures.

See the description of the vfile_ I/O module for further details.

Name: video_data_

The video_data_ subroutine is a data segment containing information about the video system.

Entry: video_data_$terminal_iocb

This is the terminal control switch IOCB pointer. If the video system is activated for the user's terminal, this pointer is nonnull, and points to the IOCB for the switch user_terminal_.

USAGE

```
declare video_data_$terminal_iocb pointer external static;
```

NOTES

User programs may use this pointer for two purposes:

1. Inquiring as to whether the video system is activated, by checking to see if the pointer is null.

2. Determining the physical characteristics and capabilities of the terminal. This may be accomplished with the get_capabilities control order, described under the window_io_ I/O module. The height and width returned will be that of the physical terminal screen.

   No other manipulations of this switch are permitted.

Name: video_utils_

This subroutine provides interfaces for activating and de-activating the video system.

Entry: video_utils_$turn_on_login_channel

This entry removes the existing attachment of the user's terminal, replacing it with the video system. When this entry returns successfully, the switch user_terminal_ is attached through tc_io_ to the user's terminal. The switch user_ i/o is attached through window_io_ to a window covering the entire screen. invoked: vertsp, can, erkl, esc, red, and ctl_char. In addition, if ^pl is set on video system invocation, ^more will be set in the video system. (For more details on modes, see the window_io_ I/O module.) Similarly, the settings of the current erase and kill characters are copied when the video system is invoked. (See "Real-Time Editing" for details.) To see how the standard I/O switch attachments change when you activate the video system on your terminal, refer to Figure A-2 in Appendix A.

USAGE

```
declare video_utils_$turn_on_login_channel entry (fixed bin (35),
     char (*));

call video_utils_$turn_on_login_channel (code, reason);
```

ARGUMENTS

code
: is a standard system error code. (Output)

reason
: contains information about the error, if there is one. (Output) (128 characters are enough to hold any message that may be returned in reason.)

NOTES

If the video system is already in service on the user's terminal, the status code video_et_$wsys_invoked is returned, and the value of reason is not defined.

If the activation of the video system fails, the original attachment of the terminal (through tty_) is restored, and information is returned in reason and code.

In particular, if the switch user_i/o is not currently attached through tty_, the code video_et_$switch_not_attached_with_tty_ is returned. This may indicate that the user has auditing or the graphic system in place. The message returned in reason advises the user to remove graphics or auditing and try again.

Entry: video_utils_$turn_off_login_channel

This entry reverses the actions of video_utils_$turn_on_login_channel. That is, it removes the window attachment of user_i/o, detaches terminal control from the user's terminal, and attaches user_i/o to the user's terminal via tty_. The settings of the following modes are copied when when the video system is revoked: vertsp, can, erkl, esc, red, and ctl_char. If ^more is set while in the video system, ^pl mode will be set after revoking the video system. (For more details on modes, see the window_io_ I/O module.) Similarly, the settings of the current erase and kill characters are copied when the video system is revoked. (See "Real-Time Editing" for details.) It is the user's responsibility to detach any windows other than user_io before calling this entry point

USAGE

```
declare video_utils_$turn_off_login_channel entry (fixed bin (35));

call video_utils_$turn_off_login_channel (code);
```

ARGUMENTS

code
: is a standard system error code. (Output) It is nonzero if and only if the video system can not be removed from the user's terminal.

Name: virtual_cpu_time_

The virtual_cpu_time_ function 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_ ();
```

ARGUMENTS

time
: is the virtual CPU time, in microseconds, used by the calling process. (Output)

NOTES

The vclock builtin function should be used in PL/I programs instead of this subroutine, because it is more efficient.

Name: window_

The window_ subroutine provides a terminal independent interface to video terminal operations. More specifically, it controls and performs I/O to a window.

The window_ subroutine is used in conjunction with the iox_ subroutine call entry points in the window_io_ I/O module. The window_ and video_utils_ subroutines together perform the same functions as the window_call command.

The virtual terminal implemented by window_ corresponds closely to common video terminals. The features of the terminal are defined implicitly by the entries below. Not all entries can be supported on all terminals. The result of calling an unsupported feature is the error code video_et_$capability_lacking. Programs can determine whether the device in question supports a given operation by using a get_capabilities control order, described under the window_io_ I/O module.

Additional terminals may be supported by defining their video attributes in the Terminal Type File (TTF). The TTF is described in the *Multics Programmer's Reference Manual*, Order No. AG91.

Some entry points require that the current cursor position be defined when they are called. The current position is defined unless a call is made to the write_raw_text entry point, or an asynchronous event changes the window contents. If the current position is not defined, these entry points will return the status code video_et_$cursor_position_undefined.

If an asynchronous event changes the state of the window, status will be set for the window. Once window status is set, all calls to window_ on that window will return the status code video_et_$window_status_pending until a get_window_status control order is used to pick up the status.

The calling sequences for all the entry points are in the include file window_dcls.incl.pl1.

Entry: window_$bell

This entry activates the terminal alarm. For most terminals, this will be the audible bell. For some it will be a visible signal.

USAGE

```
declare window_$bell entry (ptr, fixed bin (35));

call window_$bell (iocb_ptr, code);
```

ARGUMENTS

iocb_ptr
: is a pointer to an IOCB for a switch attached with window_io_. (Input)

code
: is a standard system error code. (Output)

NOTES

The current cursor position must be dfined for this call. If the cursor is in some other window on the screen when this call is made, it is moved to the current position in this window.

Entry: window_$change_column

This entry moves the cursor to a different column on the current line, without changing the line.

USAGE

```
declare window_$change_column entry (ptr, fixed bin, fixed bin (35));

call window_$change_column (iocb_ptr, new_column, code);
```

ARGUMENTS

iocb_ptr
: is a pointer to an IOCB for a switch attached with window_io_. (Input)

new_column
: is the new column. (Input)

code
: is a standard system error code. (Output)

NOTES

The current cursor position must be defined.

Entry: window_$change_line

This entry moves the cursor to a new line without changing the column.

USAGE

```
declare window_$change_line entry (ptr, fixed bin, fixed bin (35));

call window_$change_line (iocb_ptr, new_line, code);
```

ARGUMENTS

iocb_ptr
is a pointer to an IOCB for a switch attached with window_io_. (Input)

new_line
is the new line. (Input)

code
is a standard system error code. (Output)

Entry: window_$clear_region

This entry replaces the contents of the region specified with spaces, and leaves the cursor at the upper left-hand corner of the region. The region is defined by giving the upper left-hand corner (line and column), and the width and height of the region.

USAGE

```
declare window_$clear_region entry (ptr, fixed bin, fixed bin, fixed
      bin, fixed bin, fixed bin (35));

call window_$clear_region (iocb_ptr, start_line, start_col, n_lines,
      n_cols, code);
```

ARGUMENTS

iocb_ptr
is a pointer to an IOCB for a switch attached with window_io_. (Input)

start_line
is the number of the line where clearing will begin. (Input)

start_col
is the number of the column where clearing will begin. (Input)

n_lines
is the number of lines which will be cleared. (Input)

n_cols
is the number of columns which will be cleared. (Input)

code
is a standard system error code. (Output)

NOTES

The rectangular region described in cleared. The cursor position defined at (start_line, start_col).

Entry: window_$clear_to_end_of_line

This entry clears everything to the right of the cursor on the current line to spaces. Positions to the left of the cursor are not affected. The cursor is not moved.

USAGE

```
declare window_$clear_to_end_of_line entry (ptr, fixed bin (35));

call window_$clear_to_end_of_line (iocb_ptr, code);
```

ARGUMENTS

iocb_ptr
is a pointer to an IOCB for a switch attached with window_io_. (Input)

code
is a standard system error code. (Output)

NOTES

The cursor position must be defined.

Entry: window_$clear_to_end_of_window

This entry clears all of the window between the cursor and the end of the window. This includes everything to the right of the cursor on the current line, and all lines below the cursor. The cursor is not moved.

USAGE

```
declare window_$clear_to_end_of_window entry (ptr, fixed bin (35));

call window_$clear_to_end_of_window (iocb_ptr, code);
```

ARGUMENTS

iocb_ptr
is a pointer to an IOCB for a switch attached with window_io_. (Input)

code
is a standard system error code. (Output)

NOTES

The current cursor position must be defined.

Entry: window_$clear_window

This entry clears the entire window to spaces, and leaves the cursor at home.

USAGE

```
declare window_$clear_window entry (ptr, fixed bin (35));

call window_$clear_window (iocb_ptr, code);
```

ARGUMENTS

iocb_ptr
is a pointer to an IOCB for a switch attached with window_io_. (Input)

code
is a standard system error code. (Output)

NOTES

The cursor position is defined to be at line 1, column 1 after the screen is cleared.

Entry: window_$create

This entry creates a new window on the terminal screen.

USAGE

```
declare window_$create entry (ptr, ptr, ptr, fixed bin (35));

call window_$create (terminal_iocb_ptr, window_info_ptr,
      window_iocb_ptr, code);
```

ARGUMENTS

terminal_iocb_ptr
is a pointer to an IOCB for the terminal control switch. (Input) Normally this should be video_data_$terminal_iocb.

window_info_ptr
is a pointer to a standard window_position_info structure, as declared in window_control_info.incl.pl1. (Input)

window_iocb_ptr
is a pointer to a detached IOCB pointer. (Input) It may be obtained with iox_$find_iocb which must be done before the call to window_$create. For example:

```
call iox_$find_iocb ("top_window", window_iocb_ptr, code);
```

where the value returned for window_iocb_ptr is used in the call to window_$create.

code
is a standard system error code. (Output)

NOTES

The window_info_ptr must point to a window_position_info structure, as declared in window_control_info.incl.pl1. If window_position_info.width is set to zero, the window will occupy the full width of the screen. Currently windows must occupy the full
* width of the screen. If window_position_info.height is set to zero, the remainder of the screen is used. The iocb_ptr is an input argument, iox_$find_iocb may be used to obtain an iocb_ptr for a new switch.

Entry: window_$delete_chars

This entry deletes characters on the current line. Characters to the right of the cursor are moved to the left. Character positions opened up on the right margin are filled with spaces. It is an error to call this entry point if the terminal does not support the delete chars operation.

USAGE

```
declare window_$delete_chars entry (ptr, fixed bin, fixed bin (35));

call window_$delete_chars (iocb_ptr, n_chars, code);
```

ARGUMENTS

iocb_ptr
: is a pointer to an IOCB for a switch attached with window_io_. (Input)

n_chars
: is the number of characters (starting at the current cursor position) that will be removed from the screen. (Input) If n_chars is zero, no action is taken.

code
: is a standard system error code. (Output)

NOTES

The current cursor position must be defined. The number of characters specified by n_chars are deleted, and the remaining characters on the line, if any, move leftward to occupy the space.

Entry: window_$destroy

This entry destroys an existing window, leaving its IOCB in a detached state.

USAGE

```
declare window_$destroy entry (ptr, fixed bin (35));

call window_$destroy (window_iocb_ptr, code);
```

ARGUMENTS

window_iocb_ptr
: is a pointer to an IOCB attached with window_$create. (Input)

code
    is a standard system error code. (Output)

Entry: window_$edit_line

This entry allows applications to preload the video editor input buffer with a string.

USAGE

```
declare window_$edit_line entry (pointer, pointer, pointer, fixed bin
      (21), fixed bin (21), fixed bin (35));

call window_$edit_line (window_iocb_ptr, window_edit_line_info_ptr,
      buffer_ptr, buffer_len, n_returned, code);
```

ARGUMENTS

window_iocb_ptr
    is a pointer to an IOCB for a switch attached with window_io. (Input)

window_edit_line_info_ptr
    is a pointer to a window_edit_line_info structure, as declared in window_control_info.incl.pl1 (described below). (Input)

*
buffer_ptr
    is a pointer to a buffer where the users input will be put. (Input)

buffer_len
    is the size of the input buffer. (Input)

n_returned
    is the number of characters in the final output line. (Output)

code
    is a standard system error code. (Output)

NOTES

The window_edit_line_info structure is declared in window_control_info_incl.pl1:

```
dcl  1 window_edit_line_info    based (window_edit_line_info_ptr),
       2 version                char (8),
       2 line_ptr               ptr,
       2 line_length            fixed bin (21);

dcl  window_edit_line_info_version_1
                          char (8) static options (constant) init ("wed10001");

dcl  window_edit_line_info_ptr  ptr;
```

STRUCTURE ELEMENTS

version
: is the version number of the structure. This is currently window_edit_line_version_1. (Input)

line_ptr
: is a pointer to the initial text string to be loaded into the input buffer before editing begins. (Input)

line_length
: is the length of the string pointed to by line_ptr. (Input)

This page intentionally left blank.

Entry: window_$get_cursor_position

This entry is used to return the current position of the cursor. If the last operation done to the terminal was in some other window, this will not be the actual position of the cursor on the screen.

USAGE

```
declare window_$get_cursor_position entry (ptr, fixed bin, fixed bin,
      fixed bin (35));

call window_$get_cursor_position (iocb_ptr, line, col, code);
```

ARGUMENTS

iocb_ptr
    is a pointer to an IOCB for a switch attached with window_io_. (Input)

line
    is the line number. (Output)

col
    is the column position. (Output)

code
    is a standard system error code. (Output)

NOTES

The current cursor position must be defined.

Entry: window_$get_echoed_chars

This entry accepts input from the typist, echoing the characters as typed, until either a specified number of characters are read, or a break character is encountered. By default, the break characters are the control characters plus DEL (177 octal).

USAGE

```
declare window_$get_echoed_chars entry (ptr, fixed bin (21), char (*),
      fixed bin (21), char (1) varying, fixed bin (35));

call window_$get_echoed_chars (iocb_ptr, n_to_get, buffer, n_got, break,
      code);
```

ARGUMENTS

iocb_ptr
is a pointer to an IOCB for a switch attached with window_io_. (Input)

n_to_get
is the number of columns (N) between the cursor and the end of the line. (Input) At most N characters will be returned.

buffer
is the caller-supplied buffer that holds characters returned. (Input)

n_got
is the number of characters returned. (Output) Each character is echoed.

break
is the character that causes the echoing to stop. (Output) This character is not echoed.

code
is a standard system error code. (Output)

NOTES

This entry point returns no more than n_to_get characters in buffer. It reads and echoes characters until either (1) it has read n_to_get characters, or (2) it has read a break character. If it stops due to a break character, the break character is returned in break, otherwise break is equal to "".

Entry: window_$get_one_unechoed_char

This entry reads a single character, unechoed, from the terminal. Optionally, it can return instead of waiting if there are no characters available.

USAGE

```
declare window_$get_one_unechoed_char entry (ptr, char (1) varying,
      bit (1) aligned, fixed bin (35));

call window_$get_one_unechoed_char (iocb_ptr, char_read, block_flag,
      code);
```

ARGUMENTS

iocb_ptr
is a pointer to an IOCB for a switch attached with window_io_. (Input)

char_read
is the read character. (Output) If block_flag is "0"b, and no input is typed ahead, then this will be a zero length character string.

block_flag
if this flag is "1"b, input from the terminal is awaited if none is available. (Input) If it is "0"b, and no input is available, then this entry returns immediately, and sets char_read to "".

code
is a standard system error code. (Output)

NOTES

Beware of the PL/I language definition of character string comparisons when using this entry with a block flag of "0"b. In PL/I, both of the following comparisons are true:

```
(" " = " ")
("" = " ")
```

That is, a zero length varying string compares equally to a single space. To test if char_read is nonempty, use an expression like:

```
(length (char_read) > 0)
```

Entry: window_$get_unechoed_chars

This entry accepts input from the typist, leaving it unechoed, until either a specified number of characters are read, or a break character is encountered.

USAGE

```
declare window_$get_unechoed_chars entry (ptr, fixed bin (21), char (*),
     fixed bin (21), char (1) varying, fixed bin (35));

call window_$get_unechoed_chars (iocb_ptr, n_to_get, buffer, n_got,
     break, code);
```

ARGUMENTS

iocb_ptr
is a pointer to an IOCB for a switch attached with window_io_. (Input)

n_to_get
is the number of columns (N) between the cursor and the end of the line. (Input) At most N characters will be returned.

buffer
is the caller-supplied buffer that holds characters returned. (Input)

n_got
is the number of characters returned. (Output) Each character is echoed.

break
is the character that causes the echoing to stop. (Output) This character is not echoed.

code
is a standard system error code. (Output)

NOTES

This entry point will read no more than n_to_get characters from the terminal, without echoing them to the typist. The characters are returned in the buffer. Characters are read until either (1) n_to_get characters are read, or (2) a break character is read. If reading stops due to a break character, then the break character is returned in break. Otherwise break is ""

Entry: window_$insert_text

This entry inserts text at the current cursor position. Text at the cursor or to the right of the cursor is shifted to the right, to accommodate the new text. It is an error to call this entry if the terminal does not support the insertion of text.

USAGE

```
declare window_$insert_text entry (ptr, char (*), fixed bin (35));

call window_$insert_text (iocb_ptr, text, code);
```

ARGUMENTS

iocb_ptr
: is a pointer to an IOCB for a switch attached with window_io_. (Input)

text
: is the character string to be written. (Input) When converted to output, each character in this string must occupy exactly one print position. The length of this string must be such that characters moved to the right will remain on the current line in the window. If these conditions are not met, the result is undefined. The cursor is left after the last character inserted.

code
: is a standard system error code. (Output)

NOTES

The current cursor position must be defined. The string "text" must contain only printable ASCII graphics. If it contains any other characters, the status code video_et_$string_not_printable is returned.

Entry: window_$overwrite_text

This entry writes text on the window in the current cursor location. If there is any text at or to the right of the current cursor position in the window, it is overwritten with the supplied string.

USAGE

```
declare window_$overwrite_text entry (ptr, char (*), fixed bin (35));

call window_$overwrite_text (iocb_ptr, text, code);
```

ARGUMENTS

iocb_ptr
: is a pointer to an IOCB for a switch attached with window_io_. (Input)

text
: is the character string to be written. (Input) This string should consist of only printable ASCII graphics (octal codes 040 through 176 inclusive), and may not be longer than the space remaining on the current line.

code
: is a standard system error code. (Output)

NOTES

The cursor position must be defined. The string "text" may contain only printable ASCII graphics. If it contains anything else the status code video_et_$string_not_printable is returned.

Entry: window_$position_cursor

This entry moves the cursor to any requested position in the window. It defines the current cursor position if it is undefined.

USAGE

```
declare window_$position_cursor entry (ptr, fixed bin, fixed bin,
      fixed bin (35));

call window_$position_cursor (iocb_ptr, line, col, code);
```

ARGUMENTS

iocb_ptr
is a pointer to an IOCB for a switch attached with window_io_. (Input) line is the line number. (Input)

col
is the column position. (Input)

code
is a standard system error code. (Output)

Entry: window_$position_cursor_rel

The entry moves the cursor relative to the current location.

USAGE

```
declare window_$position_cursor_rel entry (ptr, fixed bin, fixed bin,
      fixed bin (35));

call window_$position_cursor_rel (iocb_ptr, line_inc, col_inc, code);
```

ARGUMENTS

iocb_ptr
is a pointer to an IOCB for a switch attached with window_io_. (Input)

line_inc
is the change in line number. (Input) If line_inc is a positive number, the cursor is moved down. If it is a negative number, the cursor is moved up. If it is zero, the cursor's line number is not changed.

col_inc
is the change in column position. (Input) If col_inc is a positive number, the cursor is moved to the right. If it is a negative number, the cursor is moved to the left. If it is zero, the cursor's column position is not changed.

code
is a standard system error code. (Output)

Entry: window_$scroll_region

This entry scrolls a region up or down a given number of lines. A positive scroll count scrolls the window up, deleting lines from the top of the window and adding new blank lines to the bottom. The cursor's new position is at the beginning of the first new blank line. A negative count scrolls the window down, deleting lines from the bottom and adding lines to the top. The cursor is left at home. If this entry is called and the terminal does not support either scrolling or insert and delete lines, the result is an error status, video_et_$capabilities_lacking.

USAGE

```
declare window_$scroll_region entry (ptr, fixed bin, fixed bin, fixed
      bin, fixed bin (35));

call window_$scroll_region (iocb_ptr, start_line, n_lines,
      scroll_distance, code);
```

ARGUMENTS

iocb_ptr
is a pointer to an IOCB for a switch attached with window_io_. (Input)

start_line
is the number of the first line of the region. (Input)

n_lines
is the number of lines that compose the region. (Input)

scroll_distance
is the distance in lines by which the region will be scrolled. (Input)

code
: is a standard system error code. (Output)

NOTES

The cursor position is defined to be column one on first_line. The region from first_line for n_lines is scrolled scroll_distance lines, which may be negative.

Entry: window_$sync

This entry synchronizes the process with the typist by writing any pending output to the terminal.

USAGE

```
declare window_$sync entry (ptr, fixed bin (35));

call window_$sync (iocb_ptr, code);
```

ARGUMENTS

iocb_ptr
: is a pointer to an IOCB for a switch attached with window_io_. (Input)

code
: is a standard system error code. (Output)

NOTES

The calling process is made to wait until the typist types something after the last text output has been transmitted to the terminal.

Entry: window_$write_raw_text

This entry is used to output a terminal dependent sequence. The current cursor position becomes undefined after this call is made. This entry should not be used to output sequences that put graphics onto the terminal screen, as the video system's internal screen image will become inconsistent. This entry is used for terminal-specific features that cannot be accessed via the video system.

USAGE

```
declare window_$write_raw_text entry (ptr, char (*), fixed bin (35));

call window_$write_raw_text (iocb_ptr, text, code);
```

ARGUMENTS

iocb_ptr
is a pointer to an IOCB for a switch attached with window_io_. (Input)

text
is any string of printable ASCII characters to be transmitted to the terminal. (Input)

code
is a standard system error code. (Output)

NOTES

Any call to window_$write_raw_text causes the cursor position to become undefined and sets the screen_invalid window status flag. Subsequent calls to write_raw_text will ignore this flag, but all other window_ entrypoints will return the status code video_et_$window_status_pending until the status flag is cleared. It is the responsibility of the application performing the raw output call to perform a get_window_status control order to clear the status flag.

Entry: window_$write_sync_read

This entry writes a prompt, synchronizes input to the output of the prompt, and reads a response. This entry is useful for queries where it is important to avoid interpreting type-ahead as a response to a question.

USAGE

```
declare window_$write_sync_read entry (ptr, char (*), fixed bin (21),
      char (*), fixed bin (21), char (1) varying, fixed bin (35));

call window_$write_sync_read (iocb_ptr, prompt, n_to_get, buffer, n_got,
      break, code);
```

ARGUMENTS

iocb_ptr
is a pointer to an IOCB for a switch attached with window_io_. (Input)

prompt
is a string of printable ASCII characters which must fit on the current line. (Input)

n_to_get
is the number of columns (N) between the cursor and the end of the line. (Input) At most N characters will be returned.

buffer
is the caller-supplied buffer that holds characters returned. (Input)

n_got
is the number of characters returned. (Output) Each character is echoed.

break
is the character that causes the echoing to stop. (Output) This character is not echoed.

code
is a standard system error code. (Output)

NOTES

The current cursor position must be defined. This entry overwrites the text string "prompt" at the current cursor position. It then reads characters typed after the prompt has been transmitted to the terminal. The characters are read in the same fashion as the get_unechoed_chars entry point. Any characters read before the prompt is transmitted, are buffered and returned to get_echoed_chars or subsequent get_unechoed_chars calls.

Name: write_allowed_

The write_allowed_ function determines whether a subject of specified authorization has access (with respect to the access isolation mechanism) to append (but not modify or destroy) data to an object of specified access class. For information on access class, see the *Multics Programmer's Reference Manual.*, Order No. AG91.

USAGE

```
declare write_allowed_ entry (bit(72) aligned, bit(72) aligned) returns
      (bit(1) aligned);
```

```
returned_bit = write_allowed_ (authorization, access_class);
```

ARGUMENTS

authorization
: is the authorization of the subject. (Input)

access_class
: is the access class of the object. (Input)

returned_bit
: indicates whether the subject is allowed to write the object. (Output)
"1"b write is allowed.
"0"b write is not allowed.

SECTION 3

SYSTEM INPUT/OUTPUT MODULES

The Multics input/output (I/O) system, described in detail in the Programmer's Reference Manual, makes use of various I/O modules to perform input and output operations. An I/O module is a system- (or user-) written program that controls a physical device and acts as an intermediary between the device and application program. The attachment may also be to something other than a peripheral device, e.g., a file in the storage system.

I/O operations in Multics involve the attachment of an I/O switch to the I/O module. The basic tool for making attachments is the iox_ subroutine (described in section 2). Alternatively, attachments can be performed from command level by use of the io_call command. The I/O facilities of the programming languages can also be used to specify the attachment.

The I/O modules contained in this section include the formats of attach descriptions, syntax of operations from command level, and information as needed concerning support for the different I/O operations.

The I/O modules described in this section and their functions are:

ansi_tape_io_
: is an mtape_ per-format module that supports I/O to and from ANSI standard tapes under control of the mtape_ I/O module.

audit_
: provides a mechanism for auditing and editing I/O on a switch.

bisync_
: performs stream I/O over a binary synchronous communications channel.

cross_ring_
: allows an outer ring to attach a switch to a preexisting switch in an inner ring to perform I/O operations.

discard_
: is a sink for unwanted output.

g115_
: performs stream I/O from/to a Honeywell Level 6 G115 data transmission terminal.

hasp_host_
: simulates record-oriented I/O to a single device of a workstation while communicating with a host system using the HASP communications protocol.

hasp_workstation_
: performs record-oriented I/O to a single device of a remote terminal that supports the HASP communications protocol.

ibm2780_
: performs stream I/O from/to a device similar to the IBM 2780 data transmission terminal.

ibm3270_
: performs stream I/O from/to a device similar to the IBM 3270 data transmission terminal.

ibm3780_
: performs stream I/O from/to a device similar to the IBM 3780 data transmission terminal.

ibm_pc_io_
: supports I/O between a Multics process and a microcomputer that runs the IBM PC-to-Host data transfer protocol.

ibm_tape_io_
: is an mtape_ per-format module that supports I/O to and from IBM standard labeled, unlabled, and DOS formatted tapes under control of the mtape_ I/O module.

mtape_
: supports I/O to and from magnetic tape volumes written in ANSI or IBM format.

rdisk_
: supports I/O from/to removable disk packs.

record_stream_
: provides a mechanism for doing record I/O on an unstructured file and stream I/O on a structured file.

remote_input_
: performs record input from a terminal I/O module that is assumed to be connected to a remote I/O device.

remote_printer_
: formats and controls stream I/O to a remote I/O terminal that has the characteristics of a line printer.

remote_punch_
: formats and controls stream I/O to a remote I/O terminal that has the characteristics of a card punch.

remote_teleprinter_
: formats and controls stream I/O from/to a logical entity that has the characteristics of a teleprinter.

signal_io_
: signals a condition whenever an iox_ I/O operation is performed.

syn_
: establishes one switch as a synonym for another.

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

tape_ibm_
: supports I/O from/to magnetic tape files according to standards established by IBM.

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

tape_nstd_
: supports I/O from/to tapes in nonstandard or unknown formats.

tty_
: supports I/O from/to terminals.

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

xmodem_io_
: supports I/O between a Multics process and a microcomputer that runs the XMODEM data transfer protocol.

This page intentionally left blank.

Name: ansi_tape_io_

The ansi_tape_io_ module is an mtape_ Per-Format module that supports I/O to and from ANSI standard tapes under control of the mtape_ I/O module. The mtape_ ANSI Per-Format module (referred to as the "ANSI PFM" in the remainder of this discussion) may be selected explicitly by the use of the mtape_ attach description control argument "-volume_type ansi", or implicitly if the volume mounted by mtape_ during attachment is recognized by RCP as being a standard ANSI tape. Tapes are processed by the ANSI PFM in accordance with ANSI specification X3.27-1978, referred to in the remainder of this document as "the Standard".

OPENING

Opening of the ANSI PFM is made by the iox_$open_file or the iox_$open entries (via the mtape_ open_file or open entries). The iox_$open_file entry provides for a character string open description, describing file processing attributes to be processed according to the wishes of the caller. The open description arguments accepted by the ANSI PFM are described below. If opening is made by the iox_$open entry, the file processing attributes are formed from the current default values of the ANSI PFM's open description arguments. The open description arguments have an initial default value, which are denoted in their respective descriptions below, or the default values may be changed by the user (see "Default Values" in the mtape_ I/O module description.).

The opening modes supported by the ANSI PFM are sequential_input and sequential_output. If the opening mode specified is sequential_output, then the mtape_ attach description must have specified the -ring control argument or the mtape_ control operation ring_in must have preceded the opening attempt.

OPEN DESCRIPTION

CONTROL ARGUMENTS

-append, -app
: specifies that the requested file is to be appended to the end of the file set as a new file. The requested opening mode must be sequential_output or the file opening will be aborted.

-no_append, -napp
: specifies that the requested file is not to be appended to the end of the file set. (Default)

-block N, -bk N
specifies the block size in bytes for output operations. For input operations, the block size is obtained from the file header label record. Permissible values are from 18 to 99996 bytes. (Default value is 2048 bytes)

-buffer_offset, -bo
specifies that each block will be recorded with an 8 character prefix. A template of a block including this prefix has the following format:

```
dcl 1 tape_block aligned based,
      2 block_size fixed dec (7, 0) unaligned,
      2 block_number fixed dec (7, 0) unaligned,
      2 block_data char (tape_block.block_size - 8) unaligned;
```

where:

block_size
is the block size in 9 bit bytes, including the 8 character prefix.

block_number
is the numerical sequence number of the block within the current physical file, starting at block number 0.

block_data
is the user specified data recorded in the block. The length of this field is governed by the user specified block length.

The block_size and block_number field are recorded in the packed fixed decimal pl1 data type, so that they may be written in the same manner without regard to interface recording mode (nine bit or binary). The buffer offset prefix length is recorded in the ANSI HDR2 label record buffer offset field (character positions 51 and 52).

-no_buffer_offset, -nbo
specifies that no block prefix will be recorded in each data block. (Default)

-comment STR, -com STR
specifies a user comment to be displayed on the user_output I/O switch after the file has been successfully opened. The comment text (STR) may be from 1 to 80 characters in length. (Default is no -comment)

-default_fixed_record N,
specifies the record length to be used for "f" or "fb" formats in the absence of a -record specification. The intended purpose of this control argument is to supply a default value for record size without having to include a -record specification in the open description. If the user wishes to explicitly specify the record length, the -record control argument should be used. Although the -default_fixed_record control argument may appear in a users open description and be processed accordingly, this would not be considered the proper method of explicitly supplying the record length. (Default value is 80)

-default_spanned_record N, -dsr N
specifies the record length to be used for "s" or "sb" formats, in the absence of a -record specification. The intended purpose of this control argument is to supply a default value for record size without having to include a -record specification in the open description. If the user wishes to explicitly specify the record length, the -record control argument should be used. Although the -default_spanned_record control argument may appear in a users open description and be processed accordingly, this would not be considered the proper method of explicitly supplying the record length. (Default value is 1044480)

-default_variable_record N, -dvr N
specifies the record length to be used for "d" or "db" formats, in the absence of a -record specification. The intended purpose of this control argument is to supply a default value for record size without having to include a -record specification in the open description. If the user wishes to explicitly specify the record length, the -record control argument should be used. Although the -default_variable_record control argument may appear in a users open description and be processed accordingly, this would not be considered the proper method of explicitly specifying the record length. (Default value is 2048)

-display, -ds
specifies that the entire open description, after it has been parsed and any necessary defaults added, is to be displayed on the user_output I/O switch.

-no_display, -nds
specifies that the open description will not be displayed on the user_output I/O switch. (Default)

-expires date, -exp date
specifies the expiration date of the file to be created, where date must be of a form acceptable to the convert_date_to_binary_ subroutine. (Default is no -expires)

-extend, -ext
specifies extension of an existing file.

-no_extend, -next
specifies that the requested file is not to be extended. (Default)

-force, -fc
specified that the expiration date of the file being overwritten is to be ignored.

-no_force, -nfc
specifies that the expiration date of a file being overwritten is not to be ignored. If the expiration date is not in the past, the user is queried for permission to overwrite the file. (Default)

-format F, -fmt F
: specifies the record format of the file to be created. Permissible values are: U, F, D, S, FB, DB, and SB. (They may be specified in either upper or lower case.) (Default value is DB)

-generate, -gen
: specifies creating a new "generation" of an existing file by replacement. The file attributes recorded in the file header remains the same as the replaced file, but the generation number in the file header is incremented by 1.

-no_generate, -ngen
: specifies that a new generation of an existing file will not be created. (Default)"

-label_entry entry, -lbe entry
: specifies the entry point of a user subroutine which will be called to process the contents of user label records on input and generate the contents of same, for subsequent writing by mtape_ on output. (See "Calling sequence for user label processing routine" below.) (Default is no -label_entry)

-last_file, -lf
: specifies that the file to be processed is the last file of the file set.

-not_last_file, -nlf
: specifies that the file to be processed may not be the last file of the file set. (Default)

-mode STR, -md STR
: specifies the encoding mode used to record the file data. Permissible values of STR are ascii, ebcdic or binary. (Default value is ascii)

-modify, -mod
: specifies modification of an existing file while retaining the file attributes as recorded in the original files header label records.

-no_modify, -nmod
: specifies that modification of an existing file is not to be performed. (Default)

-name STR, -nm STR
: specifies the file identifier of the requested file. STR can be from 1 to 17 characters. (Default is no -name)

-next_file, -nf
: specifies the file to be processed as the next (or first) file of the file set. This control argument is intended to be used when sequentially processing files. For output operations, if -name or -number are not specified, the values of their respective fields are fabricated by using the next sequential number as the file sequence number and forming the file name by concatenating the string "FILE" with the alphanumeric representation of the file number. (i.e. "FILE0001"). (Default)

-not_next_file, -nnf
specifies that the requested file is not the next file.

-number N, -nb N
specifies the file sequence number or numerical position within the file set. Permissible values range from 1 to 9999. (Default is no -number)

-record N, -rec N
specifies the logical record length in bytes. Permissible values range from 18 to 1044480 (sys_info$max_seg_size * 4) bytes, but the legality of the record size is dependent on the record format specified with the "-format" control argument and the block size. In general the record size must be <= the block size with the exception of spanned record formats (i.e. S or SB formats) where the record size may be the max allowable. (No default value. The default record size is determined by the value of the appropriate "-default_<type>_record" specification, where <type> can be either fixed, variable or spanned.) (Default is no -record)

-replace STR, -rpl STR
specifies replacement of an existing file, where STR is the file identifier to use in the search for the file to be replaced. (Default is no -replace)

CLOSING

Closing of the ANSI PFM is made by the iox_$close_file or the iox_$close entries (via the mtape_ close_file or close entries). The iox_$close_file entry provides for a character string close description, describing actions to be taken by the Per-Format module upon closing the I/O switch. If closing is made by the iox_$close entry, the close time actions are formed from the current default values of the ANSI PFMs close description arguments. The close description arguments have an initial default value, which are denoted in their respective descriptions below, or the default values may be changed by the user (see "Default Values" in the mtape_ I/O module description.).

CLOSE DESCRIPTION

CONTROL ARGUMENTS

-close_position STR, -cls_pos STR
specifies where to physically position the tape volume within the file that is being closed. The values of STR are case insensitive and may be selected from bof (for beginning of file), eof (for end of file), or leave (to leave the tape volume positioned where it is). (Default value is leave)

-comment STR, -com STR
specifies a user comment to be displayed on the user_output I/O switch, after the file has been successfully closed. The comment text (STR) may be from 1 to 80 characters in length. (Default is no -comment)

-display, -ds
: specifies that the entire close description, after if has been parsed and any necessary defaults added, is to be displayed on the user_output I/O switch.

-no_display, -nds
: specifies that the close description will not be displayed on the user_output I/O switch. (Default)

READ RECORD OPERATION

The ANSI PFM supports the iox_$read_record operation when the I/O switch is open for sequential_input. In general, format dependent logical records are extracted from physical tape blocks and written into the callers buffer area. As each tape block is exhausted, the ANSI PFM requests the mtape_ I/O module to read in the next tape block. This sequence continues until logical End of File is detected by the ANSI PFM, at which time error_table_$end_of_info is returned to the caller, and no further read_record requests will be accepted by mtape_ or the ANSI PFM until the current file is closed and another file is subsequently opened. If the callers buffer length is not long enough to contain the entire logical record, as much data as will fit in the specified buffer is returned and error_table_$long_record is returned to the caller. In this case, the ANSI PFM will position to the next logical record. If in the course of reading logical records, an End of Volume condition is detected by the ANSI PFM, automatic volume switching is initiated, which if successful, will be transparent to the caller.

WRITE RECORD OPERATION

The ANSI PFM supports the iox_$write_record entry when the I/O switch is open for sequential_output. In general, data of the specified record length is extracted from the users buffer, formatted into logical tape records and written into a physical tape block buffer. As each tape block buffer is filled, the ANSI PFM requests mtape_ to queue up the full buffer for writing and return a pointer to the next buffer to fill. This sequence continues until either: (1) The I/O switch is closed or (2) an mtape_ "volume_status" or volume_set_status" control operation is requested to be processed. In both cases, if a partially filled buffer exists, it will be queued up for writing as a short block and all unwritten buffers will be requested to be written out to tape. If the I/O switch is being closed, the ANSI PFM now writes out the End of File trailer sequence. If during the course of writing tape blocks the End of Volume condition is detected, the ANSI PFM immediatly writes out the End of Volume trailer labels and requests a volume switch to mount the tape to contain the next file section. After the new tape volume has been successfully mounted, the ANSI PFM initiates the volume label and new file section header labels and then requests that the unwritten buffers at the time of the end of volume detection be written out to tape. At this time, the write_record operation being processed at the time of the End of Volume detection is resumed.

POSITION OPERATION

The ANSI PFM supports the iox_$position operation when the I/O switch is opened for sequential_input. All positioning types legal for sequential_input are supported. (See the description of iox_$position earlier in this manual.)

READ LENGTH OPERATION

The ANSI PFM supports the iox_$read_length operation when the I/O switch is open for sequential_input. The read_length operation is implemented by actually reading the next logical record to determine its length, while discarding the actual data. After the length has been determined, backspace record position operation is executed to position to the location prior to the read_length operation. When executing read_length operations on spanned formatted records, or if the read_length operation is to determine the length of the first record of the next block, actual tape motion (i.e. read forward, and backspace block) may be necessary and will occur automatically. If a spanned record spans a volume boundary, volume switching is initiated both when doing the actual read operation and the backspace.

CONTROL OPERATION

The ANSI PFM supports all of the general mtape_ control operations described in the mtape_ I/O module description. There are no control operations that are specific to the ANSI PFM.

CALLING SEQUENCE FOR USER LABEL PROCESSING ROUTINE

In order to process user defined file labels when the "-label_entry" open description argument is used, the entry variable argument to the "-label_entry" control argument must conform to the following calling sequence in order to be called properly by mtape_ and its Per-Format modules:

```
dcl user_label_entry entry (ptr, char (*), fixed bin,
    fixed bin, fixed bin, fixed bin (35));

call user_label_entry (iocb_ptr, user_label_data, label_number,
     label_type, file_section_number, code);
```

where:

iocb_ptr
: is a pointer to the I/O control block through which the mtape_ I/O module is attached. A user_label_entry routine may wish to know more information about the file for which it is processing user labels. This can be accomplished by calling the iox_$control entry with this iocb_ptr and executing the mtape_ "file_status" control operation.

user_label_data
: is the actual contents of the user label record to be processed (INPUT) or written (OUTPUT). The length of this field will be 76 characters on input and truncated to same on output.

label_number
: is the number of the user label record within the file label group. The ANSI standard allows from 1 to 9 user label records within a file label group (UHL1 - UHL9, and UTL1 - UTL9).

label_type
: is the encoded file label group type that the user_label_entry is being called to process label records for. Its possible values are as follows:

    1 = Beginning of file (BOF) label group
    2 = End of volume (EOV) label group
    3 = End of file (EOF) label group

file_section_number
: is the section number of the file for which the user_label_entry routine is being called to process user labels for. For multivolume files, this would essentially be the number of the volume (the first volume on which a file resides being number 1) on which this file "section" resides. For single volume files, the file_section_number would always be a 1.

code
: is a standard system error code. When writing user labels, the user_label_entry routine should set code to error_table_$end_of_info in order to tell the caller that no more user labels are to be written. Otherwise, the user_label_entry is called repeatedly to generate user label data until the maximum number of user labels have been written.

SEARCHING FOR A FILE

Before a file may be either created or read, its physical position within the volume set must be located. In the case of file creation, its physical position may be non-existent, but to ensure file set integrity all of the files in the file set must be searched to ensure its non-existence. To reduce physical tape searching to a minimum, the ANSI PFM in concert with mtape_ maintains a linked list of file set members, with adequate information in each element of the linked list to identify the file it represents and its physical position within the volume set. At the time of the first opening, the above mentioned linked list of file set members does not exist. In this case, the volume set is searched sequentially forward until the desired file is found. As each file preceding the desired file is identified, a new element is added to the linked list of file set members, extracting file identity and format information from the file header and trailer labels, and obtaining the physical position of the file header from mtape_. On subsequent file openings, this linked list of file set members is searched first, and if the desired file is identified as being one of the elements, the volume set is positioned to the indicated position of the file header. If the desired file is not found in the linked list of file set members, then the volume set is searched forward from the position of the last identified file in the linked list, adding to the list as it proceeds in an attempt to find the desired file.

There are 6 open description control arguments which deal with identifying a file to be processed. These are: -append, -last_file, -name, -next_file, -number and -replace. From reading their descriptions above, it can be seen that if some of them were used together, they would form an inconsistent identity for a file to be found. (e.g. If -last_file and -next_file were used together, they may or may not describe the same file.) In order to keep the set of file identity arguments consistent for any given file, certain rules are applied when the open description is parsed as follows:

1. Open description arguments are parsed from left to right.
2. Any default arguments and their associated values are parsed before the users open description is parsed.
3. Control arguments and their associated values on the right take precedence over the same control argument and its value that preceded it. (e.g. In an open description which included "-name FILEX -name FILEY", the parsed result would be "-name FILEY".)
4. Binary control arguments (e.g. -last_file) all have an associated antonym value (i.e. -no_last_file). As each binary control argument is parsed, it takes precedence and replaces any opposite control argument that preceded it. (e.g. In an open description which included "-last_file -no_last_file" the parsed result would be -no_last_file.)
5. For each of the 6 file identity open description arguments, there are a certain set of control arguments with which it is mutually exclusive with and takes precedence over. The chart below illustrates this mutual exclusivity:

	-append	-last_file	-name	-next_file	-number	-replace
-append	*	*		*		
-last_file	*	*		*	*	*
-name		*	*	*		*
-next_file	*	*	*	*	*	
-number		*		*	*	
-replace		*	*	*		*

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 file to be created may be identified explicitly by specifying the file name and/or number (with the -name and -number open description control arguments) either together or individually. If a -name and -number control arg appear in the same open description, they must identify the same file or an error will result.

The file to be created may be identified implicitly by specifying one of the relative position control arguments, -append, -last_file or -next_file in an open description.

Implicit file replacement is also accomplished if the file to be created is identified as already existing.

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 STR 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.

The -format F, -record R and -block B control arguments, or their default values, 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 99996 characters.

db for variable length records, blocked.
Used when records are of varying lengths, the longest not in excess of 99992 characters.

sb for spanned records, blocked.
Used when the record length is fixed and in excess of 99996 characters, or variable and in excess of 99992 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 99996 characters.

NOTE: THE USE OF UNDEFINED RECORDS IS A NONSTANDARD FEATURE.

Records recorded using U format may be irreversibly 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:  amrl + 4 <= R <= 99996
F = sb | s:  amrl <= R <= 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 >= R
F = d:       B = R
F = sb | s:  18 <= B <= 99996
F = u:       amrl <= B <= 99996
```

In every case, B must be an integer in the range 18 <= B <= 99996.

NOTE: THE USE OF BLOCK LENGTHS IN EXCESS OF 2048 CHARACTERS VIOLATES THE ANSI INTERCHANGE STANDARD AND THEREFORE SHOULD NOT BE USED IN AN INTERCHANGE ENVIRONMENT.

READING A FILE

The open 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 open 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 open description need only identify the file to be read; no other control arguments are necessary. Any of the 6 file identification open description control arguments (See "Searching For a File" above.) may be used to identify the file to be read.

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 open 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 by using any of the 6 open description file identifying control arguments. (See "Searching For 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.

Any structure attribute open description control arguments specified by the user are ignored when extending a file.

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 open 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.

Any structure attribute open description control arguments specified by the user are ignored when modifying a file. The file to be modified is identified as above.

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, that all 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, that 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 open 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.

Any structure attribute open description control arguments specified by the user are ignored when generating a file. The file to be generated is identified as above.

ENCODING MODE

The ANSI PFM makes provision for three data encoding modes: ASCII, EBCDIC, and binary. Because the Standard 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 STR can be used to explicitly specify the encoding mode, where STR is the string ascii, ebcdic, or binary. The default is the string ascii.

If STR 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 STR 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 for the specific ASCII to EBCDIC mapping used by the ANSI PFM.) If STR is the string binary, any octal value can be recorded.

Like its predecessor, the tape_ansi_ I/O module, the ANSI PFM records the data encoding mode in a portion of the file labels reserved for system-defined use. When the file is subsequently read, the encoding mode is extracted from the file labels, so the -mode STR 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 <= ddd <= 366. A special case of the Julian date form is the value "00000" (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; the 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. The -force control argument unconditionally grants permission to overwrite a file without querying the user, regardless of "unexpired" status.

PROCESSING INTERCHANGE FILES

The Standard makes provision for recording record format, block length, and record length in specific fields most notably (volume name and file name) of the HDR2 file label. In addition, the ANSI PFM records the encoding mode in a portion of the HDR2 label reserved for system-defined use. Because the Standard 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 Per-Format module, or its predecessor, the tape_ansi_ I/O module) uses an encoding mode other than ASCII, the -mode STR 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 Standard 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 or the application of the appropriate defaults is required.

ASCII SUBSET

The Standard 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.

RECORD FORMATS

ANSI files are structured in one of three record formats: F, D, or S. In addition, the ANSI PFM 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 (that 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 (iobl); 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 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 = B = 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 slightly 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, that includes the length of the SCW itself. The SCW also contains a one-character record segment code, that 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. In the first example, the file is unblocked.

```
            --------    -------------    -----------------------------
data       |  200   |  |     400     |  |            1000             |
            --------    -------------    -----------------------------

            --------    -------------    ----------------    ---------
           |2|      |  |4|           |  |8|              |  |2|       |
block      |0| 200  |  |0|    400    |  |0|     795      |  |1|  205  |
           |5|      |  |5|           |  |0|              |  |0|       |
            --------    -------------    ----------------    ---------
```

In the next example, the file is blocked.

```
            --------    -------------    -----------------------------
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 Standard suggests that an entire U format block be treated as a single record. In conformance with this suggestion, the ANSI PFM 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   | |     127      | | 20 | |      156       |
           --------   --------------   ----   ----------------
```

BLOCK PADDING

The Standard 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. Unlike its predecessor, the tape_ansi_ I/O module who required that all blocks be padded out to modulo 4 characters, the ANSI PFM only requires padding to modulo 4 characters, if the file is being recorded in binary mode. In which case the ANSI PFM automatically pads every block to the correct length, using from 1 to 3 circumflex characters. In addition, the Standard does not permit recording a block of fewer than 18 characters. To conform with this requirement, the ANSI PFM 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 Standard requires that all volumes be initialized with a VOL1 label and dummy file before they are used for output. The ANSI PFM 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 Standard) to the application program. The ANSI PFM's mechanism strikes a balance between outright denial and absolute ease.

BUFFER OFFSET

The Standard 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 ANSI PFM will record a block prefix which contains block number and block length information, only if the -buffer_offset open description argument is specified. When reading a file, the block prefix area, if it exists, is ignored by the ANSI PFM, unless the file has been identified as being recorded by the ANSI PFM. If this is the case, the block prefix area (described in the -buffer_offset open description argument above), is used by the ANSI PFM for positive block position and length comparisons.

CONFORMANCE TO STANDARD

The ANSI PFM conforms to the ANSI standard for level 4 implementations with the following three exceptions:

1. Volume Initialization -- The ANSI PFM has a permission-granting mechanism that can be controlled by the application program.

2. Volume and File Accessibility -- On input, the ANSI PFM 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 ANSI PFM has a permission-granting mechanism that can be controlled by the application program.

LABEL PROCESSING

VOL1
: The label is processed on input and output. The owner-identifier field, character positions (CP) 38 to 51, holds a three-character volume authentication code, in character positions 38 through 40 and the character string "MULT001" in character positions 41 through 46.

UVL1
: This label is processed on output and ignored on input. The contents of the UVL1 label is meant to be used by Site/Tape administrators for historical information about tape usage at their particular site. The contents of the UVL1 label are as follows:

CP 01 to 04 - Label identifier ("UVL1").
CP 05 to 07 - Volume authentication code. (The same as VOL1 label CP 38 to 40).
CP 08 to 13 - Julian Creation date (" YYDDD").
CP 14 to 17 - Unused ("   ").
CP 18 to 49 - Site installation code.
CP 50 to 80 - Person.Project.Instance of creator of tape.

UVL2 - UVL9
: 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 ANSI2".

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 47 - full 32-character volume name of next volume (EOV2 only)
CP 48 - blocking attribute (all)
"0" = unblocked; "1" = blocked
CP 49 - 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 processed on output and input only if the "-label_entry" open description argument is given. Otherwise, not written on output and ignored on input.

Name: audit_

The audit_ I/O module is used to monitor input and/or output directed over a given stream I/O switch. Entries of various kinds are appended to the audit file in response to input and output on the specified switch. These are described in detail below. See the Commands manual for descriptions of the related commands attach_audit, detach_audit, and display_audit_file, and for a description of the audit editor requests.

Entry points in this module are not called directly by users; rather, they are accessed through the I/O system.

ATTACH DESCRIPTION

```
audit_ switch_name {-control_args}
```

ARGUMENTS

switch_name
is the name of an existing I/O switch that is to be monitored.

CONTROL ARGUMENTS

-truncate, -tc
truncates the audit file, if it already exists. The default is to extend the audit file.

-pathname path, -pn path
specifies the pathname of the new audit file. The default pathname is [home_dir]>[date].audit, where date is the date (in the form MM/DD/YY) returned by the date_time_ subroutine at the time of attachment.

NOTES

The attachment of audit_ does an implicit open of the switch. Attachment is simplified by use of the attach_audit command.

MODES OPERATION

Modes for the audit_ I/O module are listed below. Some modes have a complement, indicated by the circumflex character (^), that turns the mode off.

audit_trace, ^audit_trace
traces all control and mode calls to the module. An entry with a TC or TM identifier (for a control call or mode call, respectively) is placed in the audit file. This entry describes the contents of the given call. The default is off.

audit_input, ^audit_input
turns on auditing for input lines. The default is on.

audit_output, ^audit_output
turns on auditing for output lines. The default is on.

audit_edit, ^audit_edit
enables audit editing. The default is on. If audit_edit is off, the auditing requests are not recognized. (See the attach_audit command for a discussion of auditing requests.)

audit_transparent, ^audit_transparent
turns off auditing of auditing requests and editing requests, as well as their results. EL entries are still audited (see "Audit File" below). The default is off.

audit_suspend, ^audit_suspend
disables all audit capabilities. The default is off.

audit_meter, ^audit_meter
writes a metering record before each entry in the file containing the actual time of the metering, the incremental CPU time since the last metering, and the incremental page faults since the last metering. The default is off.

audit_trigger=X
sets the auditing request trigger to the character specified by X. The default is an exclamation point (!).

audit_file_size=N
sets the maximum number of records for the audit file to N. When the maximum is reached, the file is scrolled; i.e., it is treated as a circular buffer of N records. If N is the character string "unlimited", the file will grow without limit. The default is unlimited.

audit_use_editor_prompt, ^audit_use_editor_prompt
turns prompting on or off in the audit editor.

audit_editor_prompt=STR, audit_epstr=STR
sets the audit editor prompt string to STR. The audit editor prompt has the default appearance "audit editor: ", or, if the number of recursive invocations of the editor is greater than 1, "audit editor(N): ", where N is the depth of the current invocation. This string is used as an ioa_ control string, with the arguments being: a bit which is on if the level is greater than 1; and, the level. The default string is "^/audit editor^[(^d)^]:^2x".

Note that modes not preceded by "audit_" are passed on to the I/O module being audited.

CONTROL OPERATION

This I/O module supports the following control orders:

audit_truncate
truncates the audit file.

audit_modes
returns the current audit modes in a char (256) varying string pointed to by info_ptr.

OTHER OPERATIONS

The only other operation that is supported is the position operation, which is passed on to the audited I/O module.

THE AUDIT FILE

The audit file, by default, has the pathname:

```
>udd>Project_id>Person_id>[date].audit
```

where date is the first eight characters (the date portion) returned by the date_time_ subroutine, and is of the form: mm/dd/yy. This pathname can also be specified using active functions:

```
[home_dir]>[date].audit
```

The default audit file size is unlimited, and the audit file can become a multisegment file.

The entry type identifiers are:

EL edit line, returned from audit editor.

IC result of a get_chars.

IL result of a get_line.

M metering data.

OC result of a put_chars.

TC control request trace.

TM mode request trace.

AUDIT REQUESTS

The audit requests are always recognized when auditing is on. The three character request sequence is the trigger character followed by the desired request followed by a new line. However, when an unrecognized request is given, the entire line is treated as a regular input line with no special processing. The default trigger character is an exclamation mark ("!"). The requests are:

!. print "audit" and which of input and output is being audited.

!? print a brief description of available audit requests.

!e enter the audit editor.

!E enter the audit editor, with the input line processed as edit requests.

!a abbrev expand the input line.

!r replay the input line. That is, display the input line without a new line. Further input up to the next new line is appended to the redisplayed input. This is the input line which is passed through the audit_ I/O module.

!t instructs the audit_ I/O module not to log the input line, i.e., to make it transparent.

!d delete the line. It prevents the input line from ever being seen.

!n no operation. The input line to which this is appended is simply passed through the audit_ I/O module.

Name: bisync_

The bisync_ I/O module performs stream I/O over a binary synchronous communications channel.

Entry points in this module are not called directly by users; rather, the module is accessed through the I/O system.

ATTACH DESCRIPTION

```
bisync_ device {-control_args}
```

ARGUMENTS

device
is the name of the communications channel to be used for communications.

CONTROL ARGUMENTS

-ascii
uses the ASCII bisync protocol. This is the default.

-bid_limit N
sets to N the number of times a line bid is retried. The default is 30 times.

-breot
causes the get_chars operation to return any block of data ending with an end-of-transmission (EOT) character (see "Get Chars Operation" below).

-bretb
causes the get_chars operation to return any block of data ending with an end-of-text block (ETB) character. The default is to return only blocks ending with an end-of-text control character (ETX) or an intermediate text block (ITB) control character (see the discussion of the get_chars operation below).

-ebcdic
uses the EBCDIC bisync protocol.

-hangup
causes an automatic hangup when the switch is detached.

-multi_record {N}
specifies that blocking of logical records is done by the I/O module. If specified, N is the maximum number of records per block. If N is not given, the number of records per block is as many as fit.

-nontransparent
uses the nontransparent bisync protocol.

-output_etb
causes output records to the FNP channel to be terminated with the ETB character instead of with the default ETX characters. The caller of the device module (ibm3780_, ibm2780_, etc.) is expected to signal the termination of the transmission of a file (SSF or MSF) by passing down a "runout" control order. This will cause the device module and bisync_ to flush out any data being held in internal buffers. The bisync_ module will then transmit a null message with an ETX character. Subsequent calls to the bisync_put_chars entry will resume the transmission of records with the ETB character until the next runout control order.

output_etx
causes all output records to be terminated with the ETX characters. (Default)

-size N
sets to N the number of characters to be transmitted in each bisync block. The default is 256 characters.

-transparent
uses the transparent bisync protocol. This is the default.

-ttd_limit N
sets to N the maximum number of TTDs that are sent before sending an EOT. The default is 30 TTDs.

-ttd_time N
sets to N the number of seconds of temporary text delay (TTD) transmissions if output is delayed. The default is two seconds.

OPEN OPERATION

The bisync_ I/O module supports the stream_input, stream_output, and stream_input_output opening modes.

PUT CHARS OPERATION

The put_chars entry splits the data to be written into blocks according to the -size control argument in the attach description. The appropriate bisync control characters are added to the beginning and end of each block. Each block except the last is transmitted with an ETB control character at the end. The last block is transmitted with an ETX control character at the end.

GET CHARS OPERATION

The get_chars entry reads and decodes bisync blocks, removes the control characters, and returns the message text to the caller's buffer.

This page intentionally left blank.

Characters are returned up to the next logical bisync break character. Normally this is ETX. If -bretb is specified in the attach description, ETB is also considered to be a break character. If -multi_record is given, the interrecord ITB characters are also considered to be break characters. In addition, if -breot is specified, error_table_$end_of_info is returned when an EOT is read.

GET LINE OPERATION

The get_line entry reads and decodes bisync blocks, removes the control characters, and returns the message text to the caller's buffer. Characters are returned until either a newline character is placed in the buffer or the buffer is filled. The get_line entry does not distinguish between blocks ending in ETB or ITB and blocks ending in ETX.

CONTROL OPERATION

Several of the control operations supported by the bisync_ I/O module are identical to those supported by the tty_ I/O module and are documented there. They include:

abort
event_info
hangup
read_status
resetread
resetwrite
write_status

The following additional control operations are supported by this I/O module.

end_write_mode
: causes the I/0 module to block until all outstanding output has been written.

get_bid_limit
: where info_ptr points to a fixed binary bid limit that is set either to the value specified at attach or in the last set_bid_limit order.

get_bsc_modes
: returns the structure described under set_bsc_modes.

get_chars
performs a get_chars operation and returns additional information about the input. The info_ptr points to a structure of the following form:

```
dcl 1 get_chars_info,
      2  buf_ptr ptr,
      2  buf_len fixed bin(21),
      2  data_len fixed bin(21),
      2  hbuf_ptr ptr,
      2  hbuf_len fixed bin(21),
      2  header_len fixed bin(21),
      2  flags,
         3  etx bit(1) unal,
         3  etb bit(1) unal,
         3  soh bit(1) unal,
         3  eot bit(1) unal,
         3  pad bit(32) unal;
```

where:

buf_ptr, buf_len
define an input buffer for the text of the message. (Input)

data_len
is set to the number of characters of text read. (Output)

hbuf_ptr, hbuf_len
define an input buffer for the header of the message. (Input)

header_len
is set to the header's length in characters. (Output)

etx
indicates that text is terminated with an ETX character. (Output)

etb
indicates that text is terminated with an ETB character. (Output)

soh
indicates that the data includes a header. (Output)

eot
indicates that an EOT is received. (Output)

pad
is unused space in this structure. (Output)

get_multi_record_mode
where info_ptr points to a fixed binary record count. This order returns the multirecord record count. A 1 indicates single-record mode.

get_size
where info_ptr points to a fixed binary buffer size and returns the current size.

hangup_proc
sets up a procedure to be called if the communications channel hangs up. The hangup_proc input structure has the following form:

```
dcl 1 hangup_proc  aligned,
      2 entry      entry variable,
      2 datap      ptr,
      2 prior      fixed bin;
```

where:

entry
is the entry to call when a hangup is detected.

datap
is a pointer to data for the hangup procedure.

prior
is the ipc_ event call priority to be associated with hangup notification.

runout
has meaning only in multirecord mode and writes the current partially filled block.

send_nontransparent_msg
writes the data specified in nontransparent bisync mode, regardless of the current transparency mode. This order is used to send short nontransparent control sequences while in transparent mode. The info_ptr points to a structure of the following form:

```
dcl 1  order_msg,
         2  data_len fixed bin,
         2  data char (order_msg.data_len);
```

set_bid_limit
where info_ptr points to a fixed binary bid limit to replace the bid_limit specified in the attach description.

set_bsc_modes
where info_ptr points to a structure of the following form:

```
dcl 1  bsc_modes,
         2  transparent bit(1) unal,
         2  ebcdic bit(1) unal,
         2  mbz bit (34) unal;
```

The setting of the transparent and ebcdic bits then replaces the values specified in the attach description.

set_multi_record_mode
where info_ptr points to a fixed binary record count. If the count is 1, the I/O module enters single-record mode; otherwise, multirecord mode is entered, and the count specifies the maximum number of records per block. Zero (or a null info_ptr) specifies no fixed limit; i.e., as many records as fit are blocked.

set_size
where info_ptr points to a fixed binary buffer size. This new size replaces the size specified in the attach description. It cannot be larger than the size originally specified in the attach description.

Name: cross_ring_

The cross_ring_ I/O module allows an outer ring to attach a switch to a preexisting switch in an inner ring, and to perform I/O operations by forwarding I/O from the attachment in the outer ring through a gate to an inner ring. The cross_ring_ I/O module is not called directly by users; rather the module is accessed through the I/O system.

ATTACH DESCRIPTION

```
cross_ring_ switch_name N
```

ARGUMENTS

switch_name
is a previously registered switch name in ring N.

N
is a ring number from 0 to 7.

OPENING

The inner ring switch may be open or not. If not open, it will be opened on an open call. All modes are supported.

CLOSE OPERATION

The inner switch is closed only if it was opened by cross_ring_.

OTHER OPERATIONS

All operations are passed on to the inner ring I/O switch.

NOTES

This I/O module allows a program in an outer ring, if permitted by the inner ring, to use I/O services that are available only from an inner ring via cross_ring_io_$allow_cross. By the use of the cross_ring_io_$allow_cross subroutine a subsystem writer is able to introduce into an outer ring environment many features from an inner ring, thereby tailoring it to fit the user's specific needs.

The switch in the inner ring must be attached by the inner ring before cross_ring_ can be attached in the outer ring.

Name: discard_

The discard I/O module provides a sink for output and a no-op for input. All output operations are supported and return a 0 error code, but discard any data. All input operations are supported and return either error_table_$end_of_info or error_table_$no_record as described below. The control and modes operations are also supported as no-ops.

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.

LIST OF OPENING MODES

This module supports the following opening modes:

stream_input
stream_output
stream_input_output
sequential_input
sequential_output
sequential_input_output

sequential_update
keyed_sequential_input
keyed_sequential_output
keyed_sequential_update
direct_input
direct_output
direct_update

CONTROL OPERATION

This module supports the control operation in all opening modes. All orders are accepted; but they have no effect. A 0 error code is always returned, and the structure pointed to by the info pointer argument is not changed.

MODES OPERATION

This module supports modes operation in all opening modes. It always returns a null string for the old modes and a 0 error code.

GET CHARS, GET LINE, AND READ RECORD OPERATIONS

These operations always set the returned length to 0 and the error code to error_table_$end_of_info.

PUT CHARS AND WRITE RECORD OPERATIONS

These operations simply set the error code to 0 and returns.

POSITION OPERATION

This operation simply sets the error code to 0 and returns.

DELETE OPERATION

This operation sets the error code to error_table$no_record and returns.

READ AND SEEK KEY OPERATIONS

These operations set the returned length to 0 and the error code to error_table_$no_record.

READ LENGTH OPERATION

This operation sets the returned length to 0 and the error code to error_table_$no_record.

NOTES

This I/O module implements all of the indicated operations in each opening mode. (See "Opening Modes and Allowed Input/Output Operations" in Section 5 of the *Multics Programmer's Reference Manual*, Order No. AG91).

Name: g115_

The g115_ I/O module performs stream I/O to a remote I/O terminal that has the characteristics of the Honeywell Level 6 remote batch facility (G115 type). The hardware options currently supported are defined by the control arguments described below.

Entry points in this module are not called directly by users; rather, the module is accessed through the I/O system.

ATTACH DESCRIPTION

g115_ -control_args

CONTROL ARGUMENTS

The following control arguments are optional, with the exception of -comm, -device, and -tty:

-ascii
: uses the ASCII character set. This is the default. This argument is accepted for compatibility with other terminal I/O modules.

-auto_call N
: specifies the phone number, N, to be called via the auto call unit on the specified communications channel.

-comm STR
: uses the communications I/O module specified by STR. Currently, the only permissible value for STR is "rci". This argument is required for compatibility with all other I/O modules used by the I/O daemon.

This page intentionally left blank.

-device STR
attaches the subdevice specified by STR. STR can be printer, punch, reader, or teleprinter.

-physical_line_length N, -pll N
specifies the physical line length, N, of the output device. This argument is accepted for compatibility with other terminal I/O modules.

-terminal_type STR, -ttp STR
STR specifies the terminal type whose conversion, translation, and special tables defined in the user or system terminal type table (TTT) are used to convert and translate input and output to and from the device. If not specified, no conversion or translation is performed. See "Notes" below.

-tty STR
connects the remote I/O terminal to the communications channel named STR.

OPEN OPERATION

The g115_ I/O module supports stream_input, stream_output, and stream_input_output opening modes.

PUT CHARS OPERATION

The put_chars entry blocks the data to be written into blocks of up to 324 characters and transmits them to the specified communications channel.

GET CHARS OPERATION

The get_chars entry reads blocks of up to 324 characters and returns the number of characters requested up to the next record separator.

CONTROL OPERATION

This I/O module supports all the control operations supported by the tty_ I/O module, plus the following:

end_write_mode
prevents the g115_ module from returning until all outstanding output has been written to the attached channel.

hangup_proc
sets up a procedure to be called if the communications channel hangs up. The hangup_proc structure has this form:

```
dcl 1 hangup_proc aligned,
      2 entry entry variable,
      2 datap ptr,
      2 prior fixed bin;
```

where:

entry
: is the entry to call when a hangup is detected.

datap
: is a pointer to data for the hangup procedure.

prior
: is the ipc_ event call priority to be associated with hangup notification.

reset
: sets the edited mode of output conversion.

runout
: transmits any data stored in the output buffer. There is no input structure.

select_device
: selects the subdevice (printer, punch, or teleprinter) to which output is next directed. The input structure is of the form:

```
dcl device char (32);
```

MODES OPERATION

This I/O module supports the rawi and rawo modes. It also supports the nonedited and default modes, which set and reset the edited output conversion, if it has been enabled by the -terminal_type control argument.

NOTES

The only allowable values in the output conversion table are 00 and any values greater than 16. All values defined in the description of the tty_ I/O module are allowed for input conversion. Input and output translation tables can be up to 256 characters in length.

Name: hasp_host_

The hasp_host_ I/O module simulates record-oriented I/O to a single device of a workstation while communicating with a host system using the HASP communications protocol. See "Notes" below.

Entry points in this module are not called directly by users; rather, the module is accessed through the I/O system.

This I/O module must be attached to a subchannel of a communications channel configured to use the HASP ring-0 multiplexer. See the description of the HASP multiplexer in MAM Communications.

This I/O module is designed primarily for use by the Multics I/O daemon.

ATTACH DESCRIPTION

```
hasp_host_ -control_args
```

CONTROL ARGUMENTS

The following control arguments are optional, with the exception of -comm, -device, and -tty:

-comm hasp
: is required for compatibility with other I/O modules used by the I/O daemon.

-ebcdic
: is accepted for compatibility with other I/O modules used by the I/O daemon, but is ignored by this I/O module.

-device STR
: specifies the type of device for this attachment. STR must be one of teleprinter, reader, printer, or punch. The type specified by this control argument must match the type of device attached to the channel name defined below.

-physical_line_length N, -pll N
: is accepted for compatibility with other I/O modules used by the I/O daemon, but is ignored by this I/O module.

-terminal_type STR, -ttp STR
: is optional and is used to define the character set used by the remote system. STR must be the name of a terminal type defined in the site's terminal type table (TTT). See "Character Set Specification" below.

-tty channel_name
: specifies the communications channel to be attached. The channel must be a subchannel of a HASP multiplexed channel (e.g., a.h014.prt3).

OPEN OPERATION

The hasp_host_ I/O module supports the sequential_input, sequential_output, and sequential_input_output opening modes.

WRITE RECORD OPERATION

The write_record operation converts the supplied data record from ASCII to the remote system's character set, performs data compression, and transmits the record to the HASP multiplexer.

The format of the record supplied to this I/O module follows. This structure and the referenced constants are contained in the terminal_io_record.incl.pl1 include file:

```
dcl 1 terminal_io_record          aligned based,
      2 version                   fixed binary,
      2 device_type               fixed binary,
      2 slew_control,
        3 slew_type               fixed binary (18) unaligned unsigned,
        3 slew_count              fixed binary (18) unaligned unsigned,
      2 flags,
        3 binary                  bit (1) unaligned,
        3 preslew                 bit (1) unaligned,
        3 pad                     bit (34) unaligned,
      2 element_size              fixed binary,
      2 n_elements                fixed binary (24),
      2 data,
        3 bits (terminal_io_record_n_elements refer
           (terminal_io_record.n_elements))
            bit (terminal_io_record_element_size refer
             (terminal_io_record.element_size)) unaligned;
```

STRUCTURE ELEMENTS

version
: is the current version of this structure given by the value of the named constant terminal_io_record_version_1. (Input)

device_type
: is the type of device to which this record is to be written. (Input) The acceptable values are TELEPRINTER_DEVICE and READER_DEVICE.

slew_control
: is ignored by this I/O module, as the HASP communications protocol does not define slew operations for either the teleprinter or card reader. (Input)

flags.binary
: must be set to "0"b. (Input) (This I/O module does not support binary data transmission.)

flags.preslew
: must be set to "0"b. (Input)

element_size
: must be set to 9. (Input) (This I/O module only supports transmission of characters.)

n_elements
: is the number of characters in the record to be written. (Input)

data.bits
: is the actual data. (Input) This I/O module expects to be supplied ASCII characters.

READ RECORD OPERATION

The read_record operation returns a single record from the device, basically performing the inverse of the functions described for the write_record operation. Additionally, for line printer attachments, the carriage control information in the record is converted into the appropriate slew information in the terminal_io_record structure.

The format of the record that this I/O module returns in the supplied buffer is as follows. The structure and the referenced constants are contained in the terminal_io_record.incl.pl1include file:

```
dcl 1 terminal_io_record      aligned based,
      2 version               fixed binary,
      2 device_type           fixed binary,
      2 slew_control,
        3 slew_type           fixed binary (18) unaligned unsigned,
        3 slew_count          fixed binary (18) unaligned unsigned,
      2 flags,
        3 binary              bit (1) unaligned,
        3 preslew             bit (1) unaligned,
        3 pad                 bit (34) unaligned,
      2 element_size          fixed binary,
      2 n_elements            fixed binary (24),
      2 data,
        3 bits (terminal_io_record_n_elements refer
          (terminal_io_record.n_elements))
           bit (terminal_io_record_element_size refer
            (terminal_io_record.element_size)) unaligned;
```

STRUCTURE ELEMENTS

version
: is the current version of this structure given by the value of the named constant terminal_io_record_version_1. (Output)

device_type
: is the type of device from which this record is to be read. (Output) Its possible values are TELEPRINTER_DEVICE, PRINTER_DEVICE, or PUNCH_DEVICE.

slew_control
: if the input device is a line printer, it is filled in with the interpretation of the HASP carriage control record present in each line printer record; otherwise, it is always set to the value specified below. (Output)

slew_type
: for a line printer, is set to the type of slew operation to be performed before or after "printing" the data in the record and can be either SLEW_BY_COUNT or SLEW_TO_CHANNEL. (Output) For a teleprinter or punch, it is set to SLEW_BY_COUNT. (The data returned is processed by the caller of this I/O module; this processing is herein termed the "printing" of the data.)

slew_count
: for a line printer, is set to the value to be interpreted according to slew_control.slew_type above. (Output) For a teleprinter or punch it is set to 1. (Output)

flags.binary
: is always set to "0"b. (Output)

flags.preslew
: for a line printer, is set to "1"b if the slew operation above is to be performed before "printing" the data in the record, or is set to "0"b if the slew operation is to be performed after "printing". (Output) For other than the line printer, it is always set to "0"b.

element_size
: is always set to 9. (Output)

n_elements
: is set to the number of characters returned in the record. (Output)

data.bits
: is the actual returned data. (Output) This I/O module converts the data input from the remote host to ASCII.

CONTROL OPERATION

This I/O module supports the following control operations:

end_write_mode
: ensures that all previously written data has been transmitted to the HASP multiplexer and then writes an end-of-file record for the device.

hangup_proc
: is used to specify a procedure to be invoked when this attachment's channel is hung up. The info_ptr points to the following structure:

```
dcl 1 hangup_proc_info aligned,
      2 procedure entry variable,
      2 data_ptr pointer,
      2 priority fixed binary;
```

where:

procedure
    is the procedure to be invoked when the hangup occurs. (Input)

data_ptr
    is a pointer to be supplied to the procedure. (Input)

priority
    is the priority for the hangup event. (Input)

    See the ipc_ subroutine for a detailed explanation of data_ptr and priority.

read_status
    determines whether or not there are any records waiting for a process to read. The info_ptr should point to the following structure, which is filled in by the call:

```
dcl 1 info_structure aligned,
      2 ev_chan fixed bin (71),
      2 input_available bit (1);
```

where:

ev_chan
    is the event channel used to signal the arrival of input. (Output)

input_available
    indicates whether input is available (Output):
    "0"b   no input
    "1"b   input

resetread
    discards any pending input.

resetwrite
    discards any as-yet unprocessed output.

runout
    ensures that all data has been transmitted to the HASP multiplexer from where it is guaranteed to be transmitted to the terminal.

select_device and reset
    are ignored rather than rejected for compatibility with other I/O modules used by the I/O daemon.

signon_record
no_signon_record
    can only be issued on the operator's console subchannel of the multiplexer. These are described in the "SIGNON Processing" section.

MODES OPERATION

This module accepts the non_edited and default modes for compatibility with other I/O modules used by the I/O daemon, but ignores them.

CHARACTER SET SPECIFICATION

This I/O module allows the specification of the character set used by the remote system through the -terminal_type attach option.

If -terminal_type is given, the referenced terminal type must be defined in the site's (TTT) with an input and an output translation table. This module uses these translation tables to convert data from the remote system's character set to ASCII and vice versa.

If -terminal_type is not given, the remote system is assumed to use EBCDIC. In this case, the ascii_to_ebcdic_ subroutine is used to convert data sent to the system; the ebcdic_to_ascii_ subroutine is used to convert data received from the remote system.

SIGNON PROCESSING

Before communicating with certain remote systems, Multics must send the SIGNON record. This specially formatted record identifies Multics to the remote system.

For these systems, the Multics multiplexer must be configured to use signon_mode (see MAM Communications). Before data transmission is permitted, the signon_record control order must be issued on an I/O switch attached to the operator's console subchannel of the multiplexer.

If the remote system does not expect a SIGNON record, the no_signon_record control order can be used to validate that the multiplexer channel is properly configured.

SIGNON_RECORD CONTROL ORDER

This control order supplies a SIGNON record for transmission to the remote system. The info_ptr must locate the following structure, which is declared in the include file hasp_signon_record_info.incl.pl1:

```
dcl 1 signon_record_info        aligned based,
      2 version                 fixed binary,
      2 pad                     bit (36),
      2 event_channel           fixed binary (71),
      2 record                  character (80) unaligned;
```

STRUCTURE ELEMENTS

version
: is the current version of this structure. It must have the value of the named constant SIGNON_RECORD_INFO_VERSION_1.

pad
: is reserved for future expansion and must be zero.

event_channel
: is an event-wait channel whose use is described below.

record
: is the actual text of the SIGNON record in ASCII. This I/O module translates the text to uppercase and the remote system's character set.

If the status code returned by this control order is zero, the calling program must block on the above event-wait channel. When the wakeup arrives, the event message indicates the success or failure of the control order. It has one of the following values (found in the named include file):

HASP_SIGNON_OK
: indicates that the remote system has accepted the SIGNON record.

HASP_SIGNON_REJECTED
: indicates that the remote system has rejected the record; the caller should try again with a different record.

HASP_SIGNON_HANGUP
: indicates that the remote system has rejected the record and disconnected the multiplexer.

If the status code returned by the control order is error_table_$invalid_state, the multiplexer is not configured to send a SIGNON record.

NO_SIGNON_RECORD CONTROL ORDER

This control order validates that the multiplexer is not configured to send a SIGNON record to the remote system. This order does not accept an info structure.

If the returned status code is error_table_$invalid_state, the multiplexer is configured to send a SIGNON record, and a signon_record must be issued on this subchannel.

NOTES

As stated above, this I/O module is used to simulate the operation of a single device of a HASP workstation.

If the simulated device is a card reader, the caller supplies records to this module that are then formatted and transmitted to the remote host; in other words, a card reader attachment through this switch is an output-only attachment.

Similarly, this I/O module receives records from the remote host when the simulated device is either a line printer or a card punch. Thus, line printers and card punches attached through this I/O module are input-only devices.

Special I/O daemon software is provided to allow Multics to simulate the operations of a workstation in order to submit jobs to remote systems and receive those jobs' output print and punch files. This workstation simulator uses this I/O module for communications with the remote host.

Name: hasp_workstation_

The hasp_workstation_ I/O module performs record-oriented I/O to a single device of a remote terminal that supports the HASP communications protocol.

Entry points in this module are not called directly by users; rather, the module is accessed through the I/O system.

This module must be attached to a subchannel of a communications channel configured to use the HASP ring 0 multiplexer. (See the description of the HASP multiplexer in MAM Communications.)

The module is designed primarily for use by the Multics I/O daemon. It expects output for the operator's console and line printers to have been properly formatted by the prt_conv_ module.

ATTACH DESCRIPTION

```
hasp_workstation_ -control_args
```

CONTROL ARGUMENTS

The following control arguments are optional, with the exception of -comm, -device, and -tty:

-comm hasp
is required for compatibility with other I/O modules used by the I/O daemon.

-device STR
specifies the type of device for this attachment. STR must be one of "teleprinter", "reader", "printer", or "punch". The type specified by this control argument must match the type of device attached to the channel name defined below

-ebcdic
is accepted for compatibility with other I/O modules used by the I/O daemon, but is ignored by this I/O module.

-forms STR
specifies the type of forms to be used to print output directed through this attachment. STR is an arbitrary string of at most 32 characters whose interpretation is site dependent. This control argument is only permitted for a line printer. (Default is the null string.)

-inside_page STR
specifies the sequence of carriage control operations to be used to move to the top of the next "inside" page. An "inside" page is the page on which the I/O daemon prints head sheets. This control argument is only permitted for a line printer. The format of STR is described in "Carriage Control Specifications" below. (Default is "c1".)

-outside_page STR
specifies the sequence of carriage control operations to be used to move to the top of the next "outside" page. An "outside" page is the page on which the I/O daemon prints tail sheets. This control argument is only permitted for a line printer. The format of STR is described in "Carriage Control Specifications" below. (Default is "c1".)

-physical_line_length N, -pll N
is accepted for compatibility with other I/O modules used by the I/O daemon, but is ignored by this I/O module.

-terminal_type STR, -ttp STR
is used to define the character set used by the remote terminal. STR must be the name of a terminal type defined in the site's Terminal Type Table (TTT). See "Character Set Specification" below for more information, including the default character set used if this control argument is omitted.

-top_of_page STR
specifies the sequence of carriage control operations to be used to move to the top of the next page. This control argument is only permitted for a line printer. The format of STR is described in "Carriage Control Specifications" below. (Default is "c1".)

-tty channel_name
specifies the communications channel to be attached. The channel must be a subchannel of a HASP multiplexed channel (eg: a.h014.prt3).

OPEN OPERATION

The hasp_workstation_ I/O module supports the sequential_input, sequential_output, and sequential_input_output opening modes.

WRITE RECORD OPERATION

The write_record entry converts the supplied data record from ASCII to the remote terminal's character set, converts the supplied slew control into the proper carriage control sequences for line printer attachments, performs data compression, and transmits the record to the HASP multiplexer.

The format of the record supplied to this I/O module follows. This structure and the referenced constants are contained in the include file terminal_io_record.incl.pl1

```
dcl 1 terminal_io_record       aligned based,
      2 version                fixed binary,
      2 device_type            fixed binary,
      2 slew_control,
        3 slew_type            fixed binary (18) unaligned unsigned,
        3 slew_count           fixed binary (18) unaligned unsigned,
      2 flags,
        3 binary               bit (1) unaligned,
        3 preslew              bit (1) unaligned,
        3 pad                  bit (34) unaligned,
      2 element_size           fixed binary,
      2 n_elements             fixed binary (24),
      2 data,
        3 bits (terminal_io_record_n_elements refer
            (terminal_io_record.n_elements))
             bit (terminal_io_record_element_size refer
              (terminal_io_record.element_size)) unaligned;
```

STRUCTURE ELEMENTS

version
: is the current version of this structure given by the value of the named constant terminal_io_record_version_1. (Input)

device_type
: is the type of device to which this record is to be written. (Input). The acceptable values are TELEPRINTER_DEVICE, PRINTER_DEVICE, or PUNCH_DEVICE.

slew_control
: specifies the slew operation to be performed after printing the data in the record, and need only be supplied by the caller if device_type is PRINTER_DEVICE. (Input)

  slew_type
  : specifies the type of slew operation. (Input). The possible values are SLEW_BY_COUNT, SLEW_TO_TOP_OF_PAGE, SLEW_TO_INSIDE_PAGE, SLEW_TO_OUTSIDE_PAGE, or SLEW_TO_CHANNEL.

  slew_count
  : is interpreted according to the value of slew_control.slew_type. (Input)

flags.binary
: must be set to "0"b. (Input) This I/O module does not support binary data transmission.

flags.preslew
: must be set to "0"b. (Input). This I/O module does not support slew operations before printing the record's data.

element_size
: must be set to 9. (Input). This I/O module only supports transmission of characters.

n_elements
: is the number of characters in the record to be written. (Input)

data.bits
: is the actual data. (Input). This I/O module expects to be supplied ASCII characters.

READ RECORD OPERATION

The read_record entry returns a single record from the device, basically performing the inverse of the functions described for the write_record operation.

The format of the record module returned by this I/O module in the supplied buffer follows. This structure and the referenced constants are contained in the include file terminal_io_record.

```
dcl 1 terminal_io_record      aligned based,
      2 version               fixed binary,
      2 device_type           fixed binary,
      2 slew_control,
      2 slew_type             fixed binary (18) unaligned unsigned,
        3 slew_count          fixed binary (18) unaligned unsigned,
      2 flags,
        3 binary              bit (1) unaligned,
        3 preslew             bit (1) unaligned,
        3 pad                 bit (34) unaligned,
      2 element_size          fixed binary,
      2 n_elements            fixed binary (24),
      2 data,
        3 bits (terminal_io_record_n_elements refer
           (terminal_io_record.n_elements))
           bit (terminal_io_record_element_size refer
            (terminal_io_record.element_size)) unaligned;
```

STRUCTURE ELEMENTS

version
: is the current version of this structure given by the value of the named constant terminal_io_record_version_1. (Output)

device_type
: is the type of device from which this record is read. (Output). Its possible values are TELEPRINTER_DEVICE or READER_DEVICE.

slew_control.slew_type
: is always set to SLEW_BY_COUNT. (Output)

slew_control.slew_count
: is always set to 1. (Output)

flags.binary
: is always set to "0"b. (Output)

flags.preslew
: is always set to "0"b. (Output)

element_size
: is always set to 9. (Output)

n_elements
: is set to the number of characters returned in the record. (Output)

data.bits
is the actual returned data. (Output). This I/O module converts the data input from the remote workstation to ASCII.

CONTROL OPERATIONS

This I/O module supports the following control operations:

end_write_mode
ensures that all previously written data has been transmitted to the HASP multiplexer and then writes an end-of-file record for the device.

hangup_proc
is used to specify a procedure to be invoked when this attachment's channel is hung up. The info_ptr points to the following structure:

```
dcl 1 hangup_proc_info        aligned,
      2 procedure             entry variable,
      2 data_ptr              pointer,
      2 priority              fixed binary;
```

where:

procedure
is the procedure to be invoked when the hangup occurs. (Input)

data_ptr
is a pointer to be supplied to the procedure. (Input)

priority
is the priority for the hangup event. (Input)

A detailed explanation of data_ptr and priority can be found in the description of the ipc_ subroutine.

runout
ensures that all data has been transmitted to the HASP multiplexer from where it is guaranteed to be transmitted to the terminal.

read_status
determines whether or not there are any records waiting for a process to read. The info_ptr should point to the following structure which is filled in by the call:

```
dcl 1 info_structure          aligned,
      2 ev_chan               fixed bin (71),
      2input_availabible      bit (1);
```

where:

ev_chan
is the event channel used to signal the arrival of input. (Output)

input_available
indicates whether input is available (Output):
"0"b no input
"1"b input

resetread
flushes any pending input.

resetwrite
flushes any as-yet unprocessed output.

select_device and reset
are ignored rather than rejected for compatibility with other I/O modules used by the I/O daemon.

MODES OPERATION

This module accepts the "non_edited" and "default" modes for compatibility with other I/O modules used by the I/O daemon, but ignores them.

CHARACTER SET SPECIFICATION

This I/O module allows the specification of the character set used by the remote workstation through the -terminal_type attach option.

If -terminal_type is given, the referenced terminal type must be defined in the site's TTT with both an input and output translation table. This module uses these translation tables to convert data from the remote workstation's character set to ASCII and vice versa.

If -terminal_type is not given, the remote system is assumed to use EBCDIC as its character set. In this case, the subroutine ascii_to_ebcdic_ is used to convert data sent to the workstation; the subroutine ebcdic_to_ascii_ is used to convert data received from the remote system.

CARRIAGE CONTROL SPECIFICATIONS

Multics I/O daemon software uses three special slew operations: skip to top of the next page, skip to top of the next inside page, and skip to the top of the next outside page. By default, this I/O module assumes that all of these slew operations can be simulated on the remote workstation's line printer by skipping to channel one. However, through use of the -top_of_page, -inside_page, and -outside_page control arguments, any sequence of carriage motions can be specified to simulate these slew operations.

The format of this carriage control specification is:

```
Tn:Tn:Tn:...
```

where n is a numeric value and T represents how to interpret that numeric value. T can be either c representing skip to channel n, or s representing slew n lines.

For example, the string:

```
c7:s5:c12
```

means skip to channel seven, space five lines, and finally skip to channel 12.

Name: ibm_pc_io_

The ibm_pc_io_ I/O module is used to transfer ASCII files between a Multics process and a microcomputer that runs the IBM PC-to-Host data transfer protocol. It performs 7-bit stream I/O over an asynchronous communications channel using the data transfer protocol for the IBM Personal Computer as defined by IBM in their *Asynchronous Communication Support Manual,* Order No. 6024032.

Entry points in this module are not called directly by users; rather the module is accessed through the I/O system, usinng the micro_transfer command.

ATTACH DESCRIPTION

```
ibm_pc_io_ switch
```

ARGUMENTS

switch
: is the name of the target I/O switch. The switch must be open for stream_input_output. The I/O module for the target switch must be supported by the timed_io_ module. The user is responsible for setting any modes required by the protocol. For example, modes for the user_i/o switch would be: "^8bit,breakall,^echoplex,rawi,^crecho,^lfecho,^tabecho,rawo"

OPEN OPERATION

The ibm_pc_io_ I/O module supports the stream_input and stream_output opening modes.

CLOSE OPERATION

When opened for stream_output, the close entry transmits any remaining data in the internal buffer before closing the switch. See Buffering below.

PUT CHARS OPERATION

The put_chars entry transmits the data one line at a time in variable length data blocks. The end-of-line character is a carriage return. Lines exceeding 250 characters are transmitted in multiple blocks. See Notes below.

GET CHARS OPERATION

The get_chars entry reads protocol blocks and returns the message text to the caller's buffer. For further explanation of the get_chars entry, see the iox_$get_chars entry.

GET LINE OPERATION

The get_line entry reads protocol blocks and returns the message text to the caller's buffer. Characters are returned until either a carriage return character is placed in the buffer or the buffer is filled.

CONTROL OPERATION

This operation is not supported.

MODES OPERATION

This operation is not supported.

BUFFERING

The IBM PC-to-Host protocol uses variable length data packets. Data not ending with a carriage return character is stored in an internal buffer by the the ibm_pc_io_ I/O module.

NOTES

A line is a string of characters terminated by a carriage return character, 015 (octal). Only 250 characters can be transmitted in each line. When a line of text contains more than 250 characters, it is divided and one or more carriage returns inserted before transmission. For example, a call to ibm_pc_io_ to transmit a 260 character line would result in two lines being transmitted, one containing the first 249 characters plus a carriage return and the second containing the last 11 characters.

The IBM PC-to-Host data transfer protocol does not check for errors during transmission.

No particular line speed is guaranteed when transferring data between Multics and a microcomputer. Line speed is dependent on the microcomputer and the load of the FNP and communication system for Multics. Due to the nature of the IBM PC-to-Host protocol, files may not be successfully transferred to Multics over high-speed lines. The actual limit depends on the site configuration and current load.

DEFINITIONS

CR$	Carriage Return (Hex 0D) (Oct 15)
XON$	XON Character (Hex 11) (Oct 21)
XOFF$	XOFF Character (Hex 13) (Oct 23)
IBG$	Begin Transmission Code (Hex 1C) (Oct 34)
ITM$	Terminate Transmission Code (Hex 17) (Oct 27)

TRANSMISSION MEDIUM LEVEL PROTOCOL

Asynchronous, 7 data bits.

Files must be ASCII text files and have no lines longer than 250 characters.

MESSAGE BLOCK LEVEL PROTOCOL

The standard transmission portion of a message block is a variable length character block, maximum 250 characters, followed by a carriage return.

FILE LEVEL PROTOCOL

When writing programs that implement the IBM PC-to-Host protocol, users should follow the procedures listed below:

The Sending Program

1. The program should loop, reading the communications line and waiting for reception of a text line ending with the control characters IBG$ CR$.

2. When such a line is received, the program should send a text line ending with IBG$ CR$. (This line may contain an informative message as well, such as "Starting file transmission.")

3. The program then transmits the file. Each line in the file should be sent as a line ending in a Carriage Return (CR$)

4. While transmission is taking place, the program should monitor the input from the communications line and take the following actions:

   a. If an XOFF$ CR$ is seen, stop transmission of lines. When an XON$ CR$ is seen, resume transmission.

   b. If a line ending in ITM$ CR$ is seen, stop all transmission. This line will contain as text the reason the receiving microcomputer has requested termination.

   c. When all lines in the file have been sent, the program should send a line ending in ITM$ CR$. (This line can contain an appropriate message, such as "File transmission completed.")

The Receiving Program

1. The program should loop, sending out a message ending in IBG$ CR$ every 15 to 20 seconds. This message may also contain text, such as "Ready to receive file.")

2. During the loop in Step 1, the communications line must be monitored continually for messages from the microcomputer. When a line ending in IBG$ CR$ is received, the program moves on to step 3.

3. Each line received (after the one ending in IBG$ CR$) should be stored as a file record. As these lines end with Carriage Returns (CR$), the program might delete the CR$ before storing a line. Before storing a line, the program checks it to see if it ends in ITM$ CR$. If it does, the program does not store that line, but closes the file and stops operation.

4. The program can stop transmission by the microcomputer by sending a line ending with an ITM$ CR$. This line may also contain a message giving the reason for the termination.

5. If the program is receiving lines faster that they can be stored, it can suspend transmission by sending a line consisting of an XOFF$ CR$ to the microcomputer. When it has caught up with the input, it can start up transmission by sending a line consisting of an XON$ CR$ to the microcomputer.

Name: ibm_tape_io_

The ibm_tape_io_ module is an mtape_ Per-Format module that supports I/O to and from IBM standard labeled, unlabeled and DOS formated tapes under control of the mtape_ I/O module. The mtape_ IBM Per-Format module (referred to as the "IBM PFM" in the remainder of this discussion) may be selected explicitly by the use of the mtape_ attach description control argument "-volume_type ibm", or implicitly if the volume mounted by mtape_ during attachment is recognized by RCP as being a standard IBM tape. Tapes are processed by the IBM PFM in accordance with IBM documents: GC26-3795-3 (OS/VS Tape Labels), GC33-5374-1 (DOS/VSE Tape Labels), and GC26-3783-5 (OS/VS Data Management Guide). All of these documents are collectively referred to as "the Standard" in the remainder of this discussion.

OPENING

Opening of the IBM PFM is made by the iox_$open_file or the iox_$open entries (via the mtape_ open_file or open entries). The iox_$open_file entry provides for a character string open description, describing file processing attributes to be processed according to the wishes of the caller. The open description arguments accepted by the IBM PFM are described below. If opening is made by the iox_$open entry, the file processing attributes are formed from the current default values of the IBM PFM's open description arguments. The open description arguments have an initial default value, which are denoted in their respective descriptions below, or the default values may be changed by the user (see "Default Values" in the mtape_ I/O module description.).

The opening modes supported by the IBM PFM are sequential_input and sequential_output. If the opening mode specified is sequential_output, then the mtape_ attach description must have specified the "-ring" control argument or the general mtape_ control operation "ring_in" must have preceded the opening attempt.

OPEN DESCRIPTION

CONTROL ARGUMENTS

-append, -app
: specifies that the requested file is to be appended to the end of the file set as a new file. The requested opening mode must be sequential_output or the file opening will be aborted.

-no_append, -napp
: specifies that the requested file is not to be appended to the end of the file set. (Default)

-block N, -bk N
specifies the block size in bytes for output operations and is also required for input operations for unlabeled or DOS formated tapes. For input operations on standard labeled tapes, the block size is obtained from the file header label record. Permissible values are from 18 to 99996 bytes for "F", and 1044480 for "U" and "FB" formats, and 18 to 32760 bytes for "V", "VB", "VS" and "VBS" formats. (Default value is 8192 bytes.)

-comment STR, -com STR
specifies a user comment to be displayed on the user_output I/O switch after the file has been successfully opened. The comment text (STR) may be from 1 to 80 characters in length and 1044480 for "U". (Default is no -comment)

-default_fixed_record N, -dfr N
specifies the record length to be used for "f" or "fb" formats in the absence of a -record specification. The intended purpose of this control argument is to supply a default value for record size without having to include a -record specification in the open description. If the user wishes to explicitly specify the record length, the -record control argument should be used. Although the -default_fixed_record control argument may appear in a users open description and be processed accordingly, this would not be considered the proper method of explicitly supplying the record length. (Default value is 80)

-default_spanned_record N, -dsr N
specifies the record length to be used for "vs" or "vbs" formats, in the absence of a -record specification. The intended purpose of this control argument is to supply a default value for record size without having to include a -record specification in the open description. If the user wishes to explicitly specify the record length, the -record control argument should be used. Although the -default_spanned_record control argument may appear in a users open description and be processed accordingly, this would not be considered the proper method of explicitly supplying the record length. (Default value is 1044480)

-default_variable_record N, -dvr N
specifies the record length to be used for "v" or "vb" formats, in the absence of a -record specification. The intended purpose of this control argument is to supply a default value for record size without having to include a -record specification in the open description. If the user wishes to explicitly specify the record length, the -record control argument should be used. Although the -default_variable_record control argument may appear in a users open description and be processed accordingly, this would not be considered the proper method of explicitly specifying the record length. (Default value is 8192)

-display, -ds
specifies that the entire open description, after it has been parsed and any necessary defaults added, is to be displayed on the user_output I/O switch.

-no_display, -nds
specifies that the open description will not be displayed on the user_output I/O switch. (Default)

-dos
specifies that the file to be processed is in IBM DOS format. IBM DOS files contain only 1 header label (the HDR1 label) and do not retain any information as to file format, block length and record length. It is therefore necessary to specify the -block, -record and -format control arguments (or allow the default values for same to be used) even when opening an IBM DOS file for input.

-no_dos, -ndos
specifies that the file to be processed is not in IBM DOS format but is in fact in IBM standard OS/VS format. (Default)

-expires date, -exp date
specifies the expiration date of the file to be created, where date must be of a form acceptable to the convert_date_to_binary_ subroutine. (Default is no -expires)

-extend, -ext
specifies extension of an existing file.

-no_extend, -next
specifies that the requested file is not to be extended. (Default)

-force, -fc
specified that the expiration date of the file being overwritten is to be ignored.

-no_force, -nfc
specifies that the expiration date of a file being overwritten is not to be ignored. If the expiration date is not in the past, the user is queried for permission to overwrite the file. (Default)

-format F, -fmt F
specifies the record format of the file to be created. Permissible values are: U, F, V, VS, FB, VB, and VBS. (They may be specified in either upper or lower case.) (Default value is VB)

-label_entry entry, -lbe entry
specifies the entry point of a user subroutine which will be called to process the contents of user label records on input and generate the contents of same, for subsequent writing by mtape_ on output. (See "Calling sequence for user label processing routine" below.) (Default is no -label_entry)

-last_file, -lf
specifies that the file to be processed is the last file of the file set.

-not_last_file, -nlf
specifies that the file to be processed may not be the last file of the file set. (Default)

-mode STR, -md STR
specifies the encoding mode used to record the file data. Permissible values of STR are ascii, ebcdic or binary. (Default value is ebcdic)

-modify, -mod
specifies modification of an existing file while retaining the file attributes as recorded in the original files header label records.

-no_modify, -nmod
specifies that modification of an existing file is not to be performed. (Default)

-name STR, -nm STR
specifies the file identifier of the requested file. STR can be from 1 to 17 characters. (Default is no -name)

-next_file, -nf
specifies the file to be processed as the next (or first) file of the file set. This control argument is intended to be used when sequentially processing files. For output operations, if -name or -number are not specified, the values of their respective fields are fabricated by using the next sequential number as the file sequence number and forming the file name by concatenating the string "FILE" with the alphanumeric representation of the file number. (i.e. "FILE0001"). (Default)

-not_next_file, -nnf
specifies that the requested file is not the next file.

-number N, -nb N
specifies the file sequence number or numerical position within the file set. Permissible values range from 1 to 9999. (Default is no -number)

-record N, -rec N
specifies the logical record length in bytes. Permissible values range from 18 to 1044480 (sys_info$max_seg_size * 4) bytes, but the legality of the record size is dependent on the record format specified with the "-format" control argument and the block size. In general the record size must be <= the block size with the exception of "spanned record" formats (i.e. VS or VBS formats) where the record size may be the max allowable. (No default value. The default record size is determined by the value of the appropriate "-default_<type>_record" specification, where <type> can be either fixed, variable or spanned.)

-replace STR, -rpl STR
specifies replacement of an existing file, where STR is the file identifier to use in the search for the file to be replaced. (Default is no -replace)

-system_use
: specifies that when opening for output, certain fields of the HDR2 and EOV2 label records will be used to record the recording mode ASCII, EBCDIC or BINARY), and the volume name of the next volume in the volume sequence list. The fields used for these purposes are HDR2 character position 40 for recording mode (recorded as an EBCDIC "1", "2", or "3" for ASCII, EBCDIC, or BINARY respectively), and EOV2 character positions 41 - 46 for the next volume name. The IBM OS/VS Tape Labels specification marks these fields as "reserved for future use". It is therefore recommended that the "-system_use" control argument not be used in an interchange environment.

-no_system_use
: specifies that the HDR2 and EOV2 label record fields mentioned above will not be corrupted. (Default)

CLOSING

Closing of the IBM PFM is made by the iox_$close_file or the iox_$close entries (via the mtape_ close_file or close entries). The iox_$close_file entry provides for a character string close description, describing actions to be taken by the Per-Format module upon closing the I/O switch. If closing is made by the iox_$close entry, the close time actions are formed from the current default values of the IBM PFMs close description arguments. The close description arguments have an initial default value, which are denoted in their respective descriptions below, or the default values may be changed by the user (see "Default Values" in the mtape_ I/O module description.).

CLOSE DESCRIPTION

CONTROL ARGUMENTS

-close_position STR, -cls_pos STR
: specifies where to physically position the tape volume within the file that is being closed. The values of STR are case insensitive and may be selected from bof (for beginning of file), eof (for end of file), or leave (to leave the tape volume positioned where it is. (Default value is leave)

-comment STR, -com STR
: specifies a user comment to be displayed on the user_output I/O switch, after the file has been successfully closed. The comment text (STR) may be from 1 to 80 characters in length. (Default is no -comment)

-display, -ds
: specifies that the entire close description, after if has been parsed and any necessary defaults added, is to be displayed on the user_output I/O switch.

-no_display, -nds
: specifies that the close description will not be displayed on the user_output I/O switch. (Default)

READ RECORD OPERATION

The IBM PFM supports the iox_$read_record operation when the I/O switch is open for sequential_input. In general, format dependent logical records are extracted from physical tape blocks and written into the callers buffer area. As each tape block is exhausted, the IBM PFM requests the mtape_ I/O module to read in the next tape block. This sequence continues until logical End of File is detected by the IBM PFM, at which time error_table_$end_of_info is returned to the caller, and no further read_record requests will be accepted by mtape_ or the IBM PFM until the current file is closed and another file is subsequently opened. If the callers buffer length is not long enough to contain the entire logical record, as much data as will fit in the specified buffer is returned and error_table_$long_record is returned to the caller. In this case, the IBM PFM will position to the next logical record. If in the course of reading logical records, an End of Volume condition is detected by the IBM PFM, automatic volume switching is initiated, which if successful, will be transparent to the caller.

WRITE RECORD OPERATION

The IBM PFM supports the iox_$write_record entry when the I/O switch is open for sequential_output. In general, data of the specified record length is extracted from the users buffer, formatted into logical tape records and written into a physical tape block buffer. As each tape block buffer is filled, the IBM PFM requests mtape_ to queue up the full buffer for writing and return a pointer to the next buffer to fill. This sequence continues until either: (1) The I/O switch is closed or (2) an mtape_ "volume_status" or volume_set_status" control operation is requested to be processed. In both cases, if a partially filled buffer exists, it will be queued up for writing as a short block and all unwritten buffers will be requested to be written out to tape. If the I/O switch is being closed, the IBM PFM now writes out the End of File trailer sequence. If during the course of writing tape blocks the End of Volume condition is detected, the IBM PFM immediatly writes out the End of Volume trailer labels and requests a volume switch to mount the tape to contain the next file section. After the new tape volume has been successfully mounted, the IBM PFM initiates the volume label and new file section header labels and then requests that the unwritten buffers at the time of the end of volume detection be written out to tape. At this time, the write_record operation being processed at the time of the End of Volume detection is resumed.

POSITION OPERATION

The IBM PFM supports the iox_$position operation when the I/O switch is opened for sequential_input. All positioning types legal for sequential_input are supported. (See the description of iox_$position earlier in this manual.)

READ LENGTH OPERATION

The IBM PFM supports the iox_$read_length operation when the I/O switch is open for sequential_input. The read_length operation is implemented by actually reading the next logical record to determine its length, while discarding the actual data. After the length has been determined, backspace record position operation is executed to position to the location prior to the read_length operation. When executing read_length operations on spanned formatted records, or if the read_length operation is to determine the length of the first record of the next block, actual tape motion (i.e. read forward, and backspace block) may be necessary and will occur automatically. If a spanned record spans a volume boundary, volume switching is initiated both when doing the actual read operation and the backspace.

CONTROL OPERATION

The IBM PFM supports all of the general mtape_ control operations described in the mtape_ I/O module description. There are no control operations that are specific to the IBM PFM.

CALLING SEQUENCE FOR USER LABEL PROCESSING ROUTINE

In order to process user defined file labels when the "-label_entry" open description argument is used, the entry variable argument to the "-label_entry" control argument must conform to the following calling sequence in order to be called properly by mtape_ and its Per-Format modules:

```
dcl user_label_entry entry (ptr, char (*), fixed bin,
    fixed bin, fixed bin, fixed bin (35));

call user_label_entry (iocb_ptr, user_label_data, label_number,
     label_type, file_section_number, code);
```

WHERE

iocb_ptr
: is a pointer to the I/O control block through which the mtape_ I/O module is attached. A user_label_entry routine may wish to know more information about the file for which it is processing user labels. This can be accomplished by calling the iox_$control entry with this iocb_ptr and executing the mtape_ "file_status" control operation.

user_label_data
: is the actual contents of the user label record to be processed (INPUT) or written (OUTPUT). The length of this field will be 76 characters on input and truncated to same on output.

label_number
: is the number of the user label record within the file label group. The IBM standard allows from 1 to 9 user label records within a file label group (UHL1 - UHL9, and UTL1 - UTL9).

label_type
: is the encoded file label group type that the user_label_entry is being called to process label records for. Its possible values are as follows:

  1 = Beginning of file (BOF) label group
  2 = End of volume (EOV) label group
  3 = End of file (EOF) label group

file_section_number
: is the section number of the file for which the user_label_entry routine is being called to process user labels for. For multivolume files, this would essentially be the number of the volume (the first volume on which a file resides being number 1) on which this file "section" resides. For single volume files, the file_section_number would always be a 1.

code
: is a standard system error code. When writing user labels, the user_label_entry routine should set code to error_table_$end_of_info in order to tell the caller that no more user labels are to be written. Otherwise, the user_label_entry is called repeatedly to generate user label data until the maximum number of user labels have been written.

SEARCHING FOR A FILE

Before a file may be either created or read, its physical position within the volume set must be located. In the case of file creation, its physical position may be non-existent, but to ensure file set integrity all of the files in the file set must be searched to ensure its non-existence. To reduce physical tape searching to a minimum, the IBM PFM in concert with mtape_ maintains a linked list of file set members, with adequate information in each element of the linked list to identify the file it represents and its physical position within the volume set. At the time of the first opening, the above mentioned linked list of file set members does not exist. In this case, the volume set is searched sequentially forward until the desired file is found. As each file preceding the desired file is identified, a new element is added to the linked list of file set members, extracting file identity and format information from the file header and trailer labels, and obtaining the physical position of the file header from mtape_. On subsequent file openings, this linked list of file set members is searched first, and if the desired file is identified as being one of the elements, the volume set is positioned to the indicated position of the file header. If the desired file is not found in the linked list of file set members, then the volume set is searched forward from the position of the last identified file in the linked list, adding to the list as it proceeds in an attempt to find the desired file.

There are 6 open description control arguments which deal with identifying a file to be processed. These are: -append, -last_file, -name, -next_file, -number and -replace. From reading their descriptions above, it can be seen that if some of them were used together, they would form an inconsistent identity for a file to be found. (e.g. If -last_file and -next_file were used together, they may or may not describe the same file.) In order to keep the set of file identity arguments consistent for any given file, certain rules are applied when the open description is parsed as follows:

1. Open description arguments are parsed from left to right.

2. Any default arguments and their associated values are parsed before the users open description is parsed.

3. Control arguments and their associated values on the right take precedence over the same control argument and its value that preceded it. (e.g. In an open description which included "-name FILEX -name FILEY", the parsed result would be "-name FILEY".)

4. Binary control arguments (e.g. -last_file) all have an associated antonym value (i.e. -no_last_file). As each binary control argument is parsed, it takes precedence and replaces any opposite control argument that preceded it. (e.g. In an open description which included "-last_file -no_last_file" the parsed result would be "-no_last_file".)

5. For each of the 6 file identity open description arguments, there are a certain set of control arguments with which it is mutually exclusive with and takes precedence over. The chart below illustrates this mutual exclusitivity:

	-append	-last_file	-name	-next_file	-number	-replace
-append	*	*		*		
-last_file	*	*		*	*	*
-name		*	*	*		*
-next_file	*	*	*	*	*	
-number		*		*	*	
-replace		*	*	*		*

FILE IDENTIFIERS

Associated with every file is a name (file identifier) and a number (file sequence number). The file identifier must be 17 characters or less. When creating a file, the file identifier must be composed of one or more components of one to eight characters, with adjacent components separated by a period. The first character of each component 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. If a file identifier (of an existing file) does not meet the naming conventions established for files created on the Multics system, the file must be referenced using the -number control argument and a file sequence number.

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 file to be created may be identified explicitly by specifying the file name and/or number (with the -name and -number open description control arguments) either together or individually. If a -name and -number control arg appear in the same open description, they must identify the same file or an error will result.

The file to be created may be identified implicitly by specifying one of the relative position control arguments, -append, -last_file or -next_file in an open description.

Implicit file replacement is also accomplished if the file to be created is identified as already existing.

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 STR 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.

The -format F, -record R and -block B control arguments, or their default values, 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 99996 characters (not less than 32760).

vb for variable-length records.
  Used when records are of varying lengths, the longest not in excess of 32752 characters.

vbs for spanned records.
Used when the record length is fixed and in excess of 32760 characters, or variable and in excess of 32752 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 1044480 characters.

When the -record 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 = vb  | v:   amrl + 4 <= R <= 32756
F = vbs | vs:  amrl <= R <= 1044480
F = u:         R is undefined
               (the -record control argument should not be used.)
```

When the -block 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 = vb:        b >= R + 4
F = v:         B = R + 4
F = vbs | vs:  20 <= B <= 32760
F = u:         amrl <= B <= 1044480
```

ENCODING MODE

The IBM PFM makes provision for three data encoding modes: EBCDIC, binary, 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 control argument can be used to explicitly specify the encoding mode.

If STR 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 STR 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 for the specific ASCII to EBCDIC mapping used by the IBM PFM.) If STR is the string binary, any 9-bit byte value can be recorded. However, data written on IBM equipment with binary mode may not be compatible with Multics, or vice versa.

Unless the "-system_use" open description control argument is used, the -mode argument must be used when subsequently processing an ASCII or binary file, or the default mode must be changed accordingly. (If not used, the list_tape_contents command does not supply the specific mode in its report).

PADDING

Unlike its predecessor, the tape_ibm_ I/O module, the IBM PFM does not require block padding on output to a modulo 4 characters, unless the recording mode selected is binary. In this case the IBM PFM automatically pads every block with from 1 to 3 blanks to satisfy the modulo 4 requirement.

READING A FILE

The open 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 open 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 open description need only identify the file to be read; no other control arguments are necessary. Any of the 6 file identification open description control arguments (See "Searching For a File" above.) may be used to identify the file to be read.

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.

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 open 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 by using any of the 6 open description file identifying control arguments. (See "Searching For 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.

Any structure attribute open description control arguments specified by the user are ignored when extending a file.

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 open 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.

Any structure attribute open description control arguments specified by the user are ignored when modifying a file. The file to be modified is identified as above.

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 <= ddd <= 366. A special case of the Julian date form is the value "00000" (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; the 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. The -force control argument unconditionally grants permission to overwrite a file without querying the user, regardless of "unexpired" status.

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 VB or VBS format, blocked spanned records, so that blanks are not inserted.

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 IBM PFM 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 IBM PFM 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 < 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 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|7| 375 |  |8|8| 280|  |0|0|     800      |
            |3|9|     |  |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|7| 375   |8| 280   |  |0|0|     800      |
            |7|9|       |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 four-character SDW contains a record segment length in binary, that includes the length of the SDW itself, plus a binary record segment code in binary, that 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. For unblocked files:

```
          -------   -----------   -----------------------------
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|   |
          -------   -----------   ----------------   --------
```

For blocked files:

```
          -------   -----------   -----------------------------
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   | |     127      | | 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 IBM PFM 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 control argument, the IBM PFM attempts to append it to the file set, resulting in an error.

CONFORMANCE TO STANDARD

With two exceptions, the IBM PFM conforms to the Standard: the IBM PFM ignores the data set security field in the HDR1 label on input, and records it as 0 on output; if the -system_use open description argument is used, characters positions 40 - 46 or the HDR2/EOF2/EOV2 labels, are recorded with the file recording mode and the next volume name (EOV2 only). (See label Processing below.)

LABEL PROCESSING

VOL1

The label is processed on input and output. The owner-name and address-code-field, character positions (CP) 42 to 51, holds a three-character volume authentication code, in character positions 42 to 44 and the character string "MULT001" in character positions 45 to 51.

HDR1/EOF1/EOV1

The labels are processed on input and output. The system-code-field, CP 61 to 73, is recorded as "MULTICS IBM2 ".

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 || "   "
```

If the -system_use open description argument is used on output, then the "Reserved for future use" field, character positions 40 to 46 is recorded as follows.

CP 40 - data encoding mode (all)
"1" = ASCII, 9 mode
"2" = EBCDIC, 9 mode
"3" = binary
CP 41 to 46 - volume name of the next volume (EOV2 only).

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 processed on output and input only if the "-label_entry" open description argument is given. Otherwise, not written on output and ignored on input.

UNLABELED TAPES

The IBM PFM 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.

In order to process unlabeled IBM tapes, the mtape_ attach description must contain the "-no_labels" and "-volume_type ibm" control arguments. The following open description control arguments do not apply when processing unlabeled tapes and are ignored:

-name	-extend
-replace	-modify
-expires	-dos
-force	

Volume switching is handled somewhat differently for unlabeled tapes. When the IBM PFM detects two consecutive tape marks 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 IBM PFM 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.)

The use of unlabeled tapes is strongly discouraged, particularly as an interchange medium. Since no format or volume identification information exists on the recorded media itself, it would be very difficult for a foreign site to retrieve data off of an unlabeled tape without extensive written instructions. Unlabeled tapes should only be used in a highly controlled environment.

Name: ibm2780_

The ibm2780_ I/O module performs stream I/O to a remote I/O terminal that has the characteristics of an IBM 2780 data transmission terminal.

Entry points in this module are not called directly by users; rather, the module is accessed through the I/O system.

This module in turn constructs an attach description for the module specified in the -comm control argument, passing the attach information for ascii or ebcdic, tty, transparent or nontransparent, and all other attach information specified by the caller.

ATTACH DESCRIPTION

```
ibm2780_ -control_args
```

CONTROL ARGUMENTS

The following control arguments are optional, with the exception of -comm and -tty:

-ascii
: transmits control information and data in ASCII.

-carriage_ctl STR
: the eight-character string STR, taken two characters at a time, sets the four carriage control characters that specify the advance of 0, 1, 2, and 3 lines. The default set of characters is ESCM, ESC/, ESCS, and ESCT, where the mnemonic ESC means the ASCII escape character.

-comm STR
: uses the communications I/O module specified by STR.

-device STR
: specifies that this attachment is associated with the device STR. Currently, it is accepted only for compatibility with other I/O modules.

-ebcdic
: converts control information and data to its EBCDIC representation before transmission. This is the default.

-horizontal_tab, -htab
supports tab control on the remote I/O terminal printer. Tabs are set every 10 spaces. The default is no tab control.

-multi_record
transmits multiple records (up to seven) as a block, rather than separately. The default is single-record transmission.

-nontransparent
uses a nontransparent communication protocol. This is the default.

-printer_select STR
the two-character string STR sets the printer select. The default printer select string is ESC/.

-physical_line_length N, -pll N
sets the maximum character width of the remote I/O terminal printer to N characters. The default is 80 characters. This variable is used to set tabs and pad records if the transparent option is specified.

-punch_select STR
the two-character string STR sets the punch select. The default punch select string is ESC4.

-slew_ctl STR
the six-character string STR, taken two characters at a time, sets the slew control characters that specify top of form, inside page, and outside page. The default set of characters is ESCA, ESCA, and ESCA.

-terminal_type STR, -ttp STR
STR specifies the terminal type whose conversion, translation, and special tables defined in the user or system terminal type table (TTT) are used to convert and translate input and output to and from the device. If not specified, no conversion or translation is performed. For more information about the allowable conversion values see "Notes" below.

-transparent
uses a transparent communication protocol.

-tty STR
connects the remote I/O station to the communications channel named STR.

OPEN OPERATION

The ibm2780_ I/O module supports stream_input, stream_output, and stream_input_output opening modes.

PUT CHARS OPERATION

The put_chars entry splits the data to be written into blocks of 80 or 400 characters, depending on whether multirecord mode is enabled, and transmits the number of characters specified to the specified communications I/O module. The blocks are of fixed or variable length, depending on whether transparent mode is enabled or not, respectively.

GET CHARS OPERATION

The get_chars entry reads characters up to 80 or 400 characters, depending on whether multirecord is enabled, and returns the number requested, up to the next record separator.

CONTROL OPERATION

This I/O module supports all the control operations supported by the communications I/O module specified in the attach description. In addition, it supports the following:

select_device
: selects the subdevice (printer, punch, or teleprinter) to which output is next directed. The input structure is of the form:

```
dcl device char(32) based;
```

set_bsc_modes
: sets the character mode, either ascii or ebcdic, and transparency. The input structure is defined as follows:

```
dcl 1 set_bsc_modes aligned,
      2 char_mode bit(1), unaligned,
      2 transparent bit(1) unaligned;
```

where:

char_mode
: is "1"b if ebcdic and "0"b if ascii.

transparent
: is "1"b if transparency is enabled and "0"b if not.

set_multi_record_mode
: sets the number of records per block. The input structure is of the form:

```
dcl record_number fixed bin based;
```

MODES OPERATION

This module supports the nonedited and default modes, which set and reset the edited output conversion, if it has been enabled by the -terminal_type control argument.

NOTES

The only allowable values in the output conversion table are 00 and any values greater than 16. All values defined in the description of the tty_ I/O module are allowed for input conversion. Input and output translation tables can be up to 256 characters in length.

Name: ibm3270_

The ibm3270_ I/O module performs stream I/O to and from an IBM 3270 Information Display System (or any compatible device) over a binary synchronous communications channel.

NOTE: Do not use this module to communicate with a 3270 device over a multiplexed channel. Use the tty_ module in that case.

This module description assumes a knowledge of the IBM 3270 communications protocol as described in the *IBM 3270 Information Display System Component Description*, Order No. GA27-2749-4.

Entry points in this module are not called directly by the user; rather, the module is accessed through the I/O system.

ATTACH DESCRIPTION

```
ibm3270_ device {-control_args}
```

ARGUMENTS

device
  is the name of the communications channel to be used.

CONTROL ARGUMENTS

-ascii
  uses the ASCII bisync protocol and character code.

-async
  specifies that the I/O module is to return to its caller immediately after performing a read order (described below under "Control Operation") when input is not available, rather than blocking and waiting for a response from the device.

-ebcdic
  uses the EBCDIC bisync protocol and character code. This is the default.

OPEN DESCRIPTION

This I/O module supports only the stream_input_output opening mode. If the -async control argument is specified in the attach description, the open operation may return the status code error_table_$request_pending; in this case, the caller should perform an event_info order (see "Control Operation") and block on the returned event channel; when the process receives a wakeup on this channel, the open operation should be retried.

CONTROL OPERATION

This I/O module supports all the orders supported by the tty_ I/O module, as well as those described below. All orders are supported when the I/O switch is open, except for event_info, which is supported when the I/O switch is attached.

event_info
: returns the name of the event channel over which wakeups are sent when input or status is received from the communications channel. The info_ptr must point to an aligned fixed binary (71) number, in which the value of the event channel is returned. This order should be used if the -async control argument appears in the attach description (see "Attach Description" above).

general_poll
: causes a general poll operation to be initiated at the 3270 controller. Once the I/O switch is open, either a general_poll order or a poll order must be issued before any input can be received; however, the general_poll order does not have to be repeated, as polling is automatically resumed when appropriate by the I/O module. The info_ptr is not used.

get_input_message_size
: is used to obtain the maximum input message size. The info_ptr must point to a fixed binary variable in which the maximum message size is returned as a result of the call. This size is the one most recently specified by a set_input_message_size order. If no set_input_message_size order has been done since the switch was attached, a size of 0 is returned.

poll
: causes a specific poll operation to be performed on a single device connected to the controller. The info_ptr must point to a fixed binary number containing the identification number of the device to be polled. To ensure that the device is polled as soon as possible, this order usually should be preceded by a stop_general_poll order.

read
: causes input or status information from a single device to be returned, if any is available. If no status or input is available for any device on the communications channel, then the process blocks if the -async control argument is not specified in the attach description; if it is specified, a status code of error_table_$request_pending is returned.

The info_ptr must point to a user-supplied structure of the following form:

```
dcl 1 read_ctl aligned,
      2 version fixed bin,
      2 areap ptr,
      2 read_infop ptr,
      2 max_len fixed bin,
      2 max_fields fixed bin;
```

where:

version
    is the version number of the structure. (Input). It must be 1.

areap
    is a pointer to an area in which the read_info structure is allocated. (Output)

read_infop
    is a pointer to the read_info structure. (Output)

max_len
    is the largest number of characters that can be returned in a single data field. (Output)

max_fields
    is the largest number of data fields that can be returned in the read_info structure. (Output)

A read_info structure is allocated by the I/O module at the address specified by read_ctl.read_infop. This structure must be freed by the calling program. The read_info structure has the following form:

```
dcl 1 read_info aligned based (read_ctl.read_infop),
      2 version fixed bin,
      2 next_read_infop ptr,
      2 controller fixed bin,
      2 device fixed bin,
      2 reason,
        3 key fixed bin,
        3 sub_key fixed bin,
        3 code fixed bin(35),
      2 status,
        3 bits bit(12) unal,
        3 fill bit(24) unal,
      2 cursor_position fixed bin,
      2 max_fields fixed bin,
      2 max_len fixed bin,
      2 mod_fields fixed bin,
      2 data (read_ctl.max_fields refer (read_info.max_fields)),
        3 field_position fixed bin,
        3 contents char (read_ctl.max_len
           refer (read_info.max_len)) var;
```

where:

version
: is the version number of this structure. The structure described here is version 1.

next_read_infop
: is a pointer to the next read_info structure used by the I/O module. (The calling program should not attempt to make use of this item.)

controller
: is the identification number of the 3270 controller from which the data or status has been received.

device
: is the identification number of the particular device (attached to the specified controller) that produced the data or status information.

reason
: describes the event that caused the structure to be filled in.

key
: identifies the nature of the event, which is either an error or status condition, or an action on the part of the 3270 operator. It can have any of the following values:

1 an error was detected at the device. A status code describing the error is returned in reason.code (see "code" below).
2 the device reported status. The particular status is described by status.bits (see "status" below).
3 the operator pressed the ENTER key.
4 the operator pressed one of the program function (PF) keys. The particular key is identified by reason.sub_key (see "sub_key" below).
5 the operator pressed one of the program attention (PA) keys. The particular key is identified by reason.sub_key (see "sub_key" below).
6 the operator pressed the CLEAR key.
7 the operator inserted a card in the identification card reader.
8 the operator used the selector pen on an "attention" field.
9 the operator pressed the TEST REQUEST key.

sub_key
: is the number of the PF or PA key pressed if reason.key is 4 or 5, respectively.

code
: is a status code describing an error at the device if reason.key is 1.

status
: contains the device status if reason.key is 2.

cursor_position
: is the current position of the cursor on the display screen.

max_fields
: is the number of elements in the data array (below).

max_len
    is the length of the longest contents string (below).

mod_fields
    is the number of elements in the data array (below) that are actually filled in in this instance of the structure.

data
    describes the data fields containing the input. No data fields are provided if reason.key is 1, 2, 5, or 6.

field_position
    is the starting buffer address of the data field.

contents
    is the contents of the data field. It is always a null string if reason.key is 8.

set_input_message_size
    specifies the length, in characters, of the largest input message that is expected. The info_ptr must point to a fixed binary number containing the message size. A size of 0 indicates that there is no maximum message size. Use of this order when a maximum message size is defined greatly increases the efficiency of the channel.

stop_general_poll
    causes automatic general polling to stop; polling is not resumed until a general_poll order is issued. The info_ptr is not used.

write
    causes commands and data to be sent to the 3270. The info_ptr must point to a user-supplied structure of the following form:

```
dcl 1 write_info aligned,
      2 version fixed bin,
      2 controller fixed bin,
      2 device fixed bin,
      2 from_device fixed bin,
      2 command fixed bin,
      2 write_ctl_char,
        3 bits unal,
          4 print_format bit(2) unal,
          4 start_printer bit(1) unal,
          4 sound_alarm bit(1) unal,
          4 keyboard_restore bit(1) unal,
          4 reset_mdt bit(1) unal,
        3 copy_bits bit(2) unal,
        3 pad bit(28) unal,
      2 max_fields fixed bin,
      2 max_len fixed bin,
      2 mod_fields fixed bin,
      2 data (max_write_fields
         refer (write_info.max_fields)),
        3 orders unal,
          4 set_buffer_addr bit(1),
          4 start_field bit(1),
          4 insert_cursor bit(1),
          4 program_tab bit(1),
          4 repeat_to_addr bit(1),
          4 erase_to_addr bit(1),
        3 attributes unal,
          4 protected bit(1),
          4 numeric bit(1),
          4 display_form bit(2),
          4 reserved bit(1),
          4 mdt bit(1),
        3 pad1 bit(12) unal,
        3 field_position fixed bin,
        3 contents char (max_write_len
           refer (write_info.max_len)) var;
```

where:

version
: is the version number of the structure. It must be 1.

controller
: is the identification number of the 3270 controller to which the data is to be sent.

device
: is the identification number of the device on that controller to which the data is to be sent.

from_device
  is the identification number of the device to be used as the "from" device for a copy command.

command
  is the command to be sent to the device. It can have any of the following values:

  1 write
  2 erase/write
  3 copy
  4 erase all unprotected
  5 read modified
  6 read buffer

write_ctl_char
  contains the low-order 6 bits of the write control character (WCC) to be inserted in the data stream. If command (above) is 3 (copy), this field contains the low-order 6 bits of the copy control character (CCC), except that the keyboard_restore and reset_mdt bits are replaced by the copy_bits (below).

copy_bits
  contains the two low-order bits of the copy control character if command is 3. These are the bits that specify what type of data is to be copied.

max_fields
  is the number of elements in the data array (below).

max_len
  is the maximum length of any contents string (below).

mod_fields
  is the number of elements of the data array actually filled in in this instance of the structure.

data
  describes the individual data fields to be sent to the device.

orders
  identify orders to be inserted in the output stream.

set_buffer_addr
  indicates a set buffer address (SBA) order. The field_position (below) contains the buffer address to be set.

start_field
: indicates a start field (SF) order. The attribute character for the field is derived from attributes (below). If an SBA order is also indicated, the field starting address is contained in field_position (below); otherwise, the current device buffer address is used. The contents string, if nonnull, is written starting after the attribute character.

insert_cursor
: indicates an insert cursor (IC) order. If an SBA order is also indicated, the cursor is positioned to the address specified in field_position (below); otherwise, it is set to the current device buffer address. If contents is nonnull, the data is written starting at the new cursor position.

program_tab
: indicates a program tab (PT) order. If an SBA order is also indicated, the tab is inserted at the address specified in field_position (below); otherwise, it is inserted at the current device buffer address. If contents is nonnull, the data is written at the start of the field following the tab.

repeat_to_addr
: indicates a repeat to address (RA) order. The starting address is the current device buffer address; the ending address is specified in field_position (below). Neither an SBA order nor an EUA order can be indicated in the same field. The contents string must consist of a single character, which is to be repeated up to the address immediately preceding field_position.

erase_to_addr
: indicates an erase unprotected to address (EUA) order. The starting address is the current device buffer address; the ending address is specified in field_position (below). Neither an SBA order nor an RA order can be indicated in the same field. If contents is nonnull, the data is written starting at the address specified in field_position.

attributes
: contains the low-order six bits of the attribute character to be assigned to a field if start_field (above) is "1"b.

field_position
: is the device buffer address to be set if set_buffer_addr (above) is "1"b, or the ending address if repeat_to_addr or erase_to_addr (above) is "1"b.

contents
: is the data to be written. It may be a null string.

Name: ibm3780_

The ibm3780_ I/O module performs stream I/O to a remote I/O terminal that has the characteristics of an IBM 3780 data transmission terminal. The hardware options currently supported are defined by the control arguments described below.

Entry points in this module are not called directly by users; rather, the module is accessed through the I/O system.

This module in turn constructs an attach description for the module specified in the -comm control argument, passing the attach information for ascii or ebcdic, tty, transparent or nontransparent, and all other attach information specified by the caller.

ATTACH DESCRIPTION

```
ibm3780_ -control_args
```

CONTROL ARGUMENTS

The following control arguments are optional, with the exception of -comm and -tty:

-ascii
transmits control information and data in ASCII.

-carriage_ctl STR
the eight-character string STR, taken two characters at a time, sets the four carriage control characters that specify the advance of 0, 1, 2, and 3 lines. The default set of characters is ESCM, ESC/, ESCS, and ESCT where the mnemonic ESC means the ASCII escape character.

-comm STR
uses the communications I/O module specified by STR.

-device STR
specifies that this attachment is associated with the device STR.

-ebcdic
converts control information and data to its EBCDIC representation before transmission. This is the default.

-horizontal_tab, -htab
supports tab control on the remote I/O terminal printer. Tabs are set every 10 spaces. The default is no tab control.

-multi_record
transmits multiple records, up to six, as a block, rather than separately. The default is single-record transmission.

-nontransparent
uses a nontransparent communication protocol. This is the default.

-physical_line_length N, -pll N
: sets the maximum character width of the remote I/O terminal printer to N characters. The default is 80 characters (120 if -device specifies printer). This variable is used to set tabs and pad records if the transparent option is specified.

-printer_select STR
: the one-character string STR sets the printer select. The default printer select string is DC1.

-punch_select STR
: the one-character string STR sets the punch select. The default punch select string is DC2.

-slew_ctl STR
: the six-character string STR, taken two characters at a time, sets the slew control characters that specify top of form, inside page, and outside page. The default set of characters is ESCA, ESCA, and ESCA.

-terminal_type STR, -ttp STR
: STR specifies the terminal type whose conversion, translation, and special tables defined in the user or system terminal type table (TTT) are used to convert and translate input and output to and from the device. If not specified, no conversion or translation is performed. For more information about the allowable conversion values see "Notes" below.

-transparent
: uses a transparent communication protocol.

-tty STR
: connects the remote I/O station to the communications channel named STR.

OPEN OPERATION

The ibm3780_ I/O module supports stream_input, stream_output, and stream_input_output opening modes.

PUT CHARS OPERATION

The put_chars entry splits the data to be written into blocks of 80 or 512 characters, depending on whether multirecord mode is enabled, and transmits the number of characters specified to the specified communication I/O module. The blocks are of fixed or variable length, depending on whether transparent mode is enabled or not, respectively.

GET CHARS OPERATION

The get_chars entry reads characters up to 80 or 512 characters, depending on whether multirecord mode is enabled, and returns the number requested, up to the next record separator.

CONTROL OPERATION

This I/O module supports all the control operations supported by the communications I/O module specified in the attach description. In addition, it supports the following:

select_device
: selects the subdevice (either printer, punch, or teleprinter) to which output is next directed. The input structure is of the form:

```
dcl device char(32) based;
```

set_bsc_modes
: sets the character mode, either ascii or ebcdic, and transparency. The input structure is defined as follows:

```
dcl 1 set_bsc_modes aligned,
      2 char_mode bit(1), unaligned,
      2 transparent bit(1) unaligned;
```

where:

char_mode
: is "1"b if ebcdic and "0"b if ascii.

transparent
: is "1"b if transparency is enabled and "0"b if not.

set_multi_record_mode
: sets the number of records per block. The input structure is of the form:

```
dcl record_number fixed bin based;
```

MODES OPERATION

This module supports the nonedited and default modes, which set and reset the edited output conversion, if it has been enabled by the -terminal_type control argument.

NOTES

The only allowable values in the output conversion table are 00 and any values greater than 16. All values defined in the description of the tty_ I/O module are allowed for input conversion. Input and output translation can be up to 256 characters in length.

Name: mtape_

The mtape_ I/O module supports physical and logical I/O to or from magnetic tape volume(s), in any one of several formats, including:

```
ANSI standard format
IBM standard format
IBM Disk Operating System (DOS) format
IBM unlabeled format
```

Entries in this module are not called directly by users; rather, the module is accessed through the I/O system. See the *Multics Programmer's Reference Manual,* Order No. AG91 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.

block
: a collection of characters written to or read from a tape volume 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 blocks pertaining to a single subject. A file may be recorded on all or part of a volume, or on more than one volume.

file set
: a collection of one or more related files, recorded consecutively on a volume set.

per-format module
: an externally callable subroutine with several standard entry points. The naming convention for per-format modules is in the form of <volume_type>_tape_io_ where <volume_type> is the character string description of the volume label type as returned by RCP on tape input or requested by the user by the use of the -volume_type attach description argument or the default volume type on tape output. For a discussion of the definition and use of per-format modules, see "Per-format Modules" below.

record
: related information treated as a unit of information. A record is the smallest unit of information which can be written to tape.

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.

volume set
: a collection of one or more volumes on which one and only one file set is recorded. Volume sets may have any of the following relationships to a file set.

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

ATTACH DESCRIPTION

In addition to the I/O module name, only information relevant to the entire volume or volume set is supplied in the attach description. For the specification of information pertaining to files and file sets, refer to the section titled "Open Description" below. The attach description is a contiguous character string and has the following form:

```
mtape_ vn1 {-comment vn1_str} vn2 {-comment vn2_str} .......
       vnN {-comment vnN_str} {-control_args}
```

ARGUMENTS

vni
: is a volume specification. In the simplest (and typical) case, a volume specification is a volume name. Occasionally, keywords must be used with the volume name. For a discussion of volume names and keywords see "Volume Specification" below.

-comment vni_str, -com vni_str
: allows the optional specification of a message to be displayed on the operators console at the time volume vni is to be mounted. The comment text, vni_str, may be from 1 to 64 characters in length and must be quoted if it contains embedded white space.

vn1 vn2 ... vnN
comprise the volume sequence list. It can specify all volumes of the voluem set, followed by other volumes to be added to the volume set as new files are written. 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 current volume sequence list becomes exhausted during volume switching,
the user is queried for the next volume name when new volumes are needed. For more information, see "Volume Switch" below.

CONTROL ARGUMENTS
is a sequence of one or more attach control arguments. A control argument may appear only once.

-default_volume_type STR, -dvt STR
specifies the volume type (STR) to be used for Per-Format module selection when an unreadable or unlabeled tape is mounted for potential output operations and no -volume_type control argument is given. Permissable values for this control argument are ansi, or ibm. (Default value is ansi.)

-density N, -den N
specifies the recording density for output operations in bits per inch (BPI). For input operations, the density is determined and set automatically by RCP. Permissible values are 200, 556, 800, 1600 and 6250. (Default density is 1600 BPI.)

-device N, -dv N
specifies the number of tape devices that will be requested to be used simultaneously for multivolume operations. Permissible values are from 1 to 63. (Default is 1 device.)

-display, -ds
specifies that the entire attach description, after it has been parsed and any necessary defaults added, will be displayed on the user_output I/O switch.

-no_display, -nds
specifies that the attach description will not be displayed. (Default)

-error, -err
specifies that verbose error messages will be displayed when exception conditions (e.g. unrecoverable tape errors) are detected. (Default)

-no_error, -nerr
specifies that only error codes will be returned upon detection of exception conditions.

-label, -lbl
specifies that volume and file label records exist and or are to be recorded by the selected Per-Format module. (Default)

-no_label, -no_labels, -nlbl
specifies that volume and file label records do not exist or are not to be recorded by the selected Per-Format module. If this control argument is given when attempting to select a Per-Format module that does not accept unlabeled tape volumes, the attachment is aborted.

-ring, -rg
specifies that volumes are to be mounted with write rings installed.

-no_ring, -nrg
specifies that volumes are to be mounted with no write rings installed. (Default)

-speed N1{,N2,...,Nn}, -ips N1{,N2,...,Nn]}
specifies desired tape drive speed(s) in inches per second (IPS). permissible values are 75, 125 and 200. If more than one speed device is to be used, the optional second and third speed specification must be separated by commas as shown. If this control argument is omitted, RCP will pick any available speed device.

-system, -sys
specifies that the user is requesting to be considered a system process.

-no_system, -nsys
specifies that the user is not to be considered a system process. (Default)

-track N, -tk N
specifies the track type of the tape drive to be used. Permissible values are 7 or 9. (Default is 9 track.)

-volume vni, -vol vni
specifies that the following volume name (vni) begins with a hyphen (-) and would otherwise be considered a control argument.

-volume_type STR, -vt STR
specifies the volume type to be used in Per-Format module selection. Permissable values for this control argument are ansi, or ibm. (No Default. The volume type is determined by RCP for labeled volumes and by the -default_volume_type specification for unlabeled or unreadable volumes.)

-wait, -wt
specifies that when tape devices are not immediately available from RCP for a requested volume mount, the mtape_ I/O module should wait for the number of minutes specified by the -wait_time control argument (or its default value), before reporting an error on the initial volume mount or subsequent volume switching.

-no_wait, -nwt
: specifies that the mtape_ I/O module will not wait for an available device to become free, but instead report an error immediately. (Default)

-wait_time N, -wtm N
: specifies the time (in minutes) that the mtape_ I/O module will wait for unavailable tape drives to become available for volume mounts when the -wait control argument is specified. Permissible values range from 1 to 1440 minutes (24 hours). (Default wait time is 10 minutes.)

VOLUME SPECIFICATION

The volume name (also called the slot identifier) is an identifier physically written on, or affixed to, the volume's reel or container.

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 or the volume name may be interpreted as a control argument.

VOLUME SWITCHING

Volume switching is defined to be the act of switching from the current volume being processed, upon detection of physical end of volume, to another volume to continue processing. When writing tape, physical end of volume is detected when the end of tape reflective foil is passed across the appropriate tape drive sensor. This generates an exception status which mtape_ translates to the error_table_$end_of_volume error code. This error code is returned to the current Per-Format module upon completion of the write operation. The Per-Format module must now write the end of volume trailer sequence on the tape. This could consist of writing an end of file mark followed by a varying number of end of volume label records followed by 2 consecutive end of file marks for labeled formats, or simply 2 consecutive end of file marks for unlabeled formats. The Per-Format module finds the name of the next volume by searching for the next entry in the volume sequence list in the attach description. If the volume sequence list is exhausted, the user is queried for the next volume name. The Per-Format module must now initiate volume switching, which is performed by a common mtape_ entry.

Note that when end of volume was detected, the I/O was terminated after successfully writing the block when the end of volume condition was first recognized. Due to the asynchronous write-behind feature of mtape_, there may have been buffers that were not yet written. Part of the contract of the mtape_ volume switch entry is to preserve these unwritten buffers and copy their contents from the buffer area of the old volume to the buffer area of the new volume. After the new volume has been mounted (or re-activated, if it was already mounted), and appropriate validity checks done on the new volume, the Per-Format module now writes the volume labels and new file section header labels. (If dealing with unlabeled media, this step is omitted.) The Per-Format module now must request any saved and unwritten buffers to be written out by mtape_. The Per-Format module then resumes writing data to the new tape volume.

When reading tape, the physical end of volume is detected by the Per-Format module upon recognition of the end of volume trailer sequence. Volume switching is performed as described above, with the exception that there are no unprocessed buffers to contend with. In some formats, the name of the next volume is recorded in one of the end of volume label records. If this is the case, the Per-Format module will force a volume switch to the recorded volume name, even if it is different from the next volume name in the volume sequence list.

The mtape_ I/O module builds and maintains a linked list of file attribute structures as each file is processed or recognized in the course of searching for other files. Among other things, the file attribute structure contains information as to the file identifier, file sequence number and indices of the starting and ending volume set member which contain this file. In the course of opening a file, a search of this linked list of file attribute structures is made to determine if the requested file has already been processed or otherwise recognized during this attachment. If an entry for the requested file is found, then the volume set member on which the file resides is compared to the volume currently mounted. If this match is made then the physical file position on the volume is determined (from information contained in the file attribute structure) and the current volume is positioned to the beginning of the requested file. If an entry for the requested file is found in the linked list of file attribute structures, but the starting volume set member that contains this file is different from the current volume, then volume switching is initiated as described above. If no entry for the requested file is found in the linked list of file attribute structures, then a physical search for the requested file is initiated, starting from the current position of the current volume forward through each file position performing volume switching as above when necessary. As each file is identified, even though it is not the requested file, a file attribute structure is built for it and linked into the chain of other file attribute structures.

RESOURCE DISPOSITION

The mtape_ 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.

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.

In general, the decision of whether write rings are to be installed or not is made at attach time. This decision is effected by either the explicit use of the "-ring" attach description argument, or the current default value of the ring specification (See "Default Values" below). If output operations are to be performed on the volume set, then use installation of write rings must be specified or an error will result when attempting to open a file for output. The write ring decision may be effected after the attach is complete by the use of the "ring_in" control operation described below.

ERROR PROCESSING AND RECOVERY

All error recovery of data type errors is initiated by the physical tape interface module, tape_ioi_. During read operations, the error recovery procedures used consists of initiating the I/O for each block read with automatic hardware retry. If an error persists after automatic retry has been performed, then a total of up to 8 retry operations are attempted, each time varying the hardware read threshold and deskew in different combinations until the block has been read without error. If a data error ocurrs while writing a block, tape_ioi_ initiates error recovery procedures which consist of backspacing across the block in error, erasing and then re-writing the block for a total of up to 8 times until the block has been written successfully. If either a read or a write error cannot be recovered in this manner, mtape_ and the user are informed that an unrecoverable error exists. mtape_ locks the file and allows no further I/O
until the file is closed and subsequently re-opened.

OPENING

Opening is made through the iox_$open_file entry which supports a character string "open description" argument for supplying file specific attributes to the per-format modules (See "Open Description" below). The iox_$open entry is supported in the sense that it will forward the call to the mtape_$open_file entry, supplying a default open description. This default open description is different for each per-format module. Refer to the section titled "Per-format Modules" below, for details.

The ANSI and IBM Per-format Modules have a record oriented interface and support the sequential_input and sequential_output opening modes only.

An I/O switch can be opened and closed any number of times in the course of a single attachment. All openings are governed by the same attach description.

OPEN DESCRIPTION

The open description is an ASCII character string argument to the iox_$open_file entry and provides a means of specifying attributes and position information of the desired file to be processed.

For input operations on one of the supported labeled volume types, a minimal or null open description may be specified since all file attributes may be obtained from the file header label records or from default values (see "Default Values" below). Only file position information (e.g. File number or name) need be specified. If the user wishes to process a file set in a sequential manner (i.e. read the first file followed by the second file, etc.) and if the mtape_ default open description argument of "-next_file" is in force, then a null open description may be used. For output operations or input operations for unlabeled volume types, all file attributes and positioning information must be specified either in the open description or by using their corresponding default values.

Only those open description specifications that are generic to all of the supported volume types are defined below. For open description specifications that are particular to a given volume type, see their definition in the section titled "Per-Format Modules" below.

In general, the open description consists of zero, one or more control arguments.

ARGUMENTS

-append, -app
: specifies that the requested file is to be appended to the end of the file set as a new file. The requested opening mode must be sequential_output or the file opening will be aborted.

-no_append, -napp
: specifies that the requested file is not to be appended to the end of the file set. (Default)

-block N, -bk N
: specifies the block size in bytes for output operations and is also required for input operations for IBM unlabeled or DOS formatted tapes. For input operations on standard labeled IBM or ANSI tape files, the block size is obtained from the the file header label record. Permissible values are from 18 to 99996 bytes. (Defaults are 2048 bytes for ANSI and 8192 bytes for IBM formats.)

-comment STR, -com STR
: specifies a user comment to be displayed on the user_output I/O switch, after the file has been successfully opened. The comment text (STR) may be from 1 to 80 characters in length.

-default_fixed_record N, -dfr N
specifies the record length to be used for "f" or "fb" formats in the absence of a -record specification. The intended purpose of this control argument is to supply a default value for record size without having to include a -record specification in the open description. If the user wishes to explicitly specify the record length, the -record control argument should be used. Although the -default_fixed_record control argument may appear in a users open description and be processed accordingly, this would not be considered the proper method of explicitly supplying the record length. The default value of N is set to 80 (for 80 character records) for both the ANSI and IBM PFMs. This default value may be changed by the default setting mechanism (see "Default Values" below).

-default_spanned_record N, -dsr N
specifies the record length to be used for ANSI "s" or "sb" formats, or IBM "vs" or "vbs" formats, in the absence of a -record specification. The intended purpose of this control argument is to supply a default value for record size without having to include a -record specification in the open description. If the user wishes to explicitly specify the record length, the -record control argument should be used. Although the -default_spanned_record control argument may appear in a users open description and be processed accordingly, this would not be considered the proper method of explicitly supplying the record length. The default value of N is set to 1044480 (sys_info$max_seg_size * 4) for both the ANSI and IBM PFMs. This default value may be changed by the default setting mechanism (see "Default Values" below).

-default_variable_record N, -dvr N
specifies the record length to be used for ANSI "d" or "db" formats, or IBM "v" or "vb" format in the absence of a -record specification. The intended purpose of this control argument is to supply a default value for record size without having to include a -record specification in the open description. If the user wishes to explicitly specify the record length, the -record control argument should be used. Although the -default_variable_record control argument may appear in a users open description and be processed accordingly, this would not be considered the proper method of explicitly specifying the record length. The default value of N is set equal to the default block size (i.e. 2048 for ANSI and 8192 for IBM). This default value may be changed by the default setting mechanism (see "Default Values" below).

-display, -ds
specifies that the entire open description, after it has been parsed and any necessary defaults added, is to be displayed on the user_output I/O switch.

-no_display, -nds
specifies that the open description will not be displayed on the user_output I/O switch. (Default)

-expires date, -exp date
specifies the expiration date of the file to be created, where date must be of a form acceptable to the convert_date_to_binary_ subroutine.

-extend, -ext
specifies extension of an existing file.

-no_extend, -next
specifies that the requested file is not to be extended. (Default)

-force, -fc
specifies that the expiration date of the file being overwritten is to be ignored.

-no_force, -nfc
specifies that the expiration date of a file being overwritten is not to be ignored. If the expiration date is not in the past, the user is queried for permission to overwrite the file. (Default)

-format F, -fmt F
specifies the record format of the file. Permissible values for ANSI: U, F, D, S, FB, DB, and SB; For IBM: U, F, V, VS, FB, VB, and VBS. (They may be specified in either upper or lower case.) (Default values are DB for ANSI format and VB for IBM formats.)

-label_entry entry, -lbe entry
specifies the entry point of a user subroutine which will be called to process the contents of user label records on input and generate the contents of same, for subsequent writing by mtape_ on output. (See "Calling sequence for user label processing routine" described in the ansi_tape_io_ or ibm_tape_io_ description.)

-last_file, -lf
specifies that the file to be processed is the last file of the file set.

-not_last_file, -nlf
specifies that the file to be processed may not be the last file of the file set. (Default)

-mode STR, -md STR
specifies the encoding mode used to record the file data. Permissible values of STR are ascii, ebcdic or binary. (Default for ANSI format is ascii, for IBM format the default is ebcdic.)

-modify, -mod
specifies modification of an existing file while retaining the file attributes as recorded in the original files header label records.

-no_modify, -nmod
specifies that modification of an existing file is not to be performed. (Default)

-name STR, -nm STR
specifies the file identifier of the requested file. STR can be from 1 to 17 characters.

-next_file, -nf
specifies the file to be processed as the next (or first) file of the file set. This control argument is intended to be used when sequentially processing files. For output operations, if -name or -number are not specified, the values of their respective fields are fabricated by using the next sequential number as the file sequence number and forming the file name by concatenating the string "FILE" with the alphanumeric representation of the file number. (i.e. "FILE0001"). (Default)

-not_next_file, -nnf
specifies that the requested file is not the next file.

-number N, -nb N
specifies the file sequence number or numerical position within the file set. Permissible values range from 1 to 9999.

-record N, -rec N
specifies the logical record length in bytes. Permissible values range from 18 to 1044480 (sys_info$max_seg_size * 4) bytes, but the validity of the record size is dependent on the record format specified with the -format control argument and the block size. In general the record size must be <= the block size with the exception of "spanned record" formats (i.e. ANSI S or SB formats and IBM VS or VBS formats), where the record size may be the max allowable. (No default value. The default record size is determined by the value of the appropriate "-default_<type>_record" specification, where <type> can be either fixed, variable or spanned.) For a discussion of valid combinations of record format, record size and block size, refer to "Per-Foremat Modules" below.

-replace STR, -rpl STR
specifies replacement of an existing file, where STR is the file identifier to use in the search for the file to be replaced.

CLOSE OPERATION

The I/O switch must be open. Closing is made through the iox_$close_file entry which supports a character string "close description" argument for supplying file specific attributes to the per-format modules (See "Close Description" below). The iox_$close entry is supported in the sense that it will forward the call to the mtape_$close_file entry, supplying a null close description.

CLOSE DESCRIPTION

The close description is an ASCII character string argument to the iox_$close_file entry and provides a means of specifying actions to be taken when closing the current file.

In general, the close description consists of zero, one or more control arguments.

CONTROL ARGUMENTS

-close_position STR, -cls_pos STR
: specifies where to physically position the tape volume within the bounds of the file that is being closed. The values of STR are case insensitive and may be selected from "bof" (for beginning of file), "eof" (for end of file) and "leave" to leave the tape positioned where it is. (Default close position is "leave".)

-comment STR, -com STR
: specifies a user comment to be displayed on the user_output I/O switch, after the file has been successfully closed. The comment text (STR) may be from 1 to 80 characters in length.

-display, -ds
: specifies that the entire close description, after it has been parsed and any necessary defaults added, is to be displayed on the user_output I/O switch.

-no_display, -nds
: specifies that the close description will not be displayed on the user_output I/O switch. (Default)

CONTROL OPERATION

The mtape_ I/O module supports a variety of control operations.

```
file_status, fst                    file_set_status, fsst
force_end_of_volume, feov           hardware_status, hws
ring_in, rin
volume_status, vst                  volume_set_status, vsst
```

In the descriptions below, info_ptr is the information pointer specified in an iox_$control entry point call.

CONTROL OPERATIONS FROM COMMAND LEVEL

All control operations supported by this I/O module can be executed from command level by using the io_call command. The general format is:

```
io_call control switchname operation -control_arg
```

ARGUMENTS

switchname
: is the name of the I/O switch that is attached through the I/O module to an ANSI tape file-set.

operation
: is any of the control operations previously described.

FILE_STATUS OPERATION

This operation returns a structure that contains the current status of the file specified in the open description. If the I/O switch has never been opened, no information can be returned; this situation is indicated by mtape_fst.file_state = 0. If the switch was opened, but is now closed, the current status of the file is its status just prior to closing. The returned structure is defined in the include file mtape_file_status.incl.pl1. If a info_ptr is nonull, it must point to the mtape_fst structure shown below. If the info_ptr is given as null, then mtape_ will allocate the structure defined below in a temporary area on behalf of the user.

```
dcl  1 mtape_fst aligned based (info_ptr),
       2 version char (8),
       2 file_type fixed bin,
       2 file_state fixed bin,
       2 error_code fixed bin (35),
       2 file_id char (32),
       2 file_seq fixed bin,
       2 begin_vol_index fixed bin,
       2 end_vol_index fixed bin,
       2 file_sections fixed bin,
       2 generation fixed bin,
       2 gen_version fixed bin,
       2 creation char (6),
       2 expiration char (6),
       2 file_format char (3),
       2 block_len fixed bin,
       2 reclen fixed bin (21),
       2 recording_mode char (6),
       2 block_count fixed bin (35),
       2 read_errors fixed bin (35),
       2 write_errors fixed bin (35);
```

version
: is the current structure version. If info_ptr is nonnull on input, the caller must set the version element to fst_version_1. Otherwise mtape_ will set the version element in the structure it allocates.

file_type
: is the encoded file set type as determined by RCP, and could have one of the following values.

```
4 = IBM file set
5 = ANSI file set
```

file_state
is the current state of this file and could have one of the following values:

```
0 = No information available (I/O switch never opened)
1 = File not open
2 = File open
3 = File open and locked for error
```

The "locked for error" state referenced above is defined as an error or circumstance that prevents continued processing of this file. For example, parity error while reading, reached end of information, no next volume available, etc.

error_code
is the error code when mtape_fst.file_state is equal to 3 above, otherwise equal to 0.

file_id
is the file name or identifier as recorded in the appropriate file label record. This field will be blank for unlabeled formats.

file_seq
is the numerical order of this file within the file set.

begin_vol_index
is the numerical index of the first volume set member on which this file resides.

end_vol_index
is the numerical index of the last volume set member on which this file resides.

file_sections
is a count of the number of volumes on which this file resides.

generation
is the generation number of this file for those formats that support several "generations" of files. If this is the first generation, or if the format does not support several generations, then this field will be equal to 0.

gen_version
is the generation version number for those formats that support file generations. If this is the first generation, or if the format does not support several generations, then this field will be equal to 0.

creation
is the Julian creation date of this file in the form " yyddd".

expiration
is the expiration creation date of this file in the form " yyddd". If no expiration date was specified at file creation time, then the field will contain the string "00000".

file_format
is the encoded alphabetic representation of the volume type specific file format (e.g. "VBS" for IBM variable spanned block format, "DB" for ANSI variable record, blocked format, etc.).

block_len
is the maximum block length of each block within this file.

reclen
is the maximum record length of the logical records within this file.

recording_mode
is the numeric indication of the recording mode of this file. The following values are defined:

```
0 = Unknown or unspecified
1 = ASCII
2 = EBCDIC
3 = Binary
```

block_count
is the number of tape blocks contained in this file. If the file is still open, this number represents the number of blocks processed thus far.

read_errors
is a count of the number of read errors (recoverable as well as unrecoverable) encountered while reading this file. Note that read errors recovered by the hardware auto retry are not counted. Only Multics reread operations (with varying threshold and deskew settings) are counted.

write_errors
is a count of the number of write errors (recoverable as well as unrecoverable) encountered while writing this file.

FILE_SET_STATUS OPERATION

This operation may be used to obtain information about the entire file set as opposed to just the current file. If nonnull, the info_ptr should point to temporary segment which the mtape_ I/O module will fill with a structure as defined below. If the info_ptr is given as null, then mtape_ will allocate the structure for the user in a temporary area managed by mtape_. This structure is also defined in the include file mtape_file_status.incl.pl1.

```
dcl  1 mtape_fsst aligned based (info_ptr),
       2 version char (8),
       2 file_set_id char (32),
       2 file_type fixed bin,
       2 nfiles fixed bin,
       2 fs_stat (mtape_fsst_nfiles
                 refer (mtape_fsst.nfiles)),
         3 file_state fixed bin,
         3 error_code fixed bin (35),
         3 file_id char (32),
         3 file_seq fixed bin,
         3 begin_vol_index fixed bin,
         3 end_vol_index fixed bin,
         3 file_sections fixed bin,
         3 generation fixed bin,
         3 gen_version fixed bin,
         3 creation char (6),
         3 expiration char (6),
         3 file_format char (3),
         3 block_len fixed bin,
         3 reclen fixed bin (21),
         3 recording_mode char (6),
         3 block_count fixed bin (35),
         3 read_errors fixed bin (35),
         3 write_errors fixed bin (35);
```

version
: is the current structure version. If info_ptr is nonnull on input, then the caller must set the version field to fsst_version_1. If null, mtape_ will set the version element in the structure it allocates.

file_set_id
: is the file set identifier recorded in the file labels. This is usually the volume name of the first volume in the volume set.

file_type
: is the encoded file set type as determined by RCP, and could have one of the following values.

```
4 = IBM file set
5 = ANSI file set
```

nfiles
    is the number of files in the file set.

fs_stat
    is an array of structures of file set members, which appears below in sequential order. If the user allocates this structure, then the variable mtape_fsst_nfiles (declared in the mtape_file_status include file) must be set to the maximum number of files in the file set.

file_state through write_errors
    is the status of each file in the file set and are the same as the identical names described in the mtape_fst structure for the file_status control operation above.

FORCE_END_OF_VOLUME OPERATION

This operation forces the end of a volume condition and initiates volume switching when writing a file. The I/O switch must be open for output. This operation is equivalent to detection of the end of tape reflective strip. The info_ptr should be a null pointer.

HARDWARE_STATUS OPERATION

This operation returns a structure that contains the raw IOM status and the english language description of this status, generated by the last tape I/O operation. The I/O switch must be open. The returned structure is shown below and is defined in the include file mtape_hardware_status.incl.pl1. If info_ptr is nonnull, it must point to the mtape_hardware_status structure shown below. If info_ptr is given as null, then mtape_ will allocate the return structure in a temporary area on behalf of the user.

```
dcl  1 mtape_hardware_status aligned based (info_ptr),
       2 version char (8),
       2 description char (256) varying,
       2 pad bit (36),
       2 iom_status bit (72),
       2 iom_lpw bit (72);
```

ARGUMENTS

version
    is the current structure version. If info_ptr is nonnull on input, then the caller must set the version element to hwst_version_1. If null, mtape_will set the version number in the structure it allocates.

description
    is the English language description of this hardware status.

iom_status
: is the raw I/O status words returned from the last I/O operation. A definition for the format of these status words can be found in the include file iom_status.incl.pl1.

iom_lpw
: is the I/O List Pointer Word as it appeared at the termination of the last I/O operation. A definition of the format of the LPW can be found in the include file iom_lpw.incl.pl1.

RING_IN OPERATION

This operation will cause subsequent volume mounts to be requested with write rings installed. The I/O switch must be closed and the info_ptr set to null. The effect of this operation is to cause the current volume to be demounted and the write ring indicator to be set in the internal data base maintained by mtape_. At the time of the next file opening, the appropriate volume will be requested to be mounted with a write ring installed. If write rings have already been requested to be installed, either by the use of the -ring attach description argument, or by a previous invocation of the ring_in control operation, then the ring_in control operation is considered a "no-op" and has no effect.

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. The returned structure is defined in the include file mtape_volume_status.incl.pl1 as well as below. If info_ptr is nonnull, then it must point to the mtape_vst structure shown below. If the info_ptr is given as null, then mtape_ will allocate the structure in a temporary area on behalf of the user.

```
dcl  1 mtape_vst aligned based (info_ptr),
       2 version char (8),
       2 volume_type fixed bin,
       2 volume_name char (32),
       2 volume_id char (32),
       2 mounted bit (1),
       2 device_name char (8),
       2 volume_index fixed bin,
       2 mounts fixed bin,
       2 tot_error_stats,
         3 read,
           4 errors fixed bin (35),
           4 operations fixed bin (35),
         3 write,
           4 errors fixed bin (35),
           4 operations fixed bin (35),
         3 orders,
           4 errors fixed bin (35),
           4 operations fixed bin (35),
         3 successful_retry (7) fixed bin (35),
       2 rel_error_stats,
         3 read,
           4 errors fixed bin (35),
           4 operations fixed bin (35),
         3 write,
           4 errors fixed bin (35),
           4 operations fixed bin (35),
         3 orders,
           4 errors fixed bin (35),
           4 operations fixed bin (35),
         3 successful_retry (7) fixed bin (35);
```

ARGUMENTS

version
: is the current structure version. If info_ptr is nonnull on input, the caller must set the version element to vst_version_1. If null, then mtape_ will set the version element in the structure it allocates.

volume_type
: is the encoded volume type as determined by RCP, and could have one of the following values.

```
4 = IBM labeled volume
5 = ANSI labeled volume
6 = Unlabeled volume
```

volume_name
is the name of the current volume as specified in the volume sequence list (i.e. attach description).

volume_id
is the name of the current volume as recorded in the volume label. For unlabeled volumes, this field will be blank.

mounted
is a flag indicating the mounted state of the current volume. A "1"b indicates that the volume is mounted, "0"b indicates that the volume is not currently mounted.

device_name
is the name of the tape device that the current volume is mounted on (e.g. "tape_01"). If the volume is currently unmounted, this field will be blank.

volume_index
is the numerical order of this volume within the volume set.

mounts
is a count the number of times the current volume has been mounted during this attachment.

tot_error_stats
is a block representing the error statistics for the current volume inclusive of all of the mounts during the current attachment. The block includes the number of errors perportioned with the number of operations for read, write and order (i.e. non-data transfer operations, like forward space file), as well as a metering array of the number of times that read operations were successfully retried for each of the combinations of deskew window and threshold changes.

rel_error_stats
is the same as tot_error_stats above except it is the error statistics for the current mount only.

VOLUME_SET_STATUS OPERATION

This operation may be used to obtain information about the entire volume set as opposed to just the current volume. If nonnull, the info_ptr should point to a temporary segment which the mtape_ I/O module will fill with a structure as in a temporary area defined below. If the info_ptr is given as null, then mtape_ will allocate the structure for the user. This structure is also defined in the include file mtape_volume_status.incl.pl1.

```
dcl  1 mtape_vsst aligned based (info_ptr),
       2 version char (8),
       2 volume_type fixed bin,
       2 nvolumes fixed bin,
       2 vs_stat (mtape_vsst_nvolumes
                  refer (mtape_vsst.nvolumes)),
         3 volume_name char (32),
         3 volume_id char (32),
         3 mounted bit (1),
         3 device_name char (8),
         3 volume_index fixed bin,
         3 mounts fixed bin,
         3 tot_error_stats,
           4 read,
             5 errors fixed bin (35),
             5 operations fixed bin (35),
           4 write,
             5 errors fixed bin (35),
             5 operations fixed bin (35),
           4 orders,
             5 errors fixed bin (35),
             5 operations fixed bin (35),
           4 successful_retry (7) fixed bin (35),
         3 rel_error_stats,
           4 read,
             5 errors fixed bin (35),
             5 operations fixed bin (35),
           4 write,
             5 errors fixed bin (35),
             5 operations fixed bin (35),
           4 orders,
             5 errors fixed bin (35),
             5 operations fixed bin (35),
           4 successful_retry (7) fixed bin (35);
```

ARGUMENTS

version
: is the current structure version. If info_ptr is nonnull on input, the caller must set the version element to vsst_version_1. If null, then mtape_ will set the version element in the structure it allocates.

ARGUMENTS
: is the encoded volume type as determined by RCP, and could have one of the following values.

```
4 = IBM labeled volume
5 = ANSI labeled volume
6 = Unlabeled volume
```

nvolumes
: is the number of volumes in the volume set.

vs_stat
: is an array of structures of volume set members, which appears below in sequential order. If the user allocates this structure, then the variable mtape_vsst_nvolumes (declared in the mtape_volume_status include file) must be set to the maximum number of volumes in the volume set.

volume_name through rel_error_stats
: is the status of each volume in the volume set and are the same as the identical names described in the mtape_vst structure for the volume_status control operation above.

DETACH OPERATION

The I/O switch must be closed. Detachment is made through the iox_$detach entry which supports a character string "detach description" argument for supplying volume-set specific information for the disposition of the volume-set (See "Detach Description" below). The iox_$detach_iocb entry is supported in the sense that it will forward the call to the mtape_$detach entry, supplying a null detach description.

DETACH DESCRIPTION

The detach description is an ASCII character string argument to the iox_$detach entry and provides a means of specifying actions to be taken when detaching the current volume set.

In general, the detach description consists of zero, one or more control arguments.

CONTROL ARGUMENTS

-comment STR, -com STR
: allows the optional specification of a message to be displayed on the operators console at the time the volume set is to be detached. The comment text, STR, may be from 1 to 64 characters in length and must be quoted if it contains embedded white space.

-display, -ds
: specifies that the entire detach description, after it has been parsed and any necessary defaults added, will be displayed on the user_output I/O switch.

-unload
: specifies that any members of the volume set currently mounted are to be demounted at the time of detachment.

-rewind
: specifies that any members of the volume set currently mounted are to be rewound to load point at the time of detachment. This is the default in the absence of the -unload control argument.

MODES OPERATION

The mtape_ I/O module does not support the modes operation.

POSITION OPERATION

The mtape_ I/O module supports all appropriate positioning modes when the I/O switch is open for sequential_input.

READ LENGTH OPERATION

The I/O switch must be open for sequential_input.

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. Unlike previous tape I/O modules, non-mod 4 byte records may be written.

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, that is signalled by the command_query_ subroutine. Alternately, the answer command (described in the *Multics Commands and Active Functions Manual,* Order No. AG92) 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 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 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 file is locked and no further I/O may be performed until the file is closed and subsequently re-opened.

status_code = error_table_$unexpired_volume
This can occur only when the I/O switch is open for 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", the file is locked and no further I/O may be performed until the file is closed and subsequently re-opened.

status_code = error_table_$uninitialized_volume
A volume requires reinitialization or user verification before it can be used to perform any I/O. The I/O module distinguishes among two causes by setting command_question_info.query_code as follows:

query_code = 2
the first block of the tape is not a valid volume label for the volume type specified in the "-volume_type" attach description control argument. This query code can occur only if the I/O switch is opened for output.

query_code = 3
the volume identifier recorded in the volume label is incorrect. The volume identifier does not match the volume name.

If the I/O switch is opened for output, the user will be asked whether he wants to initialize or re-initialize the volume. If the response is "yes", the volume is reinitialized and processing continues. If the response is "no", the file is locked and no further I/O is possible without closing the I/O switch and subsequently re-opening. If the I/O switch is opened for input, the user will be asked whether he wants to continue processing in spite of the discrepancy.

status_code = error_table_$unexpired_file
This can occur only when the I/O switch is open for output. A file that must be extended, modified, regenerated, 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", the file is locked and no further I/O is possible without closing the I/O switch and subsequently re-opening.

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 output, the file is locked and no further I/O is possible without closing the I/O switch and subsequently re-opening. 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 may be a volume name, optionally followed by a mount message. Even if the volume name begins with a hyphen, it must *not* be preceded by the -volume control argument. If a mount message is to be specified, the response takes the following form:

```
volume_name -comment STR
```

ARGUMENTS

STR
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.

PER-FORMAT MODULES

In order to process a variety of different tape volume formats, the mtape_ I/O module employs standard subroutine interfaces to what are known as Per-Format modules. The generic name of each of these Per-Format modules or subroutines is <vol_type>_tape_io_, where <vol_type> represents the identified name of the volume format which is to be processed. For this release, two Per-Format modules have been supplied. They are:

ansi_tape_io_
For ANSI standard tape formats

ibm_tape_io_
For IBM standard labeled, unlabeled and DOS tape formats

See the ansi_tape_io_ module or the ibm_tape_io_ module for format-specific details of these two Per-Format modules.

In addition to the ANSI and IBM Per-Format modules, dummy versions of GCOS, Multics and RAW Per-Format modules have been provided as well. When one of these dummy Per-Format modules are selected, the Per-Format module initialization entry will display a message stating that the selected Per-Format module has not yet been fully implemented and then the attachment is aborted.

Selection of the appropriate Per-Format module to process the desired volume set is performed at attach time. If a -volume_type control argument was specified in the attach description, then the value of this control argument is used exclusively in the selection of the Per-Format module. If no -volume_type control argument was present in the attach description, then volume type information returned by RCP after a successful volume mount is used to select the appropriate Per-Format module. In the event that the mounted volume was unreadable or of an unrecognized format, the value of the -default_volume_type control argument is used to select a Per-Format module. The selected Per-Format module is then found in the storage system via the standard object search rules.

DATA BUFFERING AND CONTROL

For the purposes of this discussion, a data buffer is defined to be a contiguous area of storage, in the ioi_workspace, in which the data to be written to one tape block is stored for tape output or which will contain the data from 1 tape block on tape input. The size of the data buffer determines the maximum size of the tape block to be written or read. There are two types of data buffers used by mtape_ and its Per-Format modules. These are synchronous and asynchronous buffers.

Synchronous buffers are used to perform I/O on volume and file label records in a synchronous or one at a time manner. In the current implementation, only one of these buffers is allocated.

Asynchronous buffers are used to perform I/O on user data being written to or read from a tape volume. These buffers are allocated after the I/O switch has been opened, when it can be determined what the block size will be, either from information in the file header labels or from the open description. As many of these asynchronous buffers are allocated as will fit in the users maximum workspace size, up to a maximum number of 16. Asynchronous buffers are deallocated when the I/O switch is detached or, when processing multiple files, upon opening subsequent files. As a performance feature, if it is determined that the buffers for the new file are to be the same size as those that were previously allocated, the buffers are not deallocated. This removes the necessity of re-allocating them. The size of the ioi_workspace is dynamically changed when allocating or deallocating buffers.

As the name implies, asynchronous buffers are used in an asynchronous manner, due to the write-behind and read-ahead features of mtape_. When the mtape_ I/O routine is called to write a block of data, no I/O is actually queued unless half of the buffers are filled. At this time, all filled buffers are queued for writing. Upon the next call to write a block, the mtape_ I/O routine will queue up the new block immediately, thus allowing this latest buffer to be linked into the current I/O channel program in an effort to keep the tape device running at rated speed if data is available to write. When reading data, a read is queued for all available buffers, each time a block is requested to be read.

The reading and writing of data blocks (i.e. buffers), is completely demand driven from the Per-Format modules. Since the block format is Per-Format module dependent, and the unit of information passed in an iox_$read_record or iox_$write_record call is a logical record, which may or may not fill one or more blocks, only the Per-Format module in execution knows when a block is full or empty. The interface presented to the Per-Format modules by mtape_ is that of processing blocks of data in a synchronous manner. A pointer to the current buffer is maintained by the mtape_ I/O routine in a database which is common to both mtape_ and the Per-Format modules. The Per-Format module uses this pointer as if it was pointing to its only buffer. Each time a request is received by mtape_ to read or write a block, the current buffer pointer is incremented to the next buffer in the set, before control is returned to the Per-Format module. Physical file and block position counters are also maintained by mtape_ in the above mentioned common database. Whenever an exception condition presents itself which would cause the actual tape position to be different from that reflected in these counters (e.g. end of file condition, end of tape, etc.), the physical file and block position counters are re-synchronized before the exception condition is reported to the Per-Format module.

In summary, the Per-Format module performs logical record multiplexing, while mtape_ performs block multiplexing.

ARGUMENT PROCESSING

With the advent of, and the mtape_ I/O modules use of, the new I/O system entries, iox_$open_file, iox_$close_file, and iox_$detach, the task of argument processing has become much more complex. In addition to a standard I/O system attach description, open, close and detach descriptions must also be processed to extract open, close and detach time control information.

A new argument processing technology has been developed for mtape_ to reduce this complexity and offer a centralized and consistent means of performing the essential task of argument processing for mtape_ and all of its associated Per-Format modules.

In order to implement this new argument processing technology, a few simple rules were imposed. They are:

1. All "binary" control arguments (i.e. control arguments with no associated value) must have an antonym or inverse control argument to allow expressing the inverse state (e.g. -ring; -no_ring).

2. Arguments are processed from left to right and all control arguments may appear multiple times. The right most value (or state, if a binary control argument) takes precedence at the exclusion of any previous values of the same control argument. (e.g. processing an attach description that includes "-den 1600 -ring -den 6250 -no_ring" would yield "-den 6250 -no_ring").

3. With the exception of the allowable values of the -volume attach description control argument, any group of characters preceded by a "-" is assumed to be a control argument.

4. With the exception of the volume names in the volume sequence list of an attach description, any group of characters that is not preceded by a "-" is assumed to be a value of the control argument last processed.

DEFAULT VALUES

As an ease of use feature, all control arguments and associated values that a user may specify in an attach, open, close or detach description, are supplied with reasonable default values by mtape_ and its associated Per-Format modules. For each description type, the supplied default values are stored in the data space of a standard value segment as an ASCII string. These strings are collectively known as "default_linear_forms", and have unique value names so that they can be readily associated with a particular description type.

The default_linear_forms are found on the system, using the standard search_path mechanism, via the mtape_arguments search list.

The default values are processed at the same time each description type is processed. Due to the left to right argument exclusion rule mentioned above in "Argument Processing", the insertion of default values is a simple matter. Whenever a particular description type is to be processed, the appropriate default_linear_form is found and processed before any user supplied description. After the default_linear_form has been processed, the user supplied description values are processed "on top" of the default_linear_form. The result is a combination of default as well as user supplied values, which make up a complete and unambiguous attach, open, close or detach description.

Although the mtape_ default values were picked to be reasonable and to fit most situations, it is well recognized that they may not be suitable for all tape processing needs at all sites or by all individuals. Therefore, a mechanism has been provided which will allow a site or an individual user to tailor their own default values to fit their particular needs. This user settable default mechanism is embodied in three commands. The mtape_set_defaults, mtape_get_defaults and mtape_delete_defaults commands in concert will allow minipulation of default values. See the *Multics Commands and Active Functions Manual,* Order No. AG92 for a description of these commands.

Name: rbf_

The rbf_ I/O module performs record oriented I/O to a remote I/O terminal that has the characteristics of the Honeywell Level 6M Satellite remote batch facility operating over an X.25 connection. The hardware options currently supported are defined by the control arguments described below.

Entry points in this module are not called directly by users; rather, the module is accessed through the I/O system.

ATTACH DESCRIPTION

```
rbf_ -control_args
```

CONTROL ARGUMENTS
are optional with the exception of -device, -comm, and -tty. All other control arguments are passed through as part of the attach description for the communications I/O module specified via -comm.

-ascii
: uses the ASCII character set. This is the default. This argument is accepted for compatibility with other terminal I/O modules.

-comm STR
: uses the communications I/O module specified by STR where STR must be "tty_".

-device STR
: attaches the subdevice specified by STR. STR can be printer, punch, reader or teleprinter.

-physical_line_length N, -pll N
: specifies the physical line length, N, of the output device. This argument is accepted for compatibility with other terminal I/O modules.

-terminal_type STR, -ttp STR
: specifies the terminal type whose translation tables defined in the user or system terminal type table (TTT) are used to translate input and output to and from the device. If not specified, no translation is performed. Input and output translation tables can be up to 256 characters in length.

-tty STR
: connects the remote I/O terminal to the logical communications channel named STR.

OPEN OPERATION

The rbf_ I/O module supports the sequential_input, sequential_output, and sequential_input_output opening modes.

WRITE RECORD OPERATION

The write_record entry performs the appropriate translation on the data record, converts the supplied slew control into the proper carriage control sequences for line printer attachments and performs data compression. The records are then transmitted to the specified communications channel.

The format of the record supplied to this I/O module follows. This structure and the referenced constants are contained in terminal_io_record.incl.pl1:

```
dcl 1 terminal_io_record aligned based,
      2 version          fixed bin,
      2 device_type      fixed bin,
      2 slew_control,
        3 slew_type      fixed bin (18) unaligned unsigned,
        3 slew_count     fixed bin (18) unaligned unsigned,
      flags,
        3 binary         bit(1) unaligned,
        3 preslew        bit(1) unaligned,
        3 pad            bit(34) unaligned,
      2 element_size     fixed bin,
      2 n_elements       fixed bin (24),
      2 data,
        3 bits (terminal_io_record_n_elements refer
          (terminal_io_record.n_elements))
                         bit (terminal_io_record_element_size refer
                           (terminal_io_record.element_size)) unaligned;
```

STRUCTURE ELEMENTS

version
: is the current version of this structure. This version of the structure is given by the value of the named constant terminal_io_record_version_1.

device_type
: is the type of device to which this record is to be written. The acceptable values are TELEPRINTER_DEVICE, PRINTER_DEVICE, or PUNCH_DEVICE.

slew_control
: need only be supplied by the caller if device_type is PRINTER_DEVICE and specifies the slew operation to be performed after printing the data in the record.

    slew_type
    : specifies the type of slew operation. The possible values are SLEW_BY_COUNT, SLEW_TO_TOP_OF_PAGE, SLEW_TO_INSIDE_PAGE, SLEW_TO_OUTSIDE_PAGE, or SLEW_TO_CHANNEL.

    slew_count
    : is interpreted according to the value of slew_control.slew_type.

flags.binary
: must be set to "0"b. (This I/O module does not support binary data transmission.)

flags.preslew
: must be set to "0"b. (This I/O module does not support slew operations before printing the record's data.)

element_size
: must be set to 9. (This I/O module only supports transmission of characters.)

n_elements
    is the number of characters in the record to be written.

data.bits
    is the actual data.

READ RECORD OPERATION

The read_record entry reads characters from the communications channel and returns a single record from the device, basically performing the inverse of the functions described for the write_record operation.

The format of the record this I/O module returns in the supplied buffer follows. This structure and the referenced constants are contained in terminal_io_record.incl.pl1:

```
dcl 1 terminal_io_record aligned based,
      2 version           fixed bin,
      2 device_type       fixed bin,
      2 slew_control,
        3 slew_type       fixed bin (18) unaligned unsigned,
        3 slew_count      fixed bin (18) unaligned unsigned,
      flags,
        3 binary          bit(1) unaligned,
        3 preslew         bit(1) unaligned,
        3 pad             bit(34) unaligned,
      2 element_size      fixed bin,
      2 n_elements        fixed bin (24),
      2 data,
        3 bits (terminal_io_record_n_elements refer
          (terminal_io_record.n_elements))
                          bit (terminal_io_record_element_size refer
                            (terminal_io_record.element_size)) unaligned;
```

STRUCTURE ELEMENTS

version
    is the current version of this structure. This version of the structure is given by the value of the named constant terminal_io_record_version_1.

device_type
    is the type of device from which this record is read. Its possible values are TELEPRINTER_DEVICE or READER_DEVICE.

slew_control.slew_type
    is always set to SLEW_BY_COUNT.

slew_control.slew_count
    is always set to 1.

flags.binary
    must be set to "0"b.

flags.preslew
    must be set to "0"b.

element_size
    must be set to 9.

n_elements
    is set to the number of characters returned in the record.

data.bits
    is the actual returned data.

CONTROL OPERATION

This I/O module supports all the control operations supported by the tty_ I/O module. In addition, it supports the following:

runout
    transmits any data stored in the output buffer. There is no input structure.

end_write_mode
    prevents rbf_ from returning until all outstanding output has been written to the attached channel. There is no input structure.

MODES OPERATION

This I/O module supports the rawi, rawo, and 8bit modes.

NOTES

The select_device, reset, and binary_punch control orders are ignored, but are accepted for compatibility with other I/O modules.

Name: rdisk_

The rdisk_ I/O module supports I/O from/to disk packs. Sequential and indexed file types are supported.

Entries in this module are not called directly by users; rather, the module is accessed through the I/O system. For a general description of the I/O system and a discussion of files, see the Programmer's Reference Manual.

All errors encountered by rdisk_ are reported via the sub_err_ subroutine with the "flags" variable set to ACTION_DEFAULT_RESTART. The "info_ptr" variable passed to sub_err_ is set to null.

ATTACH DESCRIPTION

```
rdisk_ device_id pack_id {-control_args}
```

ARGUMENTS

device_id
    is a character string identifying the type number of the required disk device. The supported disk devices are listed in the table below, along with the character string to use for device_id:

```
device_id           Device Type
---------           -----------

d181                DSU181
d190                DSU190
d191 or d400        DSU190/MSU0400 with the high-efficiency
                              format (40 sectors/track)
3380                MSU3380
3381                MSU3381
d451                MSU0451
d500                MSU0500
d501                MSU0501
```

pack_id
    is a character string identifying the disk pack to be mounted.

CONTROL ARGUMENTS

-device, -dv DEVICE_NAME
: indicates what disk drive DEVICE_NAME is to be attached. This is useful when attaching to drives that do not have removable media. DEVICE_NAME has the form of:

  dskX_NN{S}

  where:

  X
  : is the subsystem name.

  NN
  : is the device number.

  S
  : is the subvolume name. Only valid for 3380 and 3381 device_ids. Valid subvolume names for 3380 are a and b, for the 3381 a, b and c.

-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.)

-sys
: indicates that the attachment is being made by a system process and that a disk drive reserved for system functions is to be assigned.

-write
: indicates that the disk pack may be written on. If omitted, the operator is instructed to mount the pack with the PROTECT button pressed so that writing is inhibited.

NOTES

The attachment causes the specified disk pack to be mounted on a drive of the specified type.

When the -device option is given with a subvolume, rdisk_ will perform the address conversions in the same manner as the file system IO. More that one subvolume of a device may be attached by a process. The device may only be attached to one process at any one time. If the subvolume name is not given for a device that supports subvolumes, then rdisk_ will not convert the addresses and the entire device may be accessed.

This page intentionally left blank.

OPENING

The following opening modes are supported:

sequential_input
sequential_output
sequential_update
direct_input
direct_update

Notice that if the opening mode is of the output or update type, the attach description must include the -write control argument so that the operator does not press the PROTECT button when the pack is mounted.

POSITION OPERATION

This operation is supported for only the sequential_input and sequential_update opening modes. The type and quantity values are interpreted as follows:

TYPE	QUANTITY	ACTION
-1	--	position to the beginning of the disk pack.
+1	--	position to the end of the disk pack.
0	N	skip N sectors (forward if N > 0; backward if N < 0).
2	N	position to sector N.

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.) This operation is not supported for the sequential_output opening mode.

REWRITE RECORD OPERATION

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 spaces in sequential modes and binary zeros in direct modes. A code of 0 is returned in this case. (See "Notes" below.) This operation is supported for only the update opening modes.

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.) This operation is supported for only the direct opening modes.

LIST OF CONTROL ORDERS

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 varying character string (maximum of 32 characters) containing the identifier of the pack to be mounted. This operation is not allowed for MSU0500 or MSU0501 devices.

device_info
: causes information pertaining to the attached disk device to be returned to the user. The info_ptr should point to a structure of the following form:

```
dcl 1 device_info_table    aligned,
      2 subsystem_name     char (4),
      2 device_name        char (8),
      2 sect_per_dev       fixed bin (35),
      2 cyl_per_dev        fixed bin,
      2 sect_per_cyl       fixed bin,
      2 sect_per_track     fixed bin,
      2 num_label_sect     fixed bin,
      2 num_alt_sect       fixed bin,
      2 sect_size          fixed bin (12);
```

where:

subsystem_name
: is the name of the disk subsystem in use (i.e., D191).

device_name
: is the name of the disk device in use (i.e., dska_04).

sect_per_dev
: is the total number of non-T&D sectors on the disk pack.

cyl_per_dev
: is the total number of non-T&D cylinders on the disk pack.

sect_per_cyl
: is the number of data sectors on each cylinder of a disk pack.

sect_per_track
: is the number of data sectors on each track.

num_label_sect
: is the number of data sectors to reserve for label information.

num_alt_sect
: is the number of data sectors to reserve for alternate track area.

sect_size
: is the number of 36-bit words in each data sector.

format_trk
: causes a format track command to be issued to the track that was indicated by a preceding seek_key operation. This operation is not allowed for MUS0500 or MSU0501 devices. The info_ptr should point to a user supplied structure of the following form:

```
dcl 1 format_trk_info   aligned,
     (2 hz              bit (2),
      2 ti              bit (2),
      2 adcyl           fixed bin (16),
      2 adhd            fixed bin (16)) unaligned;
```

where:

hz
: is a bit pattern indicating the state of the header bypass switch. The hz bits are defined as follows:

h z	bit pattern meaning.
0 0	format home address and all data records.
0 1	verify home address and record one, format home address and all data records.
1 0	skip home address, format all data records.
1 1	verify home address and data record one, skip home address and format all data records.

ti
: is a bit pattern indicating the state of the track indicator bits. The ti bits are defined as follows:

t i	bit pattern meaning.
0 0	format track good.
0 1	format track alternate.
1 0	format track defective with alternate track assigned.
1 1	format track defective with no alternate track assigned.

adcyl, adhd
: are the alternate or defective cylinder and head numbers used when the track indicator bits equal "01"b or "10"b. These two fields are defined as follows:

1. If the track indicator bits are set to "01"b (alternate track), then adcyl and adhd should be equal to the defective cylinder and head number for which the alternate track is being formatted.

2. If the track indicator bits are set to "10"b (defective with alternate assigned), then adcyl and adhd should be equal to the cylinder and head number of the alternate track.

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);
```

rd_trk_header
causes a read track header command to be issued to the track that was indicated by a preceding seek_key operation. This operation is not allowed for MUS0500 or MSU0501 devices. The raw track header information is passed to the user in a structure (pointed to by info_ptr) of the following form:

```
dcl 1 trk_header_info   aligned,
     (2 ha_cyl          bit (16),
      2 ha_head         bit (16),
      2 pad1            bit (2),
      2 ha_ti           bit (2),
      2 pad2            bit (10),
      2 rcd_0_ti        bit (2),
      2 rcd_0_cyl       bit (16),
      2 rcd_0_head      bit (16),
      2 rcd_0_rn        bit (8),
      2 pad3            bit (24),
      2 rcd_0_data (8), bit (8),
      2 pad4            bit (4)) unaligned;
```

where:

ha_cyl
is the cylinder number read from the track home address.

ha_head
is the head number read from the track home address.

ha_ti
is the track indicator bits (defined above in the format_trk order) read from the track home address.

rcd_0_ti
: is the track indicator bits read from record zero. If the ha_ti bits indicate "10"b, then rcd_0_ti should equal "01"b for alternate track. If ha_ti indicates "01"b, then rcd_0_ti should equal "10"b for defective track. Otherwise rcd_0_ti will equal ha_ti.

rcd_0_cy, rcd_0_head
: are the cylinder and head number read from record zero. If ha_ti indicates "10"b, then rcd_0_cyl and rcd_0_head equal the cylinder and head number of the alternate track. If ha_ti indicates "01"b, then rcd_0_cyl and rcd_0_head contain the cylinder and head number of the defective track. Otherwise, rcd_0_cyl and rcd_0_head equal ha_cyl and ha_head.

rcd_0_rn
: is the record number for record zero (normally equal to zero).

rcd_0_data
: is the eight data bytes in record zero (not a normal data record) and is normally equal to zero.

padn
: are unused bits that are returned as "0"b.

tsize
: 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 or position operation is to treat any key or position within this area as an invalid key. (The default is on.)

raw, ^raw
: specifies that the entire disk pack is available to the user, including the T&D cylinder (the last cylinder on the disk pack). (The default is off.)

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 or position operation is to treat any key within that area as an invalid key. (The default is off.) This mode cannot be enabled for a MSU0500 or MSU0501 disk.

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

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 spaces. A code of 0 is returned in this case. (See "Notes" below.) This operation is supported for only the sequential_output opening mode. A series of writes will write successive records.

CLOSING

The closing has no effect on the physical device. For the sequential_output opening mode, the effect is as if an end-of-file flag is placed just beyond the end of the available disk area.

DETACHING

The detachment causes the disk pack to be detached from the users process.

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, access to the IOI gates, and access to the "acs" segment (e.g., >sc1>rcp>dskb_18.acs) that is used by the site to control access to the disk drive.

For input and update opening modes, the file occupies the entire available disk area (see the getbounds control order). For the sequential_output opening mode, the file is considered to be empty. That is, an open followed by a write records data in the first sector of the available disk area.

For direct opening modes, the entire disk pack is treated as an indexed 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.

For the sequential_input and sequential_update opening modes, if an attempt is made to read beyond the end of the user-accessible area, the code error_table_$end_of_info is returned. For all other opening modes, if an attempt is made to read or write beyond the end of the user-accessible area on disk, the code error_table_$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 write and rewrite operations, unless overridden by a -size control argument in the attach description, or by a setsize control order.

The following items must be considered when using this I/O module with language input/output:

DEVICE ATTACHMENT AND FILE OPENING

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. After opening, the desired modes should be set and the current sector bounds should be obtained through direct calls to pl1_io_$get_iocb_ptr, iox_$modes, and iox_$control.

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 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.

INPUT

PL/I: The input record length (buff_len) is determined by the size of the variable specified in the into option.

For the sequential_input and sequential_update opening modes, use the PL/I read statement with the into option to read data. Use the ignore option to skip forward within the file. An open statement followed by a read statement will read in the first record. Successive reads will obtain successive records.

For the direct_input opening mode, use the PL/I read statement with the into and key options. The set option should not be used. The key should be a character string containing the character representation of the desired sector number.

The PL/I get statement can be used with the sequential_input opening mode if the record_stream_ I/O module is referenced in the attach description of the open statement.

FORTRAN: 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.

For the sequential_input opening mode, use the unformatted sequential read statement.

For the direct_input opening mode, use the unformatted keyed version of the FORTRAN read statement. The key must be an integer, whose value is the desired sector number.

OUTPUT

PL/I: The size of the variable referenced in the from option determines the length of the record written to disk.

For the sequential_output opening mode, use the write statement with the from option. An open statement followed by a write statement will start writing at the beginning of the available area on the disk pack.

For the sequential_update opening mode, use the rewrite statement with the from option. A previous read statement must have been used to designate which record will be updated.

For the direct_update opening mode, use the rewrite statement with the from and key options. The key should be a character string containing the character representation of the desired sector number.

The PL/I put statement can be used with the sequential_output opening mode if the record_stream_ I/O module is referenced in the attach description of the open statement.

FORTRAN: The size of the output record is determined by the amount of data specified in the write list.

For the sequential_output opening mode, use the unformatted sequential write version of the FORTRAN write statement.

For the direct_update opening mode, use the unformatted keyed version of the write statement. The key should be a character string containing the character representation of 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:

switch
  is the name of the I/O switch.

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_

The record_stream_ 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

```
record_stream_ {switch_name} {-control_args}
```

ARGUMENTS

switch_name
  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.

CONTROL ARGUMENTS

The following control the transformation of records into a stream of bytes and vice versa, or control the target attachment:

-nnl
: transforms a record into a stream without appending a newline character.

-length N
: converts the stream of bytes 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 switch_name 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. The implications of the opening mode are as follows:

stream_input
: The target I/O switch must be either 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 either 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 either 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 either 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 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 argument may be one of the following:

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_ -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: remote_input_

The remote_input_ I/O module performs record input from a terminal I/O module, which is assumed to be connected to a remote I/O device, such as a Honeywell Level 6 remote batch facility (G115 type), an IBM 2780, or an IBM 3780. Except for hardware restrictions, this module performs some code conversion and control in such a way that remote and local card reading are the same.

Entry points in this module are not called directly by users; rather, the module is accessed through the I/O system.

This module in turn constructs an attach description for the module specified in the -terminal control argument, passing the other attach information specified by the caller.

ATTACH DESCRIPTION

```
remote_input_ -control_args
```

CONTROL ARGUMENTS

-device STR
: STR defines the device type that this I/O module is attempting to simulate. The acceptable values for STR are reader, printer_in, and punch_in. This control argument is optional. If not supplied, a device type of reader is assumed.

-physical_line_length N, -pll N
: This control argument is accepted and ignored for compatibility with other device-level I/O modules. It is not passed on to the terminal I/O module.

-record_len N
: defines the maximum record length (buffer size) for data from the terminal I/O module in characters. The accepted ranges are 80 to 160 for the device type of reader, and 10 to 1024 otherwise. If this control argument is not given, the maximum for the device type is assumed.

-runout_spacing N, -runsp N
This control argument is accepted and ignored for compatibility with other device-level I/O modules. It is not passed on to the terminal I/O module.

-terminal STR
STR specifies the terminal I/O module to be attached by this device I/O module. (Required)

All other attach control arguments are assumed to belong to the terminal I/O module. These are passed on as part of its attach description. The -device option passed on to the terminal I/O module specifies one of the following devices: reader, printer, or punch. See the description of the terminal I/O module for a full definition of required and optional control arguments.

OPEN OPERATION

The remote input I/O module supports the stream_input opening mode. The terminal I/O module switch is in turn opened with the sequential_input or stream_input modes.

GET CHARS OPERATION

The get_chars entry reads one record from the terminal I/O module and returns up to the number of specified characters. If the number of characters in the record is greater than the requested number, error_table_$data_loss is returned along with the data.

CONTROL OPERATION

The remote_input_ device I/O module supports the following control operations:

get_count
returns the current record count. This is the count of records read from the terminal I/O module since the last reset control operation. This operation is not passed on to the terminal I/O module.

The info_pointer must point to the following structure. (This structure is taken from the counts structure in prt_order_info.incl.pl1 for compatibility with procedures that use several device I/O modules.)

```
dcl 1 counts aligned based,
      2 prt_data_pad (4) fixed bin,
      2 record_count fixed bin (35),
      2 prt_pad fixed bin;
```

The variable record_count will contain the returned value. This corresponds with the variable line_count from the other structure.

reset
sets the current record count to 0 and passes the control operation on to the terminal I/O module.

All other control operations are passed on to the terminal I/O module.

MODES OPERATION

This I/O module supports the modes defined by the terminal I/O module specified in the attach description.

Name: remote_printer_

The remote_printer_ I/O module presents a stream I/O interface to the caller and performs record output to a printer, which is assumed to be part of a remote I/O device, such as a Honeywell Level 6 remote batch facility (G115 type), an IBM 2780, or an IBM 3780. Except for hardware restrictions, this module performs all the necessary code conversion and control in such a way that remote and local printing are the same.

Entry points in this module are not called directly by users; rather, the module is accessed through the I/O system.

This module in turn constructs an attach description for the module specified in the -terminal control argument, passing the attach information for horizontal tabbing, physical line length, and all other attach information specified by the caller.

ATTACH DESCRIPTION

```
remote_printer_ -control_args
```

CONTROL ARGUMENTS

The following control arguments are optional, with the exception of -terminal:

-horizontal_tab, -htab
printer has a horizontal tab feature. The default is no tab control.

-physical_line_length N, -pll N
printer has a maximum line width of N characters. The default is 132 characters.

-physical_page_length N, -ppl N
printer has a maximum line count per page of N. The default is 66 lines.

-terminal STR
uses the terminal I/O module specified by STR. This control argument is required.

OPEN OPERATION

The remote printer I/O module supports the stream_output opening mode.

PUT CHARS OPERATION

The put_chars entry converts a character string delimited by a newline character to an image suitable for printing and transmits this image to the terminal I/O module. This operation is repeated until all the characters specified by the caller have been transmitted.

CONTROL OPERATION

This I/O module supports all the control operations supported by the terminal I/O module specified in the attach description. In addition, it supports the following control orders:

channel_stops
: sets the channel stop data used for slew to channel control sequences during a put_chars operation. The info pointer defines the channel_stops input array as found in the prt_order_info include file. Array element N defines the stops for line number N. Bit M of an array element defines a stop for channel M. The initial value is no stops defined. Once defined, the stops remain in effect until the next channel_stops control operation.

end_of_page
: advances the paper to the bottom of the current page, one line below the point where page labels are printed. If page labels are set the label is printed. The info pointer is not used and may be null.

get_count
: returns accounting information. The info pointer defines the counts output structure as found in the prt_order_info include file. The page and line counts are reset by the reset control operation.

get_error_count
: returns the error count since the output module was attached. The info pointer defines the output variable ret_error_count as found in the prt_order_info include file.

get_position
: returns the position data defined by the position_data structure in the prt_order_info include file. The data resembles that of the get_count control operation, but the structure adds the total characters printed since the last reset to allow the caller to start the next put_chars operation at the following character when the module returns due to 1pg or stopN mode. The data structure is also used for the set_position operation (see below).

inside_page
: advances the paper to the formfeed position of the next inside page. An inside page is a top page when the listing is folded correctly. Separator bars for the head sheet are printed over the perforations at the bottom of an inside page. The info pointer is not used and may be null.

outside_page
: advances the paper to the formfeed position of the next outside page. An outside page is a bottom page when the listing is folded correctly. The info pointer is not used and may be null.

page_labels
: sets the top and bottom page labels to be printed for each logical page. The info pointer may be null to reset page labels to blank. Otherwise, the info pointer defines the page_labels input structure as found in the prt_order_info include file.

paper_info
: sets the physical characteristics of the paper in the printer. The info pointer defines the paper_info input structure as found in the prt_order_info include file. Once set, the paper_info remains in effect until the next paper_info control operation. If the printer has a software loadable VFC image, a new image is loaded and the printer placed out of synchronization for the operator to align the paper. Otherwise, the code error_table_$no_operation is returned so the caller can request the operator to load the appropriate VFU tape and set the required lines per inch switch to complete the operation. The defaults are: page length, 66; line length, 136; lines per inch, 6.

reset
: resets the output module to its default state: default modes, no page labels, line count = 0, page count = 1, and total chars = 0. The info pointer is not used and may be null.

resetwrite
: cancels any data buffered for output. It is used to clear the output module after an error so the paper can be resynchronized. The info pointer is not used and may be null.

runout
: causes all buffered data to be output before returning to the caller. It is used to synchronize the program with the actual device. The info pointer is not used and may be null.

set_position
: sets the internal counters in the output module. The info pointer defines the position_data input structure as found in the prt_order_info include file. This is the reverse of the get_position control operation. It is used to start the accounting data at the correct point when restarting an I/O daemon request in the middle.

MODES OPERATION

This I/O module supports all the modes supported by the terminal I/O module specified in the attach description. In addition, it supports the following modes:

lpg, ^lpg
: causes the output module to return to the caller when the end of the current page is reached (i.e., at the formfeed position for the next logical page). If there are unprocessed characters at this point, the code error_table_$request_pending is returned. The default is ^lpg.

ctl_char, ^ctl_char
: causes the output module to pass nonprinting characters to the device as is. Carriage movement characters (newline, formfeed, carriage return, backspace, and horizontal and vertical tab) are interpreted normally. The ASCII escape character (octal 033) is also transmitted directly, unless esc mode is enabled. If ctl_char mode is disabled, the treatment of nonprinting characters is determined by the setting of non_edited mode. The default is ^ctl_char.

esc, ^esc
: enables searching for escape sequences in the input string, which enables slew to channel orders. The default is ^esc.

non_edited, ^non_edited
: causes the output module to print the applicable octal ASCII code preceded by a backslash (\) for nonprinting characters, and to use the nonedited output conversion table in the specified TTT for the remote device. The ^non_edited value causes any such characters to be omitted from the output. The setting of this mode is ignored when ctl_char is in effect. The default is ^non_edited.

noskip, ^noskip
: suppresses the automatic insertion of blank lines at the end of a logical page (i.e., it allows the printer to print over the perforations). It has the side effect of setting the logical page length to its default value. The default is ^noskip.

print, ^print
: specifies that processed characters from the input string are to be printed. The ^print value allows a string to be processed for output, sets page and line counts, and honors the lpg and stopN modes, but without actually printing the processed characters. The default is print.

single, ^single
: specifies that any formfeed or vertical tab characters from the input string are to be converted to newline characters (i.e., it suppresses runaway paper feeding). The default is ^single.

truncate, ^truncate
: truncates the output if the line exceeds the line length. The ^truncate value allows the line to be wrapped onto the next line if it is too long. The default is ^truncate.

plN
: sets the logical page length to N lines. At the end of a logical page, the printer skips to the next formfeed position (unless noskip mode is set). The value of N must be greater than one, and can be greater than a physical page. The default value is physical page length minus lines per inch.

llN
: sets the logical line length to N characters. The value of N must be greater than the indentation (see below) and must not be greater than the physical line length of the device. The default value is the physical line length.

inN
: sets the indentation to N characters. The value of N must be 0 or a positive integer which is less than the logical line length. The default value is 0.

stopN
: sets the output module to return to the caller every N pages even though the processing of the input string has not been completed. If there is unprocessed input remaining, a code of error_table_$request_pending is returned. A value of 0 means do not return until all input is processed. The counter of how many pages to process before returning is reset when a new value is given. The default value is 0.

default
: causes all of the above modes to be reset to their default values. This mode is also passed to the terminal I/O module for processing.

POSITION OPERATION

This I/O module supports all the position operations supported by the terminal I/O module specified in the attach description.

Name: remote_punch_

The remote_punch_ I/O module presents a stream I/O interface to the caller and performs record output to a card punch, which is assumed to be part of a remote I/O device, such as a Honeywell Level 6 remote batch facility (G115 type), an IBM 2780, or an IBM 3780. Except for hardware restrictions, this module performs all the necessary code conversion and control in such a way that remote and local card punching are the same.

Entry points in this module are not called directly by users; rather, the module is accessed through the I/O system.

This module in turn constructs an attach description for the module specified in the -terminal control argument, passing the other attach information specified by the caller.

ATTACH DESCRIPTION

```
remote_punch_ -control_arg
```

CONTROL ARGUMENTS

-card_ll N
    specifies the length of records (cards) supported by the terminal I/O module. (Default is 80.)

-device STR
    defines the type of device to be simulated by this I/O module and can be either "punch" or "reader_simulator". This specification is passed to the terminal I/O module as "-device punch" or "-device reader", respectively. (Default is "punch".)

-horizontal_tab, -htab
    specifies that the device supports the horizontal tab character. (Default is the use of the appropriate number of spaces.)

-non_edited
    specifies that nonprinting characters can be passed directly to the terminal I/O module. (Default is that these characters are not passed.)

-runout_spacing N, -rnsp N
-physical_page_length N, -ppl N
    are accepted and ignored for compatibility with other device I/O modules.

-terminal STR
    STR specifies the terminal I/O module to be attached to this device I/O module. (Required)

All other attach arguments are passed directly to the terminal I/O module.

OPEN OPERATION

The remote punch I/O module supports the stream_output opening mode.

PUT CHARS OPERATION

The put_chars entry splits the data to be written into records of the size given by -card_ll and transmits these records to the terminal I/O module. This operation is repeated until all the characters specified by the caller have been transmitted.

CONTROL OPERATION

The remote_punch device I/O module supports the following control operations:

binary_punch
: requests that all subsequent data be punched in binary (rather than RMCC) if supported by the terminal I/O module. This control order is then passed on to the terminal I/O module.

get_count
: returns the current record count, which is the number of records written to the terminal I/O module since the last reset control operation. This operation is not passed on to the terminal I/O module. The info_ptr must point to the following PL/1 structure. (This structure is taken from the counts structure in prt_order_info.incl.pl1 for compatibility with procedures that use several device I/O modules.)

```
dcl 1 counts aligned based,
      2 prt_data_pad (4) fixed bin,
      2 record_count fixed bin (35),
      2 prt_pad fixed bin;
```

The variable record_count will contain the returned value. This corresponds with the variable line_count from the other structure.

reset
: sets the current record count to zero, returns to punching in RMCC (remote Multics card code), and passes the order to the terminal I/O module.

All other control operations are passed directly to the terminal I/O module for processing.

MODES OPERATION

This I/O module supports the RMCC output card mode defined in the Programmer's Reference Manual. It also supports the two modes non_edited and default, which enable and disable edited output conversion, if output conversion has been enabled by the terminal I/O module.

POSITION OPERATION

This I/O module supports all the position operations supported by the terminal I/O module specified in the attach description.

Name: remote_teleprinter_

The remote_teleprinter_ I/O module presents a stream I/O interface to the caller and performs record I/O to a terminal or printer, which is assumed to be part of a remote I/O device, such as a Honeywell Level 6 remote batch facility (G115 type), an IBM 2780, or an IBM 3780.

Entry points in this module are not called directly by users; rather, the module is accessed through the I/O system.

This module in turn constructs an attach description for the module specified in the -terminal control argument, passing the attach information for ASCII or EBCDIC, horizontal tabbing, physical line length, and all other attach information specified by the caller.

ATTACH DESCRIPTION

```
remote_teleprinter_ -control_args
```

CONTROL ARGUMENTS

The following control arguments are optional, with the exception of -terminal:

-horizontal_tab, -htab
: output device has a horizontal tab feature. The default is no tab control.

-physical_line_length N, -pll N
: output device has a maximum line width of N characters. The default is 80 characters.

-physical_page_length N, -ppl N
: output device has a maximum line count per page of N. The default is 66 lines.

-runout_spacing N, -runsp N
: outputs N newline characters with each runout operation. This allows the operator to see messages still under the printer mechanism for terminals that have only a printer as an output device. The default is 0.

-terminal STR
: uses the terminal I/O module specified by STR. This control_arg is required.

OPEN OPERATION

The remote_teleprinter_ I/O module supports the stream_input_output opening mode.

PUT CHARS OPERATION

The put_chars entry converts a character string ending in a newline character to an image suitable for printing and transmits this image to the terminal I/O module.

GET CHARS OPERATION

The get_chars entry reads the number of specified characters from the terminal I/O module.

GET LINE OPERATION

The get_line entry reads one record from the terminal I/O module, appends a new line, and returns as many characters as requested by the caller, or the whole record if it is shorter. If the record is longer than requested, error_table_$data_loss is returned.

CONTROL OPERATION

This I/O module supports all the control operations supported by the terminal I/O module specified in the attach description. In addition, it supports all the control operations supported by the I/O module remote_printer_.

MODES OPERATION

This I/O module supports all the modes supported by the terminal I/O module specified in the attach description. In addition, it supports all the modes supported by the I/O module remote_ printer_.

POSITION OPERATION

This I/O module supports all the position operations supported by the terminal I/O module specified in the attach description.

Name: signal_io_

The signal_io_ I/O module signals a condition whenever an iox_ I/O operation is performed. The condition has an info structure that allows a handler of the condition to either abort the operation or complete it by setting values in the structure and restarting the condition signal. When the condition is restarted, the signal_io_ I/O module returns control to the caller of iox_ and returns the output data in the structure as corresponding parameters of the iox_ call.

Applications using this I/O module must have a handler on the stack at all times to handle the signal_io_ condition.

ATTACH DESCRIPTION

```
signal_io_
```

OPEN OPERATION

All opening modes are supported.

I/O OPERATIONS (get_chars, get_line, put_chars, read_record, rewrite_record, delete_record, read_length, position, seek_key, read_key, write_record, control, modes)

All operations are supported in appropriate opening modes. See NOTES for a discussion of handing the condition associated with these operations.

NOTES

When this module is called through iox_ to perform an I/O operation as listed above, it signals the "signal_io_" condition with an info structure given here. The condition is restartable.

Applications using this module must establish a handler for the condition that calls find_condition_info_ to locate the info structure. If the condition is not handled, the default_error_handler_ will print a default error message, unless the condition is associated with user_i/o, user_output, user_input or error_output. For these I/O switches, terminates the process.

The returned_error_code in signal_io_info is initially set to error_table_$action_not_performed, so if the condition is restarted without first having the structure filled in, the iox_ call will return error_table_$action_not_performed.

This condition does NOT pass through the condition walls established when for new command levels. If the application is attaching, for example, user_i/o via this module, it must establish a command level intermediary procedure (via cu_$set_cl_intermediary) that establishes a new handler for the signal_io_ condition before calling the standard intermediary (located via cu_$get_cl_intermediary).

For example:

```
        on signal_io_ call SIGNAL_IO_HANDLER;
        call cu_$get_cl_intermediary (std_cl_intermediary);
        call cu_$set_cl_intermediary (NEW_COMMAND_LEVEL);

        {attach/open switch, do work}

        revert signal_io_;
        call cu_$set_cl_intermediary (std_cl_intermediary);

NEW_COMMAND_LEVEL:
        procedure (flags);

dcl 1 flags aligned,
      2 reset_sw bit (1) unaligned,
      3 pad bit (35) unaligned;

      on signal_io_ call SIGNAL_IO_HANDLER;

      call std_cl_intermediary (flags);
      return;
      end NEW_COMMAND_LEVEL;
```

INFO STRUCTURE

```
declare   signal_io_info_ptr             pointer;
declare   1 signal_io_info               aligned,
            2 header                     aligned like condition_info_header,
            2 iocb_name                  char (32) unaligned,
            2 iocb_ptr                   pointer,
            2 operation                  char (32),
            2 control_order_info_ptr     pointer,
            2 position_type              fixed bin,
            2 position_amount            fixed bin (35),
            2 data_ptr                   pointer,
            2 data_length                fixed bin (21),
            2 returned_data_length       fixed bin (21),
            2 returned_error_code        fixed bin (35),
            2 old_modes                  aligned,
              3 pointer                  pointer,
              3 length                   fixed bin (21),
            2 new_modes                  aligned,
              3 pointer                  pointer,
              3 length                   fixed bin (21),
            2 key                        char (256) varying;

declare   (
```

```
        SGI_OP_GET_LINE             init ("get_line"),
        SGI_OP_GET_CHARS            init ("get_chars"),
        SGI_OP_PUT_CHARS            init ("put_chars"),
        SGI_OP_MODES                init ("modes"),
        SGI_OP_POSITION             init ("position"),
        SGI_OP_CONTROL              init ("control"),
        SGI_OP_READ_RECORD          init ("read_record"),
        SGI_OP_WRITE_RECORD         init ("write_record"),
        SGI_OP_REWRITE_RECORD       init ("rewrite_record"),
        SGI_OP_DELETE_RECORD        init ("delete_record"),
        SGI_OP_SEEK_KEY             init ("seek_key"),
        SGI_OP_READ_KEY             init ("read_key"),
        SGI_OP_READ_LENGTH          init ("read_length")
        )
char (32) int static options (constant);

declare signal_io_io_buffer
 char (signal_io_info.data_length) based (signal_io_info.data_ptr);
declare signal_io_order_name
 char (signal_io_info.data_length) based (signal_io_info.data_ptr);
declare signal_io_old_modes
  char (signal_io_info.old_modes.length)
  based (signal_io_info.old_modes.pointer);
declare signal_io_new_modes
  char (signal_io_info.new_modes.length)
  based (signal_io_info.new__modes.pointer);
```

ARGUMENTS

header
: is the standard structure declared in condition_info_header.incl.pl1. The current version is zero. No fields here should be changed by handlers.

iocb_name
: is the name of the switch. This allows multiple switches to be serviced by the same handler. This field should not be changed by a handler.

iocb_ptr
: is the IOCB pointer for the switch. This allows multiple switches to be serviced by the same handler. This field should not be changed by a handler.

operation
: is the name of the IOX operation that caused this signal. This will be one of the the named constants SGI_OP_* declared in signal_io_info.incl.pl1. This should not be changed by handlers.

control_order_info_ptr
: is the info pointer associated with control orders. For operations other than control, this pointer is null. This should not be changed by handlers.

position_type
: is the type of position requested in a position operation. This should not be changed by handlers.

position_amount
: is the position distance requested in a position operation. This should not be changed by handlers.

data_ptr
: is a pointer to data to be written on a write operation, or a pointer to a data buffer to be filled on a read operation. In a control operation, this points to the character string name of the control order. This should not be changed by handlers. On read operations, the buffer pointer to by this pointer may be filled in.

data_length
: is the length of the data buffer in nine-bit characters. It should be used correspondingly to data_ptr.

returned_data_length
: is the amount of data read, or found in a record. On get_chars or get_line or read_record operations, the handler should set this to the amount of data placed in the buffer. On read_length or seek_key or read_key operations, the length of the record should be put here.

returned_error_code
: is the error code to be returned to the caller of iox_.

old_modes
: is a substructure. It describes a character string that should be set, in a modes operation, to the old modes. The pointer and length should not be changed by the handler.

new_modes
: is a substructure. It describes a character string that will be set, in a modes operation, to the desired new modes. The pointer and length should not be changed by the handler.

key
: is a keyed record key. On seek_key operations, this will be set to the desired key. On read_key operations this should be set by the handler to the key to be returned.

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 the Programmer's Reference Manual for a general description of the input/output system and a discussion of synonym attachments.

ATTACH DESCRIPTION

```
syn_ switch_name {-control_arg}
```

ARGUMENTS

switch_name
is the name of the I/O switch, y, for which the attached switch, x, is to be a synonym.

CONTROL ARGUMENTS

-inhibit names, -inh names
specifies which I/O operations are to be inhibited. The name arguments are separated by spaces and must be chosen from the following:

open	close
get_line	put_chars
get_chars	write_record
read_record	delete_record
rewrite_record	position
read_length	read_key
seek_key	modes
control	

SWITCH 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 the *American National Standards Institute's ANSI X3.27-1978,* "Magnetic Tape Labels and File Structure for Information Interchange". This document is referred to below as "the Standard". In addition, the I/O module provides a number of features that are extensions to, but outside of, the Standard. 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 the Programmer's Reference Manual 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 Standard.

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

```
tape_ansi_ vn1 vn2 ...  vnN {-control_args}
```

ARGUMENTS

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, that 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 names 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 that 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 that they may become members. These volumes are known as volume set candidates. (See "Volume Switching" below.)

CONTROL ARGUMENTS

-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.)

-clear, -cl
: specifies that internal information on a file-set which the I/O module retains from previous attachments is to be deleted. This control argument can be used when it is desired to change attributes of a file-set which are maintained across attachments for a given process, e.g. density or label standard. For the initial attachment to a file-set in a given process, this control argument has no effect.

-create, -cr
: specifies that a new file is to be created. (See "Creating a File" below.)

-density N, -den N
: specifies the density at which the file-set is recorded, where N can be 800, 1600, or 6250 bits per inch. (See "File Set Density" 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.)

-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.)

-extend, -ext
specifies extension of an existing file. (See "Extending a File" below.)

-force, -fc
specifies that the expiration date of the file being overwritten is to be ignored. (See "File Expiration" 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.)

-generate, -gen
specifies generation of an existing file. (See "Generating a File" below.)

-mode STR, -md STR
specifies the encoding mode used to record the file data, where STR is the string ascii, ebcdic, or binary. The default is ascii. (See "Encoding Mode" below.)

-modify, -mod
specifies modification of an existing file. (See "Modifying a File" below.)

-name STR, -nm STR
specifies the file identifier of the file where STR 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.)

-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.)

-replace STR, -rpl STR
specifies the file identifier of the file to be replaced, where STR must be from 1 to 17 characters. If no file with file identifier STR exists, an error is indicated. (See "Creating a File" below.)

-retain STR, -ret STR
specifies retention of resources across attachments, where STR 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.)

-speed S1{,S2,...,SN}, -ips S1{,S2,...,SN}
    specifies desired tape drive speeds in inches per second, where Si can be 75, 125, or 200 inches per second. (See "Device Speed Specification" below.)

The following sections define each control argument in the contexts that 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 control arguments are required to create a file, where STR 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 STR 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 STR 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 99996 characters.

db for variable length records, blocked.
Used when records are of varying lengths, the longest not in excess of 99992 characters.

sb for spanned records, blocked.
Used when the record length is fixed and in excess of 99996 characters, or variable and in excess of 99992 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 99996 characters.

NOTE: THE USE OF UNDEFINED RECORDS IS A NONSTANDARD FEATURE.

Records recorded using U format may be irreversibly 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:  amrl + 4 <= R <= 99996
F = sb | s:  amrl <= R <= 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 >= R
F = d:         B = R
F = sb | s:    18 <= B <= 99996
F = u:         amrl <= B <= 99996
```

In every case, B must be an integer in the range 18 <= B <= 99996.

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 STR control argument, the -number N control argument, or both in combination. If the -name STR 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 STR 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 STR 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 STR 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, that all 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, that 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 STR 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 STR can be used to explicitly specify the encoding mode, where STR is the string ascii, ebcdic, or binary. The default is the string ascii. (If -mode STR is not specified, the list_tape_contents command does not supply the specific mode in its report.)

NOTE: THE USE OF ENCODING MODES OTHER THAN ASCII IS A NONSTANDARD FEATURE.

If STR 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 STR 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 for the specific ASCII to EBCDIC mapping used by the I/O module.) If STR 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 STR 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 STR 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 <= ddd <= 366. A special case of the Julian date form is the value "00000" (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; the 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 (also called the slot identifier) is an 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, the volume identifier must be used in the volume specification field of the attach description.

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 or the volume name may be interpreted as a control argument.

Occasionally, it is necessary for a user to communicate some additional information to the operator in connection with a mount request. This can be done through the use of the -comment control argument:

```
     vni -comment STR
or:
     -volume vni -comment STR
```

where the -comment STR keyword and text specify that a given message is to be displayed on the operator's console whenever volume vni is mounted (a comment can be specified after each volume name supplied). STR can be from 1 to 64 characters. STR can 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 (that 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, that 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 <= N <= 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 equal to 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 three densities: 800, 1600, and 6250 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 can be 800, 1600, 6250 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 = 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	-generate
-expires	-modify
-extend	-replace
-force	

DEVICE SPEED SPECIFICATION

The -speed control argument is used to specify acceptable tape device speeds in inches per second. The module only attaches a device that matches a speed specified by this control argument. If more than one speed is specified, the module attaches a device that matches one of the speeds. If more than one device is attached, and more than one speed is specified, the devices will not necessarily all be of the same speed.

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 STR control argument is used to specify retention of resources across attachments, where STR specifies the detach-time resource disposition. If STR is the string all, all devices and volumes remain assigned to the process. If STR 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. If retention of any devices or volumes has been specified at or subsequent to attach time using the retention control operation, the unassign_resource command cannot be used. Instead, use the retain_none or retention -none control operation before detaching the I/O module. (See the retention, retain_none, retain_all operations 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, that 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, that is signalled by the command_query_ subroutine. Alternately, the answer command (described in the the Commands manual) 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 the Programmer's Reference Manual 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
A volume requires reinitialization or user verification 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. This query code can occur only if the I/O stream is opened for sequential_output.

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. This query code can occur only if the I/O stream is opened for sequential_output.

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. This query code can occur only if the I/O stream is opened for sequential_output.

If the I/O stream is opened for sequential_output, the user will be asked whether he wants to initialize or re-initialize the volume. If the I/O stream is opened for sequential_input, the user will be asked whether he wants to continue processing in spite of the discrepancy. 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 NOT be preceded by the -volume control argument. If a mount message is to be specified, the response takes the following form:

```
volume_name -comment STR
```

where STR 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 STR 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 (that 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 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 = B = 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, that includes the length of the SCW itself. The SCW also contains a one-character record segment code, that 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 the Multics system 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

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, holds a three-character volume authentication code.

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 47 - full 32-character volume name of next volume (EOV2 only)
CP 48        - blocking attribute (all)
               "0" = unblocked; "1" = blocked
CP 49        - 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. (See reset_error_lock operation below.) 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 eleven control operations.

hardware_status	retention
status	retain_none
volume_status	retain_all
file_status	reset_error_lock
feov	volume_density
close_rewind	

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 descriptions of the com_err_ and convert_status_code_ subroutines.) The I/O switch must be open.

This page intentionally left blank.

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) supplies 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 tape_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, 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.

retention, retain_none, retain_all Operations

These operations cause the tape resources currently in use, i.e., tape drives(s) and tape volume(s), to be unassigned or retained at detach time according to the specified retention argument or operation. The info_ptr points to a fixed binary number with value as defined below:

1 retention -none or retain_none
causes none of the tape resources currently in use to remain assigned at detach time.

2 retention -volume
causes the tape volume(s) currently in use to remain assigned at detach time.

3 retention -device
causes the tape drives(s) currently in use to remain assigned at detach time.

4 retention -all or retain_all
causes all of the devices and volumes currently in use to remain assigned at detach time.

reset_error_lock Operation

This operation unlocks the files so that further I/O is possible subsequent to a parity-type I/O error while reading. Such an error is indicated by a previous iox_$read_record or iox_$position call having returned the status code error_table_$tape_error. In this case, the value of tape_file_status.event_lock is error_table_$tape_error. (See the file_status operation above.) The I/O switch must be open for sequential_input. The info_ptr should be a null pointer.

NOTE: IF RECORDS ARE BLOCKED AND/OR SPANNED, THE VALIDITY OF ANY RECORDS READ SUBSEQUENT TO A PARITY-TYPE I/O ERROR IS NOT GUARANTEED. (The parity error is reported for the first read of a logical record in the block. The actual location of the error in the block is unknown.)

volume_density Operation

This operation returns the encoded density of the volume set. The I/O switch need not be open. The variable to which info_ptr points is declared as follows:

```
declare volume_density fixed bin;
```

The values returned and their meanings are listed below:

```
value          meaning
-----          -------

 -1            none specified yet
  2            800
  3            1600
  4            6250
```

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.

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.

CONTROL OPERATIONS FROM COMMAND LEVEL

All control operations supported by this I/O module can be executed from command level by using the io_call command. The general format is:

```
io_call control switchname operation -control_arg
```

ARGUMENTS

switchname
    is the name of the I/O switch that is attached through the I/O module to an ANSI tape file-set.

operation
    is any of the control operations previously described and summarized below.

operation	abbreviation	control_arg
status	st	-all
hardware_status	hst	
reset_error_lock	rel	
file_status	fst	
volume_status	vst	
retention	ret	-none, -volume, -device, -all
retain_all	reta	
retain_none	retn	
close_rewind	crw	
feov	feov	

CONTROL ARGUMENTS

are operation control arguments valid only for the retention and the status operations. A control argument is required for the retention operation.

-all
    causes all of the devices and volumes currently in use to remain assigned at detach time.

-device
    causes the tape drives(s) currently in use to remain assigned at detach time.

-none
    causes none of the tape resources currently in use to remain assigned at detach time.

-volume
    causes the tape volume(s) currently in use to remain assigned at detach time.

The -all control argument is optional for the status operation. This control argument prints all available tape status information such as the device status, the volume status, the file status, and the hardware status. The -all control argument is only for use with the status operation through the io_call command. It is not defined for use in the status operation with iox_$control directly.

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 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 access the old file. Each opening for sequential_output creates a new file, and each subsequent opening for sequential_input accesses 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 12/31/83
```

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, 1983. 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 <= 99996
-clear           -cl
-create          -cr
-density N       -den N      N = 800 | 1600  | 6250
-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 STR        -md STR     STR = ascii | ebcdic | binary
-modify          -mod
-name STR        -nm STR     STR <= 17 characters
-number N        -nb N       1 <= N <= 9999
-record R        -rec R      1 <= R <= 1044480
-replace         -rpl        STR <= 17 characters
-retain STR      -ret STR    STR = all | none
-ring            -rg
```

The following is a list of positional keywords:

```
-comment STR     -com STR    STR <= 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 the Programmer's Reference Manual 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
: 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

```
tape_ibm_ vn1 vn2 ...  vnN {-control_args}
```

ARGUMENTS

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 that 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 that 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 that they may become members. These volumes are known as volume set candidates. (See "Volume Switching" below.)

CONTROL ARGUMENTS
A control argument may appear only once.

-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.)

-clear, -cl
specifies that internal information on a file-set which the I/O module retains from previous attachments is to be deleted. This control argument can be used when it is desired to change attributes of a file-set which are maintained across attachments for a given process, e.g. density or label standard. For the initial attachment to a file-set in a given process, this control argument has no effect.

-create, -cr
specifies that a new file is to be created. (See "Creating A File" below.)

-density N, -den N
specifies the density at which the file set is recorded, where N can be 800, 1600, or 6250 bits per inch. (See "File Set Density" 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.)

-dos
specifies that a file was produced by, or is destined for, a DOS installation. (See "DOS Files" 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.)

-extend, -ext
specifies extension of an existing file. (See "Extending a File" below.)

-force, -fc
specifies that the expiration date of the file being overwritten is to be ignored. (See "File Expiration" 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.)

-mode STR, -md STR
specifies the encoding mode used to record the file data, where STR is the string ebcdic, ascii, or binary; the default is ebcdic. (See "Encoding Mode" below.)

-modify, -mod
specifies modification of an existing file. (See "Modifying a File" below.)

-name STR, -nm STR
specifies the file identifier of the file, where STR is from 1 to 17 characters. (See "Creating A File" below.)

-no_labels, -nlb
specifies that unlabeled tapes are to be processed. (See "Unlabeled Tapes" 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.)

-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.)

-replace STR, -rpl STR
specifies the file identifier of the file to be replaced, where STR must be from 1 to 17 characters. If no file with file identifier STR exists, an error is indicated. (See "Creating A File" below.)

-retain STR, -ret STR
specifies retention of resources across attachments, where STR 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.)

-speed S1{,S2,...,SN}, -ips S1{,S2,...,SN}
specifies desired tape drive speeds in inches per second, where Si can be 75, 125, or 200 inches per second. (See "Device Speed Specification" 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.

File Identifiers

Associated with every file is a name (file identifier) and a number (file sequence number). The file identifier must be 17 characters or less. When creating a file, the file identifier must be composed of one or more components of one to eight characters, with adjacent components separated by a period. The first character of each component 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. If a file identifier (of an existing file) does not meet the naming conventions established for files created on the Multics system, the file must be referenced using the -number control argument and a file sequence number.

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 control arguments are required to create a file, where STR is the file identifier. If no file having file identifier STR 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. Either a file identifier or a file sequence number is sufficient to uniquely identify a particular file in the file set. The -number and -replace 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 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 control argument must be explicitly specified. (See "Volume Initialization" below.)

The -format, -record and -block 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 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 32760 characters.

vb for variable-length records.
Used when records are of varying lengths, the longest not in excess of 32752 characters.

vbs for spanned records.
Used when the record length is fixed and in excess of 32760 characters, or variable and in excess of 32752 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 32760 characters.

When the -record 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 = vb  | v:   amrl + 4 <= R <= 32756
F = vbs | vs:  amrl <= R <= 1044480
F = u:         R is undefined
               (the -record control argument should not be used.)
```

When the -block 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 = vb:          b >= R + 4
F = v:           B = R + 4
F = vbs | vs:    20 <= B <= 32760
F = u:           amrl <= B <= 32760
```

In every case, B must be an integer in the range 20 <= B <= 32760, and, when the I/O switch is opened for sequential_output, must satisfy mod (B,4) = 0.

Since the structure attribute control arguments are interdependent, care must be taken to ensure that specified values are consistent.

Padding

Since the Multics system 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 (B,4) = 0, and pads a record, if necessary, to meet this requirement. (Warning: this padding may cause IBM-system rejection of a block if block length is not a multiple of the record length.) The following rules govern padding on output:

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 R, the record is padded on the right with blanks to length R. The last (or only) record of the file may be padded on the right with N blanks, where 0 <= N <= 19 is sufficient to satisfy B >= 20, and mod (B,4) = 0.

F = f: if iobl is less than R, the record is padded on the right with blanks to length R. Because the specified value of B must satisfy B >= 20, mod (B,4) = 0, and R = 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 <= N <= 12 is sufficient to satisfy B >= 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 <= N <= 12 is sufficient to satisfy B >= 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 <= N <= 12 is sufficient to satisfy B >= 20, and mod (B,4) = 0.

F = vs: every record or record segment is padded on the right with N blanks, where 0 <= N <= 12 is sufficient to satisfy B >= 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 <= N <= 12 is sufficient to satisfy B >= 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, that 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 control argument, the -number control argument, or both in combination. If the -name 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 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 and -number 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 control argument, the -number 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 control argument, the -number 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 three data encoding modes: EBCDIC, binary, 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 control argument can be used to explicitly specify the encoding mode (if not used, the list_tape_contents command does not supply the specific mode in its report).

If STR 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 STR 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 for the specific ASCII to EBCDIC mapping used by the I/O module.) If STR is the string binary, any 9-bit byte value can be recorded. However, data written on IBM equipment with binary mode may not be compatible with Multics, or vice versa.

Because the data encoding mode is not recorded in the file labels, the -mode ascii and the -mode binary control arguments must always be specified when subsequently processing an ASCII or binary file, respectively.

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 control argument is used to specify an expiration date where date must be of a form acceptable to the convert_date_to_binary_ subroutine; the 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 (also called the slot identifier) is an identifier physically written on, or affixed to, the reel or container of the volume. 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, the volume identifier must be used in the volume specification field of the attach description.

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 or the volume name may be interpreted as a control argument.

Occasionally, it is necessary for a user to communicate some additional information to the operator in connection with a mount request. This can be done through the use of the -comment control argument:

```
     vni -comment STR
or:
     -volume vni -comment STR
```

where the -comment STR keyword and text specify that a given message is to be displayed on the operator's console whenever volume vni is mounted (a comment can be specified after each volume name supplied). STR can be from 1 to 64 characters. STR can 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, 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 user is queried as described above. If either of these methods results in a volume name, volume switching occurs and processing of the file continues.

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 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 <= N <= 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 equals the previous value of N.

File Set Density

The I/O module makes provision for three densities: 800, 1600, and 6250 bpi (bits per inch). Every file in a file set must be recorded at the same density; otherwise, an error is indicated.

The -density 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 can be 800, 1600, and 6250 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 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

Device Speed Specification

The -speed control argument is used to specify acceptable tape device speeds in inches per second. The module only attaches a device that matches a speed specified by this control argument. If more than one speed is specified, the module attaches a device that matches one of the speeds. If more than one device is attached, and more than one speed is specified, the devices will not necessarily all be of the same speed.

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 control argument is used to specify retention of resources across attachments, where STR specifies the detach-time resource disposition. If STR is the string all, all devices and volumes remain assigned to the process. If STR 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. If retention of any devices or volumes has been specified at or subsequent to attach time using the retention control operation, the unassign_resource command cannot be used. Instead, use the retain_none or retention -none control operation before detaching the I/O module. (See the retention, retain_none, retain_all operations 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 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 the Programmer's Reference Manual 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 is probably 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
A volume requires reinitialization or user verification 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. This query code can occur only if the I/O stream is opened for sequential_output.

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. This query code can occur only if the I/O stream is opened for sequential_output.

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. This query code can occur only if the I/O stream is opened for sequential_output.

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_$unexpired_file
This can occur only when the I/O switch is open for sequential_output. A file that 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 NOT be preceded by the -volume control argument. If a mount message is to be specified, the response takes the following form:

```
volume_name -comment STR
```

where STR 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 = vb

2. block length - the default is B = 8192

3. record length -
   F = u: undefined
   F = fb | f: R = block length
   F = vb | v: R = block length - 4
   F = vbs | vs: R = 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, -block, and -record 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 fb 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 < 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 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, that includes the length of the SDW itself. (See "DOS Files" above.) The SDW also contains a one-character record segment code in binary, that 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 control argument, the I/O module attempts to append it to the file set, resulting in an error.

Conformance To Standard

With one exception, the I/O module conforms to the Standard: 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, holds a three-character volume authentication code.

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. (See reset_error_lock operation, below.) 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.

List of Control Orders

The I/O module supports eleven control operations.

hardware_status	retention
status	retain_none
volume_status	retain_all
file_status	reset_error_lock
feov	volume_density
close_rewind	

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 descriptions of the convert_status_code_ and the com_err_ subroutines.) The I/O switch must be open. The structure to which info_ptr points, device_status.incl.pl1, is declared as follows:

This page intentionally left blank.

```
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) supplies 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 tape_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 (feov) 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.

retention, retain_none, retain_all Operations

These operations cause the tape resources currently in use, i.e., tape drives(s) and tape volume(s), to be unassigned or retained at detach time according to the specified retention argument or operation. The info_ptr points to a fixed binary number with value as defined below:

1 retention -none or retain_none
causes none of the tape resources currently in use to remain assigned at detach time.

2 retention -volume
causes the tape volume(s) currently in use to remain assigned at detach time.

3 retention -device
causes the tape drives(s) currently in use to remain assigned at detach time.

4 retention -all or retain_all
causes all of the devices and volumes currently in use to remain assigned at detach time.

reset_error_lock Operation

This operation unlocks the files so that further I/O is possible subsequent to a parity-type I/O error while reading. Such an error is indicated by a previous iox_$read_record or iox_$position call having returned the status code error_table_$tape_error. In this case, the value of tape_file_status.event_lock is error_table_$tape_error. (See file_status operation, above.) The I/O switch must be open for sequential_input. The info_ptr should be a null pointer.

NOTE: IF RECORDS ARE BLOCKED AND/OR SPANNED, THE VALIDITY OF ANY RECORDS READ SUBSEQUENT TO A PARITY-TYPE I/O ERROR IS NOT GUARANTEED. (The parity error is reported for the first read of a logical record in the block. The actual location of the error in the block in unknown.)

volume_density OPERATION

This operation returns the encoded density of the volume set. The I/O switch need not be open. The variable to which info_ptr points is declared as follows:

```
declare volume_density fixed bin;
```

The values returned and their meanings are listed below:

```
value        meaning
-----        -------

 -1          none specified yet
  2          800
  3          1600
  4          6250
```

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.

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 argument 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.)

Control Operations from Command Level

All control operations supported by this I/O module can be executed from command level by using the io_call command. The general format is:

```
io_call control switchname operation -control_arg
```

where:

switchname
: is the name of the I/O switch that is attached through the I/O module to an IBM tape file-set.

operation
: is any of the control operations previously described and summarized below.

operation	abbreviation	control_arg
status	st	-all
hardware_status	hst	
reset_error_lock	rel	
file_status	fst	
volume_status	vst	
retention	ret	-none, -volume, -device, -all
retain_all	reta	
retain_none	retn	
close_rewind	crw	
feov	feov	

CONTROL ARGUMENTS
are operation control arguments valid only for the retention and the status operations. A control argument is required for the retention operation.

-none
: causes none of the tape resources currently in use to remain assigned at detach time.

-volume
: causes the tape volume(s) currently in use to remain assigned at detach time.

-device

-all
: causes all of the devices and volumes currently in use to remain assigned at detach time.

The -all control argument is optional for the status operation. This control argument prints all available status information such as the device status, the volume status, the file status, and the hardware status. The -all control argument is only for use with the status operation through the io_call command. It is not defined for use in the status operation with iox_$control directly.

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:

```
                  short
control_arg       form        value of argument
-----------       -----       -----------------

-block B          -bk B       20 <= B <= 32760
                              mod (B,4) = 0 if open for
                                 sequential_output
-clear            -cl
-create           -cr
-density N        -den N      N = 800 | 1600 | 6250
-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 STR         -md STR     STR = ebcdic | ascii | binary
-modify           -mod
-name STR         -nm STR     STR <= 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 STR      -rpl STR    STR <= 17 characters
-retain STR       -ret STR    STR = all | none
-ring             -rg
```

The following is a list of positional keywords:

```
-comment STR      -com STR    STR <= 64 characters
-volume VNI       -vol VNI    volume name <= 6 characters
```

Name: tape_mult_

The tape_mult_ I/O module supports I/O to and from Multics standard tapes.

ATTACH DESCRIPTION

```
tape_mult_ reelid {-control_args}
```

ARGUMENTS

reelid
is the name of the tape reel to be mounted for this attachment.

CONTROL ARGUMENTS

-comment STR, -com STR
specifies a comment string that is displayed to the operator. It can be used to give the operator any special instructions that are relevant to this attachment. The comment string must be enclosed within quotes if it contains blanks or other spacing characters.

-density N, -den N
specifies the density setting of the attached tape drive, where N can be 800, 1600, or 6250 bpi. The defaults are 800 for 7-track, and 1600 for 9-track. When opened for reading, the specified density is used only as a first guess. If the tape cannot be read at that density, tape_mult_ tries the other density.

-error_tally, -et
when opened for stream_input, displays an error summary on the user_output stream upon closing the tape I/O switch. This error summary includes: total number of read errors; number of errors that were successfully recovered for each of 1 to 7 backspace/re-read retries (each re-read using a different threshold and/or de-skew setting); number of errors that could not be recovered by backspace/re-reading but were successfully recovered by reading forward and finding a good copy of the original record in error; and the number of times that both backspace/re-read and read forward recovery failed, but successful recovery was accomplished by backspacing two files, forward-spacing two files (thus positioning the tape at the beginning of the current file after tape motion past the tape cleaner and head in both directions dislodges any buildup of oxide particles on the tape or head surface) and then reading forward until original record in error was read successfully. This information is obtained from metering data kept in the tape_mult_ work segment, defined by tmdb.incl.pl1.

-speed S1{,S2,...,SN}, -ips S1{,S2,...,SN}
specifies desired tape drive speeds in inches per second, where Si can be 75, 125, or 200 inches per second. (See "Device Speed Specification" below.)

-track N, -tk N
specifies the track type of the tape drive that is to be attached, where N may be either 9 or 7. The default is 9.

-write, -wrt
mounts the tape reel with a write ring. The default is to mount the tape reel without a write ring.

-system, -sys
increases tape performance by using more I/O buffers and other performance optimizations. Access to >system_control_1>rcp>workspace.acs or rcp_sys_ is required to use this control argument.

-volume_set_name STR, -vsn STR
specifies the contents of the volume set name field located in the tape label record (see the Programmer's Reference Manual for a description of the standard Multics tape label record). When opened for writing, STR is written into the volume_set_id field of the tape label record. If this control argument is not specified, the volume_set_id field will be set to blanks. When opened for reading, the volume_set_id field of the tape label is compared to STR. If they match or if the volume_set_id field is padded with blanks, the open operation is allowed to be completed. If the volume_set_id field and STR do not match and the volume_set_id is not padded with blanks, error_table_$bad_label is returned. STR can be up to 32 characters in length.

DEVICE SPEED SPECIFICATION

The -speed control argument is used to specify acceptable tape device speeds in inches per second. The module only attaches a device that matches a speed specified by this control argument. If more than one speed is specified, the module attaches a device that matches one of the speeds.

OPENING

The opening modes supported by tape_mult_ are stream_input and stream_output. If the opening mode is stream_output, the attach description must have specified the -write control argument.

READ RECORD OPERATION

The get_chars operation reads Multics standard records until either the caller's buffer is filled, or until the end of the tape volume is encountered. If not all the characters on a tape record fit into the caller's buffer, they are saved by the I/O module for the next get_chars call.

WRITE RECORD OPERATION

The put_chars operation formats the data into Multics standard records of 1024 data words each. Each record is written as it is filled. A partially filled record is not written onto the tape until it is filled with a subsequent put_chars operation, an error_count order is done, or the switch is closed.

LIST OF CONTROL ORDERS

The tape_mult_ I/O module supports the control operation with three orders.

error_count
: This order is supported only for the stream_output opening mode. It causes all output currently buffered to be written. An up-to-date error count is returned in the (fixed bin) variable referenced by the info_ptr argument.

boot_program
: This order allows the specification of a boot program to be written into the tape label record (see the programmer's Reference Manual for a discussion of the bootable Multics tape label record format and function). The specified boot program must be coded in absolute self-relocating ALM assembly language and must be less than or equal to 832 (1500 octal) locations in length. The specified boot program is overlayed starting at absolute location 300 (octal) in the tape label record. When a Multics tape containing a bootable label record is bootloaded, control is transferred to location 300 via the tape label record transfer vector, the first 8 words of a bootable Multics tape label record. The I/O switch must be closed when this control order is executed. The specified boot program is written onto the tape label record when the tape is subsequently opened for output. The info_ptr must point to a structure of the following form:

```
dcl  1 boot_program_info            based (info_ptr),
       2 version                    fixed bin,
       2 boot_program_ptr           pointer,
       2 boot_program_text_length   fixed bin (21),
       2 boot_program_name          char (32) unaligned;
```

where:

version
: is the version number of this structure, currently 1.

boot_program_ptr
: is a pointer to the beginning of the text section of the specified boot program.

boot_program_text_length
: is the length in 36-bit words of the text section of the specified boot program.

boot_program_name
: if nonblank, is the name of the boot program that the user wants recorded in the boot_pgm_path field of the label record. If boot_program_name is blank, then the absolute pathname of the boot program is written into the boot_pgm_path field of the label record.

get_boot_program
This order allows a boot program to be extracted from the tape label when the tape is opened for input. This control order must be issued immediately after the tape is opened for input and before the first read operation is begun. If it is executed later, then error_table_$no_operation is returned. The info_ptr must point to the boot_program_info structure defined above for the boot_program control order. The user must set the version number. Then a pointer to a buffer, containing the extracted boot_program, its length, and the entryname portion of the boot_program_pathname, is returned to the user. If the get_boot_program control order is executed on a tape that has a standard tape label record, boot_program_ptr is set to null.

CONTROL OPERATIONS FROM COMMAND LEVEL

All control operations can be performed from the io_call command, as follows:

```
io_call control switch order_arg
```

ARGUMENTS

switch
is the name of the I/O switch.

order_arg
must be one of the following:

error_count
boot_program PATH
get_boot_program

Name: tape_nstd_

The tape_nstd_ I/O module supports I/O to/from tapes in nonstandard or unknown formats. This module makes no assumptions about the format of the tape and returns one logical record for each physical record on the tape. Since the information upon the tape, including tape marks, is not interpreted by this I/O module, the user must detect the logical end of information on the reel. This I/O module functionally replaces ntape_.

Entry points in the module are not called directly by users; rather, the module is accessed through the iox_ subroutine. See the Programmer's Reference Manual for a general description of the I/O system and for a discussion of files.

ATTACH DESCRIPTION

```
tape_nstd_ reel_num {-control_args}
```

ARGUMENTS

reel_num
is the tape reel number.

CONTROL ARGUMENTS

block N, -bk N
specifies the maximum record length, in bytes, for this attachment. The default value for N is 11200. Values of N greater than 11888 require access to either the >system_library_1>rcp_sys_ gate or >sc1>rcp>workspace.acs (see "Buffer Size" below).

-comment STR -com STR
specifies a comment string that is displayed to the operator. It can be used to give the operator any special instructions that are relevant to this attachment. The comment string must be enclosed within quotes if it contains blanks or other spacing characters.

-density N, -den N
specifies the initial density to be used for this attachment. Acceptable values for N are 200, 556, 800, 1600 and 6250; the default is 800 bpi.

-speed S1{,S2,...,SN}, -ips S1{,S2,...,SN}
specifies desired tape drive speeds in inches per second, where Si can be 75, 125, or 200 inches per second. (See "Device Speed Specification" below.)

-track N, -tk N
means that the tape is N track. Acceptable values for N are 7 and 9. If no track argument is supplied then 9 track is assumed.

-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.

DEVICE SPEED SPECIFICATION

The -speed control argument is used to specify acceptable tape device speeds in inches per second. The module only attaches a device that matches a speed specified by this control argument. If more than one speed is specified, the module attaches a device that matches one of the speeds.

OPEN OPERATION

The opening modes supported are sequential_input, sequential_output, and sequential_input_output. If an I/O switch attached via the tape_nstd_ I/O module is to be opened for output or input_output, the -write control argument must occur in the attach description.

LIST OF CONTROL ORDERS

The following control operations are implemented by this I/O module:

backspace_file
: positions the tape before the file mark next encountered while rewinding the tape (if no file mark is encountered then the tape is left at load point).

backspace_record
: positions the tape before the previous record on the tape (or file mark if the current record is preceded by a file mark).

bcd
: sets hardware mode to binary coded decimal (BCD). See "Hardware Modes" below.

binary
: sets hardware mode to binary (this is the default). See "Hardware Modes" below.

data_security_erase
: erases the tape media from its current position to the end of tape (EOT) reflective marker. Additional "erase" control orders can be issued to erase any data written beyond the EOT reflective marker. No more than 40 additional erase control orders should be issued since the tape volume could run off the supply reel.

d200
: sets density to 200 bpi.

d556
: sets density to 556 bpi.

d800
: sets density to 800 bpi. This is the default.

d1600
: sets density to 1600 bpi.

d6250
: sets density to 6250 bpi.

erase
: erases tape for a distance of three inches from the current position.

fixed_record_length
: specifies that no record length information is expected by the caller since all data records are of a fixed length specified by a fixed bin(21) value. The record length is specified in bytes.

forward_file
positions the tape past the next file mark encountered on the tape.

forward_record
positions the tape after the next record (or file mark if one follows the current record) encountered on the tape.

io_call
supports the io_call command protocol for orders that expect nonnull info pointers. This order is prepared to interpret and print the status returned by the saved_status and request_status orders.

nine
sets hardware mode to eight/nine bit conversion. See "Hardware Modes" below.

protect
sets write inhibit regardless of the presence of a write permit ring in the tape reel. The tape unit will remain write inhibited until the tape is detached.

request_status
interrogates the tape controller and returns its status as a bit(12) aligned quantity.

reset_status
causes all resettable statuses of the tape unit to be reset.

retry_count
specifies a fixed bin(17) value which is the number of times an operation is to be retried before returning an error to the caller. The default value for the retry count is 10.

rewind
rewinds the tape to load point.

saved_status
returns the last status returned from the tape controller as a bit(12) aligned quantity.

unload
rewinds the tape and unloads it (done automatically when the tape is detached).

write_eof
writes an end of file mark (EOF).

HARDWARE MODES

In BCD mode, allowed only for 7-track drives, 6-bit characters are translated and then put on tape one character per frame. The translation is reversed on input.

In nine mode, on output four 8-bit bytes are written from each word ignoring the high order bit of each 9-bit byte (by truncating it). On input, 8-bit characters are converted to 9-bit characters by forcing the high order bit to zero (by appending a zero-bit). This mode should be used to put ASCII or EBCDIC data on tape for transfer to other systems with 8-bit bytes.

In binary mode, all 36 bits of each word are read or written. This mode should be used for native Multics applications where binary data is written to tape.

9-track write	9 8-bit bytes (2 words) are written to 9 frames on tape.
9-track read	9 frames are read into 9 8-bit bytes (2 words).
7-track write	6 6-bit frames from each word.
7-track read	6 frames on tape are read into 6 6-bit characters (1 word).

7-track is 6 data + 1 parity track.
9-track is 8 data + 1 parity track.

CLOSE OPERATION

The close operation rewinds the tape reel. The tape remains mounted, and positioned at the load point. No further I/O operations may be performed unless the I/O switch is opened again.

DETACH OPERATION

The detach operation unloads the tape.

READ RECORD OPERATION

The logical record returned by the read_record operation contains m=ceil(n/36) words, 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 m-n bits are 0's. The buffer supplied to the read_record operation must be word aligned. Read requests are retried 10 times before reporting an error unless a retry_count control order has been used to change the retry count.

WRITE RECORD OPERATION

The logical record supplied to the write_record operation must be word aligned, and must contain 0 mod 36 data bits.

NOTES

This I/O module violates those iox_ conventions that seem ill suited to processing raw tapes. In particular, read record and skip record operations may pass file marks. For example, if a tape contains two records, A and B, separated by a file mark, then the first read request would read record A, a second read request would return error_table_$end_of_info, and a third read request would return record B.

BUFFER SIZE

The maximum number of bytes that may be transmitted on a read_record or write_record operation is 180224, less overhead. This limit may be administratively restricted to a lower value. To use the full capability, the caller may need access to >system_library_1>rcp_sys_ or >sc1>rcp>workspace.acs.

Name: tc_io_

The tc_io_ I/O module supports terminal independent I/O to the screen of a video terminal.

Entry points in this module are not called directly by users; rather, the module is accessed through the I/O system interfaces iox_.

ATTACH DESCRIPTION

```
tc_io_ {device} {-control_args}
```

ARGUMENTS

device
: * is the channel name of the device to be attached If a device is not given, the -login_channel control argument must be given.

CONTROL ARGUMENTS

-login_channel
: specifies attachment to the user's primary login channel. If a device is not specified, then the user's login channel is used. This control argument flags this switch for reconnection by the process disconnection facility. If the user's login device should hang up, this switch will be automatically closed, detached, attached, and opened on the user's new login channel when the user reconnects, if permission to use this facility is specified in the SAT and PDT for the user.

-destination DESTINATION
: specifies that the attached device is to be called using the address DESTINATION. In the case of telephone auto_call lines, DESTINATION is the telephone number to be dialed. See the dial_manager_ subroutine in the *Multics Subroutines and I/O Modules* manual, Order No. AG93, for more details.

-no_block
: specifies that the device is to be managed asynchronously. The tty_ subroutine will not block to wait for input to be available or output space to be available. * This control argument should not be used on the login channel, because it will cause the command listener to loop calling get_chars.

-no_hangup_on_detach
: prevents the detach entry point from hanging up the device. This is not meaningful for the login channel.

-hangup_on_detach
: causes the detach entry point to hang up the device automatically. This is not meaningful for the login channel.

OPEN OPERATION

Opens the module for stream_input_output.

GET LINE OPERATION

The get_line operation is not supported.

CONTROL OPERATION

The following control orders are supported:

clear_screen
: clears the entire terminal screen. The info_ptr is null. It is intended for use when the screen image may have been damaged due to communications problems, for example.

get_capabilities
: returns information about the capabilities of the terminal. The info structure is described in the description of the "get_capabilities" control order in the window_io_ module.

get_break_table
: returns the current break table. The info pointer should point to a break table, declared as follows (window_control_info.incl.pl1):

```
dcl 1 break_table_info  aligned based (break_table_ptr),
      2 version         fixed bin,
      2 breaks          (0:127) bit (1) unaligned;
```

STRUCTURE ELEMENTS

version
: must be set by the caller to break_table_infc_version_1. (Input)

breaks
: has a "1"b for each character that is a break character. (Output)

set_break_table
: sets the break table. The info pointer should point to a break table as defined by the get_break_table order, above. By default, the break table has "1"b for all nonprintable characters, and "0"b elsewhere. Applications that set the break table must be careful to reset it afterwards, and establish an appropriate cleanup handler.

set_line_speed
: sets the speed of the terminal's connection to Multics. The info_ptr should point to a fixed binary number representing the line speed in characters per second. Negative line speeds are not allowed.

set_term_type
: changes the terminal type. The info pointer should point to a set_term_type_info structure, described below. This sets window_status_pending for all windows and sets the ttp_change field in the window_status structure along with the screen_invalid. This operation re-initializes all the terminal specific video system information such as the video sequences, length and width of the screen, and capabilities. It is equivalent to doing "window_call revoke; stty -ttp new_terminal_type; window_call invoke", except no windows are destroyed. The set_term_type_info structure is declared in set_term_type_info.incl.pl1:

```
dcl 1 set_term_type_info        aligned based (sttip)
      2 version                 fixed bin
      2 name char               (32) unaligned
      2 flags                   unaligned
        3 send_initial_string   bit (1)
        3 set_modes             bit (1)
        3 ignore_line_type      bit (1)
        3 mbz                   bit (33);
```

STRUCTURE ELEMENTS

version
: is the version of this structure. (Input) It must be stti_version_1.

name
: is the name of the new terminal type. (Input)

NOTES

The send_initial_string, set_modes and ignore_line_type flags are all ignored by the video system. The initial string will always be sent.

reconnection
: determines the new terminal type (which may or may not be the same as before the disconnection). Performs a set_term_type control order to inform the rest of the system of the change in terminal type. If the set_term_type fails then the video_utils_$turn_off_login_channel is invoked in an attempt to re-attach tty_. Reconnection (a field in window_status) is set to indicate to an application doing get_window_status that a reconnection has occurred.

  The window_status_info structure is declared in window_status.incl.pl1.

Name: tty_

The tty_ 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 the Programmer's Reference Manual for a general description of the I/O system.

ATTACH DESCRIPTION

```
tty_ {device} {-control_args}
```

ARGUMENTS

device
: is the channel name of the device to be attached (channel names are described in Appendix B of the Programmer's Reference manual). If a device is not given, one of the -login_channel or -dial_id control arguments must be given. The star convention is allowed.

CONTROL ARGUMENTS

-destination DESTINATION
: this control argument specifies that the attached device is to be called using the address DESTINATION. In the case of telephone auto_call lines, DESTINATION is the telephone number to be dialed. Use of this control argument requires the dialok attribute.

When the destination specifies an X.25 address it may optionally be preceded by "*" or "x29," to indicate that an X.29 (PAD) call should be made. For example, a destination of

x.29,3106:mitmul or
*3106:mitmul

specifies an X.29-type call on TYMNET.

-dial_id STR
specifies that dial connections are to be accepted on the dial_id STR. Use of this control argument requires the dialok attribute. The dial command is then used to connect a terminal on the dial_id STR. If STR is not a registered dial_id, then the Person_id.Project_id of the process being connected to must be supplied to the dial command. For example:

```
! dial STR Person.Project
```

To become a registered server, the process must have rw access to `>sc1>rcp>dial.STR.acs`.

-hangup_on_detach
causes the detach entrypoint to hang up the device automatically. This is not meaningful for the login channel.

-login_channel
specifies attachment to the user's primary login channel. This control argument flags this switch for reconnection by the process disconnection facility. If the user's login device should hang up, this switch is automatically closed, detached, attached, and opened on the user's new login channel when the user reconnects, if permission to use this facility is specified in the SAT and PDT for the user.

-no_block
specifies that the device is to be managed asynchronously. The tty_ module does not block to wait for input to be available or output space to be available (see "Buffering" below). This control argument should not be used on the login channel, because it causes the command listener to loop calling get_line.

-no_hangup_on_detach
prevents the detach entrypoint from hanging up the device. This is not meaningful for the login channel.

-no_suppress_dial_manager
enables dial_manager_, and is the default.

-resource STR
specifies the desired characteristics of a channel. STR (which can be null) consists of reservation attributes separated by commas. The channel used by a dial-out operation must have the characteristics specified in the reservation string. Reservation attributes consist of a keyword and optional argument. Attributes allowed are:

baud_rate=BAUD_RATE
line_type=LINE_TYPE

where BAUD_RATE is a decimal representation of the desired channel line speed and LINE_TYPE is a valid line type, chosen from line_types.incl.pl1 (see "set_line_type", below).

-required_access_class STR
specifies the access class that must be associated with the channel. STR is an access class string. The access class specified must be the same as the process's authorization unless the process has the "comm" privilege turned on, in which case the access class specified must be less than or equal to the process's authorization.

-suppress_dial_manager
prevents tty_ from using dial_manager_ to attach the specified channel. If the channel cannot be attach via a call to hcs_, the attach operation fails.

NOTES

The device specified must be available to the attaching process. The user's login device is always available. Any devices acquired with the dial_manager_ subroutine are also available. If the device is in slave service, and the user has appropriate access to its access control segment (rw to >sc1>rcp>NAME.acs), tty_ attempts to make it available using the privileged_attach mechanism of dial_manager_. If the -destination control argument is specified, the dial_out mechanism is used (the user must have rw access to >sc1>rcp>NAME.acs). If the -dial_id control argument is specified, the allow_dials or registered_server mechanism is used.

OPENING

The opening modes supported are stream_input, stream_output, and stream_input_output.

EDITING

To control editing, use the modes operation. Details on the various modes are given below.

BUFFERING

This I/O module blocks to await either the availability of input characters or the availability of output buffer space, unless the -no_block control argument is specified in the attach description. If the -no_block attach description control argument is specified, the behavior of the iox_$put_chars, iox_$get_chars, and iox_$get_line calls changes. If the put_chars entrypoint cannot write all the characters supplied, it returns a nonstandard status code consisting of the negative of the number of characters actually not written (returns -(number not written)). Any positive status code should be interpreted as a standard system status code. The get_chars and get_line entrypoints will return zero status codes and zero characters read if there is no input available.

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 can be performed on the switch during the interruption. If the interrupted operation is get_line, get_chars, or put_chars and another get_line, get_chars, or put_chars operation is performed during the interruption, the "start" control operation should be issued before the interrupted operation is resumed.

GET CHARS OPERATION

The get_chars operation reads as many characters as are available, up to, but not exceeding, the number requested by the caller. No error code is returned if the number of characters read is less than the number requested. At least one character is always returned (unless the number requested is zero). The characters read may comprise only a partial input line, or several input lines; no assumptions can be made in this regard.

GET LINE OPERATION

The get_line operation is supported. No error code is returned if the read operation occurs with the input buffer length at zero. For further explanation, see the iox_$get_line entry.

PUT CHARS OPERATION

The put_chars operation is supported (see the iox_$put_chars entry).

CONTROL OPERATION

The following orders are supported when the I/O switch is open. Except as noted, the info_ptr should be null. The orders are divided into categories. Local orders perform a specific function one time only, global orders change the way the system interfaces with the terminal, and other orders fit in neither category. Control orders are performed through the iox_$control entry.

LIST OF LOCAL ORDERS

abort
: flushes the input and output buffers.

get_chars_timeout
: performs a get_chars operation, with a timeout specified. The preferred method to using this control order is to use the timed_io_$get_chars subroutine. The info_ptr points to the following structure (declared in io_timeout_info.incl.pl1):

```
dcl 1 input_timeout_info,
      2 timeout              fixed bin (71),
      2 buffer_pointer       ptr,
      2 buffer_length        fixed bin (21),
      2 characters_read      fixed bin (21);
```

where the buffer_pointer, buffer_length, and characters_read elements are the same as the corresponding arguments to iox_$get_chars (Input). The timeout element is the number of microseconds tty_ will wait before returning error_table_$timeout (Input).

get_line_timeout
: performs a get_line operation, with a timeout specified. The preferred method to using this control order is to use the timed_io_$get_line subroutine. The info_pointer points to the same structure as that specified for get_chars_timeout.

hangup
: disconnects the telephone line connection of the terminal, if possible. This makes the terminal unavailable for further use.

interrupt
: sends an out-of-band interrupt signal (quit signal) to the terminal.

listen
: sends a wakeup to the process once the line associated with this device identifier is dialed up.

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 "Notes" below).

position
: the I/O switch must be open for stream input. The I/O module reads and discards the number of lines specified by the call.

printer_on
: causes the printer mechanism of the terminal to be re-enabled (see "Notes" below).

put_chars_timeout
performs a put_chars operation, with a timeout specified. The preferred method to using this control order is to use the timed_io_$put_chars subroutine. The info_ptr points to the following structure (declared in io_timeout_info.incl.pl1):

```
dcl 1 output_timeout_info,
      2 timeout              fixed bin (71),
      2 buffer_pointer       ptr,
      2 buffer_length        fixed bin (21),
      2 characters_written   fixed bin (21);
```

resetread
flushes the input buffer.

resetwrite
flushes the output buffer.

*
wru
initiates the transmission of the answerback of the device, if it is so equipped. This operation is allowed only for the process that originally attached the device (generally the initializer process). The answerback can subsequently be read by means of the get_chars input/output operation.

LIST OF GLOBAL CONTROL ORDERS

accept_printer_off
causes subsequent printer_off and printer_on orders to be accepted if possible.

get_channel_info
returns the name of the attached channel and its hardcore device index. The info_ptr must point to the following structure (defined in tty_get_channel_info.incl.pl1):

```
dcl 1 tty_get_channel_info        aligned based,
      2 version                   fixed bin,
      2 devx                      fixed bin,
      2 channel_name              char (32);
```

where:

version
is the version of this structure. (Input). It must be set to tty_get_channel_info_version.

devx
is the hardcore device index for the channel. (Output)

channel_name
is the name of the channel. (Output)

get_delay
is used to find out what delay values are currently in effect. The info_ptr points to the structure described for set_delay (below), which is filled in as a result of the call (except for the version number, which must be supplied by the caller).

get_editing_chars
is used to find out what input editing characters are in effect. The info_ptr points to the structure described below for set_editing_chars, which is filled in as a result of the call (except for the version number, which must be supplied by the caller).

get_framing_chars
causes the framing characters currently in use to be returned (see the set_framing_chars order below). If no framing characters have been supplied, NUL characters are returned. The info_ptr must point to a structure like the one described for the set_framing_chars order; this struc
the call.

get_ifc_info
causes the characters currently in use for input flow control to be returned (see the input_flow_control_chars order below). The info_ptr must point to a structure like the one described for the input_flow_control_chars order, which is filled in as a result of the call. If no characters are currently set, the count fields are set to 0.

get_input_translation
get_output_translation
get_input_conversion
get_output_conversion
these orders are used to obtain the current contents of the specified table. The info_ptr points to a structure like the one described for the corresponding "set" order below, which is filled in as a result of the call (except for the version number, which must be supplied by the caller). If the specified table does not exist (no translation or conversion is required), the status code error_table_$no_table is returned.

get_ofc_info
causes the characters and protocol currently in use for output flow control to be returned (see the output_flow_control_chars order below). The info_ptr must point to a structure like the one described for the output_flow_control_chars order, which is filled in as a result of the call. If no output flow control protocol is currently in use, the count fields are set to 0, and both suspend_resume and block_acknowledge are set to "0"b.

get_special
is used to obtain the contents of the special_chars table currently in use. The info_ptr points to the following structure (defined in tty_convert.incl.pl1):

```
dcl 1 get_special_info_struc    aligned,
      2 area_ptr                ptr,
      2 table_ptr               ptr;
```

where:

area_ptr
points to an area in which a copy of the current special_chars table is returned. (Input)

table_ptr
is set to the address of the returned copy of the table. (Output)

hangup_proc
allows you to establish a procedure that is called when the communications line to the device hangs up.

input_flow_control_chars
specifies the character(s) to be used for input flow control for terminals with line speed input capability. The terminal must be in iflow mode for the feature to take effect. The info_ptr must point to the following structure (declared in flow_control_info.incl.pl1):

```
dcl 1 input_flow_control_info   aligned,
      2 suspend_seq             unaligned,
        3 count                 fixed bin(9) unsigned,
        3 chars                 char (3),
      2 resume_seq              unaligned,
        3 count                 fixed bin(9) unsigned,
        3 chars                 char (3),
      2 timeout                 bit(1);
```

where:

suspend_seq
: is the character sequence that the system sends to tell the terminal to stop sending input, or that the terminal sends to inform the host that it is suspending input. count is an integer from 0 to 3 that specifies the number of characters in the sequence. chars are the characters themselves. At present, only sequences of length 0 or 1 are supported.

resume_seq
: is the character sequence to be sent by the system to the terminal to tell it to resume transmission of input. count is an integer from 0 to 3 that specifies the number of characters in the sequence. chars are the characters themselves.

timeout
: is "1"b if the resume character is to be sent to the terminal after input has ceased for one second, whether or not a suspend character has been received.

output_flow_control_chars
: enables either of two output flow control protocols and specifies the characters to be used for output flow control. The terminal must be in oflow mode for the feature to take effect. The info_ptr must point to the following structure (declared in flow_control_info.incl.pl1):

```
dcl 1 output_flow_control_info  aligned,
      2 flags                   unaligned,
        3 suspend_resume        bit(1),
        3 block_acknowledge     bit(1),
        3 mbz                   bit(16),
      2 buffer_size             fixed bin(18) unsigned unaligned,
      2 suspend_or_etb_seq      unaligned,
        3 count                 fixed bin(9) unsigned,
        3 chars                 char (3),
      2 resume_or_ack_seq       unaligned,
        3 count                 fixed bin(9) unsigned,
        3 chars                 char (3);
```

where:

suspend_resume
: is "1"b to specify a suspend/resume protocol.

block_acknowledge
: is "1"b to specify a block acknowledgement protocol.

buffer_size
: is the number of characters in the terminal's buffer if block_acknowledge is "1"b; otherwise, it is ignored.

suspend_or_etb_seq
: is the character sequence sent by the terminal to tell the system to suspend output if suspend_resume is "1"b, or the end_of_block character sequence if block_acknowledge is "1"b. count is an integer from 0 to 3 that specifies the number of characters in the sequence. chars are the characters themselves.

resume_or_ack_seq
: is the character sequence sent by the terminal to indicate that output can be resumed if suspend_resume is "1"b, or the character sequence sent by the terminal to acknowledge completion of a block if block_acknowledge is "1"b. count is an integer from 0 to 3 that specifies the number of characters in the sequence. chars are the characters themselves.

refuse_printer_off
: causes subsequent printer_off and printer_on orders to be rejected, except when in echoplex mode.

set_delay
: sets the number of delay characters associated with the output of carriage-motion characters. The info_ptr points to the following structure (defined in tty_convert.incl.pl1):

```
dcl 1 delay_struc        based aligned,
      2 version          fixed bin,
      2 default          fixed bin,
      2 delay,
        3 vert_nl        fixed bin,
        3 horz_nl        float bin,
        3 const_tab      fixed bin,
        3 var_tab        float bin,
        3 backspace      fixed bin,
        3 vt_ff          fixed bin;
```

where:

version
: is the version number of the structure. It must be 1.

default
: indicates, if nonzero, that the default values for the current terminal type and baud rate are to be used and that the remainder of the structure is to be ignored.

vert_nl
: is the number of delay characters to be output for all newlines to allow for the linefeed (-127 <= vert_nl <= 127). If it is negative, its absolute value is the minimum number of characters that must be transmitted between two linefeeds (for a device such as a TermiNet 1200).

horz_nl
: is a number to be multiplied by the column position to obtain the number of delays to be added for the carriage return portion of a newline (0 <= horz_nl <= 1). The formula for calculating the number of delay characters to be output following a newline is:

```
ndelays = vert_nl + fixed (horz_nl*column)
```

const_tab
: is the constant portion of the number of delays associated with any horizontal tab character (0 <= const_tab <= 127).

var_tab
: is the number of additional delays associated with a horizontal tab for each column traversed (0 <= var_tab <= 1). The formula for calculating the number of delays to be output following a horizontal tab is:

```
ndelays = const_tab + fixed (var_tab*n_columns)
```

backspace
: is the number of delays to be output following a backspace character (-127 <= backspace <= 127). If it is negative, its absolute value is the number of delays to be output with the first backspace of a series only (or a single backspace). This is for terminals such as the TermiNet 300 that need delays to allow for hammer recovery in case of overstrikes, but do not require delays for the carriage motion associated with the backspace itself.

vt_ff
: is the number of delays to be output following a vertical tab or formfeed (0 <=vt_ff <= 511).

set_editing_chars
: changes the characters used for editing input. The info_ptr points to the following structure (declared in tty_editing_chars.incl.pl1):

```
dcl 1 editing_chars     aligned,
      2 version         fixed bin,
      2 erase           char(1) unaligned,
      2 kill            char(1) unaligned;
```

where:

version
    is the version number of this structure. It must be 2.

erase
    is the erase character.

kill
    is the kill character.

The following rules apply to editing characters:

1. The two editing characters cannot be the same.

2. No carriage-movement character (carriage return, newline, horizontal tab, backspace, vertical tab, or formfeed) can be used for either of the editing functions.

3. NUL and space cannot be used for either editing function.

4. If the editing character is an ASCII control character, it will not have the desired effect unless ctl_char mode is on (see "Modes Operation" below).

set_framing_chars
    specifies the pair of characters that the terminal generates surrounding input transmitted as a block or "frame." These characters must be specified in the character code used by the terminal. This order must be used for blk_xfer mode (see below) to be effective. The info_ptr must point to a structure with the following format:

```
dcl 1 framing_chars aligned,
      2 frame_begin char(1) unaligned,
      2 frame_end char(1) unaligned;
```

set_input_conversion
    provides a table to be used in converting input to identify escape sequences and certain special characters. The info_ptr points to a structure of the following form (defined in tty_convert.incl.pl1):

```
dcl 1 cv_trans_struc      aligned,
      2 version           fixed bin,
      2 default           fixed bin,
      2 cv_trans          aligned,
        3 value           (0 : 255) fixed bin(8) unaligned;
```

where:

version
is the version number of the structure. It must be 1.

default
indicates, if nonzero, that the default table for the current terminal type is to be used, and the remainder of the structure is ignored.

values
are the elements of the table. This table is indexed by the value of a typed input character, and the corresponding entry contains the ASCII character resulting from the translation. If the info_ptr is null, no translation is to be done.

NOTE: In the case of a terminal that inputs 6-bit characters and case-shift characters, the first 64 characters of the table correspond to characters in lower shift, and the next 64 correspond to characters in upper shift.

The table is indexed by the ASCII value of each input character (after translation, if any), and the corresponding entry contains one of the following values (mnemonic names for these values are defined in tty_convert.incl.pl1):

0 ordinary character
1 break character
2 escape character
3 character to be thrown away
4 formfeed character (to be thrown away if page length is nonzero)
5 this character and immediately following character to be thrown away

set_input_translation
provides a table to be used for translation of terminal input to ASCII. The info_ptr points to a structure of the following form (declared in tty_convert.incl.pl1):

```
dcl 1 cv_trans_struc       aligned,
      2 version            fixed bin,
      2 default            fixed bin,
      2 cv_trans           aligned,
        3 value            (0 : 255) char(1) unaligned;
```

where version, default, and value are as described in the cv_trans_struc structure used with the set_input_conversion order above.

set_line_type
sets the line type associated with the terminal to the value supplied. The info_ptr should point to a fixed binary variable containing the new line type. Line types can be any of the following named constants defined in the include file line_types.incl.pl1:

LINE_ASCII
device similar to 7-bit ASCII using Bell 103-type modem protocol.

*

LINE_SYNC
synchronous connections, no protocol.

LINE_G115
ASCII synchronous connection, Model G-115 remote computer protocol.

LINE_BSC
binary synchronous protocol.

*

LINE_VIP
device similar to Honeywell Model 7700 Visual Information Projection (VIP) stand-alone terminal.

LINE_ASYNC1
LINE_ASYNC2
LINE_ASYNC3
site-supplied asynchronous protocols.

LINE_SYNC1
LINE_SYNC2
LINE_SYNC3
site-supplied synchronous protocols.

LINE_POLLED_VIP
: device similar to Honeywell Model 7700 Visual Projection System (VIP) polled terminal concentrator subsystem.

LINE_X25LAP
: X.25 network connection using the link access protocol (LAP).

LINE_COLTS
: special software channel used for Communications Online Test and Diagnostics System.

This operation is not permitted while the terminal is in use.

set_output_conversion
: provides a table to be used in formatting output to identify certain kinds of special characters. The info_ptr points to a structure like that described for set_input_conversion (above). The table is indexed by each ASCII output character (before translation, if any), and the corresponding entry contains one of the following values (mnemonic names for these values are defined in tty_convert.incl.pl1):

0 ordinary character.
1 newline.
2 carriage return.
3 horizontal tab.
4 backspace.
5 vertical tab.
6 formfeed.
7 character requiring octal escape.
8 red ribbon shift.
9 black ribbon shift.
10 character does not change the column position.
11 this character together with the following one do not change the column position (used for hardware escape sequences).
12 character is not to be sent to the terminal.
17 or greater a character requiring a special escape sequence. The indicator value is the index into the escape table of the sequence to be used, plus 16. The escape table is part of the special characters table; see the set_special order below.

set_output_translation

provides a table to be used for translating ASCII characters to the code to be sent to the terminal. The info_ptr points to a structure like that described for set_input_translation. The table is indexed by the value of each ASCII character, and the corresponding entry contains the character to be output. If the info_ptr is null, no translation is to be done.

NOTE: For a terminal that expects 6-bit characters and case-shift characters, the 400(8) bit must be turned on in each entry in the table for a character that requires upper shift, and the 200(8) bit must be on in each entry for a character that requires lower shift.

set_special

provides a table that specifies sequences to be substituted for certain output characters, and characters that are to be interpreted as parts of escape sequences on input. Output sequences are of the following form (defined in tty_convert.incl.pl1):

```
dcl 1 c_chars      based aligned,
      2 count      fixed bin(8) unaligned,
      2 chars(3)   char(1) unaligned;
```

where:

count

is the actual length of the sequence in characters (0 <= count <= 3). If count is zero, there is no sequence.

chars

are the characters that make up the sequence.

The info_ptr points to a structure of the following form (defined in tty_convert_incl.pl1):

```
dcl 1 special_chars_struc      aligned based,
      2 version                fixed bin,
      2 default                fixed bin,
      2 special_chars
        3 nl_seq               aligned like c_chars,
        3 cr_seq               aligned like c_chars,
        3 bs_seq               aligned like c_chars,
        3 tab_seq              aligned like c_chars,
        3 vt_seq               aligned like c_chars,
        3 ff_seq               aligned like c_chars,
        3 printer_on           aligned like c_chars,
        3 printer_off          aligned like c_chars,
        3 red_ribbon_shift     aligned like c_chars,
        3 black_ribbon_shift   aligned like c_chars,
        3 end_of_page          aligned like c_chars,
        3 escape_length        fixed bin,
        3 not_edited_escapes   (sc_escape_len refer
                                 (special_chars.escape_length))
                                 like c_chars,
        3 edited_escapes       (sc_escape_len refer
                                 (special_chars.escape_length))
                                 like c_chars,
        3 input_escapes        aligned,
          4 len                fixed bin(8) unaligned,
          4 str                char (sc_input_escape_len refer
                                 (special_chars.input_escapes.len))
                                 unaligned,
        3 input_results        aligned,
          4 pad                bit(9) unaligned,
          4 str                char (sc_input_escape_len refer
                                 (special_chars.input_escapes.len))
                                 unaligned;
```

where:

version
: is the version number of this structure. It must be 1.

default
: indicates, if nonzero, that the default values for the current terminal type and baud rate are to be used and that the remainder of the structure is to be ignored.

nl_seq
: is the output character sequence to be substituted for a newline character. The nl_seq.count generally should be nonzero.

cr_seq
: is the output character sequence to be substituted for a carriage-return character. If count is zero, the appropriate number of backspaces is substituted. However, either cr_seq.count or bs_seq.count should be nonzero (i.e., both should not be zero).

bs_seq
: is the output character sequence to be substituted for a backspace character. If count is zero, a carriage return and the appropriate number of spaces are substituted. However, either bs_seq.count or cr_seq.count, should be nonzero (i.e., both should not be zero).

tab_seq
: is the output character sequence to be substituted for a horizontal tab. If count is zero, the appropriate number of spaces is substituted.

vt_seq
: is the output character sequence to be substituted for a vertical tab. If count is zero, no characters are substituted.

ff_seq
: is the output character sequence to be substituted for a formfeed. If count is zero, no characters are substituted.

printer_on
: is the character sequence to be used to implement the printer_on control operation. If count is zero, the function is not performed.

printer_off
: is the character sequence to be used to implement the printer_off control operation. If count is zero, the function is not performed.

red_ribbon_shift
: is the character sequence to be substituted for a red-ribbon-shift character. If count is zero, no characters are substituted.

black_ribbon_shift
: is the character sequence to be substituted for a black-ribbon-shift character. If count is zero, no characters are substituted.

end_of_page
: is the character sequence to be printed to indicate that a page of output is full. If count is zero, no additional characters are printed, and the cursor is left at the end of the last line.

escape_length
is the number of output escape sequences in each of the two escape arrays.

not_edited_escapes
is an array of escape sequences to be substituted for particular characters if the terminal is in "^edited" mode. This array is indexed according to the indicator found in the corresponding output conversion table (see the description of the set_output_conversion order above).

edited_escapes
is an array of escape sequences to be used in edited mode. It is indexed in the same fashion as not_edited_escapes.

input_escapes
is a string of characters each of which forms an escape sequence when preceded by an escape character.

input_results
is a string of characters each of which is to replace the escape sequence consisting of an escape character and the character occupying the corresponding position in input_escapes.

set_term_type
sets the terminal type associated with the channel to one of the types defined in the terminal type table. The info_ptr should point to the following structure (declared in set_term_type_info.incl.pl1):

```
dcl 1 set_term_type_info      aligned based (sttip),
      2 version               fixed bin,
      2 name                  char (32) unaligned,
      2 flags,
        3 send_initial_string bit(1) unaligned,
        3 set_modes           bit(1) unaligned,
        3 ignore_line_type    bit(1) unaligned,
        3 mbz                 bit(33);
```

where:

version
is the version number of the above structure. It must be 1.

name
is the name of the terminal type to be set.

send_initial_string
is "1"b if the initial string for the terminal type is to be transmitted to the terminal; otherwise, it is "0"b.

set_modes
is "1"b if the default modes for the terminal type are to be set; otherwise, it is "0"b.

ignore_line_type
is "1"b if the terminal type to be set need not be compatible with the line type; otherwise, it is "0"b.

mbz
must be "0"b.

set_wakeup_table
specifies a wakeup table, i.e., a set of wakeup characters that controls the dispatching of input wakeups. The wakeup table operates in conjunction with wake_tbl mode. The wakeup table has no effect until wake_tbl mode is enabled. Once enabled, the standard method of generating input wakeups (normally one wakeup for each line) is suspended. Thereafter, wakeups are only generated when wakeup characters are received or when the buffer gets too full. The wakeup table cannot be changed while wake_tbl mode is enabled. The info_ptr should point to the following structure (declared in set_wakeup_table_info.incl.pl1):

```
dcl  swt_infop ptr;
dcl  swt_info_version_1 fixed bin static options (constant) init (1);

dcl 1 swt_info aligned based (swt_infop),
      2 version fixed bin,
      2 new_table like wakeup_table,
      2 old_table like wakeup_table;

dcl wakeup_tablep ptr;

dcl 1 wakeup_table aligned based (wakeup_tablep),
      2 wake_map (0:127) bit (1) unal,
      2 mbz bit (16) unal;
```

where:

version
is the version number of this structure. (Input). It must be 1.

new_table
is the wakeup table to set. (Input)

old_table
is set to the value of the current wakeup table that is being replaced. (Output). If no current wakeup table exists, all entries are set to "0"b.

wake_map
is an array having one entry for each character in the ASCII character set. (Input). A value of "1"b defines a wakeup character. All other entries must be "0"b. If all entries are "0"b, the current wakeup table, if any, is deleted.

mbz
must be "0"b. (Input)

The primary application for the wakeup table mechanism is to reduce overhead incurred by text editors, such as qedx, while in input mode. While in input mode, a user process must wake up for each line of input even though no processing is immediately required. In wake_tbl mode, a process is only awoken when input mode is exited or a large amount of input has been accumulated. However, since wake_tbl mode causes more input to be buffered in ring 0 than before, a quit signal is likely to discard more input than before. If a user does not wish to lose input, he simply should avoid quitting while in input mode.

If a user does quit out of input mode, he does not remain in wake_tbl mode (under normal circumstances). The default modes established by the standard quit handler include ^wake_tbl. A start command restores wake_tbl mode.

LIST OF MISCELLANEOUS CONTROL ORDERS

copy_meters
causes the current cumulative meters associated with the channel to be copied to unwired storage, so that the statistics for the channel can be determined both for the life of the system and for the current dialup. This order can only be issued by the "owning" process (normally the initializer). The info_ptr should be null.

get_event_channel
returns the identifier of the ipc_ event channel associated with the channel. The info_pointer should point to a fixed bin (71) aligned quantity into which the channel identifier is stored. If the switch is not yet open and the set_event_channel order has not been given, the result is zero.

This control order, which replaces the event_info control order, is accepted with the switch open or closed. For more information on event management, see the set_event_channel control order.

quit_disable
causes quit signal processing to be disabled for this device.

quit_enable
causes quit signal processing to be enabled for this device. (Quit signal processing is initially disabled.)

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 (defined in tty_read_status_info.incl.pl1), which is filled in by the call:

```
dcl 1 tty_read_status_info  aligned based,
      2 event_channel       fixed bin (71),
      2 input_pending       bit (1);
```

where:

event_channel
is the event channel used to signal the arrival of input.

input_available
indicates whether input is available.
"0"b no input
"1"b input

send_initial_string
transmits an initialization string to the terminal in raw output (rawo) mode. Due to the use of raw output mode, the string must comprise character codes recognized by the terminal. If the info_ptr is null, the initial string defined for the terminal type is used. Otherwise, the info_ptr should point to the following structure:

```
dcl 1 send_initial_string_info aligned,
      2 version                fixed bin,
      2 initial_string         char(512) varying;
```

where:

version
is the version number of the above structure. It must be 1.

initial_string
is the initial string to be sent.

set_default_modes
sets the modes to the default modes for the terminal type.

set_event_channel
specifies the ipc_ event channel that receives wakeups for this attachment. Wakeups are received for input available, output completed, and state changes such as hangups and quits. The channel can be event wait or event call. If it is event call, the -no_block control argument must be present in the attach description for correct operation.

The info_pointer should point to a fixed bin (71) aligned quantity containing a valid ipc_ channel identifier. No check for the validity of the channel is made. If the channel is invalid, incorrect operation results.

If this control order is not given before the opening of the switch, tty_ attempts to allocate a fast event channel. Fast event channels cannot be converted to call channels and receive no associated message. If tty_ cannot allocate a fast channel, an ordinary event wait channel is created and used. This control order is accepted while the switch is closed or open. If the switch is open, the new channel replaces the old one.

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 wakeups may have been lost or discarded.

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 that contains the new identifier.

terminal_info
returns information about the terminal. The info_ptr should point to the following structure (declared in terminal_info.incl.pl1):

```
dcl 1 terminal_info              aligned,
      2 version                  fixed bin,
      2 id                       char (4) unaligned,
      2 term_type                char (32) unaligned,
      2 line_type                fixed bin,
      2 baud_rate                fixed bin,
      2 reserved (4)             fixed bin;
```

where:

version
is the version number of the above structure. (Input). It must be 1.

id
is the terminal identifier derived from the answerback. (Output)

term_type
is the terminal type name. (Output)

line_type
is the line type number. (Output)

baud_rate
is the baud rate at which the terminal is running. (Output)

reserved
is reserved for future use.

write_status
tells whether or not there is any write-behind output that has not been sent to the terminal. The info_ptr should point to the following structure, which is filled in by the call:

```
dcl 1 info_structure     aligned,
      2 ev_chan          fixed bin(71),
      2 output_pending   bit(1);
```

where:

ev_chan
is the event channel used to signal the completion of output.

output_pending
indicates whether output is pending.
"0"b no output
"1"b output

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. For these modes, the complement is displayed with the mode. Normal defaults are indicated for those modes that are generally independent of terminal type. The modes string is processed from left to right; thus, if two or more contradictory modes appear within the same modes string, the rightmost mode prevails.

LIST OF MODES

8bit, ^8bit
causes input characters to be received without removing the eighth (high-order) bit, which is normally interpreted as a parity bit. This mode is valid for HSLA channels only. (Default is off.)

blk_xfer, ^blk_xfer
specifies that the user's terminal is capable of transmitting a block or "frame" of input all at once in response to a single keystroke. The system cannot handle such input correctly unless blk_xfer mode is on and the set_framing_chars order has been issued. (Default is off.)

breakall, ^breakall
enables a mode in which all characters are assumed to be break characters, making each character available to the user process as soon as it is typed. This mode only affects get_chars operations. (Default is off.)

can, ^can
performs standard canonicalization on input. (Default is on.)

can_type=overstrike, can_type=replace
specifies the method to be used to convert an input string to canonical form. Canonicalization is only performed when the I/O switch is in "can" mode. (Default is can_type=overstrike.)

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.)

crecho, ^crecho
echoes a carriage return when a line feed is typed. This mode can only be used with terminals and line types capable of receiving and transmitting simultaneously.

ctl_char, ^ctl_char
specifies that ASCII control characters that do not cause carriage or paper motion are to be accepted as input, except for the NUL character. If the mode is off, all such characters are discarded. (Default is off.)

default
is a shorthand way of specifying erkl, can, ^rawi, ^rawo, ^wake_tbl, and esc. The settings for other modes are not affected.

echoplex, ^echoplex
echoes all characters typed on the terminal. The same restriction applies as for crecho; it must also be possible to disable the terminal's local copy function.

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.)

erkl, ^erkl
performs "erase" and "kill" processing on input. (Default is on.)

esc, ^esc
enables escape processing on all input read from the device. (Default is on.)

force
specifies that if the modes string contains unrecognized or invalid modes, they are to be ignored and any valid modes are to be set. If force is not specified, invalid modes cause an error code to be returned, and no modes are set.

fulldpx, ^fulldpx
allows the terminal to receive and transmit simultaneously. This mode should be explicitly enabled before enabling echoplex mode.

hndlquit, ^hndlquit
echoes a newline character and performs a resetread of the associated stream when a quit signal is detected. (Default is on.)

iflow, ^iflow
specifies that input flow control characters are to be recognized and/or sent to the terminal. The characters must be set before iflow mode can be turned on.

init
sets all switch type modes off, sets line length to 50, and sets page length to zero.

lfecho, ^lfecho
echoes and inserts a line feed in the user's input stream when a carriage return is typed. The same restriction applies as for crecho.

llN, ^ll
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. If ^ll is specified, line length checking is disabled. In this case, if a line of more than 255 column positions is output by a single call to iox_$put_chars, some extra white space may appear on the terminal.

no_outp, ^no_outp
causes output characters to be sent to the terminal without the addition of parity bits. If this mode and rawo mode are on, any 8-bit pattern can be sent to the terminal. This mode is valid for HSLA channels only. (Default is off.)

oddp, ^oddp
causes any parity generation that is done to the channel to assume odd parity. Otherwise, even parity is assumed for line types other than 2741 and 1050. This mode is valid for HSLA channels only. (Default is off.)

oflow, ^oflow
specifies that output flow control characters are to be recognized when sent by the terminal. The characters and the protocol to be used must be set before oflow mode can be turned on.

plN, ^pl
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 any break character, the output continues with the next page. The warning message is normally the string "EOP", but can be changed by means of the set_special control order. The string is displayed on a new line after N consecutive output lines are sent to the screen (including long lines that are folded as more than one output line). To have the end-of-page string displayed on the screen without scrolling lines off the screen, N should be set to one less than the page length capability of the screen. However, if the end-of-page string is a null string, output stops at the end of the last line of the page or screen and N should be the actual page length capability of the screen.. If ^pl is specified, end-of-page checking is disabled. (See scroll mode below.)

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.)

prefixnl, ^prefixnl
controls what happens when terminal output interrupts a partially complete input line. In prefixnl mode, a newline character is inserted in order to start the output at the left margin; in ^prefixnl mode, the output starts in the current column position. Polite mode controls when input may be interrupted by output; prefixnl controls what happens when such an interruption occurs. (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.)

red, ^red
sends red and black shifts to the terminal.

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.)

scroll, ^scroll
specifies that end-of-page checking is performed in a manner suited to scrolling video terminals. If the mode is on, the end-of-page condition occurs only when a full page of output is displayed without intervening input lines. The mode is ignored whenever end-of-page checking is disabled. (Default is off.)

tabecho, ^tabecho
echoes the appropriate number of spaces when a horizontal tab is typed. The same restriction applies as for crecho.

tabs, ^tabs
inserts tabs in output in place of spaces when appropriate. If tabs mode is off, all tab characters are mapped into the appropriate number of spaces.

vertsp, ^vertsp
performs the vertical tab and formfeed functions, and sends appropriate characters to the device. Otherwise, such characters are escaped. (Default is off.)

wake_tbl, ^wake_tbl
causes input wakeups to occur only when specified wakeup characters are received. Wakeup characters are defined by the set_wakeup_table order. This mode cannot be set unless a wakeup table has been previously defined.

NOTES

The status code error_table_$action_not_performed is returned by the printer_on and printer_off control operations if the special characters table currently in effect indicates that this terminal cannot perform the printer_on or printer_off operation. The status code error_table_$no_table is returned by the get_input_translation, get_output_translation, get_input_conversion, get_output_conversion, and get_special control orders if the specified table does not exist. A code of zero is returned otherwise.

To assist the user in determining how to alter the tables described above, the following paragraphs provide a summary of the processing of input and output strings in ring 0.

INPUT PROCESSING

Translation The characters are translated from the terminal's code to ASCII, using the input_translation table. If there is no input_translation table, this step is omitted.

Canonicalization

The input string is rearranged (if necessary) into canonical form.

Editing

Performs erase and kill processing.

Break and escape processing

The characters in the input string are looked up in the input_conversion table and treated accordingly. If a character is preceded by an escape character (as determined from the table), it is looked up in the input_escapes array in the special_chars table and, if found, replaced by the corresponding character from the input_results array.

OUTPUT PROCESSING

Capitalization

Lowercase letters are replaced by uppercase for terminals in capo mode; uppercase letters are prefixed by escape characters if appropriate.

Formatting

The characters in the output string are looked up in the output_conversion table described above. Carriage-movement characters are replaced by sequences found in the special_chars table, followed by delay characters if so indicated by the delay table. Ribbon-shift characters are likewise replaced by appropriate sequences. Any character whose indicator in the output_conversion table is greater than 16 is replaced by the (indicator-16)th sequence in either the not_edited_escapes or edited_escapes array in the special_chars table.

Translation

The result of step 2 is translated from ASCII to the terminal's code, using the output_translation table. If there is no output_translation table, this step is omitted.

CONTROL OPERATIONS FROM COMMAND LEVEL

Some control operations can be performed from the io_call command, as follows:

```
io_call control switch_name order_arg
```

ARGUMENTS

switch_name
    is the name of the I/O switch.

order_arg
    can be any control order described above under "Control Operation" that can accept a null info_ptr, as well as read_status, write_status, terminal_info, and the following (which must be specified as shown):

    store_id id
        where id is the new answerback string.

    set_term_type type {-control_args}
        where type is the new terminal type and -control_args can be any of -initial_string (-istr), -modes, and -ignore_line_type.

    set_line_type line_type
        where line_type is the new line type.

    line_length N
        where N is the new line length.

The following control orders can be used as active functions:

[io_call control switch_name read_status]
    returns true if input is available; otherwise, false.

[io_call control switch_name write_status]
    returns true if output is pending; otherwise, false.

[io_call control switch_name terminal_info terminal_type]
    returns the current terminal type.

[io_call control switch_name terminal_info baud]
    returns the baud rate.

[io_call control switch_name terminal_info id]
    returns the terminal identifier (answerback).

[io_call control switch_name terminal_info line_type]
    returns the current line type.

Name: tty_printer_

The tty_printer_ I/O module performs stream I/O to a standard terminal (e.g., TN1200, ROSY, Diablo, VIP7760, or IBM3270 printer) to make it operate as a remote printer. The hardware options currently supported are defined by the control arguments described below.

The tty_printer_ I/O module can also be used to direct its stream I/O through the syn_ I/O module to another I/O switch (e.g., user_i/o or to a file switch through vfile_).

Entry points in this module are not called directly by users; rather, the module is accessed through the I/O system. It is normally attached through the remote_printer_ I/O module, and all attach options are passed through remote_printer_ to tty_printer_.

ATTACH DESCRIPTION

```
tty_printer_ -control_args
```

CONTROL ARGUMENTS

The following control arguments are optional, with the exception of -comm, -device, and -tty:

-auto_call N
: specifies the phone number, N, to be called via the automatic call unit on the specified communications channel.

-comm STR
: uses the communications I/O module specified by STR. Normally, STR is either tty_ or syn_.

-device STR
: attaches the switch as the device type specified by STR. STR is normally printer or teleprinter.

-horizontal_tab, -htab
: specifies that horizontal tab characters are to be sent to the device.

-physical_line_length N, -pll N
: specifies the physical line length, N, of the output device.

-terminal_type STR, -ttp STR
: STR specifies the terminal type whose conversion, translation, and special tables defined in the user or system terminal type table (TTT) are used to convert and translate input and output to and from the device. If not specified, the default terminal type is used.

-tty STR
: defines the target communications channel to be STR, where STR is an I/O switch name if the communications I/O module is syn_.

-vtab
: specifies that vertical tab characters are to be sent to the device.

OPEN OPERATION

The tty_printer_ I/O module supports stream_input, stream_output, and stream_input_output opening modes.

PUT CHARS OPERATION

The put_chars entry passes the data directly to the communications I/O module without any conversion.

GET CHARS/GET LINE OPERATION

The get_chars and get_line entries pass the operation directly to the communications I/O module.

CONTROL OPERATION

This I/O module passes all undefined control operations to the communications I/O module. In addition, it supports the control operations listed below. Unless otherwise specified, there are no input control structures.

select_device
: selects the device characteristics for which output is next directed. The device is the one associated with the I/O switch by the -device option at attachment. The input structure is of the form:

```
dcl device char(32);
```

runout
: transmits any data stored in the output buffer.

hangup_proc
: sets up a specified event call channel to be signalled over, and a procedure to be called, if the communications channel hangs up. The hangup_proc input structure has the following form:

```
dcl 1 hangup_proc aligned,
      2 entry     entry variable,
      2 datap     ptr,
      2 prior     fixed bin;
```

where:

entry
    is the entry to call when a hangup is detected.

datap
    is a pointer to data for the hangup procedure.

prior
    is the ipc_ event call priority to be associated with hangup notification.

reset
    sets the ^edited mode of output conversion and enables the tabs and vertsp modes if required by attachment options.

get_error_count
    returns the current count of errors detected since attachment. The input structure is of the form:

```
dcl error_count fixed bin;
```

hangup
    is used to hang up the device communications connection. This control operation is trapped if the communications I/O module is syn_, otherwise it is passed on.

MODES OPERATION

This I/O module passes all modes operations to the communications I/O module.

NOTES

This I/O module is normally attached through a remote device I/O module (e.g. remote_printer_ or remote_teleprinter_.) Attachment to tty_printer_ is specified in the remote_device attach description by "-terminal tty_printer_" along with any attach options listed above. The -device attach option is supplied by the remote_device I/O module.

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 the Programmer's Reference Manual 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}
```

ARGUMENTS

path
: is the absolute or relative pathname of the file.

CONTROL ARGUMENTS

are arranged below according to the file types for which they are appropriate. The categories following are: any file type, any file type except indexed, blocked files, indexed files, sequential files, and unstructured files.

For any type of file, control_args can be chosen from the following:

-extend
: specifies extension of the file if it already exists. The position is normally set at the end of file. This control argument is only meaningful with openings for output or input_output; otherwise, it is ignored.

-old
: indicates that a new file is not to be created if an attempt is made to open a nonexistant file for output, input_output, or update. If the file does not exist, the user is informed of this at open time.

For any file type except indexed, control_args can be chosen from the following:

-append
: in input_output openings, this control argument 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.

-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.

For blocked files, control_args can be chosen from the following:

-share {N}
: allows a 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.) If specified, N is the maximum time in seconds that this process waits to perform an operation on the file. A value of -1 means the process may wait indefinitely. The default value of N is 1.

-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_end
: permits positioning beyond end of file (last record or byte written) and then appending to the file without encountering an error. Instead, the end of file position is advanced leaving any intervening bytes zero, or leaving record positions beyond the previous end of file with the appearance of being logically deleted (see "Logically Absent Records" below).

For indexed files, control_args can be chosen from the following:

-share {N}
: see the -share control argument under blocked files above.

-dup_ok
: indicates that the creation of duplicate keys is permitted (see "Duplicate Keys" below).

-exclusive
: causes the exclusion of all shared references in other openings for the duration of this opening. The file must be opened for modification. This control argument conflicts with the -share control argument. Shared readers in other openings are otherwise excluded only while an update operation is in progress, or while the file has been explicitly locked to exclude readers via the set_file_lock order.

-stationary
: causes newly created records to be of the stationary type, and forces vfile_ to maintain a reference count during the addition and removal of keys from such records. The use of this control argument is recommended for applications with multiply keyed records, or when record level synchronization is required (see "Multiply Keyed Records" and "Record Locks" below).

-transaction tcf_sw, -trans tcf_sw
: indicates that all operations on this switch are performed within transactions associated with a control file attached to the I/O switch named tcf_sw. The file must be indexed with stationary type records (see the transaction_call command and transaction_call_ subroutine in the Transaction Processing manual for more information).

* For unstructured files, control_args may be chosen from the following:

-header {N}
: 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 that 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.

-no_trunc
: indicates that a put_chars order into the middle of an unstructured file (stream_input_output) is permitted, and no truncation is to occur in such cases. This control argument 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.

-no_end
: see the -no_end control argument under blocked files above.

The -extend, -append, and -no_trunc control arguments conflict; only one of these 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. The mode must be compatible with any control arguments given in the attach description.

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.

MODES OPERATION

This operation is not supported.

CONTROL OPERATION

The following orders are supported by the vfile_ I/O module. The orders in the first column are those most often used. The remaining orders include various features of indexed files that require somewhat more knowledge of internal file structure than is expected of most users. The letters following the names of the orders indicate the type of file for which the orders apply. The letters and their meanings are: A - any file, B - blocked file, I - indexed file, S - sequential file, and U - unstructured file.

add_key	I	read_position	B,S,U	
delete_key	I	reassign_key	I	*
error_status	I,S	record_status	B,I,S	
exclude	I	seek_head	I	*
file_status	A	select	I	
get_key	I	set_file_lock	I	
max_rec_len	B	set_wait_time	I	
min_block_size	I	truncate	B,S,U	

Detailed descriptions of the orders are given, in alphabetical order, at the end of the general discussion.

CONTROL OPERATIONS FROM COMMAND LEVEL

All control orders can be performed using the io_call command. The general format is:

```
io_call control switch_name order {optional_args}
```

ARGUMENTS

order
is any of the control orders supported by vfile_, or the short name of the control order, if it has one.

optional_args
are required for certain orders as indicated in the descriptions of the orders.

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 order. In cases where an invalid multiple opening does occur, I/O operations 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 wait up to the limit specified by N in the -share control argument or from the last set_wait_time order. If the operation is not carried out because of the limit N, the code error_table_$file_busy is returned.

   Two codes 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 order) has already been inserted (by some other opening).

   The code error_table_$asynch_change is returned on a subsequent reference to an item previously referenced in the same transaction, if an asynchronous
* change is detected.

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 (blocked, indexed, sequential, and unstructured) may be obtained with the vfile_status command or vfile_status_ subroutine.

The remainder of this discussion will treat each of the four file structures separately.

BLOCKED FILES

The following paragraphs describe exceptions and provide information applicable to blocked files. For general information on the subjects mentioned here, see the Programmer's Reference Manual.

Position Operation

In addition to the standard iox_ positioning, another type of positioning is available with 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 whose ordinal position (0, 1, 2, ...) is given. The zero position is just beyond the file header.

Write Operation

The write operation is supported in files open for update. The effect is to append a record to the file or replace the next record, depending on the next record position.

Rewrite Operation

No record may be written over 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

Deletions are supported by marking the current record as logically deleted. If the last record is deleted, the end of file position is moved back to just beyond the previous nondelcted record.

Logically Absent Records

Within the limits of efficiency imposed by the choice of implementation, the concept of deletion is interchangeably defined for the different types of files. In certain situations, however, it is necessary to distinguish between the various ways in which a record may appear to be absent from a file.

In a blocked file the space occupied by deleted records is reusable. The appearance of a deletion is less absolute than in a sequential file, for example. For the purpose of this discussion, records implicitly allocated when the file is extended with the -no_end attach control argument are equivalent to those that have been deleted. When a record is deleted, its position is reserved and marked as logically absent. The end of file position is maintained just beyond the last nondeleted record.

Records that have been marked as logically absent are made invisible to the user in most situations, so that the distinction between logical and absolute deletion can often be disregarded. The exception to this is that the position operation and the get_key and seek_head orders permit one to locate a logically deleted record without regard to the logical presence or absence of records. Operations that reference the current record treat its being logically absent as an error, returning the code error_table_$no_record. Operations referencing the length or contents of the next record, on the other hand, do not treat its being logically absent as an error, but scan sequentially for the next instance of a nondeleted record or end of file. Records are scanned in ascending order, except immediately following a successful backward position skip operation, in which case scanning is done in reverse order.

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. 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.

Any type of file may be properly adjusted with the vfile_adjust command if an interrupted opening has occurred.

INDEXED FILES

The following paragraphs describe exceptions and provide information applicable to indexed files. For general information on the subjects mentioned, see the Programmer's Reference Manual.

Position Operation

The type 2 position operation is not supported for indexed files.

Rewrite Operation

For indexed files, if the current record is not of the stationary type, and the current position is "outside" the index (e.g., after a delete_key order or use of the record_status order with the locate switch), then the new record length must be small enough to fit in the old record without reallocation; otherwise, the code error_table_$long_record is returned.

Delete Operation

In an indexed file, stationary records having multiple keys (reference count>1) are deleted by being marked as logically deleted, until a later time when garbage collection automatically takes place (see "Logically Absent Records" below).

Logically Absent Records

Records that have been marked as logically absent are made invisible to the user in most situations, so that the distinction between logical and absolute deletion can often be disregarded. The exception to this is that the position order permits one to locate a logically deleted record without regard to the logical presence or absence of records. Operations that reference the current record treat its being logically absent as an error, returning the code error_table_$no_record. Operations referencing the length or contents of the next record, on the other hand, do not treat its being logically absent as an error, but scan sequentially for the next instance of a nondeleted record or end of file. Records are scanned in ascending order, except immediately following a successful backward position skip operation, in which case scanning is done in reverse order.

For records that are not of the stationary type, or for some cases of stationary records, the effect of a deletion is absolute, and the record's storage is immediately recovered. In the case of multiply keyed stationary records (reference count>1), however, the record is logically deleted. The presence of such records is automatically masked until garbage collection occurs. This behavior ensures that there are never inconsistencies of the form where an index entry refers to a record that is no longer valid (see "Multiply Keyed Records" below).

Garbage collection of keys belonging to logically deleted records takes place automatically upon their detection in a file opened for modification. Only when the last reference has been removed is the body of a logically deleted record entirely freed, so that every bit of its storage can be reused.

In shared openings, garbage collection of the record's last key and stationary header is postponed until the effective collection delay time since the logical deletion has elapsed. This is done to ensure that any passive reference to the record prior to its
* deletion can find the record header afterwards and detect the asynchronous change. The delay should be set by the user, with the set_wait_time order, to a duration that exceeds the maximum time a transaction (or a single vfile_ operation) can be in
* progress or until an intermediate call is made to transaction_call_$status with the verify option.

Scanning over logically absent records in an indexed file is subject to one additional constraint not applicable to blocked files. Specifically, after a successful seek_head or get_key order with rel_type=0 (head must match), scanning is limited to the last key whose head matches that previously specified. This only applies to an operation that references the next record in the immediately following operation.

A temporary form of logical deletion is also available through the use of the select and exclude orders with indexed files. Records made to appear absent through these orders are not altered in any way, so there is certainly no reuse of storage in this case. Except for leaving the file statistics unchanged, this form of logical deletion may be regarded as absolute, insofar as all subsequent operations (including position) behave as if such logically absent records and their keys never existed in the first place.

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 order locates the first instance of a set of duplicate keys. A write_record order 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. Orders that can advance the next record position (i.e., read_record, rewrite_record, get_key, and position with a type argument of 0) permit one to locate intermediate instances of duplicate keys.

Multiply Keyed Records

The vfile_ I/O module allows any number of keys to be associated with a given record in an indexed file through the use of the add_key control order. In conjunction with the use of duplicate keys, arbitrary many-to-many relationships can be established between keys and records. The appearance of each of a record's keys is completely independent of the existence of any other keys, i.e., there is no distinction between primary and secondary keys.

The orders delete_key and reassign_key permit keys to be removed and reassigned from one record to another. Information about which and how many keys belong to a given record may be obtained with the get_key and record_status orders.

When duplicate keys are allowed, it may be necessary to specify more than one key in order to uniquely identify a record. The use of successive select orders permits one to achieve this effect by progressively narrowing down a cross section of the file to be made visible. Conversely, the exclude order permits records to be found by the process of elimination.

File statistics are maintained giving the number of keys, number of duplicate keys, and number of records separately, as well as the total length of keys, and the total length of duplicate keys. With regard to these values, the count of duplications does not include the first instance of a key. A count of keys is maintained for each record as an option, specified by the -stationary control argument.

In general, when multiple keys are present in a file subject to random updates, it is recommended that the -stationary control argument be used. Otherwise it is the user's responsibility to maintain a consistent relationship between the index and the records when records are either deleted or reallocated. This problem can be avoided when stationary records are used under the -stationary control argument.

Stationary records have the property that the descriptor defining one's location never changes during an update. If such a record must be reallocated, the new address of the contents of the record is stored in the header of the initial allocation, and an indication is made that the record is found indirectly. The reference count of keys kept with each stationary record permits deletions to take place in a manner that postpones the removal of an allocation until all references to it have been removed.

Record Locks

Record locks provide a basis for synchronizing concurrent access at the individual record level. The setting and clearing of record locks is explicitly controlled by the user via the record_status order.

There are two types of records that may be locked. The more general facility requires that records be of the stationary type, created under the -stationary attach control argument. Each stationary record has a lock, modifier code, and a time of its last modification. It is a fundamental property of such a record that the storage occupied by its synchronization elements resides in a fixed location for the life of the record.

Thus, it is never necessary to lock the file in order to lock a stationary record if its descriptor is known. Use of the time last modified permits purely passive synchronization (i.e., without locking) to be done at the record level. This involves the use of a protocol such as the following:

1. Obtain the record's time_last_modified with record_status, which may abort if the record lock remains set for longer than the allowed wait_time currently in effect. The record lock is always examined before returning the record's time_last_modified.

2. Reference the record's contents via its pointer, obtained from the from the previous call to record_status.

3. Use the block_ptr obtained from record_status to reexamine its time_last_modified. If unchanged, the passive reference is verified, and the operation is done. On the other hand, if the time_last_modified has changed, go back to step 1.

In order to synchronize access at the record level without having to lock the file, it is necessary that record locks have a fixed location. Stationary records should therefore always be used, except when it is known that there will be no deletions or rewrites requiring reallocation.

A different implementation of locks applies to nonstationary records. The modifier and time_last_modified are not supported, and the record lock is only supported if the user is careful to maintain allocations of sufficient size.

When the capacity of an allocated record block exceeds its contents by at least four bytes, the last word of the block is treated as a record lock. A nonzero lock identifies the process that set it. The user can ensure that record allocations leave room for a lock by using the min_block_size order with a residue specification of at least four bytes.

All orders that reference the length or contents of an existing record (e.g., seek_key, but not seek_head) also check the lock of the record (if one exists). If the record is not locked, the operation proceeds normally. Otherwise, the returned error code reflects the state of the lock, indicating that the contents of the record may be in an inconsistent state. In this case, if the order does not explicitly involve changing the file, it proceeds normally and the returned code is: error_table_$record_busy, if the record is locked by another live process; error_table_$lock_is_invalid, if the record's lock is set, but not by an existing process; or error_table_$locked_by_this_process, if the record is locked in the caller's process.

Attempting a rewrite_record or delete_record order on a record locked by another process has no effect other than to return the code error_table_$record_busy (file is unchanged). If the lock is invalid, these orders return the code error_table_$invalid_lock_reset and zero the lock. If the lock was set by the caller, the code returned is error_table_$locked_by_this_process. In either of these latter cases, the operation is successful.

If a record has been locked by a transaction, the above error codes are suppressed, except for the case of record_busy on an attempt to alter a record locked by a live process. If the record is not locked by another live process and the record's modifier can not be found in the transaction control file, or if the caller has not used the -transaction attach option, then the code error_table_$higher_inconsistency is returned.

When a record that is locked by the user's process is rewritten, its lock remains set, as long as the minimum block size specification currently in effect leaves enough room for a record_lock.

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 opening for input only. 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, unless stationary type records are used. (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 is automatically deleted from the file.

Any type of file may be properly adjusted with the vfile_adjust command if an interrupted opening has occurred.

SEQUENTIAL FILES

The following paragraphs describe exceptions and provide information applicable to sequential files. For general information on the subjects mentioned, see the Programmer's Reference Manual.

Position Operation

The type 2 position operation is not supported for sequential files.

Write Operation

The write operation is supported in files open for update. The effect is to append a record to the file or replace the next record, depending on the next record position.

*

Rewrite Operation

For sequential files, 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.

*

Delete Operation

For sequential files, the record is logically deleted but the space it occupies is not recovered.

*

Logically Absent Records

In a sequential file, the logical effect of a deletion is absolute in the sense that the result is the same as if the record had never been written in the first place. No subsequent operation can make the presence of such a record known to the user, although the storage that it occupies is not reused for the life of the file. The logical position of a deleted record is not reserved, but is assigned to the following nondeleted record, with the logical positions of subsequent records diminished accordingly.

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 opening for input only. 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.

Any type of file may be properly adjusted with the vfile_adjust command if an interrupted opening has occurred.

*

UNSTRUCTURED FILES

The following paragraphs describe exceptions and provide information applicable to unstructured files. For general information on the subjects mentioned, see the Programmer's Reference Manual.

Position Operation

In addition to the standard iox_ positioning, another type of positioning is available with files that are open for input or input_output. When the type argument of the iox_$position entry point is 2, this specifies direct positioning to the byte whose ordinal position (0, 1, 2,...) is given. The zero position is just beyond the file header, if a header is present.

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 opening for input only. 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.

Any type of file may be properly adjusted with the vfile_adjust command if an interrupted opening has occurred.

CONTROL ORDER DESCRIPTIONS

The remainder of this section describes the control orders, which are arranged alphabetically. Information concerning the usage of control orders from command level is located at the end of each description. The short names of the orders (where they exist) are provided in the format lines; use of either the long or the short name is acceptable in a command line. For general, rather than order specific, information, see "Control Operations from Command Level" earlier in this description.

Name: add_key, ak

The add_key order creates a new index entry with a given key and record descriptor.

The I/O switch must be open for direct_output, direct_update, keyed_sequential_output, or keyed_sequential_update. Current and next record positions are unchanged.

Associations may be formed between any number of keys and a single record via this order. Duplicate keys may be added if the file is attached with the -dup_ok control argument, or if the file already contains duplications; otherwise, the code error_table_$key_duplication is returned (see "Duplicate Keys" above.)

When the -stationary control argument is used, the addition of a key to a stationary type record increments the record's reference count, in addition to inserting a new index entry (see "Multiply Keyed Records" above under "Indexed Files").

The current implementation restricts a user from adding more than 65,535 (2**16-1) keys to a single stationary record, returning the code error_table_$too_many_refs.

This page intentionally left blank.

This order and the delete_key, reassign_key, and get_key orders do not reference the length or contents of a record. This permits one to avoid the use of actual records altogether in any given indexed file.

For this order, the info_ptr argument must point to a structure (declared in the include file ak_info.incl.pl1) of the following form:

```
dcl 1 ak_info              based (info_ptr),
      2 flags              aligned,
        3 input_key        bit(1) unal,
        3 input_desc       bit(1) unal,
        3 mbz              bit(34) unal,
      2 descrip            fixed bin(35),
      2 key_len            fixed bin,
      2 key                char(256 refer(ak_info.key_len));
```

STRUCTURE ELEMENTS

input_key
: indicates whether the new key is given in the info structure. (Input)
"0"b indicates that the current key for insertion is the new key. If this value is undefined, the code error_table_$no_key is returned.
"1"b indicates that the key to be added is the key_string contained in this info structure.

input_desc
: indicates whether the new descriptor is given in the info structure. (Input)
"0"b indicates that the current record defines the new descriptor. If the current record is undefined, the code error_table_$no_record is returned.
"1"b indicates that the user-supplied descriptor in this info structure is the new descriptor.

mbz
: must be set to zero by the user. (Input)

descrip
: is used only if the variable input_descrip is set to "1"b. (Input) The descriptor is stored into the index together with its associated key. Any 36-bit quantity may be supplied, although in general this number is a result of a previous record_status or get_key order. Descriptors are used by operations that reference the contents or length of a record, in order to obtain the record's address.

key_len
: is the length of the key_string. (Input)

key
: is used only if ak_info.input_key is set to "1"b. (Input) It defines the key to be added to the index with the appropriate record descriptor.

COMMAND LEVEL

Syntax:

```
io_call control switch_name ak {args}
```

where:

flags
: is a string of two bits corresponding to the switch settings for input_key and input_descrip. If one argument is given, it is interpreted as a key to be added to the current record, i.e., flags defaults to "10"b.

key
: is a character string that must be given if flags.input_key is set.

descrip
: is an octal descriptor that must be supplied if flags.input_descrip is set.

Name: delete_key, dk

The delete_key order deletes a specified index entry.

The I/O switch must be open for direct_update or keyed_sequential_update. The current and next file positions are left unchanged, with the following exception: if the deleted index entry is at the next record position, then the next record position is advanced to the following index entry, or becomes undefined in direct openings.

When the -stationary control argument is used, the deletion of a key from a stationary type record decrements its reference count. The user cannot remove the last key in such a case (i.e., causing the reference count to vanish); the code error_table_$last_reference is returned.

For this order, the info_ptr argument may be null, or it may point to a structure whose form is identical to the structure (declared in the include file ak_info.incl.pl1) for the add_key order, in this way:

```
dcl 1 dk_info like ak_info
```

where:

input_key
  indicates whether the key is given in the info structure. (Input)
  "0"b indicates that the key associated with the current file position defines the key of the index entry that is to be deleted. If current position is undefined or outside the index (e.g., after deleting the current key of the current record), the code error_table_$no_key is returned.
  "1"b indicates that the user-supplied key_string defines the key of the entry to be deleted. If no such key is found, the code error_table_$no_key is returned.

input_descrip
  indicates whether the descriptor is given in the info structure. (Input)
  "0"b indicates that the index entry to be deleted is associated with the current record. If the current record is undefined, the code error_table_$no_record is returned.
  "1"b indicates that the entry to be deleted is associated with the user-supplied descriptor. If no such entry exists, the code error_table_$no_record is returned.

descriptor
  is used only if delete_key_info.input_descrip equals "1"b. (Input) The entry that is deleted is the first whose descriptor matches this value, among those entries with the specified key.

key_length
  is the length of the key_string. (Input)

key_string
  if delete_key_info.input_key equals "1"b, this argument defines the key for which the index entry with the specified record descriptor is to be deleted. (Input)

mbz
  must be set to zero by the user. (Input)

If the info_ptr argument is null, the index entry at the current file position is deleted, i.e., the effect is the same as that of setting both arguments, input_key and input_descrip, to "0"b.

NOTES

The interpretation of the descriptor argument as a record locator is not mandatory, since the add_key and reassign_key orders permit the user to set the descriptor portion of an index entry to an arbitrary 36-bit value.

The descriptor itself may be thought of as a one-word record that is read by the get_key order.

COMMAND LEVEL

Syntax:

```
io_call control switch_name dk {args}
```

where args are the same as for add_key above (flags, key, descrip). Optionally, if no arguments are given, the order is equivalent to a delete_key order with no info structure (null info_ptr).

Name: error_status, er

The error_status order is accepted when the I/O switch is open and attached to an indexed or sequential file. The order returns information about the most recent attempt to position beyond either the beginning or the end of file in the current opening.

For this order the info_ptr argument must point to a structure of the following form:

```
dcl 1 error_info      based (info_ptr),
      2 version       fixed bin,
      2 type          fixed bin,
      2 requested     fixed bin(34),
      2 received      fixed bin(34);
```

STRUCTURE ELEMENTS

version
    must be set to one by the user. (Input)

type
    indicates the type of error that has occurred. (Output)
    0   no errors
    1   attempt to position beyond end or beginning of file.

requested
    gives the value of the position skip argument that led to the most recent error. (Output)

received
    gives the actual number of records successfully skipped before encountering end or beginning of file (negative if backwards skip). (Output)

COMMAND LEVEL

Syntax:

```
io_call control switch_name er
```

where this prints the requested and received counts for the most recent skip error.

Name: exclude, ex

The exclude order causes subsequent vfile_ operations to behave as if a subset of records and their keys are absent from an indexed file. The exclude order excludes all of the keys associated with the records identified in the order. This process may remove more keys from the index than are identified explicitly in the order.

The subset of interest may be specified in terms of ranges of keys, a list of record descriptors, or an identifying number for a previously formed subset.

Various items of information that may be returned include a subset number, count of distinct descriptors, or an identifying number for a previously formed subset. However, status_only may not be requested via exclude.

None of the file position designators (current and next record positions) are affected by this order.

If no selection or exclusion is currently in effect (subset number=0), then a new subset will be formed defining the set of record descriptors to be excluded. The subset number in this case will be negative, indicating that a pure exclusion is in effect.

If a pure exclusion is initially in effect, a subsequent exclude order has the effect of enlarging the current subset (i.e., the set of things to be excluded), and the current subset number is unchanged.

If a selection is in effect (subset number > 0), then the effect of a subsequent exclude order is to remove the specified descriptors from the current subset, again leaving the current subset number unchanged.

For this order, the info_ptr argument must point to one of the following structures (all declared in the include file select_info.incl.pl1):

```
dcl 1 common_sl_info          based (info_ptr),
      2 flags                 aligned,
        3 list_type           fixed bin(3) unal,
        3 status_only         bit(1) unal,
        3 output_descriptors  bit(1) unal,
        3 delete_old_subsets  bit(1) unal,
        3 mbz                 bit(11) unal,
        3 version             fixed bin(17) unal,
      2 array_limit           fixed bin(19),
      2 subset_no             fixed bin,
      2 count                 fixed bin(34),
      2 desc_arrayp           ptr;
```

where common_sl_info.desc_arrayp may point to the following structure:

```
dcl desc_array(1:common_sl_info.count) fixed bin(35)
       based (sl_info.desc_arrayp);
```

or:

```
dcl 1 hi_sl_info              based (info_ptr),
      2 common                like common_sl_info,
      2 interval(1:sl_array_limit refer (hi_sl_info.array_limit)),
        3 first_head,
          4 length            fixed bin,
          4 kptr              ptr unal,
        3 last_head,
          4 length            fixed bin,
          4 kptr              ptr unal;
```

or:

```
dcl 1 da_sl_info              based (info_ptr),
      2 common                like common_sl_info,
      2 desc_array(1:sl_array_limit refer(da_sl_info.array_limit))
                              fixed bin(35);
dcl sl_array_limit            fixed bin;
dcl sl_info_version_0         static options(constant) internal
                              fixed bin init(0);
```

STRUCTURE ELEMENTS

flags.list_type
is a code indicating the manner in which this info structure specifies a subset: (Input)

list_type=0
causes the reuse of a subset formed earlier in this opening, whose subset number is given in sl_info.subset_no.

list_type=1
indicates that the subset is specified in terms of ranges of keys, or index intervals, using a structure like hi_sl_info. The code error_table_$no_record is returned if no index entries in the current subset are found in the specified set of intervals.

list_type=2
indicates that a list of descriptors with a structure like da_sl_info will be used to define the subset of interest.

flags.status_only
if set, status information will be returned for the current subset without making any subset changes. (Input)

flags.output_descriptors
if set, causes a sorted list of descriptors for the resulting subset to be output into the structure desc_array. (Input)

flags.delete_old_subsets
if set, and list_type = 1 or 2, causes all existing subsets to be deleted. The current subset number must be 0. (Input)

version
is the version number for this info structure, which should be set to sl_info_version_0. (Input)

array_limit
gives the number of array elements in this info structure. (Input)

subset_no
is an identifying number for the resulting subset, which permits its subsequent reuse in the same opening. A new subset is defined for each select order unless the user explicitly specifies reselection by giving the subset number as an input argument (list_type=0). The default subset is the identity case (i.e., the whole file), denoted by subset_no=0. A negative subset number indicates that the current subset is defined in terms of a set of records to be excluded. (Input/Output)

count
: is the number of distinct record descriptors for the resulting current subset. (Output)

desc_arrayp
: is used only if the flag, output_descriptors, is set. (Input/Output) If null, the required desc_array structure will be allocated in system_free_area, and its address will be returned in desc_arrayp. Otherwise, desc_arrayp is assumed to point to an already allocated structure of sufficient size, in which the sorted list of descriptors (with duplications removed) is returned.

desc_array
: is an optionally returned list of record descriptors in the current subset, sorted and with duplications removed. (Output)

first_head.length
: is the number of bytes in the key string that defines the starting head for this range of keys. (Input)

first_head.kptr
: gives the location of the character string that specifies the first head of this index interval. Every key in the interval must have a head that is greater than or equal its first_head. (Input)

last_head.length
: is the number of bytes in the key string that defines the end of this index interval. (Input) If this number is negative, then one of the following applies:

  last_head.kptr=first_head.kptr
  : indicates that this interval pertains only to keys that exactly match the given first_head.

  last_head.kptr^=first_head.kptr
  : specifies that this is an "open" interval, whose largest key must have a head that is less than the given last_head. The length of the last_head is the absolute value of last_head.length in this case.

  Otherwise, if last_head.length>=0, then a "closed" interval is specified, whose keys must have heads that are less than or equal to the given last_head.

last_head.kptr
: gives the address of the last_head. (Input) If this is equal to first_head.kptr and last_head.length<0, the effect is the same as if both the first and last heads for this interval are the same and have been padded with blanks. Indicating an exact key match in this manner permits the user to avoid having to pad each key to 256 characters, which might require considerably more storage and processing.

da_sl_info.desc_array
: contains a list of record descriptors that define the subset of interest. (Input) The list may be unordered and may contain duplications.

COMMAND LEVEL

Syntax:

```
io_call control switch_name ex {args}
```

where:

-brief, -bf
suppresses the printing of the current subset number, descriptor count, and any error messages except the errors no_operation and bad_arg.

-list, -ls
prints the list of descriptors for the resulting subset.

-delete_old_subsets, -dos
deletes all existing subsets, before the new subset is created. This is incompatible with the -list control argument. The current subset number must be 0.

(-head, -key) interval_spec1 ({-or, -or_key} interval_spec2 ({-or, -or_key} interval_spec3...))
specifies the subset in terms of ranges of keys where:

-head
indicates that the following interval starts with the first key whose head is greater than or equal to the specified first_key. This control argument is the default.

-key
indicates that the following interval is defined as those keys exactly matching the specified first_key . A last_key may not be given for this interval.

interval_spec
is of the form:

```
first_key ({-thru, -to} last_key)
```

where:

first_key
is a character string that defines the starting point for a range, or interval of keys.

last_key
is a character string giving the head that defines the end of an index interval. Its default value is that of the first_key.

-thru
separates the first and last key specifications for a closed index interval, i.e. keys with heads that are less than or equal to the last_key.

-to
separates the first and last key specifications for an open index interval. Keys whose head is equal to or greater than the last_key are not included in this case.

-or
delimits the start of another interval specification that is of the default type; i.e., the following interval starts with the first key whose head is greater than or equal to the next first_key specification.

-or_key, -ork
delimits the start of an interval specification of the type that follows the -key control argument.

{-or, -or_key} interval_spec1 (interval_spec2...)
is the same as above, except:

-or
if the first argument, it is taken as the default delimiter, and should be omitted between interval specifications following on this command line.

-or_key, -ork
if the first argument, it is taken as the default delimiter, and should be omitted between interval specifications following on this command line.

{-desc} descriptor_list
specifies the subset in terms of a list of record descriptors where:

-desc, -ds
indicates that the subset specification for this order is in terms of a list of descriptors that follows.

descriptor_list
is a list of octal record descriptors.

{-reset} subset_number
specifies the subset in terms of an identifying number for a previously formed subset where:

-reset, -rs
indicates that a previously formed subset is to be reused. If no subset_number follows, subset 0 is assumed.

subset_number
is the identifying subset number for the subset to be reused.

Name: file_status, fs

The file_status order is accepted when the I/O switch is attached (open or closed). Various items of information about the file are returned. The info_ptr argument must point to a structure identical to one of those required for the vfile_status_ subroutine.

COMMAND LEVEL

Syntax:

```
io_call control switch_name fs
```

where the output is the same as that of the vfile_status command.

Name: get_key, gk

The get_key order returns both the key and the record descriptor for the specified index entry in a file opened for keyed_sequential_input or keyed_sequential_update.

An index entry may be specified in terms of the current or next positions, by association with a given descriptor, or by bearing the specified relation to a given key. If the requested index entry does not exist, one of the codes, error_table_$no_record or error_table_$no_key is returned, whichever is appropriate. Optionally, the user can indicate that the final position is to be left unchanged. Otherwise the current and next record positions are set to the specified index entry.

For this order, the info_ptr argument must point to a structure (declared in the include file ak_info.incl.pl1) of the following form:

```
dcl 1 gk_info                     based (info_ptr),
      2 flags                     aligned,
        3 input_key               bit(1) unal,
        3 input_desc              bit(1) unal,
        3 desc_code               fixed bin(2) unal,
        3 position_specification  unal,
          4 current               bit(1) unal,
          4 rel_type              fixed bin(2) unal,
          4 head_size             bit(9) unal,
        3 reset_pos               bit(1) unal,
        3 mbz                     bit(8) unal,
        3 version                 fixed bin(8) unal,
      2 descrip                   fixed bin(35),
      2 key_len                   fixed bin(17),
      2 key                       char(256 refer(gk_info.key_len));
  dcl gk_info_version_0 fixed bin static init(0);
```

STRUCTURE ELEMENTS

input_key
: if set to "1"b indicates that the key in this info structure is an input argument, which must bear the specified relationship to a key in the index. Otherwise the key of interest is located through either the next or the current position, according to the setting of flags.current. (Input)

input_desc
: if set to "1"b indicates that the desired index entry must have a descriptor that is equal to that given in this structure as an input argument. Otherwise the descriptor may either have any value or must be that of the current record, as specified by the setting of flags.desc_code. (Input)

desc_code
: is used only if flags.input_desc="0"b to specify the desired descriptor portion of an index entry. If desc_code=0, then any descriptor is satisfactory. If desc_code=1, then the index entry of interest must be associated with the current record. No other desc_code settings are defined in this implementation. (Input)

current
: applies only if flags.input_key="0"b. If set to "1"b, this indicates that the current index entry is the one of interest. This control argument conflicts with the setting of flags.input_desc to "1"b. Otherwise, if flags.current="0"b, the next record position is used as a starting point to find the desired index entry by scanning for the next occurrence of the desired descriptor, until end of file is encountered, or until the next key ceases to satisfy an immediately preceding successful seek_head order. Index entries are always scanned in ascending order, except immediately after a successful backwards position skip operation, when scanning is done in reverse. (Input)

rel_type
: applies only if flags.input_key="1"b. This indicates the desired relationship that the head of a key in the index must have with the key_string given in this info structure. Allowed values and their meanings are the same as those for the seek_head order. (Input)

head_size
: applies only if flags.input_key="1"b, specifying the number of characters in the key_string contained in the desired head. (Input)

reset_pos
: if set to "1"b, the state of the file position designators will be left unchanged by this order. Otherwise the current and next record positions will be left at the specified index entry. (Input)

version
: is the version of this info structure, which should be set to gk_info_version_0. (Input)

descrip
: is the record locator portion of the specified index entry. If flags.input_desc="1"b, this is an input argument. Descriptors may also be input to the control orders add_key, delete_key, reassign_key, and record_status (see "Notes" below). (Input/Output)

key_len
: is the length of the key for the specified index entry. (Input/Output)

key
: if flags.input_key="1"b, this is an input argument that contains the desired key head. The value that is returned is the key of the specified index entry. (Input/Output)

mbz
: must be set to zero by the user. (Input)

NOTES

If flag.input_key is equal to "1"b, both head_size and key_len should be initialized to zero or assigned an appropriate value.

COMMAND LEVEL

Syntax:

```
io_call control switch_name gk {args}
```

where:

-brief, -bf
: suppresses printing of the key, its descriptor, and any error messages except for the errors no_operation and bad_arg.

-head key_string ({-rel_type} N)
: indicates that the following argument is to be taken as the key giving the head that must bear the specified relationship to the key of the desired index entry. This argument need not be given if the key_string cannot be confused with one of the optional arguments to this order.

  key_string
  : if specified, this is the string that defines the key portion of the index entry of interest. If no key_string is specified, and -cur_pos is not given, then the next record position is used.

  -rel_type, -rel
  : applies only when a key_string is specified. This argument must be followed by a number that defines a valid relationship between the given key_string and the head of a key in the index. If not specified, -rel 0 is assumed, when applicable.

N
: is the first N characters of the key. The allowed values are:
  0 head = search_key
  1 head >= search_key
  2 head > search_key
  See the seek_head order for more information.

-cur_pos
: indicates that the index entry of interest is at the current record position. This control argument conflicts with a key_string specification.

-current, -cur
: specifies that the desired index entry belongs to the current record. If neither this nor the -desc control argument is given, the first index entry encountered that satisfies the key_specification is specified by default.

-desc DESCRIPTOR, -ds DESCRIPTOR
: specifies that the desired index entry has a given descriptor, which must be the next argument. DESCRIPTOR is an octal record descriptor, like those returned by this order.

-reset, -rs
: causes the final position to be left unchanged. If not specified, the final positions correspond to the specified index entry.

-substr offset{,length}
: specifies a substring of the key to be returned where OFFSET is the starting character position of the key to be returned, and LENGTH is the length of the part of the key to be returned. If LENGTH is omitted, the entire tail of the key is returned.

If this order is issued through an io_call active function, only the key is returned. Use of the -brief control argument suppresses any error messages except for the no_operation and bad_arg errors.

If no arguments are supplied, the next index entry is located.

Name: max_rec_len, mx

The max_rec_len order is accepted when the I/O switch is open and attached to a blocked file. The order 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 bin(21),    /*output*/
      2 new_max_recl  fixed bin(21);    /*input*/
```

COMMAND LEVEL

Syntax:

```
io_call control switch_name mx {arg}
```

where:

arg
: is an integer specifying a new maximum record length.

This prints the old maximum record length.

Name: min_block_size, mb

The min_block_size order determines the minimum size for blocks of record space that are subsequently allocated by write_record or rewrite_record operations (documented in the iox_ subroutine). The specification remains in effect for the duration of the current opening or until another call to this order is issued. The I/O switch must be attached to an indexed file open for output or update.

For this order, the info_ptr argument must point to a structure of the following form:

```
dcl 1 min_blksz_info   based (info_ptr),
      2 min_residue    fixed bin(21),
      2 min_capacity   fixed bin(21);
```

STRUCTURE ELEMENTS

min_residue
: specifies the minimum unused capacity of a record block (bytes); i.e., the difference between the record's length and the maximum length it can attain without requiring reallocation. (Input)

min_capacity
: specifies the minimum total record capacity (bytes); i.e., the maximum length that the record can attain without requiring reallocation. (Input)

When the I/O switch is initially opened, both these parameters are set to zero.

The current implementation imposes the following constraints on allocated record blocks:

1. The minimum allocation is eight full words, including two header words for the block length and record length. Six more words of overhead are required for stationary type records. The minimum nonnull record capacity is, therefore, 24 bytes, or 0 bytes in the case of stationary records.

2. The size of an allocated block is always an even number of full words, i.e., a multiple of eight bytes.

The formula below gives the allocation size, block_words, used for a rewrite_record, write_record, or record_status order with a given buffer length, buff_len:

block_words = 0 (no allocation if and only if buff_len and min_residue and min_capacity all are equal to 0. A nonnull allocation is always left when rewriting a stationary record or upon record creation under the -stationary attach option.)

otherwise:

block_words = max (8,(max(buff_len + min_residue, min_capacity) + 7) / 8)

COMMAND LEVEL

Syntax:

```
io_call control switch_name mb {args}
```

where:

min_res
is an integer. The default is 0.

min_cap
is an integer. The default is 0.

Name: read_position, rp

The read_position order is accepted when the I/O switch is open and attached to a nonindexed file. The order 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 bin(34),   /*output*/
      2 last_position   fixed bin(34);   /*output*/
```

COMMAND LEVEL

Syntax:

```
io_call control switch_name rp
```

where this prints the next record (or byte) and end of file positions.

Name: reassign_key, rk

The reassign_key order causes the descriptor portion of a specified index entry to be replaced with a given value.

The I/O switch must be open for direct_update or keyed_sequential_update. The file position designators are not changed.

When the -stationary control argument is used, the reference counts of any stationary records involved are adjusted accordingly, as described for add_key and delete_key. The code, error_table_$last_reference is returned if the user is prevented from removing a record's last key.

The current implementation restricts a user from adding more than 65,535 (2**16-1) keys to a single stationary record, and returns the code error_table_$too_many_refs.

For this order, the info_ptr argument must point to a structure (declared in the include file ak_info.incl.pl1) of the following form:

```
dcl 1 rk_info                based (info_ptr),
      2 flags                aligned,
        3 input_key          bit(1) unal,
        3 input_old_desc     bit(1) unal,
        3 input_new_desc     bit(1) unal,
        3 mbz                bit(33) unal,.
      2 old_descrip          fixed bin(35),
      2 new_descrip          fixed bin(35),
      2 key_len              fixed bin,
      2 key                  char(256 refer(rk_info.key_len));
```

STRUCTURE ELEMENTS

input_key
: indicates whether the key is given in the info structure. (Input)
  "0"b indicates that the index entry to be reassigned has as its key the current key for insertion. If undefined, the code error_table_$no_key is returned.
  "1"b indicates that the key_string argument defines the key portion of the index entry to be reassigned. If the key_string is not found in the index, the code error_table_$no_key is returned.

input_old_desc
: indicates whether the old descriptor is given in the info structure. (Input)
  "0"b indicates that the entry to be changed is associated with the current record. If the current record is undefined, the code error_table_$no_record is returned.
  "1"b indicates that the old_descrip argument defines the descriptor portion of the index entry to be changed.

input_new_desc
: indicates whether the new descriptor is given in the info structure. (Input)
  "0"b indicates that the specified index entry is to be reassigned to the current record. If the current record is undefined, the code error_table_$no_record is returned.
  "1"b indicates that the argument new_descrip is to supply the new value for the descriptor portion of the specified index entry.

old_descrip
: is used only if rk_info.input_old_desc equals "1"b. (Input) The entry that is reassigned is the first whose descriptor matches this value, among those index entries with the specified key.

new_descrip
: is used only if rk_info.input_new_desc equals "1"b. This value replaces the old descriptor of the specified index entry. (Input)

key_len
    is the length of the key_string. (Input)

key
    if rk_info.input_key equals "1"b, this argument defines the key for which the index entry with the specified descriptor is to be reassigned. (Input)

COMMAND LEVEL

Syntax:

```
io_call control switch_name rk flags {args}
```

where:

flags
    is a string of three bits corresponding to the switch settings input_key, input_old_desc, input_new_desc.

args
    can be chosen from the following:

    key
        is a character string that must be given if flags.input_key is set.

    old_descrip
        is an octal number required if flags.input_old_desc is set.

    new_descrip
        is an octal number required if flags.input_new_desc is set.

Name: record_status, rs

The record_status order returns information about a specified record in an indexed, sequential, or blocked file, and optionally permits the user to manipulate the lock of the record or to allocate an empty record or both (indexed files only).

An argument is provided that permits the user to entirely avoid using the index in accessing and creating records (see "Notes" below).

In blocked and sequential files, the current and next record positions may optionally be set to a given record number.

The I/O switch must be open and attached to a structured file. The next record position is not altered or used by this order, unless the locate_pos_sw flag is set (unindexed files only). The current record position is always set to the record referenced.

If the file is sequential or blocked, the only nonzero flag setting permitted is locate_pos_sw.

The I/O switch must be open for output or update in order to lock, unlock, or create a record.

For this order, the info_ptr argument must point to a structure (declared in the include file record_status.incl.pl1) of the following form:

```
dcl  1 rs_info                  based (info_ptr) aligned,
       2 version                fixed bin,
       2 flags                  aligned,
         3 lock_sw              bit(1) unal,
         3 unlock_sw            bit(1) unal,
         3 create_sw            bit(1) unal,
         3 locate_sw            bit(1) unal,
         3 inc_ref_count        bit(1) unal,
         3 dec_ref_count        bit(1) unal,
         3 locate_pos_sw        bit(1) unal,
         3 mbz1                 bit(29) unal,
       2 record_length          fixed bin(21),
       2 max_rec_len            fixed bin(21),
       2 record_ptr             ptr,
       2 descriptor             fixed bin(35),
       2 ref_count              fixed bin(17),
       2 time_last_modified     fixed bin(71),
       2 modifier               fixed bin(34),
       2 block_ptr              ptr unal,
       2 last_image_modifier    fixed bin(35),
       2 mbz2(1)                fixed bin;

dcl  rs_info_version_2 static internal fixed bin init(2);
```

STRUCTURE ELEMENTS

version
    is provided for compatibility with possible future versions of this info structure. (Input) The user should set this argument to rs_info_version_2.

lock_sw
    indicates whether an attempt is made to lock the specified record within the wait time limit given at attachment or subsequently set via the set_wait_time order. (Input)
    "1"b yes
    "0"b no

Possible error codes are:

error_table_$invalid_lock_reset
error_table_$locked_by_this_process
error_table_$record_busy
error_table_$no_room_for_lock
error_table_$higher_inconsistency

The code no_room_for_lock is returned if the allocated record block is too small to contain a lock. (See "Records Locks" above under "INDEXED FILES".) The code higher_inconsistency is returned if the lock was set by a transaction which cannot be adjusted, either because it is another transaction in the caller's process, or because the lock was set by a dead process and no tcf entry can be found for the record modifier.

If the first modification of a record in a transaction is to lock (and not unlock) via record_status, then vfile_ automatically initializes an afterimage for the record with a copy of its before image. The record_ptr returned in this case points to the afterimage, so that based manipulations of the record via its pointer do not affect the before image; this guarantees that modifications made in this manner can be rolled back. Afterimage initialization is suppressed by setting rs_info.unlock_sw.

unlock_sw
indicates whether an attempt is made to unlock the record. (Input)
"1"b yes
"0"b no

Possible error codes are:

error_table_$lock_not_locked
error_table_$locked_by_other_process
error_table_$no_room_for_lock

If both lock_sw and unlock_sw are set to "1"b, the locking takes place first and determines the resultant error code. (This permits one to clear an invalid lock in a single operation.)

When the -transaction attach option applies, records can not be unlocked explicitly, since they must be left locked until the transaction completes; unlocking is then done automatically. The only permissible use of setting rs_info.unlock_sw under -trans is in the case where rs_info.lock_sw is also set, in which case, the effect is to suppress setting the record's afterimage and to return a pointer to the before image allocation, leaving the record locked. This usage permits explicit synchronization for avoiding interference and deadlocks without incurring the added expense of preparing an afterimage when one has no immediate intention to rewrite. Based modifications of the record contents should not be made via the record_ptr returned by record_status in this case, but passive based references are allowed. The only valid way to perform based alterations of a record in a transaction is by obtaining a pointer to its afterimage.

create_sw
indicates whether a new record is allocated using the record_len and max_rec_len arguments as input parameters. (Input)
"1"b yes
"0"b no

The contents of the record are set to zero, and its lock is set in the same operation if lock_sw equals "1"b. Depending upon the setting of locate_sw, the new record may be entered into the index. If locate_sw equals "0"b, the current key for insertion is added to the index as a key for the new record. Otherwise, no index entry is created and the key for insertion becomes undefined.

locate_sw
indicates how the record of interest is located. (Input)
"0"b if create_sw also equals "0"b, this indicates that the current record position defines the record of interest. Otherwise, the current key for insertion is used. If the relevant position designator is undefined, the code error_table_$no_record or error_table_$no_key is returned, whichever is appropriate.
"1"b if create_sw equals "0"b, this indicates that the descriptor argument is an input parameter defining the location of the record of interest. When such references are permitted in a shared file, users must observe certain protocols to ensure proper synchronization of access at the record level. Record locks are provided for this purpose. If create_sw equals "1"b, this causes the new record to be created without a key.

inc_ref_count
if set to "1"b, the record must be of the stationary type, or the code error_table_$no_room_for_lock is returned. (Input) The effect of setting this flag is to increment the reference count of the record.

The current implementation prevents a user from causing a reference count to exceed 65,535 (2**16-1), returning the code error_table_$too_many_refs.

dec_ref_count
if set to "1"b and the record is of the stationary type, this causes its reference count to be decremented. (Input) Users are not normally expected to manipulate the reference count of a record explicitly in this manner, unless list structures are maintained having direct references to records in terms of their descriptors within other records (see "Multiply Keyed Records" below).

The code error_table_$last_reference is returned when the user is prevented from removing the last reference to a nondeleted record.

locate_pos_sw
if set to "1"b, the current and next record positions are first set to the record whose ordinal position is given in rs_info.record_length. (Input) The file must be either blocked or sequential. If the file is sequential, then the descriptor of the record must also be supplied as an input argument.

record_length
: gives the record's length in bytes. (Input/Output) If create_sw equals "1"b, this argument is input.

max_rec_len
: if create_sw equals "1"b this argument is input and, if nonzero, overrides any minimum block size specification that may currently be in effect (see min_block_size order below). (Input/Output) The returned value gives the maximum length that the record can attain (bytes) without requiring reallocation. When this argument is used as an input parameter, the resultant maximum record length is the smallest number greater than or equal to max_rec_len that corresponds to an implemented (nonzero) block size.

record_ptr
: is a pointer to the first byte of the allocated record, or is set to null if no allocated record exists. (Output)

descriptor
: is a process-independent locator for the specified record. (Input/Output) This value is used as an input argument when locate_sw equals "1"b and create_sw equals "0"b. The actual structure of each descriptor (for indexed or blocked files) is as follows:

```
dcl 1 descrip_struct   based (addr(descriptor)) aligned,
      2 comp_num       fixed bin(17) unal,
      2 word_offset    bit(18) unal;
```

where:

comp_num
: is the multisegment file component number of the segment containing the record.

word_offset
: is the word offset of the block of storage containing the allocated record, relative to the base of its file component.

A zero descriptor designates an unallocated (zero-length) record.

Descriptors may also be arguments to the add_key, delete_key, reassign_key, and get_key orders. Note that at any given time within a single file each record is uniquely located by its descriptor, which remains valid only for the life of a single allocation.

ref_count
is returned only if the record is of the stationary type, in which case this is the reference count of the record. (Output) When the -stationary control argument is used, vfile_ automatically maintains the reference counts of stationary records to reflect the number of keys on each record (see "Multiply Keyed Records" above).

time_last_modified
applies only for stationary records. (Output) Contains a standard system clock time for the most recent modification made to the current record.

modifier
if nonzero, this is the identifying number of a transaction on whose behalf the record was locked. (Input/Output) When rs_info.lock_sw is set, the user should set this value to 0 before calling record_status.

block_ptr
points to the start of the allocated block for the record. (Output) The time_last_modified of a stationary record may be directly examined by using the following structure:

```
dcl 1 stat_block               aligned based (block_ptr),
      2 double words(2)        fixed bin(71),
      2 half_word              fixed(17) unal,
      2 time_last_modified     fixed bin(53) unal;
```

Obtaining the time_last_modified via block_ptr avoids the expense of another call to record_status.

last_image_modifier
is the transaction number for the most recent modification of this record. (Output) If zero, then the most recent modification was not made under the -transaction option.

mbz1, mbz2
must be set to zero by the user. (Input)

Notes

If locate_sw is set to "1"b, the resultant current record position moves "outside" of the index in the sense that there is if so, a subsequent rewrite_record or delete_record order behaves differently from the usual case. The difference is that no corresponding index entry is changed or deleted to reflect the change to the record.

Extreme caution must be exercised when using the orders that take a descriptor as an input argument, especially in a shared environment. The user is responsible for ensuring that previously obtained descriptors and pointers are still valid when they are used. Also, it is important to maintain the index in a consistent state, i.e., each index entry should designate a valid record if a record reference may be attempted.

COMMAND LEVEL

Syntax:

```
io_call control switch_name rs {args}
```

where:

-brief, -bf
: suppresses the printing of status information. If omitted, record_length, max_rec_len, record_ptr, and record descriptor (in octal) are printed; in addition, time_last_modified, reference_count, and modifier are printed for stationary type records.

flags, -pos
: is a string of seven bits, corresponding to the switch settings for lock_sw, unlock_sw, create_sw, locate_sw, inc_ref_count, dec_ref_count, and locate_pos_sw. This argument defaults to "0000000"b if not given. The setting of locate_pos_sw may also be expressed by the use of the -pos control argument as an abbreviation for the corresponding specification of flags.

recl
: is an integer that must be given when flags.create_sw is set. This determines the new record length.

maxl
: is an optionally supplied integer that may be given with recl to specify a maximum record length. This defaults to recl if not given.

descrip
: is an octal record descriptor required when flags.locate_sw is set and flags.create_sw is not set.

pos_spec
: is a number or pair of numbers specifying the record's ordinal position (followed by the record's descriptor if the file is sequential). This specification is required and applies only when flags.locate_pos_sw is set.

Name: seek_head, sh

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 bin,
      2 n                     fixed bin,
      2 search_key            char (256 refer (info.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 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

COMMAND LEVEL

Syntax:

```
io_call control switch_name sh {args} search_key
```

where:

-brief, -bf
suppresses any error message except the no_operation and bad_arg errors.

rel_type
is a single digit, 0, 1, or 2. If omitted, the last argument is interpreted as a search_key, with a default rel_type of 0.

search_key
is any character string.

Name: select, sl

The select order causes subsequent vfile_ operations to behave as if a subset of all the records and their keys were present in an indexed file. The select order selects all of the keys associated with the records identified in the order. This process may select more keys from the index than are identified explicitly in the order.

Use (and include file) is the same as that described for the exclude order, except that status_only may be requested via select.

The subset of interest may be specified in terms of ranges of keys, a list of record descriptors, or an identifying number for a previously formed subset.

Various items of information that may be returned include a subset number, count of distinct descriptors, and a sorted list of descriptors.

None of the file position designators (current and next record positions) are affected by this order.

New records may not be created while a selection is in effect. If attempted, the code error_table_$no_record is returned.

COMMAND LEVEL

Syntax:

```
io_call control switch_name sl {args}
```

where args are the same as those described for the exclude order.

If no control arguments are given, the only effect is to print the status information for the current subset.

Name: set_file_lock, sf

The set_file_lock order is accepted when the I/O switch is open for output or update and attached to an indexed file with the -share control argument. For this order, the info_ptr argument must point to a variable of the following form:

```
dcl set_lock_flag bit(2) aligned based (info_ptr);
```

This order causes the file to be locked (if possible within the wait_time limit) or unlocked, depending on whether the user has set the first bit of info_ptr->set_lock_flag to "1"b or "0"b, respectively.

The possible error codes are:

error_table_$locked_by_this_process
error_table_$lock_not_locked
error_table_$file_busy

The second bit of set_lock_flag indicates the class of operations that are to be excluded by locking the file. If this bit is "0"b, only operations that alter the file are excluded (passive operations do not detect this state). Otherwise, all index referencing operations are excluded. In any case, the exclusion only applies to operations outside the current opening.

COMMAND LEVEL

Syntax:

```
io_call control switch_name sf set_lock_flag
```

where:

set_lock_flag
    is a string of two bits.

Name: set_wait_time, sw

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 one of the following structures:

```
dcl new_wait_time float based (info_ptr);
```

or:

```
dcl 1 wt_info                    based (info_ptr),
      2 version                  float,  /*Input*/
      2 collection_delay_time    float;  /*Input*/
```

This order specifies a limit on the time that the user's process waits 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 argument N used with the -share control argument.

If wt_info.version equals -2 (-2.0e0), the second argument is taken as a new collection_delay_time, in seconds. This specifies the amount of time that must elapse after deleting a stationary record before its storage can be completely recovered. Initially, in any opening, a default value of 0 applies.

COMMAND LEVEL

Syntax:

```
io_call control switch_name sw {arg} new_wait_time
```

where arg can be -collection_delay_time (-cdtm), and new_wait_time is a floating point number. If -cdtm is specified, new_wait_time is taken as the new collection delay time.

Name: truncate, tc

The truncate order is accepted when the I/O switch is attached to an unindexed file open for input_output or update. The order 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.

COMMAND LEVEL

Syntax:

```
io_call control switch_name tc
```

Name: window_io_

The window_io_ I/O module supports I/O to a window. In addition to the usual iox_ entries, the module provides terminal independent access to special video terminal features, such as a moveable cursor, selective erasure, and scrolling of regions. The module provides a real-time input line editor and performs output conversion and "MORE" processing.

Entry points in this module are not called directly by users; rather, this module is accessed through the I/O system interfaces iox_ and window_.

ATTACH DESCRIPTION

```
window_io_ switch {-control_args}
```

ARGUMENTS

switch
: is the name of an I/O switch attached to a terminal via the tc_io_ I/O module. The window created by this attach operation will be mapped onto the screen of that terminal. Use window_$create to attach and open, and use window_$destroy to detach and close windows on the login terminal.

CONTROL ARGUMENTS

-first_line LINE_NO
: LINE_NO is the line number on the screen where the window is to begin. If omitted, the window starts on the topmost line of the screen (line 1).

-height N_LINES, -n_lines N_LINES
: N_LINES is the number of lines in the window. The default is to use all lines to the end of the screen.

-first_column COL_NO
: COL_NO is the column number on the screen where the window is to begin. If omitted, the window starts on the leftmost column of the screen (column 1).

-width N_COLS, -n_columns N_COLS
: N_COLS is the number of the columns in the window. The default is all columns to the end of the screen.

NOTES

The attach description control arguments must specify a region which lies within the terminal screen. If not, the attachment is not made, and the error code video_et_$out_of_terminal_bounds is returned.

When the window is attached, it is cleared and the cursor is left at home.

OPEN OPERATION

The following opening mode is supported: stream_input_output.

PUR CHARS OPERATION

This operation is used to output a character string to the window. If rawo mode (see below) is disabled, the characters are processed according to the output conversions defined for the terminal. If necessary, the string is continued on subsequent lines of the window. If output passes the last line of the window, the placement of additional lines is controlled by the setting of the more_mode mode (see below). If an output line must be erased from the window to make room for this new output, and there has been no intervening input in this window, and more_mode (see below) is enabled, the user is queried for the disposition of this new output. (See MORE processing in Section 4.)

In rawo mode, the characters are written directly to the terminal, without any of the above processing.

GET CHARS OPERATION

This operation returns exactly one character, unechoed, regardless of the size of the caller's buffer. The line editor is not invoked by this call.

GET LINE OPERATION

The get_line operation invokes the real-time input line editor, and returns a complete line typed by the user. A description of the typing conventions is given in Section 4. The put_chars and get_line operations retrieve and reset any statuses that they encounter, so that applications that make these calls need not be changed to check for video_et_$window_status_pending.

CONTROL OPERATION

The control operations below are supported. Note that many of the control operations can be issued at command level via io_call commands; these include any control orders that do not require an info structure, and those described below. The following relations must hold when changing windows (set_window_info). These relations are always true when obtaining information about a window (get_window_info):

```
0 < column + width <= screen width
0 < line + height <= scrren height
```

get_window_info
returns information about the position and extent of the window. The info ptr points to the following structure (declared in window_control_info.incl.pl1):

```
dcl 1 window_position_info  based (window_position_info_ptr),
      2 version             fixed bin,
      2 origin,
        3 column            fixed bin,
        3 line              fixed bin,
      2 extent,
        3 width             fixed bin,
        3 height            fixed bin;
```

STRUCTURE ELEMENTS

version
is the version number of this structure. (Input) It must be window_position_info_version_2.

column
is the column of the upper left-hand corner of the window. (Output) If the column of the upper left-hand corner is zero, then the first column will be used, to allow old programs written when this was a mbz field to run without modification.

line
is the line of the upper left-hand corner of the window. (Output) *

width
is the width of the window (columns). (Output)

height
is the height of the window (lines). (Output)

set_window_info
causes the window to be relocated or to change size (or both). The info ptr points to the same structure used in the "get_window_info" control order. The values have the same meaning, but are the new values for the window when setting (Input), and are returned by get_window_info (Output).

get_window_status, set_window_status
window status is used to inform the application that some asynchronous event has disturbed the contents of the window. When window status is set for a window, all calls to window_ will return video_et_$window_status_pending until the status is reset. To reset the status, make a get_window_status control order on the switch. The info pointer should point to the following structure (declared in window_control_info.incl.pl1):

*

```
dcl  1 window_status_info  aligned based (window_status_info_ptr),
       2 version           fixed bin,
       2 status_string     bit (36) aligned;
```

STRUCTURE ELEMENTS

version
is the version of this structure. (Input) It must be window_status_version_1.

status_string
is the window status information. (Input) To interpret the actual status_string, use the include file window_status.incl.pl1:

```
dcl 1 window_status_info  aligned based (window_status_info_ptr),
     2 screen_invalid     bit (1) unaligned,
     2 async_change       bit (1) unaligned,
     2 ttp_change         bit (1) unaligned,
     2 reconnection       bit (1) unaligned,
     2 pad                bit (32) unaligned;
```

STRUCTURE ELEMENTS

screen_invalid
indicates that the contents of the window have become undefined. (Input for set, Output for get) This will happen, for example, in the event of a disconnection/reconnection of the terminal.

async_change
indicates that a timer or event call procedure has made a modification to the window. (Input for set, Output for get)

ttp_change
indicates that the terminal type has changed. (Input for set, Output for get) This re-initializes all the terminal specific video system information such as the video sequences, length and width of the screen, and capabilities.

reconnection
: determines the new terminal type (which may or may not be the same as before the disconnection). (Input for set, Output for get) Performs a set_term_type control order to inform the rest of the system of the change in terminal type.

pad
: reserved for future expansion and must be "0"b.

NOTES

The get_window_status and get_window_status control orders are available from command level and as active functions with the following io_call commands:

```
io_call control window_switch get_window_status status_key_1
   {status_key_2} N
io_call control window_switch set_window_status status_key_1
   {status_key_N}
```

where status_key_N is either screen_invalid, asynchronous_change, ttp_change, or reconnection.

get_capabilities
: returns information about the generic capabilities of the terminal. These are the "raw" physical characteristics of the terminal. The video system may simulate those that are lacking. For example, the system simulates insert and delete characters, but does not simulate insert and delete lines. The info ptr should point to the following structure (declared in terminal_capabilities.incl.pl1):

```
dcl 1 capabilities_info      aligned based(capabilities_info_ptr),
      2 version              fixed bin,
      2 screensize,
        3 columns            fixed bin,
        3 rows               fixed bin,
      2 flags,
        3 scroll_region      bit (1) unal,
        3 insert_chars       bit (1) unal,
        3 insert_mode        bit (1) unal,
        3 delete_chars       bit (1) unal,
        3 overprint          bit (1) unal,
        3 pad                bit (28) unal,
      2 line_speed           fixed bin,
```

STRUCTURE ELEMENTS

version
is the version number of this structure. (Input) It must be capabilities_info_version_1, also declared in the include file.

columns
is the number of columns on the terminal. (Output)

rows
is the number of rows (lines) on the terminal. (Output)

scroll_region
is true if the terminal is capable of scrolling, with insert and delete lines. (Output)

insert_chars
is true if the insert_chars function is supported. (Output)

insert_mode
is true if the terminal is capable of going into and out of insert mode. (Output)

delete_chars
is true if the delete chars function is supported. (Output)

overprint
is true if the terminal is capable of printing overstrike characters. (Output) It is currently always set to "0"b (false).

pad
reserved for future expansion and must be "0"b.

line_speed
is the speed of the communications channel to the terminal, in characters per second. (Output)

reset_more
causes MORE Processing to be reset. All lines on the window may be freely discarded without querying the user.

get_editing_chars
is identical to the operation supported by the tty_ I/O module.

set_editing_chars
is identical to the operation supported by the tty_ I/O module.

NOTES

The get_editing_chars and set_editing_chars control orders are available from command level and as active functions with the following io_call commands:

```
io_call window_switch get_editing_chars
io_call control window_switch set_editing_chars erase_kill_characters
```

get_more_responses
returns information about the acceptable responses to MORE processing. The info pointer should point to the following structure (declared in window_control_info.incl.pl1):

```
dcl 1 more_responses_info  aligned based (more_responses_info_ptr),
      2 version            fixed bin,
      2 n_yeses            fixed bin,
      2 n_noes             fixed bin,
      2 yeses              char (32) unaligned,
      2 noes               char (32) unaligned;
```

STRUCTURE ELEMENTS

version
is the version number of this structure and must be set to more_responses_info_version_1, also declared in the include file. (Input)

n_yeses
is the number of different affirmative responses, from zero to 32. (Output)

n_noes
is the number of different negative responses. (Output)

yeses
is the concatenation of all the affirmative responses. (Output) Each response is one character. Only the first "n_yeses" are valid.

noes
is the concatenation of all negative responses. (Output) Each response is one character. Only the first "n_noes" are valid.

set_more_responses
sets the responses. The data structure is the same as the one used for the "get_more_responses" order except that all fields are Input. At most, 32 yeses and 32 noes may be supplied. It is highly recommended that there be at least one yes, so that output may continue. The "yes" and "no" characters must be distinct. If they are not, the error code video_et_$overlapping_more_responses is returned, and the responses are not changed.

NOTES

The get_more_response and set_more_response control orders are available from command level and as active functions with the following io_call command:

```
io_call control window_switch get_more_responses
io_call control window_switch set_more_responses yes_responses
   no_responses
```

where the yes_responses and no_responses will be used as arguments to the get_more_responses control order. If either of the response strings contains blanks or special characters, it must be quoted.

get_more_prompt set_more_prompt
: sets the prompt displayed when a more break occurs. The current more responses can be displayed as part of the more prompt, by including the proper ioa_ control codes as part of the prompt string. For example the default video system more prompt string is "More? (^a for more; ^a to discard output.)". With the default more responses of carriage return for more and the delete for discard, the final string displayed is "More (RETURN for more; DEL to discard output)." The info pointer should point to the following structure (declared in window_control_info.incl.pl1):

```
dcl 1 more_prompt_info     aligned based (more_prompt_info_ptr),
      2 version            char (8),
      2 more_prompt        char (80);
```

STRUCTURE ELEMENTS

version
: is the version number of this structure (currently more_prompt_info_version_1). (Input)

more_prompt
: is the ioa_ control string to serve as the more prompt. (Input for set, Output for get)

The get_more_prompt and set_more_prompt control orders are available from command level and as active functions with the following io_call command:

```
io_call control window_switch get_more_prompt
io_call control window_switch set_more_prompt prompt_string
```

where window_switch is a valid window_io_ switch and prompt_string is the ioa_ control string described above.

get_more_handler set_more_handler
Sets the handler for video system more breaks to the specified routine. The info pointer should point to the following structure (declared in window_control_io.incl.pl1):

```
dcl 1 more_handler_info        aligned based (more_handler_info_ptr),
      2 version                fixed bin,
      2 flags                  unaligned,
        3 old_handler_valid    bit(1),
        3 pad                  bit(35),
      2 more_handler           entry (pointer, bit(1) aligned),
      2 old_more_handler       entry (pointer, bit(1) aligned);

dcl (more_handler_info_version_3);
    fixed bin internal static options (constant) init (3);
```

STRUCTURE ELEMENTS

version
is the version number of this structure, and must be set to more_handler_info_version_3 (also declared in the include file). (Input)

more_handler
is the entry to be called at a more break. (Input for set) (Output for get) It will be passed two arguments, described below.

old_handler_valid
is a flag specifying whether some other user-supplied more handler was in effect when the order call was made. (Output) (This can only be used with get.)

old_more_handler
is the user supplied entry that was acting as more handler before the order call was made. (Output) Its value is only defined if the old_handler_valid flag is on. (This can only be used with get.)

The more handler routine is called with two arguments. The first is a pointer to a structure containing information of interest to a more handler (see below), and the second is a flag which the more handler sets to indicate whether or not output should be flushed ("1"b to continue output, "0"b to flush output).

The structure can be found in the include file window_more_handler.incl.pl1, and is declared as follows:

*

```
dcl 1 more_info            aligned base (more_info_ptr),
      2 version            fixed bin,
      2 more_mode          fixed bin,      /* which flavor */
      2 window_iocb_prt    pointer,        /* for window that MORE'd */
      2 more_prompt        character (80), /* MORE? */
      2 more_responses,
        3 n_yeses          fixed bin,
        3 n_noes           fixed bin,
        3 more_yeses       character (32) unaligned,
                                           /* at most 32 yeses */
        3 more_noes        character (32) unaligned;
```

*

STRUCTURE ELEMENTS

version
is the version number of the structure (declared as more_handler_info_version_2 in the include file). (Input)

window_iocb_ptr
is a pointer to the iocb for the window in which the more break occurred. (Input) Prompt output should be written to this switch, and responses should be read from it.

more_mode
is the current more mode. (Input) Constants for the different more modes are declared in the include file window_io_attach_data.incl.pl1.

more_prompt
is the current more prompt. (Input) This is the string "More? (^a for more; ^a to discard output)" and is user-settable.

more_responses
is the current set of more responses, and is declared similarly to the more_responses_info structure in the get_more_responses order description above. (Input)

NOTES

The get_more_handler and set_more_handler control orders are available from command level and as active functions with the following io_call command:

```
io_call window_switch get_more_handler
io_call window_switch set_more_handler more_handler
```

where more_handler is the entryname of the routine to be used as the more handler routine. The name is converted to an entry using the user's search rules and is then used as described in the set_more_handler control order.

get_break_table set_break_table
: break table determines action of the get_echoed_chars, get_unechoed_chars, and write_sync_read entry points of the window_ subroutine. The array "breaks" has a 1 for each character that is to be considered a break. By default, the break table has "1"b for all the nonprintable characters, and "0"b elsewhere. Applications that set the break table must be careful to reset it afterwards, and establish an appropriate cleanup handler.

```
dcl 1 break_table_info   aligned based (break_table_ptr),
      2 versions         fixed bin,
      2 breaks           (0:127) bit (1) unaligned;
```

STRUCTURE ELEMENTS

versions
: must be set by the caller to break_table_info_version_1. (Input)

breaks
: has a "1"b for each character that is a break character. (Input/Output)

reset_more_handler
: cancels the last user-defined more_handler. The reset_more_handler control order is available from command level with the following io_call command:

```
io_call control window_switch reset_more_handler
```

get_output_conversion
this order is used to obtain the current contents of the specified table. The info_ptr points to a structure like the one described for the corresponding "set" order below, which is filled in as a result of the call (except for the version number, which must be supplied by the caller). If the specified table does not exist (no translation or conversion is required), the status code error_table_$no_table is returned.

set_output_conversion
provides a table to be used in formatting output to identify certain kinds of special characters. The info_ptr points to the following structure (declared in tty_convert.incl.pl1). If the info_ptr is null, no transaction is to be done.

```
dcl 1 cv_trans_struc              aligned
      2 version                   fixed bin,
      2 default                   fixed bin,
      2 cv_trans                  aligned
        3 value                   (0:255) fixed bin (8) unaligned
```

STRUCTURE ELEMENTS

version
is the version number of the structure. It must be 2 and declared in tty_convert.incl.pl1.

default
indicates, if nonzero, that the table is the one that was in effect before video was invoked.

values
are the elements of the table. This table is indexed by the value of a typed input character, and the corresponding entry contains the ASCII character resulting from the translation.

get_special
is used to obtain the contents of the special_chars table currently in use. The info_ptr points to the following structure (defined in tty_convert.incl.pl1):

```
dcl 1 get_special_info_struc    aligned
      2 area_ptr                ptr,
      2 table_ptr               ptr;
```

STRUCTURE ELEMENTS

area_ptr
points to an area in which a copy of the current special_chars table is returned. (Input)

table_ptr
is set to the address of the returned copy of the table. (Output)

set_special
provides a table that specifies sequences to be substituted for certain output characters, and characters that are to be interpreted as parts of escape sequences on input. Output sequences are of the following form (defined in tty_convert.incl.pl1):

```
dcl 1 c_chars          based aligned,
      2 count          fixed bin (8) unaligned,
      2 chars (3)      char (1) unaligned;
```

STRUCTURE ELEMENTS

count
is the actual length of the sequence in characters (0<= count <=3). If count is zero, there is no sequence.

chars
are the characters that make up the sequence. The info_ptr points to a structure of the following form (defined in tty_convert.incl.pl1):

```
dcl 1 special_chars_struc      aligned based,
      2 version                fixed bin,
      2 default                fixed bin,
      2 special_chars
        3 nl_seq               aligned like c_chars,
        3 cr_seq               aligned like c_chars,
        3 bs_seq               aligned like c_chars,
        3 tab_seq              aligned like c_chars,
        3 vt_seq               aligned like c_chars,
        3 ff_seq               aligned like c_chars,
        3 printer_on           aligned like c_chars,
        3 printer_off          aligned like c_chars,
        3 red_ribbon_shift     aligned like c_chars,
        3 black_ribbon_shift   aligned like c_chars,
        3 end_of_page          aligned like c_chars,
        3 escape_length        fixed bin,
        3 not_edited_escapes   (sc_escape_len refer
                               (special_chars.escape_length))
                                 like c_chars,
        3 edited_escapes       (sc_escape_len refer
                               (special_chars.escape_length))
                                 like c_chars,
        3 input_escapes        aligned,
          4 len                fixed bin(8) unaligned,
          4 str                char (sc_input_escape_len refer
                               (special_chars.input_escapes.len))
                                 unaligned,
        3 input_results        aligned,
          4 pad                bit(9) unaligned,
          4 str                char (sc_input_escape_len refer
                               (special_chars.input_escapes.len))
                                 unaligned;
```

NOTES

Video ignores cr_seg, bs_seg, tab_seg, vt_seg, ff_seg, printer_on, printer_off, end_of_page, input_escapes, and input results.

STRUCTURE ELEMENTS

version
: is the version number of this structure. It must be 1.

default
: indicates, if nonzero, that the default values for the current terminal type and baud rate are to be used and that the remainder of the structure is to be ignored.

nl_seq
: is the output character sequence to be substituted for a newline character. The nl_seq.count generally should be nonzero.

cr_seq
: is the output character sequence to be substituted for a carriage-return character. If count is zero, the appropriate number of backspaces is substituted. However, either cr_seq.count or bs_seq.count should be nonzero (i.e., both should not be zero).

bs_seq
: is the output character sequence to be substituted for a backspace character. If count is zero, a carriage return and the appropriate number of spaces are substituted. However, either bs_seq.count or cr_seq.count, should be nonzero (i.e., both should not be zero).

tab_seq
: is the output character sequence to be substituted for a horizontal tab. If count is zero, the appropriate number of spaces is substituted.

vt_seq
: is the output character sequence to be substituted for a vertical tab. If count is zero, no characters are substituted.

ff_seq
: is the output character sequence to be substituted for a formfeed. If count is zero, no characters are substituted.

printer_on
is the character sequence to be used to implement the printer_on control operation. If count is zero, the function is not performed.

printer_off
is the character sequence to be used to implement the printer_off control operation. If count is zero, the function is not performed.

red_ribbon_shift
is the character sequence to be substituted for a red-ribbon-shift character. If count is zero, no characters are substituted.

black_ribbon_shift
is the character sequence to be substituted for a black_ribbon_shift character. If count is zero, no characters are substituted.

end_of_page
is the character sequence to be printed to indicate that a page of output is full. If count is zero, no additional characters are printed, and the cursor is left at the end of the last line.

escape_length
is the number of output escape sequences in each of the two escape arrays.

not_edited_escapes
is an array of escape sequences to be substituted for particular characters if the terminal is in "^edited" mode. This array is indexed according to the indicator found in the corresponding output conversion table (see the description of the set_output_conversion order above).

edited_escapes
is an array of escape sequences to be used in edited mode. It is indexed in the same fashion as not_edited_escapes.

input_escapes
is a string of characters each of which forms an escape sequence when preceded by an escape character.

input_results
is a string of characters each of which is to replace the escape sequence consisting of an escape character and the character occupying the corresponding position in input_escapes.

get_token_characters, set_token_characters

changes the set of characters that are used by the video system input line editor to define a word for such requests as ESC DEL. The set of characters supplied in the structure replace the existing set of characters. The info_ptr points to the following structure (declared in window_control_info.incl.pl1):

```
dcl 1 token_characters_info    aligned based
                                  (token_characters_info_ptr),
      2 version                char (8),
      2 token_characters_count fixed bin,
      2 token_characters       char (128) unaligned;
```

STRUCTURE ELEMENTS

version

is the version string for this structure. (Input) Its current value is token_characters_info_version_1, also declared in the include file.

token_characters_count

is the number of characters in the token_characters string. (Input)

token_characters

is a character string containing the new set of token characters. (Input)

NOTES

The set_token_characters and get_token_characters control orders are available from command_level and as active functions with the following io_call commands:

```
io_call control window_switch get_token_characters
io_call control window_switch set_token_characters token_char_string
```

where token_char_string is a character string containing the new set of token characters. get_token_character returns its result as a string if it was invoked as an active function, otherwise it prints out the token characters.

get_editor_key_bindings

returns a pointer to the line_editor_key_binding structure describing the key bindings. io_call support points out the pathname of each editor routine, listing only the names of builtin requests in capital letters, with the word "builtin" in parentheses. The control order prints or returns current information about the key bindings. Use the set_editor_key_bindings control order to change the bindings. This control order prints or returns current information about the key_bindings. Use the set_editor_key_bindings control order to change the bindings.

The info_ptr points to the following structure (declared in window_control_info.incl.pl1):

```
dcl 1 get_editor_key_bindings_info  aligned based
                                    (get_editor_key_binding_info_ptr),
      2 version                     char (8),
      2 flags,
        3 entire_state              bit (1) unaligned,
        3 mbx                       bit (35) unaligned,
      2 key_binding_info_ptr        ptr,
      2 entire_state_ptr            ptr;
```

STRUCTURE ELEMENTS

version
: is get_editor_key_binding_info_version_1. (Input)

entire_state
: is "1"b if the entire state is desired, "0"b if only information about certain keybindings is desired. (Input)

key_binding_info_ptr
: if entire_state = "0"b, then this points to a line_editor_key_binding_structure. (Input) The bindings component of this structure is then filled in based upon the value of each key_sequence supplied.

entire_state_ptr
: is set to point to the "state" of the key bindings, if entire_state = "1"b. (Output) This is suitable input to the set_editor_key_bindings control order.

NOTES

The get_editor_key_bindings control order is available from command level and as an active function with following io_call command:

```
io_call control window_switch get_editor_key_bindings
```

The get_editor_key_bindings control order prints or returns information about a key binding. When you use it as an active function the information is returned in a form suitable as arguments to the set_editor_key_bindings control order.

set_editor_key_bindings

A line editor routine is bound to a sequence of keystrokes via the set_editor_key bindings control order. The sequence of characters that triggers an editor request may be of any length, with multiple-key sequences working like the Emacs prefix characters. This allows the use of terminal function keys (which often send three or more character sequences) to invoke line editor requests. More than one binding can be set in one invocation of this control order.

The info_ptr points to the following structure (declared in window_control_info.incl.pl1):

```
dcl 1 set_editor_key_bindings_info aligned based
                                   (set_editor_key_bindings_info_ptr),
      2 version                     char (8),
      2 flags,
        3 replace                   bit (1) unaligned,
        3 update                    bit (1) unaligned,
        3 pad                       bit (34) unaligned.
      2 key_binding_info_ptr;
```

STRUCTURE ELEMENTS

version

is the version of the structure. (Input) It must be set_editor_key_bindings_info_version_1.

replace

if "1"b then key_binding_info is considered to be returned by a previous get_editor_key_bindings operation with entire_state = "1"b and will be used to replace the keybinding state of the editor. (Input)

update

if "1"b then key_binding_info_ptr is considered a pointer to a line_editor_key_binding_info structure, which will be used to update the keybinding state of the editor. (Input)

Note: only one of replace and update may be true, but at least one of them must be true.

key_binding_info_ptr

is a pointer received from get_editor_key bindings operation or a pointer to a line_editor_key_binding_info structure, depending on the value of the replace and update flags. (Input)

Notes on freeing: The video system's internal data structures are freed at the following times: video system revocation and when a set_editor_key_bindings control order with replace = "1"b is done.

NOTES

The set_editor_key_bindings control order is available from command level and as an active function with the following io_call command:

```
io_call control window_switch set_editor_key_bindings key_sequence1
  {user_routine1} {control_args1} ... key_sequenceN
    {user_routineN} {control_args1} {control_argsN}
```

where user_routine is the name of a user-written editor request.

control args are:

```
-external user_routine
-builtin builtin_request_name
-numarg_action numarg_action_name
```

The line_editor_key_bindings_info structure is described in Section 7.

At least one user_routine or one of -external/-builtin must be specified for each key sequence, with the rightmost editor request specifier taking precedence (for example, io control window_switch set_editor_key_binings foo -builtin FORWARD_word,) will bind control -a to the forward word builtin, not the user routine foo.

numarg_action_name
: the type of automatic numeric argument to be taken when the editor routine is invoked, must be one of the following and can only be given for external editor routines

REPEAT
: (the default is PASS). This can be entered in upper or lower case. Call the user routine n times, where n is the numeric argument supplied by the user.

REJECT
: ring the terminal bell and don't call the user routine if a numeric argument is given.

PASS
: pass any numeric argument to the user routine, without any other action.

IGNORE
: same as PASS but implies the user routine will not make use of the numeric argument.

-name STR
: specifies the name of the editor command being assigned to the key. If this is the null string, then a default name is used (for builtins this is the name of the builtin, otherwise it is segname$entrypoint). STR must be quoted if it contains whitespace.

-description STR
: specifies a description string to be associated with the key binding. If this is the null string, a default description is used. The defaults can be found in the include file window_editor_values.incl.pl1. STR must be quoted if it contains whitespace.

-info_pathname PATH
: specifies an info segment pathname to be associated with this key binding. This info segment is expected to have more information about the editor_routine. If this is not specified, it defaults to >doc>info>video_editing.gi.info if -builtin, otherwise no info segment is associated with the key. The info suffix is assumed on PATH.

MODES OPERATION

The modes operation is supported by window_io_. The recognized modes are listed below. Some modes have a complement indicated by the circumflex character (^) that turns the mode off (e.g. ^more). For these modes, the complement is displayed with that mode. Some modes specify a parameter that can take on a value (e.g. more_mode). These modes are specified as MODE=VALUE, where MODE is the name of the mode and VALUE is the value it is to be set to. Parameterized modes are indicated by the notation (P) in the following description:

more, ^more
: Turns MORE processing on. Default is on. If ^pl is set before you invoke the video system, ^more will be set when you invoke the video system.

more_mode = STR
: controls behavior when the window is filled. The value for STR may be one of the following:

  clear
  : the window is cleared, and output starts at the home position.

  fold
  : output begins at the first line and moves down the screen a line at a time replacing existing text with new text. Prompts for a MORE response when it is about to overwrite the first line written since the last read or MORE break.

  scroll
  : lines are scrolled off the top of the window, and new lines are printed in the space that is cleared at the bottom of the screen. This is the default for full width windows on all terminals capable of scrolling.

  wrap
  : output begins at the first line and moves down the screen a line at a time replacing existing text with new text. Prompts for a MORE response at the bottom of every window of output. This is the default for all terminals that are incapable of scrolling or when using partial width windows.

vertsp, ^vertsp
is only effective when more mode is on. When vertsp mode is on, output of a FF or VT will cause an immediate MORE query. When you invoke the video system, it copies the current setting of this mode before attaching the window_io_ module. The default is ^vertsp.

rawo, ^rawo
causes characters to be output with no processing whatsoever. The result of output in this mode is undefined.

can, ^can
causes input lines to be canonicalized before they are returned. When you invoke the video system, it copies the current setting of this mode before attaching the window_io_ module. The default is on.

ctl_char, ^ctl_char
specifies that ASCII control characters that do not cause newline or linefeed motion are to be accepted as input except for the NUL character. If the mode is off all such characters are discarded. When you invoke the video system, it copies the current setting of this mode before attaching the window_io_ module. The default is off.

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. When you invoke the video system, it copies the current setting of this mode before attaching the window_io_ module. The default is off.

erkl, ^erkl
controls the editing functions of get_line. When you invoke the video system, it copies the current setting of this mode before attaching the window_io_ module. The default is on, which allows erase and kill processing and the additional line editor functions.

esc, ^esc
controls input escape processing. When you invoke the video system, it copies the current setting of this mode before attaching the window_io_ module. The default is on.

rawi, ^rawi
acts as a master control for can, erkl, and esc. If this mode is on, none of the input conventions are provided. The default is on.

ll = STR
is the width of the window, in characters, and it can only be changed with the set_window_info control operation.

pl = STR
is the height of the window (i.e., number of lines), and it can only be changed with the set_window_info control operation.

red, ^red
controls interpretation of red shift and black shift characters on output. When you invoke the video system, it copies the current setting of this mode before attaching the window_io_ module. The default is ^red, which ignores them. In red mode, the character sequence given in the TTF is output. The effect is undefined and terminal-specific. In some cases, "red shifted" output appears in inverse video, but this is not guaranteed.

CONTROL OPERATIONS FROM COMMAND LEVEL

Those control operations which require no info_ptr and those additional orders described above may be performed from command level using the io_call command, as follows:

```
io_call control switch_name control_order
```

ARGUMENTS

switch_name
is the name of the I/O switch.

control_order
can be any control order described above under "Control Operation" that can accept a null info_ptr.

Name: xmodem_io_

The xmodem_io_ I/O module is used to transfer files between a Multics process and a microcomputer that runs the XMODEM data transfer protocol. It performs 8-bit stream I/O over an asynchronous communications channel using the xmodem protocol.

Entry points in this module are not called directly by users; rather the module is accessed through the I/O system.

ATTACH DESCRIPTION

```
xmodem_io_ switch {-control_args}
```

ARGUMENTS

switch
: is the name of the target I/O switch. The switch must be open for stream_input_output. The I/O module for the target switch must be supported by the timed_io_ module. The user is responsible for setting any modes required by the xmodem protocol. For example, modes for the user_i/o switch would be: "no_outp,8bit,breakall,^echoplex,rawi,^crecho,lfecho,^tabecho,rawo"

CONTROL ARGUMENTS

-error_detecting_code STR, -edc STR
: specifies the error-detecting code to be used for the file transfer, where STR may be one of the following:

  check_sum, cs
  : specifies that the checksum error-detecting code is to be used for the file transfer.

  cyclic_redundancy_code, crc
  : specifies that the CRC-CCITT error-detecting code is to be used for the file transfer. Note, because it is the receiver that determines the type of error-detecting code, this control argument is incompatible with the stream_output opening mode.

Default is check_sum.

OPEN OPERATION

The xmodem_ I/O module supports the stream_input and stream_output opening modes.

CLOSE OPERATION

When opened for stream_output, the close entry transmits any remaining data in the internal buffer before closing the switch. If there are less than 128 bytes in the buffer, the buffer is filled with the NUL ASCII character, 000 (octal), before transmission. See Buffering below.

PUT CHARS OPERATION

The put_chars entry splits the data to be written into 128-character blocks. The appropriate xmodem control characters are added to the beginning and end of each block. For further explanation of the put_chars entry, see the iox_$put_chars entry.

GET CHARS OPERATION

The get_chars entry reads and decodes xmodem blocks, removes the xmodem control characters, and returns the message text to the caller's buffer. For further explanation of the get_chars entry, see the iox_$get_chars entry.

GET LINE OPERATION

The get_line entry reads and decodes xmodem blocks, removes the control characters, and returns the message text to the caller's buffer. Characters are returned until either a newline character is placed in the buffer or the buffer is filled. For further explanation of the get_line entry, see the iox_$get_line entry.

CONTROL OPERATION

This operation is not supported.

MODES OPERATION

This operation is not supported.

BUFFERING

The xmodem protocol uses 128 data characters per packet. Data that is not a multiple of 128 characters is stored in an internal buffer by the xmodem_io_ I/O module. Thus, those users concerned with efficiency should provide a multiple of 128 data characters for I/O operations.

NOTES

No particular line speed is guaranteed when transferring data between Multics and a microcomputer. Line speed is dependent on the microcomputer and the load of the FNP and communication system for Multics. Due to the nature of the XMODEM protocol, files may not be successfully transferred to Multics over high-speed lines. The actual limit depends on the site configuration and current load.

DEFINITIONS

<soh>	01 (HEX)	01 (OCT)
<eot>	04 (HEX)	04 (OCT)
<ack>	06 (HEX)	06 (OCT)
<nak>	15 (HEX)	25 (OCT)

TRANSMISSION MEDIUM LEVEL PROTOCOL

Asynchronous, 8 data bits, no parity, one stop bit.

There are no restrictions on the contents of the data being transmitted. Any kind of data may be sent: binary, ASCII, etc. No control characters are looked for in the 128-byte data messages.

MESSAGE BLOCK LEVEL PROTOCOL

The standard transmission portion of a message block is a 132 character block without framing characters. Each block of the transfer looks like:

```
<SOH><blk #><255-blk #><..128 data bytes..><edc> where:

<SOH>          =   01 (Hex).
<blk #>        =   binary number, starts at 01 increments by 1
                   and wraps OFF (Hex) to 00 (Hex).
<255-blk #>    =   The one's complement of the block number.
<edc>          =   A one-character checksum or two-character CRC-CCITT.
                   The checksum is the sum of the data bytes only.  The
                   CRC-CCITT is a 16-bit remainder obtained by dividing
                                                            16  12  5
                   the data bit string by the polynomial X  +X  +X +1.
```

File Level Protocol

When writing programs that implement the XMODEM protocol, users should follow the procedures listed below:

In both sending and receiving programs, all errors should be retried ten times.

THE RECEIVING PROGRAM

The receiver should have a 10-second timeout and send a <nak> every time it times out. The first timeout that sends a <nak> signals the transmitter to start.

Once into receiving a block, the receiver must go into a one-second timeout for each character and the checksum. If a valid block is received, the receiver must transmit an <ack>. For invalid blocks, a <nak> must be transmitted.

The Sending Program

The sender should start transmission upon receipt of a <nak> from the receiver. If the block is received successfully (i.e., the receiver sends an <ack>), the next block should be sent. If the receiver responds with a <nak>, the transmission has failed, and the sender should retransmit the last block. When the sender has no more data, he should send an <eot> and await an <ack>. If it does not get one, the sending program should repeat the <eot>.

APPENDIX A

OBSOLETE FUNCTIONS

This section contains descriptions of those functions that have been functionally replaced with other functions whose usage is preferred. The following are documented here because they are still called by older programs:

Obsolete Function	*Replacement*	
decode_clock_value_	date_time_$from_clock	*
encode_clock_value_	date_time_$to_clock	
hcs_$del_dir_tree	delete_$path	
hcs_$delentry_file	delete_$path	
hcs_$delentry_seg	delete_$ptr	
hcs_$set_bc_seg	terminate_file_	*
hcs_$terminate_file	term_	
hcs_$terminate_name	term_$refname	
hcs_$terminate_noname	terminate_file_	
hcs_$terminate_seg	term_$seg_ptr	
hcs_$truncate_seg	terminate_file_	
ipc_$create_ev_chn	ipc_$create_event_channel	
ipc_$decl_event_call_chn	ipc_$create_event_channel	
link_unsnap_	term_$unsnap	

Name: decode_clock_value_

NOTE: All entrypoints in decode_clock_value_ are replaced by date_time_$from_clock; decode_clock_value_ is supported for compatibility only.

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

```
dcl decode_clock_value_ entry (fixed bin(71), fixed bin, fixed bin,
      fixed bin, fixed bin(71), fixed bin, char(3));

call decode_clock_value_ (clock, month, dom, year, tod, dow, zone);
```

ARGUMENTS

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

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

dom
: is the day of the month, i.e., 1 to 31. (Output)

year
: is the year, e.g., 1982. (Output)

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

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

zone
: is a three-character lowercase abbreviation of the time zone currently used by this process (for example, mst, edt). (Output)

NOTES

If the clock value does not lie within the 20th century, then zero values are returned for month, dom, year, tod, and dow.

Entry: decode_clock_value_$date_time

This entry point is given a system clock reading and returns the month, the day of the month, the year, the hour, the minute, the second, the microseconds within a second, and the day of the week. The time zone in which the decoded clock reading is expressed may be given as input, or the current time zone can be used.

USAGE

```
declare decode_clock_value_$date_time entry (fixed bin(71), fixed bin,
      fixed bin, fixed bin, fixed bin, fixed bin, fixed bin,
      fixed bin(71), fixed bin, char(3), fixed bin(35));

call decode_clock_value_$date_time (clock, month, dom, year, hour,
      minute, second, microsecond, dow, zone, code);
```

ARGUMENTS

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

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

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

year
is the year, e.g., 1982. (Output)

hour
is the hour of the day (midnight = 0, ..., 11 PM = 23). (Output)

minute
is the minute of the hour, i.e., 0 to 59. (Output)

second
is the second of the minute, i.e., 0 to 59. (Output)

microsecond
is the microsecond within a second. (Output)

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

zone
is a three-character abbreviation of the time zone in which the decoded clock value is expressed. (Input or Output)

Input
is one of the zone abbreviations given in the table of time zones, or is a null character string. A zone abbreviation may be in uppercase or lowercase. If a null string is given, the time zone currently used by this process is assumed.

Output
is a three-character lowercase abbreviation of the current time zone used by this process if a null character string was given as input.

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

error_table_$bad year
the clock reading does not represent a date within the 20th century.

error_table_$unknown_zone
the specified time zone abbreviation is not in the table of time zones.

error_table_$unimplemented_version
the current version of the table of time zones is not implemented by decode_clock_value_.

Name: encode_clock_value_

NOTE: encode_clock_value_ is replaced by date_time_$to_clock; encode_clock_value_ is supported for compatibility only.

The encode_clock_value_ subroutine takes a given month, day of the month, year, hour of the day, minute, second, microsecond, and time zone and returns a system clock reading. When given a day of the week, it performs an optional check on the clock reading to ensure that it falls on the given day.

A system clock reading is encoded as the number of microseconds from January 1, 1901 0000.0, Greenwich Mean Time (GMT) to the given date, time, and time zone.

USAGE

```
declare encode_clock_value_ entry (fixed bin, fixed bin, fixed bin,
      fixed bin, fixed bin, fixed bin, fixed bin(71), fixed bin,
      char(3), fixed bin(71), fixed bin(35));

call encode_clock_value_ (month, dom, year, hour, minute, second,
      microsecond, dow, zone, clock, code);
```

ARGUMENTS

month
: is the month (January = 1, ..., December = 12). (Input)

dom
: is the day of the month, i.e., 1 to 31. (Input)

year
: is the year, e.g., 1982. (Input)

hour
: is the hour of the day (midnight = 0, ..., 11 PM = 23). (Input)

minute
: is the minute of the hour, i.e., 0 to 59. (Input)

second
: is the second of the minute, i.e., 0 to 59. (Input)

microsecond
: is the number of microseconds that are added to the clock reading encoded from the given month, dom, year, hour, minute, and second. (Input)

dow
: is the day of the week (0 = no day of week checking, 1 = Monday, ..., 7 = Sunday). (Input)

zone
is a three-character abbreviation of the time zone in which the given day of the month and hour are expressed. (Input or Output)
Input
is one of the zone abbreviations given in the table of time zones (see the convert_date_to_binary_ subroutine), or is a null character string. A zone abbreviation may be given in uppercase or lowercase. If a null string is given, the current time zone used by the process is assumed.
Output
is a three-character lowercase abbreviation of the current time zone used by the process if a null character string was given as input.

clock
is the encoded system clock reading. (Output)

code
is a system status code. (Output) It can be one of the following:
error_table_$bad_date
the date represented by month, dom and year is an invalid date, e.g., 2/29/77.
error_table_$bad_day_of_week
the returned clock reading does not fall on the given day of the week, or dow is not a number from 0 to 7.
error_table_$bad_time
the time represented by hour, minute, second, and microsecond is invalid, e.g., 23:60 or negative time values.
error_table_$unknown_zone
the specified time zone abbreviation is not in the table of time zones.
error_table_$unimplemented_version
the current version of the table of time zones is not implemented by encode_clock_value_.

Entry: encode_clock_value_$offsets

NOTE: encode_clock_value_$offsets is replaced by date_time_$offset_to_clock; encode_clock_value_$offsets is supported for compatibility only.

This entry point takes a system clock reading, a day of the week, and year, month, day, hour, minute, second, and microsecond, offset values. The offset values may be positive, negative, or zero. It returns a clock reading that has been adjusted to fall on the given day of the week, and which is then offset by the given number of years, months, days, hours, minutes, seconds, and microseconds.

USAGE

```
declare encode_clock_value_$offsets entry (fixed bin(71), fixed bin,
      fixed bin, fixed bin, fixed bin, fixed bin, fixed bin,
      fixed bin(71), fixed bin, char(3), fixed bin(71), fixed bin(35));

call encode_clock_value_$offsets (clock_in, month_off, day_off,
      year_off, hour_off, minute_off, second_off, microsec_off,
      dow_offset, zone, clock_out, code);
```

ARGUMENTS

clock_in
: is a system clock reading. (Input)

month_off
: is an offset, in months. (Input)

day_off
: is an offset, in days. (Input)

year_off
: is an offset, in years. (Input)

hour_off
: is an offset, in hours. (Input)

minute_off
: is an offset, in minutes. (Input)

second_off
: is an offset, in seconds. (Input)

microsec_off
: is an offset, in microseconds. (Input)

dow_off
: is a day of the week offset (0 = no day of week offset, 1 = offset to next Monday, ..., 7 = offset to next Sunday). (Input)

zone
is a three-character abbreviation of the time zone in which the input clock reading is to be interpreted. (Input or Output) The choice of zone may alter which day of the week the input clock reading falls on, and may therefore affect any day of the week adjustment.

Input
is one of the zone abbreviations given in the table of time zones (see the convert_date_to_binary_ subroutine), or is a null character string. A zone abbreviation may be given in uppercase or lowercase. If a null string is given, the current time zone used by the process is assumed.

Output
is a three-character lowercase abbreviation of the current time zone used by the process if a null character string was given as input.

clock_out
is the adjusted clock reading. (Output)

code
is a system status code. (See above.) (Output)

NOTES

The order in which offsets are applied to the input clock reading can affect the adjusted clock reading. The encode_clock_value_$offsets entry point uses the order required by the convert_date_to_binary_ subroutine in all cases. The offsets are applied in the following order:

1. Decode the input clock reading into absolute date and time values specified in terms of the input time zone. The time zone can alter the day of the week the input clock reading falls on, and can therefore change the effect of the day of the week offset.

2. Apply any day of the week offset by adding days to the absolute date values from step 1 until the date falls on the given day of the week.

3. Apply any year offset to the absolute date values from step 2.

4. Apply any month offset to the absolute date values from step 3. If applying the month offset results in an invalid date (e.g., 1/31/77 +3 months yields 4/31/77), then adjust the day of the month to be the last day of the new month, taking leap years into account.

5. Apply the day offset to the absolute date values from step 4.

6. Apply the hour, minute, second, and microsecond offsets to the absolute time values from step 1.

7. Encode the absolute date values from step 5 and absolute time values from step 6 to form the adjusted clock reading.

Name: hcs_$del_dir_tree

This entry point, given the pathname of a containing directory and the entryname of a subdirectory, deletes the contents of the subdirectory from the storage system hierarchy. All segments, links, and directories inferior to that subdirectory are deleted, including the contents of any inferior directories. The subdirectory is not itself deleted. For information on the deletion of directories, see the description of the hcs_$delentry_file entry point.

USAGE

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

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

ARGUMENTS

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

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

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

NOTES

The user must have status and modify permission on the subdirectory and the safety switch of all branches in the directory must be off. If the user does not have status and modify permission on inferior directories, access is automatically set and processing continues.

If an entry in an inferior directory gives the user access only in a ring lower than his validation level, that entry is not deleted and no further processing is done on the subtree. For information about rings, see "Intraprocess Access Control" in the Programmer's Reference Manual.

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. 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 entry (char(*), char(*), fixed bin(35));

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

ARGUMENTS

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

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

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 the entryname argument 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 entry (ptr, fixed bin(35));

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

ARGUMENTS

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

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_$initiate

This entry point, when given a pathname and a reference name, makes known the segment defined by the pathname, initiates the given reference name, and increments the count of initiated reference names for the segment.

Use of the initiate_file_ subroutine is preferred if a null reference name is desired.

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);
```

ARGUMENTS

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

entryname
is the entry name of the segment. (Input)

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

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

copy_ctl_sw
is obsolete, and should be set to zero. (Input)

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

code
is a storage system status code. (Output)

NOTES

The user must have nonnull access on the segment (the entryname argument) in order to make it known.

If a segment is concurrently initiated more than a system-defined number of times, the usage count of the segment is said to be in an overflowed condition, and further initiations do not affect the usage count. This affects the use of the hcs_$terminate_noname and hcs_$terminate_name entry points. 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.

If entryname cannot be made known, a null pointer is returned for seg_ptr and the returned value of code indicates the reason for failure. Thus, the usual way to test whether the call was successful is to check the pointer, not the code, since the code may be nonzero even if the segment was successfully initiated. If entryname is already known to the user's process, code is returned as error_table_$segknown and the seg_ptr argument contains a nonnull pointer to entryname. If ref_name has already been initiated in the current ring, the code is returned as error_table_$namedup. The seg_ptr argument contains a valid pointer to the segment being initiated. If entryname is not already known, and no problems are encountered, seg_ptr contains a valid pointer and code is 0.

Name: hcs_$initiate_count

The hcs_$initiate_count entry point, when given a pathname and a reference name, causes the segment defined by the pathname to be made known and the given reference name initiated. A segment number is assigned and returned as a pointer and the bit count of the segment is returned. Use of the initiate_file_ subroutine is preferred if a null reference name is desired.

USAGE

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

call hcs_$initiate_count (dir_name, entryname, ref_name, bit_count,
      copy_ctl_sw, seg_ptr, code);
```

ARGUMENTS

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

entryname
: is the entry name of the segment. (Input)

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

bit_count
: is the bit count of the segment. (Output)

copy_ctl_sw
: is obsolete, and should be set to zero. (Input)

seg_ptr
: is a pointer to the segment. (Output)

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

NOTES

The user must have nonnull access on the segment (the entryname argument) in order to make it known.

If entryname cannot be made known, a null pointer is returned for seg_ptr and the returned value of code indicates the reason for failure. Thus, the usual way to test whether the call was successful is to check the pointer, not the code, since the code may be nonzero even if the segment was successfully initiated. If entryname is already known to the user's process, code is returned as error_table_$segknown and the seg_ptr argument contains a nonnull pointer to entryname. If entryname is not already known, and no problems are encountered, seg_ptr contains a valid pointer and code is 0. If ref_name has already been initiated in the current ring, the code is returned as error_table_$namedup. The seg_ptr argument contains a valid pointer to the segment being 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_$set_bc_seg

This entry point, given a pointer to the segment, sets the bit count of a segment in the storage system. It also sets the bit count author of that segment to be the user who called it.

The terminate_file_ subroutine performs this same function and its usage is recommended.

USAGE

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

call hcs_$set_bc_seg (seg_ptr, bit_count, code);
```

ARGUMENTS

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

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

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

Name: hcs_$terminate_file

This 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).

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.

USAGE

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

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

ARGUMENTS

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

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

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

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 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 the Programmer's Reference Manual.

Name: hcs_$terminate_name

This entry point terminates one reference name from a segment and decrements a count of initiated reference names for the segment.

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

USAGE

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

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

ARGUMENTS

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

code
is a storage system status code. (Output)

NOTES

If a segment is concurrently initiated more than a system-defined number of times, the usage count of the segment is said to be in an overflowed condition. Under such circumstances, the hcs_$terminate_name entry point does not reduce the usage count, but leaves the segment in the overflowed state. To terminate the segment, hcs_$terminate_file or hcs_$terminate_seg should be used.

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).

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.

Name: hcs_$terminate_noname

This entry point terminates a null reference name from the specified segment and decrements a count of initiated reference names for the segment.

The terminate_file_ subroutine performs this same function and its usage is recommended.

USAGE

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

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

ARGUMENTS

seg_ptr
is a pointer to the segment. (Input/Output)

code
is a storage system status code. (Output)

NOTES

If a segment is concurrently initiated more than a system-defined number of times, the usage count of the segment is said to be in an overflowed condition. Under such circumstances, the hcs_$terminate_noname entry point does not reduce the usage count, but leaves the segment in the overflowed state. To terminate the segment, hcs_$terminate_file or hcs_$terminate_seg should be used.

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.

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

This 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).

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

USAGE

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

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

ARGUMENTS

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

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

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 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 the Programmer's Reference Manual.

Name: hcs_$truncate_seg

This 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 zeros in the words beyond the specified length.

The terminate_file_ subroutine performs this same operation, and its usage is recommended.

USAGE

```
declare hcs_$truncate_seg entry (ptr, fixed bin(19), fixed bin(35));

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

ARGUMENTS

seg_ptr
: is a pointer to the segment to be truncated. (Input) Only the segment number portion of the pointer is used.

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

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 zeros.

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

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

Name: ipc_$create_ev_chn

This entry point creates an event-wait channel in the current ring.

USAGE

```
declare ipc_$create_ev_chn entry (fixed bin(71), fixed bin(35));

call ipc_$create_ev_chn (channel_id, code);
```

ARGUMENTS

channel_id
is the identifier of the event channel. (Output)

code
is a standard status code. (Output)

Name: ipc_$decl_event_call_chn

This entry point changes an event-wait channel into an event-call channel.

USAGE

```
declare ipc_$decl_event_call_chn entry (fixed bin(71), entry, ptr,
     fixed bin, fixed bin(35));

call ipc_$decl_event_call_chn (channel_id, call_chn_procedure, data_ptr,
     priority, code);
```

ARGUMENTS

channel_id
is the identifier of the event channel. (Input)

call_chn_procedure
is the procedure entry point invoked when an event occurs on the specified channel. (Input)

data_ptr
is a pointer to a region where data to be passed to and interpreted by that procedure entry point is placed. (Input)

priority
    is a number indicating the priority of this event-call channel as compared to other event-call channels declared by this process for this ring. If, upon interrogating all the appropriate event-call channels, more than one is found to have received an event, the lowest-numbered priority is honored first, and so on. (Input)

code
    is a standard status code. (Output)

Name: link_unsnap_

The link_unsnap_ subroutine restores snapped links pointing to a given segment or its linkage section. Such links then appear as if they had never been snapped (changed into ITS pairs). This is accomplished by sequentially indexing through the Linkage Offset Table (LOT) and for each linkage section listed there by searching for links to be restored.

USAGE

```
declare link_unsnap_ entry (ptr, ptr, ptr, fixed bin(17), fixed
      bin(17));

call link_unsnap_ (lot_ptr, isot_ptr, linkage_ptr, hcsc, high_seg);
```

ARGUMENTS

lot_ptr
    is a pointer to the LOT. (Input)

isot_ptr
    is a printer to the ISOT. (Input)

linkage_ptr
    is a pointer to the linkage section to be discarded. (Input)

hscs
    is one less than the segment number of the first segment that can be unsnapped. (Input)

high_seg
    is the number of LOT slots used in searching for links to be restored. (Input)

INDEX

D

F

G

I

L

N

O

T

U

CUT ALONG LINE

HONEYWELL INFORMATION SYSTEMS
Technical Publications Remarks Form

TITLE	MULTICS SUBROUTINES AND I/O MODULES	ORDER NO.	AG93-05
		DATED	FEBRUARY 1985

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

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

FROM: NAME ______________________ DATE ______________

TITLE ______________________

COMPANY ______________________

ADDRESS ______________________

____

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

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 39531 WALTHAM, MA02154

POSTAGE WILL BE PAID BY ADDRESSEE

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTN: PUBLICATIONS, MS486

Honeywell

CUT ALONG LINE

FOLD ALONG LINE

FOLD ALONG LINE

HONEYWELL INFORMATION SYSTEMS
Technical Publications Remarks Form

TITLE	MULTICS SUBROUTINES AND I/O MODULES ADDENDUM A	ORDER NO.	AG93-05A
		DATED	NOVEMBER 1986

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

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

PLEASE FILL IN COMPLETE ADDRESS BELOW.

FROM: NAME ______________________ DATE ____________

TITLE ______________________

COMPANY ______________________

ADDRESS ______________________

____

CUT ALONG LINE

CUT ALONG LINE

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

FOLD ALONG LINE

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 39531 WALTHAM, MA 02154

POSTAGE WILL BE PAID BY ADDRESSEE

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTN: PUBLICATIONS, MS486

FOLD ALONG LINE

Honeywell

MULTICS

SUBROUTINES AND I/O MODULES ADDENDUM A

SUBJECT

Additions and Changes to the Manual

SPECIAL INSTRUCTIONS

This is the first Addendum to AG93-05, dated February 1985. See "Significant Changes" in the Preface for a list of major changes. Change bars in the margins indicate technical changes and additions; asterisks denote deletions.

Note:

Insert this cover behind the manual cover to indicate the manual is updated with Addendum A.

SOFTWARE SUPPORTED

Multics Software Release 12.0

ORDER NUMBER

AG93-05A

November 1986

47166
5C287
Printed in U.S.A.

Honeywell

COLLATING INSTRUCTIONS

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

Remove	Insert
Title page, Preface	Title page, Preface
iii, blank	iii through xi
1-7, 1-8	1-7, 1-8 1-8.1, blank
1-13, 1-14	1-13, 1-14 1-14.1, blank
1-17 through 1-24	1-17 through 1-25
2-29 through 2-32	2-29 through 2-32 2-32.1, 2-32.2
2-49 through 2-52	2-49 through 2-52
2-87, 2-88	2-87, 2-88 2-88.1 through 2-88.4 2-88.5, blank
2-93 through 2-100	2-93 through 2-100 2-100.1, blank
2-137, 2-138	2-137, 2-138
2-149 through 2-152	2-149 through 2-152 2-152.1, 2-152.2
2-167, 2-168	2-167, 2-168 2-168.1, blank
2-175, 2-176	2-175, 2-176 2-176.1, blank
2-207, 2-208	2-207, 2-208 2-208.1 through 2-208.8
2-217, 2-218	2-217, 2-218 2-218.1, blank
2-227 through 2-230	2-227 through 2-230 2-230.1, blank
2-239, 2-240	2-239, 2-240 2-240.1, blank
2-243, 2-244	2-243, 2-244 2-244.1 through 2-244.4 2-244.5, blank
2-273 through 2-290	2-273 through 2-290 2-290.1 through 2-290.10
2-303, 2-304	2-303, 2-304
2-353 through 2-360	2-353 through 2-360 2-360.1, blank
2-369 through 2-384	2-369 through 2-384
2-413 through 2-416	2-413 through 2-416 2-416.1 through 2-416.4 2-416.5, blank

2-419, 2-420	2-419, 2-420 2-420.1, blank
2-427, 2-428	2-427, 2-428 2-428.1, blank
2-435, 2-436	2-435, 2-436 2-436.1, blank
2-437 through 2-450	2-437 through 2-450 2-450.1, blank
2-461, 2-462	2-461, 2-462 2-462.1, 2-462.2
2-491 through 2-502	2-491 through 2-502 2-502.1, 2-502.2 2-502.3, blank
2-519, 2-520	2-519, 2-520 2-520.1, blank
2-557 through 2-560	2-557 through 2-560 2-560.1, blank
2-565, 2-566	2-565, 2-566
2-569, 2-570	2-569, 2-570
2-583 through 2-586	2-583 through 2-586 2-586.1, 2-586.2 2-586.3, blank
2-619, 2-620	2-619, 2-620
2-635 through 2-642	2-635 through 2-642 2-642.1 through 2-642.16 2-642.17, blank
2-649, 2-650	2-649, 2-650 2-650.1, 2-650.2 2-650.3, blank
2-667, 2-668	2-667, 2-668 2-668.1 through 2-668.10 2-668.11, blank
2-681 through 2-686	2-681 through 2-686 2-686.1, blank
2-721, 2-722	2-721, 2-722 2-722.1, blank
2-731 through 2-742	2-731 through 2-742 2-742.1 through 2-742.6
2-849 through 2-858	2-849 through 2-858
2-865, 2-866	2-865, 2-866
2-885 through 2-888	2-885 through 2-888 2-888.1, blank
2-913, 2-914	2-913, 2-914

2-917, 2-918	2-917, 2-918 2-918.1, 2-918.2
2-957 through 2-960	2-957 through 2-960 2-960.1, blank
3-27, 3-28	3-27, 3-28 3-28.1, blank
3-33, 3-34	3-33, 3-34 3-34.1, blank
3-121, 3-122	3-121, 3-122 3-122.1, blank
3-153, 3-154	3-153, 3-154 3-154.1 through 3-154.20 3-154.21, blank
3-165, 3-166	3-165, 3-166 3-166.1 through 3-166.18 3-166.19, blank
3-187 through 3-194	3-187 through 3-194
3-199, 3-200	3-199, 3-200
3-221 through 3-234	3-221 through 3-234
A-1, A-2	A-1, A-2
A-11, A-12	A-11, A-12
A-19, A-20	A-19, A-20 A-21
i-1 through i-43	i-1 through i-59

Together, we can find the answers.

Honeywell

Honeywell Information Systems
U.S.A.: 200 Smith St., MS 486, Waltham, MA 02154
Canada: 155 Gordon Baker Rd., Willowdale, ON M2H 3N7
U.K.: Great West Rd., Brentford, Middlesex TW8 9DH **Italy**: 32 Via Pirelli, 20124 Milano
Mexico: Avenida Nuevo Leon 250, Mexico 11, D.F. **Japan**: 2-2 Kanda Jimbo-cho, Chiyoda-ku, Tokyo
Australia: 124 Walker St., North Sydney, N.S.W. 2060 **S.E. Asia**: Mandarin Plaza, Tsimshatsui East, H.K.

42844, 9C585, Printed in U.S.A.

AG93-05