

SERIES 60 (LEVEL 68)

SUBJECT:

Reference Guide for Advanced Multics Users, Writing Their Own Subsystems.

SPECIAL INSTRUCTIONS:

This manual is one of four manuals that constitute the Multics Programmers' Manual (MPM).

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

This manual supersedes AK92, Rev. 0, and its Addendum A. The manual has been extensively revised; therefore, marginal change indicators have not been included in this edition.

SOFTWARE SUPPORTED:

Multics Software Release 5.0

INCLUDES UPDATE PAGES ISSUED AS ADDENDUM A IN JULY 1976,
AND ADDENDUM B IN FEBRUARY 1977.

DATE:

September 1975

ORDER NUMBER:

AK92, Rev. 1

PREFACE

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

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

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

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

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

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

The MPM I/O manual contains descriptions of commands and subroutines used to perform peripheral I/O. Included in this manual are commands and subroutines that manipulate tapes and disks as I/O devices. Special purpose communications I/O, such as binary synchronous communication, is also included.

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

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

The MPM Subsystem Writers' Guide provides the advanced Multics user with a selection of some of the internal interfaces used to construct the standard Multics user interface. It also describes some specialized tools helpful to the advanced subsystem writer.

The facilities described here are subject to changes and improvements in their interface specifications. Further, at the level of the system presented by many of these interfaces, it is difficult to avoid far-reaching subsystem changes when these interfaces change. Thus, the subsystem writer is cautioned against the unnecessary use of the interfaces described in this manual.

Most interfaces described here should be used only if there is a need to bypass normal Multics procedures; i.e., in using one of these interfaces, the user risks giving up some of the desirable characteristics of Multics. For example, the standard Multics interface presents a consistency of style and interpretation to the user that the subsystem writer may find difficult to duplicate and maintain. Therefore, the subsystem writer should be cautious about unintentionally introducing different, and possibly confusing, styles and interpretations when bypassing a standard function.

However, one of the objectives of Multics is to allow the knowledgeable user to construct subsystems of almost any specification. The content of the MPM Subsystem Writers' Guide, applied with care, is intended to help fulfill this objective.

Several cross-reference facilities in the MPM help locate information:

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

CONTENTS

	Page
Section I	
Multics Standard Object Segment	1-1
Format of an Object Segment	1-1
Structure of the Text Section	1-2
Entry Sequence	1-3
Gate Segment Entry Point Transfer Vector	1-4
Structure of the Definition Section	1-5
Definition Section Header	1-7
Expression Word	1-10
Type Pair	1-10
Trap Pair	1-12
Initialization Structure for Type 5 *system and Type 6 Links	1-12
Structure of the Static Section	1-12
Structure of the Linkage Section	1-13
Linkage Section Header	1-13
Internal Storage Area	1-14
Links	1-14
First-Reference Trap	1-15
Structure of the Symbol Section	1-17
Symbol Block Header	1-17
Source Map	1-19
Relocation Information	1-20
Structure of the Object Map	1-22
Generated Code Conventions	1-24
Text Section	1-24
Entry Sequence	1-24
Text Relocation Codes	1-24
Definition Section	1-25
Definition Relocation Codes	1-25
Implicit Definitions	1-26
Linkage Section	1-26
Internal Storage	1-26
Links	1-26
Linkage Relocation Codes	1-26
Static Section	1-27
Symbol Section	1-27
Structure of Bound Segments	1-27
Internal Link Resolution	1-29
Definition Section	1-29
Binder Symbol Block	1-29
Bind Map	1-30
Section II	
Standard Execution Environment	2-1
Standard Stack and Linkage Area Formats	2-1
Multics Stack	2-1
Stack Header	2-2
Multics Stack Frame	2-5
Linkage Offset Table	2-8
Internal Static Offset Table	2-8
Subroutine Calling Sequences	2-9
Call Operator	2-10
Entry Operator	2-10
Push Operator	2-11
Return Operator	2-11

CONTENTS (cont)

Page

	Short Return Operator	2-11
	Pseudo-op Code Sequences	2-12
	Register Usage Conventions	2-13
	Argument List Format	2-13
Section III	Subsystem Programming Environment	3-1
	Writing a Process Overseer	3-1
	Process Initialization	3-1
	Process Overseer Functions	3-2
	Handling of Quit Signals	3-3
Section IV	Implementation of Input/Output Modules	4-1
	I/O Control Blocks	4-2
	I/O Control Block Structure	4-2
	Attach Pointers	4-3
	Open Pointers	4-3
	Entry Variables	4-4
	Synonyms	4-4
	Writing an I/O Module	4-5
	Design Considerations	4-5
	Implementation Rules	4-6
	Attach Operation	4-7
	Open Operation	4-8
	Close Operation	4-8
	Detach Operation	4-9
	Modes and Control Operations	4-9
	Other Operations	4-9
Section V	Reference to Commands and Subroutines by Function	5-1
	Command Repertoire	5-1
	Subroutine Repertoire	5-2
Section VI	Command Descriptions	6-1
	alm	6-4
	alm_abs, aa	6-20
	archive_sort, as	6-22
	area_status	6-23
	copy_names	6-24
	create_area	6-25
	delete_external_variables	6-26
	display_component_name, dcn	6-27
	error_table_compiler, etc	6-28
	list_external_variables	6-30
	list_temp_segments	6-31
	mbx_add_name, mban	6-33
	mbx_create, mber	6-34
	mbx_delete, mbd1	6-35
	mbx_delete_acl, mbda	6-36
	mbx_delete_name, mbdn	6-38
	mbx_list_acl, mbla	6-39
	mbx_rename, mbrn	6-40
	mbx_set_acl, mbsa	6-41
	mbx_set_max_length, mbsml	6-43
	move_names	6-44
	print_bind_map, pbm	6-45
	print_link_info, pli	6-46
	print_linkage_usage, plu	6-48
	reorder_archive	6-49
	reset_external_variables	6-51

CONTENTS (cont)

Page

set_max_length, sml	6-52
set_ring_brackets, srb	6-54
set_system_storage	6-55
set_user_storage	6-57

Section VII

Subroutine Descriptions	7-1
active_fnc_err_	7-3
aim_check_	7-5
aim_check_\$equal	7-5
aim_check_\$greater	7-5
aim_check_\$greater_or_equal	7-6
area_info_	7-6.1
ascii_to_ebcdic_	7-7
ascii_to_ebcdic_\$table	7-7
assign_	7-11
check_star_name_	7-12
check_star_name_\$path	7-12
check_star_name_\$entry	7-12
condition_interpreter_	7-14
continue_to_signal_	7-16
convert_aim_attributes_	7-17
convert_dial_message_	7-17.1
convert_status_code_	7-18
cu_	7-19
cu_\$ready_proc	7-19
cu_\$get_ready_procedure	7-20
cu_\$set_ready_procedure	7-20
cu_\$get_ready_mode	7-21
cu_\$set_ready_mode	7-21
cu_\$arg_list_ptr	7-22
cu_\$arg_ptr_rel	7-22
cu_\$af_arg_count	7-23
cu_\$af_arg_ptr	7-24
cu_\$af_return_arg	7-24
cu_\$stack_frame_ptr	7-25
cu_\$stack_frame_size	7-26
cu_\$generate_call	7-26
cu_\$set_command_processor	7-27
cu_\$get_command_processor	7-27
cu_\$set_cl_intermediary	7-28
cu_\$cl	7-28
cu_\$get_cl_intermediary	7-29
cu_\$level_set	7-29
cu_\$level_get	7-29
cu_\$decode_entry_value	7-30
cv_bin_	7-32
cv_bin_\$dec	7-32
cv_bin_\$oct	7-33
cv_entry_	7-33.1
cv_hex_	7-33.3
cv_oct_	7-34
cv_oct_check	7-34
cv_ptr_	7-34.1
decode_descriptor_	7-35
define_area_	7-36.1
dial_manager_	7-36.4
dprint_	7-37
ebcdic_to_ascii_	7-40
ebcdic_to_ascii_\$table	7-40
find_condition_info_	7-42
get_default_wdir_	7-44
get_definition_	7-44.1

CONTENTS (cont)

	Page
get_entry_name_	7-45
get_equal_name_	7-46
get_privileges_	7-48
get_ring_	7-50
get_system_free_area_	7-51
hcs_\$add_dir_inacl_entries	7-52
hcs_\$add_inacl_entries	7-54
hcs_\$del_dir_tree	7-56
hcs_\$delete_dir_inacl_entries	7-57
hcs_\$delete_inacl_entries	7-58
hcs_\$get_author	7-60
hcs_\$get_bc_author	7-61
hcs_\$get_dir_ring_brackets	7-62
hcs_\$get_max_length	7-63
hcs_\$get_max_length_seg	7-64
hcs_\$get_ring_brackets	7-65
hcs_\$get_safety_sw	7-66
hcs_\$get_safety_sw_seg	7-67
hcs_\$get_search_rules	7-68
hcs_\$get_system_search_rules	7-68.1
hcs_\$initiate_search_rules	7-69
hcs_\$list_dir_inacl	7-71
hcs_\$list_inacl	7-73
hcs_\$quota_move	7-75
hcs_\$quota_read	7-76
hcs_\$replace_dir_inacl	7-77
hcs_\$replace_inacl	7-78
hcs_\$set_dir_ring_brackets	7-79
hcs_\$set_entry_bound	7-80
hcs_\$set_entry_bound_seg	7-81
hcs_\$set_max_length	7-82
hcs_\$set_max_length_seg	7-83
hcs_\$set_ring_brackets	7-84
hcs_\$set_safety_sw	7-85
hcs_\$set_safety_sw_seg	7-86
hcs_\$star	7-87
hcs_\$star_dir_list_	7-89
hcs_\$star_list_	7-90
hcs_\$wakeup	7-93
iod_info_	7-94
iod_info_\$generic_type	7-94
iod_info_\$driver_access_name	7-94
iox_	7-96
iox_\$destroy_iocb	7-96
iox_\$err_no_operation, iox_\$err_not_open, iox_\$err_not_closed, iox_\$err_not_attached	7-96
iox_\$find_iocb_n	7-97
iox_\$look_iocb	7-97
iox_\$propagate	7-98
ipc_	7-99
ipc_\$create_ev_chn	7-99
ipc_\$delete_ev_chn	7-100
ipc_\$dcl_event_call_channel	7-100
ipc_\$decl_ev_wait_chn	7-101
ipc_\$drain_chn	7-101
ipc_\$cutoff	7-102
ipc_\$reconnect	7-102
ipc_\$set_wait_prior	7-102
ipc_\$set_call_prior	7-103
ipc_\$mask_ev_calls	7-103
ipc_\$unmask_ev_calls	7-104
ipc_\$block	7-104
ipc_\$read_ev_chn	7-105

CONTENTS (cont)

	Page
match_star_name_	7-107
msf_manager_	7-108
msf_manager_\$open.	7-108
msf_manager_\$get_ptr	7-109
msf_manager_\$adjust.	7-110
msf_manager_\$close	7-110
msf_manager_\$acl_list.	7-111
msf_manager_\$acl_replace	7-112
msf_manager_\$acl_add	7-113
msf_manager_\$acl_delete.	7-114
object_info_	7-115
object_info_\$brief	7-115
object_info_\$display	7-115
object_info_\$long.	7-116
prepare_mc_restart_	7-120
prepare_mc_restart_\$retry.	7-120
prepare_mc_restart_\$replace.	7-121
prepare_mc_restart_\$tra.	7-121
read_allowed_	7-123
read_write_allowed_	7-124
release_area_	7-124.1
signal_	7-125
stu_	7-126
stu_\$find_header	7-126
stu_\$find_block.	7-127
stu_\$get_runtime_block	7-127
stu_\$find_runtime_symbol	7-128
stu_\$decode_runtime_value.	7-129
stu_\$get_implicit_qualifier.	7-130
stu_\$get_runtime_address	7-131
stu_\$get_line_no	7-133
stu_\$get_runtime_line_no	7-134
stu_\$get_location.	7-135
stu_\$get_line.	7-135
stu_\$get_runtime_location.	7-136
stu_\$get_statement_map	7-137
stu_\$offset_to_pointer	7-137
stu_\$pointer_to_offset	7-138
stu_\$remote_format	7-139
sub_err_	7-141.1
sys_info	7-141.4
system_info_	7-142
system_info_\$installation_id	7-142
system_info_\$sysid	7-142
system_info_\$titles.	7-142
system_info_\$users	7-143
system_info_\$timeup.	7-143
system_info_\$next_shutdown	7-144
system_info_\$prices.	7-144
system_info_\$device_prices	7-145
system_info_\$shift_table	7-146
system_info_\$abs_prices.	7-146
system_info_\$io_prices	7-147
system_info_\$last_shutdown	7-147
system_info_\$access_ceiling	7-148
system_info_\$level_names	7-148
system_info_\$category_names	7-148
timer_manager_	7-150
timer_manager_\$sleep	7-151
timer_manager_\$alarm_call.	7-151
timer_manager_\$alarm_call_inhibit.	7-152
timer_manager_\$alarm_wakeup.	7-152
timer_manager_\$cpu_call.	7-152

CONTENTS (cont)

	Page
timer_manager_\$cpu_call_inhibit.	7-153
timer_manager_\$cpu_wakeup.	7-153
timer_manager_\$reset_cpu_call.	7-153
timer_manager_\$reset_cpu_wakeup.	7-153
timer_manager_\$reset_alarm_call.	7-154
timer_manager_\$reset_alarm_wakeup.	7-154
tssi_	7-155
tssi_\$get_segment.	7-155
tssi_\$get_file	7-155
tssi_\$finish_segment	7-156
tssi_\$finish_file.	7-157
tssi_\$clean_up_segment	7-157
tssi_\$clean_up_file.	7-158
tty_	7-158.1
unwinder_	7-159
vfile_	7-159.1
write_allowed_	7-160
 Appendix A Approved Control Arguments	 A-1
 Index	 i-1

ILLUSTRATIONS

Figure 1-1	Sample Definition List	1-6
Figure 1-2	Structure of a Link	1-16
Figure 1-3	Structure of a Bound Segment	1-28
Figure 2-1	Stack Header Format	2-2
Figure 2-2	Stack Frame Format	2-5
Figure 2-3	Standard Argument List	2-14
Figure 2-4	Argument Descriptor Format	2-16



SECTION I

MULTICS STANDARD OBJECT SEGMENT

A Multics object segment contains object code generated by a translator and linkage information that is used by the dynamic linking mechanism to resolve intersegment references. (See "Dynamic Linking" in the MPM Reference Guide.) The most common examples of object segments are procedure segments and data segments.

Format requirements for an object segment are primarily associated with external interfaces; thus, translator designers are permitted a great amount of freedom in the area of code and data generation. The format contains certain redundancies and unusual data structures; these are a byproduct of maintaining upward compatibility with earlier object segment formats. The dynamic linking mechanism and the standard object segment manipulation tools assume that all object segments are standard object segments.

FORMAT OF AN OBJECT SEGMENT

An object segment is divided into six sections that usually appear in the following order:

- text
- definition
- linkage
- static (if present)
- symbol
- break map (if present)

The type of information contained in each of the six sections is summarized below:

1. text contains only pure parts of the object segment (instructions and read-only data). It can also contain relative pointers to the definition, linkage and symbol sections.
2. definition contains only nonexecutable, read-only symbolic information used for dynamic linking and symbolic debugging. Since it is assumed that the definition section is infrequently referenced (as opposed to the constantly referenced text section), it should not be used as a repository for read-only constants referenced during the execution of the text section. The definition section can sometimes (as in the case of an object segment generated by the binder) be structured into definition blocks that are threaded together.

3. linkage contains the impure (i.e., modified during the program's execution) nonexecutable parts of the object segment and may consist of two types of data:
 - a. links modified at run time by the Multics linker to contain the machine address of external references, and possibly
 - b. data items to be allocated on a per-process basis such as the internal static storage of PL/I procedures.
4. static contains the data items to be allocated on a per-process basis. The static storage may be included in the linkage section in which case there is no explicit separate static section.
5. break map contains information used by the debuggers to locate breakpoints in the object segment. This section is generated by the debuggers rather than the translator and only when the segment currently contains breakpoints. Its internal format is of interest only to the debuggers.
6. symbol contains all generated items of information that do not belong in the first five sections such as the language processor's symbol tree and historical and relocation information. The symbol section may be further structured into variable length symbol blocks threaded to form a list. The symbol section contains only pure information.

The text, definition, and symbol sections are shared by all processes that reference an object segment. Usually, a copy of the linkage section is made when an object segment is first referenced in a process. That is, the linkage section is a per-process data base. The original linkage section serves only as a copying template. An exception is made for some system programs whose link addresses are filled in at system initialization time. Their linkage sections are shared by everyone who wants to use the supplied addresses. When these programs have data items in internal storage, they have a separate static section template that is copied once per process. See "Dynamic Linking" in the MPM Reference Guide and "Standard Stack and Linkage Area Formats" in Section II of this document. Normally, a segment containing break map information is in the state of being debugged and is not used by more than one process.

The object segment also contains an object map that contains the offsets and lengths of each of the sections. The object map can be located immediately before or immediately after any of the six sections. Translators normally place it immediately after the symbol section. The last word of every object segment must contain a left-justified 18-bit relative pointer to the object map.

STRUCTURE OF THE TEXT SECTION

The text section is basically unstructured, containing the machine-language representation of a symbolic algorithm and/or pure data. Its length is usually an even number of words.

Two of the items that can appear within the text section have standard formats: the entry sequence and the gate segment entry point transfer vector.

Entry Sequence

A standard entry sequence is usually provided for every externally accessible procedure entry point in an object segment. A standard entry sequence has the following format (the two structures are independent but are normally contiguous):

```
dcl 1 parm_desc_ptrs      aligned,
    2 n_args              bit(18) unaligned,
    2 descriptor_relp(n_args) bit(18) unaligned;

dcl 1 entry_sequence     aligned,
    2 descr_relp_offset  bit(18) unaligned,
    2 reserved           bit(18) unaligned,
    2 def_relp           bit(18) unaligned,
    2 flags              unaligned,
    3 basic_indicator    bit(1) unaligned,
    3 revision_1         bit(1) unaligned,
    3 has_descriptors    bit(1) unaligned,
    3 variable           bit(1) unaligned,
    3 function           bit(1) unaligned,
    3 pad                bit(13) unaligned,
    2 code_sequence(n)   bit(36) aligned;
```

where:

1. n_args is the number of arguments expected by this external entry point. This item is optional and is valid only if the flag has_descriptors equals "1"b.
2. descriptor_relp is an array of pointers (relative to the base of the text section) to the descriptors of the corresponding entry point parameters. This item is optional and is valid only if the flag has_descriptors equals "1"b.
3. descr_relp_offset is the offset (relative to the base of the text section) of the n_args item. This item is optional and is valid only if the flag has_descriptors equals "1"b.
4. reserved is reserved for future use and must be "0"b.
5. def_relp is an offset (relative to the base of the definition section) to the definition of this entry point. Thus, given a pointer to an entry point, it is possible to reconstruct its symbolic name for purposes such as diagnostics or debugging.
6. flags contains 18 binary indicators that provide information about this entry point.

basic_indicator

"1"b this is the entry point of a BASIC program
"0"b this is not the entry point of a BASIC program

revision_1

"1"b all of the entry's parameter descriptor information is with the entry sequence, i.e., none is in the definition
"0"b parameter descriptor information, if any, is with the definition

```

has_descriptors
    "1"b  the entry has parameter descriptors; i.e.,
          items      n_args,      descriptor_relp      and
          descr_relp_offset contain valid information
    "0"b  the entry does not have parameter
          descriptors

variable
    "1"b  the entry expects arguments whose number
          and types are variable
    "0"b  the number and type of arguments, if any,
          are not variable

function
    "1"b  the last parameter is to be returned by
          this entry
    "0"b  the last parameter is not to be returned by
          this entry

pad      is reserved for future use and must be "0"b

```

7. code_sequence is any sequence of machine instructions satisfying Multics standard calling conventions. See "Subroutine Calling Sequences" in Section II.

The value (i.e., offset within the text section) of the entry point corresponds to the address of the code_sequence item. (The value is stored in the formal definition of the entry point. See "Structure of the Definition" below.) Thus, if entry_offset is the value of the entry point ent1, then the def_relp item pointing to the definition for ent1 is located at word (entry_offset minus 1).

Gate Segment Entry Point Transfer Vector

For protection purposes, control must not be passed to a gate procedure at other than its defined entry points. To enforce this restriction, the first n words of a gate segment with n entry points must be an entry point transfer vector. That is, the k th word ($0 \leq k \leq n-1$) must be a transfer instruction to the k th entry point (i.e., a transfer to the code_sequence item of a standard entry sequence as described above). In this case, the value of the k th entry point is the offset of the k th transfer instruction (i.e., word k of the segment) rather than the offset of the code_sequence item of the k th entry point.

To ensure that only these entries can be used, the hardware enforced entry bound of the gate segment must be set so that the segment can be entered only at the first n locations.

STRUCTURE OF THE DEFINITION SECTION

The definition section of an object segment contains pure information that is used by the dynamic linking mechanism.

The definition section consists of a header pointing to a linked list of items describing the externally accessible named items of the object segment, followed by an unstructured area containing information describing the externally accessible named items of other object segments referenced by this object segment. The linked list is known as the definition list. The items on the list are known as definitions. The unstructured area contains expression words, type pairs, trap pairs, trap procedure information, and the symbolic names associated with external references.

A definition specifies the name of an externally accessible named item and its location in the object segment. The definition list consists of one or more definition blocks each of which consists of one or more class-3 definitions followed by zero or more definitions that are not class-3 (see "Definition Section Header" below for format). Normally, unbound object segments contain one definition block, while bound segments contain one definition block for every component object segment.

The information in the unstructured area of the definition section is used at runtime in conjunction with information in the linkage section to resolve the external references made by the object segment. This information is conceptually part of the linkage section, but is stored in the definition section so it can be shared among all the users of the segment.

Figure 1-1 shows the structure of the definition section. For more information concerning the interpretation of the information in the definition section see "Dynamic Linking" in the MPM Reference Guide.

Character strings in the definition section are stored in ALM "acc" format. This format is defined by the following PL/I declaration:

```
dcl 1 acc                aligned,
    2 length_of_string   fixed bin(8) unaligned,
    2 string              char(0 refer(length_of_string)) unaligned;
```

The first nine bits of the string contain the length of the string. Such a structure is referred to as an acc string.

The following paragraphs describe the formats of the various items in the definition section.

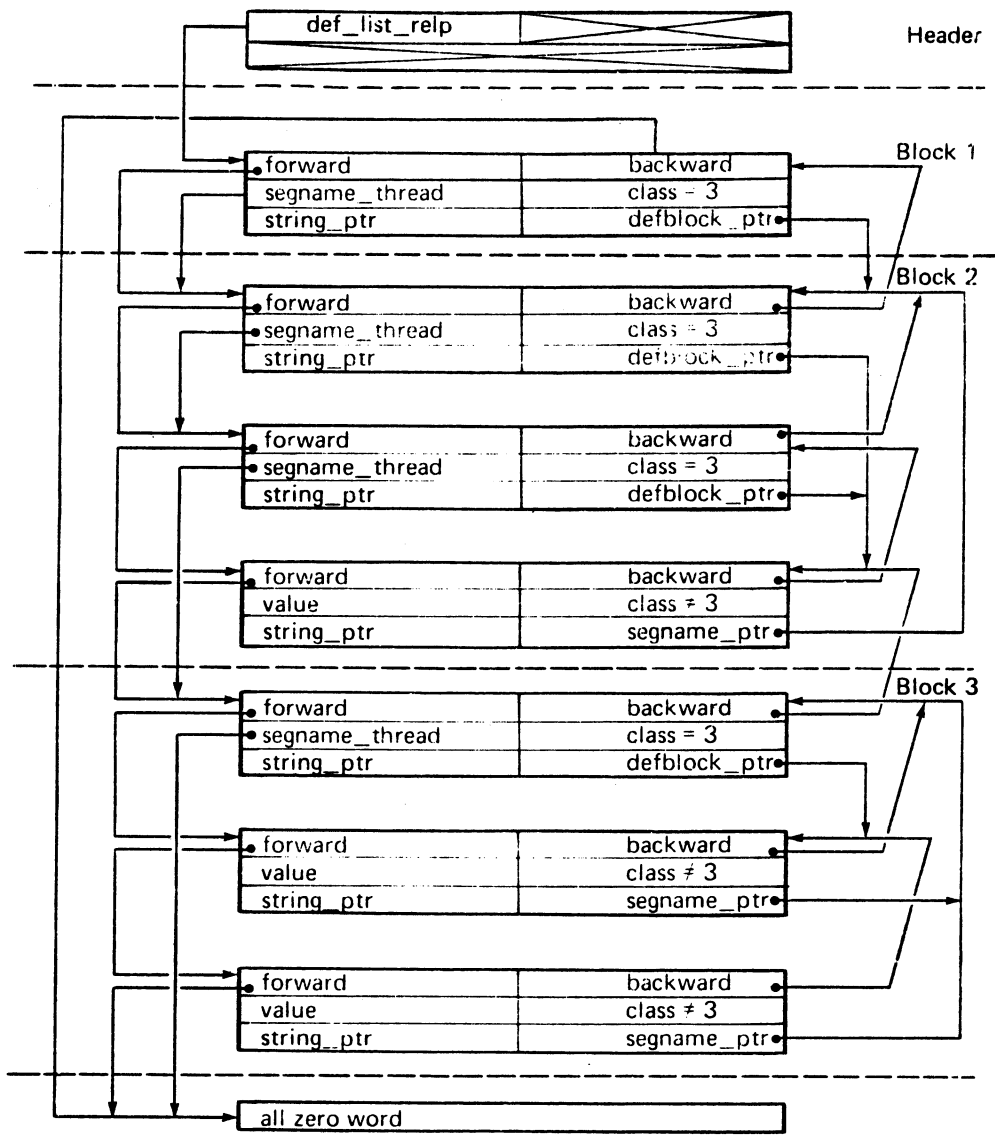


Figure 1-1. Sample Definition List

Definition Section Header

The definition section header resides at the base of the definition section and contains an offset (relative to the base of the definition section) to the beginning of the definition list.

```
dcl 1 def_header    aligned,
    2 def_list_relp bit(18) unaligned,
    2 unused        bit(36) unaligned,
    2 flags         unaligned,
    3 new_format    bit(1) unaligned initial ("1"b),
    3 ignore       bit(1) unaligned initial ("1"b),
    3 unused        bit(16) unaligned;
```

where:

1. def_list_relp is a relative pointer to the first definition in the definition list.
2. unused is reserved for future use and must be "0"b.
3. flags contains 18 binary indicators that provide information about this definition section:

new_format

```
"1"b definition section has new format
"0"b definition section has old format
```

ignore

```
"1"b if new_format equals "1"b, the Multics linker
ignores this definition.
"0"b is an old format definition
```

unused is reserved for future use and must be "0"b

The format of a definition that is not class-3 is given below.

```
dcl 1 definition    aligned,
    2 forward_thread bit(18) unaligned,
    2 backward_thread bit(18) unaligned,
    2 value           bit(18) unaligned,
    2 flags          unaligned,
    3 new_format     bit(1) unaligned,
    3 ignore         bit(1) unaligned,
    3 entry_point    bit(1) unaligned,
    3 retain         bit(1) unaligned,
    3 argcount       bit(1) unaligned,
    3 has_descriptors bit(1) unaligned,
    3 unused         bit(9) unaligned,
    2 class          bit(3) unaligned,
    2 symbol_relp    bit(18) unaligned,
    2 segname_relp   bit(18) unaligned,
    2 n_args         bit(18) unaligned,
    2 descriptor_relp (0 refer(n_args)) bit(18) unaligned;
```

where:

1. `forward_thread` is a thread (relative to the base of the definition section) to the next definition. The thread terminates when it points to a word that is 0. This thread provides a single sequential list of all the definitions within the definition section.
2. `backward_thread` is a thread (relative to the base of the definition section) to the preceding definition.
3. `value` is the offset, within the section designated by the class variable (described below), of this symbolic definition.
4. `flags` contains 15 binary indicators that provide additional information about this definition:
 - `new_format`
 - "1"b definition section has new format
 - "0"b definition section has old format
 - `ignore`
 - "1"b definition does not represent an external symbol and is, therefore, ignored by the Multics linker
 - "0"b definition represents an external symbol
 - `entry_point`
 - "1"b definition of an entry point (a variable reference through a transfer of control instruction)
 - "0"b definition of an external symbol that does not represent a standard entry point
 - `retain`
 - "1"b definition must be retained in the object segment (by the binder)
 - "0"b definition can be deleted from the object segment (by the binder)
 - `argcount`
 - "1"b (obsolete) definition includes a count of the argument descriptors (i.e., item `n_args` below contains valid information)
 - "0"b no argument descriptor information is associated with the definition
 - `has_descriptors`
 - "1"b (obsolete) definition includes an array of argument descriptor (i.e., items `n_args` and `descriptor_rop` below contain valid information)
 - "0"b no valid descriptors exist in the definition
 - `unused` is reserved for future use and must be "0"b
5. `class` this field contains a code indicating the section of the object segment to which value is relative. Codes are:
 - 0 text section
 - 1 linkage section
 - 2 symbol section
 - 3 this symbol is a segment name
 - 4 static section
6. `symbol_rop` is an offset (relative to the base of the definition section) to an aligned acc string representing the definition's symbolic name.

7. `segname_relp` is an offset (relative to the base of the definition section) to the first class-3 definition of this definition block.
8. `n_args` (obsolete) is the number of arguments expected by this external entry point. This item is present only if `argcount` or `has_descriptors` equals "1".
9. `descriptor_relp` (obsolete) is an array of pointers (relative to the base of the text section) that point to the descriptors of the corresponding entry point arguments. This item is present only if `has_descriptors` equals "1".

The obsolete items are described here to illustrate earlier versions; translators should put these items in the entry sequence of the text section. See "Entry Sequence" above.

In the case of a class-3 definition, the above structure is interpreted as follows:

```

dcl 1 segname          aligned,
    2 forward_thread  bit(18) unaligned,
    2 backward_thread bit(18) unaligned,
    2 segname_thread  bit(18) unaligned,
    2 flags           bit(15) unaligned,
    2 class           bit(3) unaligned,
    2 symbol_relp     bit(18) unaligned,
    2 first_relp      bit(18) unaligned;

```

where:

1. `forward_thread` is the same as above.
2. `backward_thread` is the same as above.
3. `segname_thread` is a thread (relative to the base of the definition section) to the next class-3 definition. The thread terminates when it points to a word that contains all 0's. This thread provides a single sequential list of all class-3 definitions in the object segment.
4. `flags` is the same as above.
5. `class` is the same as above (and has a value of 3).
6. `symbol_relp` is the same as above.
7. `first_relp` is an offset (relative to the base of the definition section) to the first nonclass-3 definition of the definition block. If the block contains no nonclass-3 definitions, it points to the first class-3 definition of the next block. If there is no next block, it points to a word that is all 0's.

The end of a definition block is determined by one of the following conditions (whichever comes first):

1. forward_thread points to an all zero word;
2. the current entry's class is not 3, and forward_thread points to a class-3 definition;
3. the current definition is class 3, and both forward_thread and first_relp point to the same class-3 definition.

The threading of definition entries is shown in Figure 1-1 above. The following paragraphs describe items in the unstructured portion of the definition section.

Expression Word

The expression word is the item pointed to by the expression pointer of an unsnapped link (see "Structure of the Linkage Section" below) and has the following structure:

```
dcl 1 exp_word      aligned,
    2 type_pair_relp bit(18) unaligned,
    2 expression     fixed bin(17) unaligned;
```

where:

1. type_pair_relp is an offset (relative to the base of the definition section) to the link's type pair.
2. expression is a signed value to be added to the offset (i.e., offset within a segment) of the resolved link.

Type Pair

The type pair is a structure that defines the external symbol pointed to by a link.

```
dcl 1 type_pair     aligned,
    2 type           bit(18) unaligned,
    2 trap_relp      bit(18) unaligned,
    2 segname_relp   bit(18) unaligned,
    2 offsetname_relp bit(18) unaligned;
```

where:

1. type assumes a value from 1 to 6:
 - 1 is a self-referencing link (i.e., the segment in which the external symbol is located is the object segment containing this link or a dynamic related section of the link) of the form:
$$\text{myself}|0+\text{expression},\text{modifier}$$
 - 2 unused; it was earlier used to define a now obsolete ITP-type link.

3 is a link referencing a specified reference name but no symbolic offset name, of the form:

refname|0+expression,modifier

4 is a link referencing both a symbolic reference name and a symbolic offset name, of the form:

refname|offsetname+expression,modifier

5 is a self-referencing link having a symbolic offset name, of the form:

myself|offsetname+expression,modifier

6 same as type 4 except that the external item is created if it is not found. (See "Dynamic Linking" in the MPM Reference Guide.) (Now Obsolete.)

2. trap_relp is an offset (relative to the base of the definition section) to either an initialization structure (if type equals 5 and segname_relp equals 5 or if type equals 6) or to a trap pair.

3. segname_relp is a code or a pointer depending on the value of type. For types 1 and 5, this item is a code that can assume one of the following values, designating the sections of the self-referencing object segment:

0 is a self-reference to the object's text section; such a reference is represented symbolically as "*text".

1 is a self-reference to the object's linkage section; such a reference is represented symbolically as "*link".

2 is a self-reference to the object's symbol section; such a reference is represented symbolically as "*symbol".

4 is a self-reference to the object's static section; such a reference is represented symbolically as "*static".

5 is a reference to an external variable managed by the linker; such a reference is represented symbolically as "*system".

For types 3, 4, and 6, this item is an offset (relative to the base of the definition section) to an aligned acc string containing the reference name portion of an external reference. (See "Constructing and Interpreting Names" in Section III of the MPM Reference Guide.)

4. offsetname_relp has a meaning depending on the value of type. For types 1 and 3, this value is ignored and must be zero. For types 4, 5, and 6, this item is an offset (relative to the base of the definition section) to an aligned acc string of an external reference. (See "Constructing and Interpreting Names" in Section III of the MPM Reference Guide for a discussion of offset names.)

Trap Pair

The trap pair is a structure that specifies a trap procedure to be called before the link associated with the trap pair is resolved by the dynamic linking mechanism. It consists of relative pointers to two links. (Links are defined under "Structure of the Linkage Section" below.) The first link defines the entry point in the trap procedure to be called. The second link defines a block of information that is passed as one of the arguments of the trap procedure. For more detailed information on trap procedures see "Dynamic Linking" in the MPM Reference Guide. The trap pair is structured as follows:

```
dcl 1 trap_pair      aligned,
    2 entry_relp    bit(18) unaligned,
    2 info_relp     bit(18) unaligned;
```

where:

1. entry_relp is an offset (relative to the base of the linkage section) to a link defining the entry point of the trap procedure.
2. info_relp is an offset (relative to the base of the linkage section) to a link defining information of interest to the trap procedure.

Initialization Structure for Type 5 *system and Type 6 Links

This structure specifies how a link target first referenced because of a type 5 *system or a type 6 link should be initialized. It has the following format:

```
dcl 1 initialization_info aligned,
    2 n_words             fixed bin,
    2 code                fixed bin,
    2 info (n_words)     bit(36) aligned;
```

where:

1. n_words is the number of words required by the new variable.
2. code indicates what type of initialization is to be performed. It can have one of the following values:
 - 0 no initialization is to be performed
 - 3 copy the info array into the newly defined variable
 - 4 initialize the variable as an area
3. info is the image to be copied into the new variable. It exists only if code is 3.

STRUCTURE OF THE STATIC SECTION

The static section is unstructured.

STRUCTURE OF THE LINKAGE SECTION

The linkage section is subdivided into four distinct components:

1. A fixed-length header that always resides at the base of the linkage section
2. A variable length area used for internal (static) storage (optional)
3. A variable length structure of links (optional)
4. First-reference trap (optional)

These four components are located within the linkage section in the following sequence:

```
header
internal storage (if present)
links (if present)
trap (if present)
```

The length of the linkage section must be an even number of words and must start on an even -word boundary; in addition, the link substructure must also begin at an even location (offset) within the linkage section.

When an object segment is first referenced in a process, its linkage section is copied into a per-process data base. At this time certain items in the copy of the header are initialized. Items not explicitly described as being initialized by the linker are set by the program that generates the object segment. In addition, the first two words of the header (containing the items `pad`, `def_section_relp`, and `first_reference_relp`) are overwritten with a pointer to the beginning of the object segment's definition section. For more information see "Dynamic Linking" in the MPM Reference Guide and "Standard Stack and Linkage Area Formats" in Section II of this manual.

Linkage Section Header

The header of the linkage section has the following format:

```
dcl 1 linkage_header      aligned,
    2 pad                bit(36),
    2 def_section_relp   bit(18) unaligned,
    2 first_reference_relp bit(18) unaligned,
    2 symbol_ptr         ptr unal,
    2 original_linkage_ptr ptr unal,
    2 unused             bit(72),
    2 links_relp         bit(18) unaligned,
    2 linkage_section_length bit(18) unaligned,
    2 object_segno       bit(18) unaligned,
    2 static_length      bit(18) unaligned;
```

where:

1. `pad` is reserved for future use and must be 0.
2. `def_section_relp` is an offset (relative to the base of the object segment) to the base of the definition section.

3. `first_reference_relp` is an offset (relative to the base of the linkage section) to the first-reference trap. This trap is activated by the linker when the first reference to this object segment is made within a given process. If the value of this item is "0"b, there is no first-reference trap.
4. `symbol_ptr` is a pointer to the object segment's symbol section. It is used by the linker to snap links relative to the symbol section. It is initialized by the linker when the header is copied.
5. `original_linkage_ptr` is a pointer to the original linkage section within the object segment. It is used by the link unsnapping mechanism and is initialized by the linker when the header is copied.
6. `links_relp` is an offset (relative to the base of the linkage section) to the first link (the base of the link array).
7. `linkage_section_length` is the entire length in words of the entire linkage section.
8. `object_segno` is the segment number of the object segment. It is initialized by the linker when the header is copied.
9. `static_length` is the length in words of the static section and is valid even when static is part of the linkage section. It is initialized by the linker if not filled in by the translator.

Internal Storage Area

The internal storage area is an array of words used by translators to allocate internal static variables and has no predetermined structure.

Links

A linkage section may contain an array of link pairs, each of which defines an external name, referenced by this object segment, whose effective address is unknown at compile time. Figure 1-2 illustrates the structure of a link.

A link must reside on an even location in memory, and must therefore be located at an even offset from the base of the linkage section. The format of a link is:

```
dcl 1 link          aligned,
    2 header_relp   bit(18) unaligned,
    2 ignore1       bit(12) unaligned,
    2 tag           bit(6) unaligned,
    2 expression_relp bit(18) unaligned,
    2 ignore2       bit(12) unaligned,
    2 modifier      bit(6) unaligned;
```


where:

1. header_relp is an offset (relative to the link itself) to the head of the linkage section. It is, in other words, the negative value of the link pair's offset within the linkage section.
2. ignore1 is reserved for future use and must be "0"b.
3. tag is a constant (46)8 that represents the hardware fault tag 2 and distinctly identifies an un-snapped link. The snapped link (ITS pair) has a distinct (43)8 tag. See "Simulated Fault" in Section VII of the MPM Reference Guide.
4. expression_relp is an offset (relative to the base of the definition section) to the expression word for this link.
5. ignore2 is reserved for future use and must be "0"b.
6. modifier is a hardware address modifier.

First-Reference Trap

It is sometimes necessary to perform certain types of initialization of an object segment when it is first referenced for execution (i.e., linked to) in a given process--for example, to store some per-process information in the segment before it is used. The first-reference trap mechanism provides this facility for use by various mechanisms, the status code assignment mechanism being an example. See "Handling of Unusual Occurrences" in Section VII of the MPM Reference Guide.

A first-reference trap consists of two relative pointers. The first points to a link defining the first reference procedure entry point to be invoked. The second points to a link defining a block of information to be passed as an argument to the first-reference procedure. For more details on first-reference traps, see "Dynamic Linking" in the MPM Reference Guide.

```
dcl 1 fr_traps      aligned,  
    2 decl_vers    fixed bin initial(1),  
    2 n_traps      fixed bin,  
    2 call_relp    bit(18) unaligned,  
    2 info_relp    bit(18) unaligned;
```

where:

1. decl_vers is the version number of the structure.
2. n_traps specifies the number of traps; it must equal 1.
3. call_relp is an offset (relative to the base of the linkage section) to a link defining a procedure to be invoked by the linker upon first reference to this object within a given process.
4. info_relp is an offset (relative to the base of the linkage section) to a link specifying a block of information to be passed as an argument to the first reference procedure; if info_relp is 0, there is no such block.

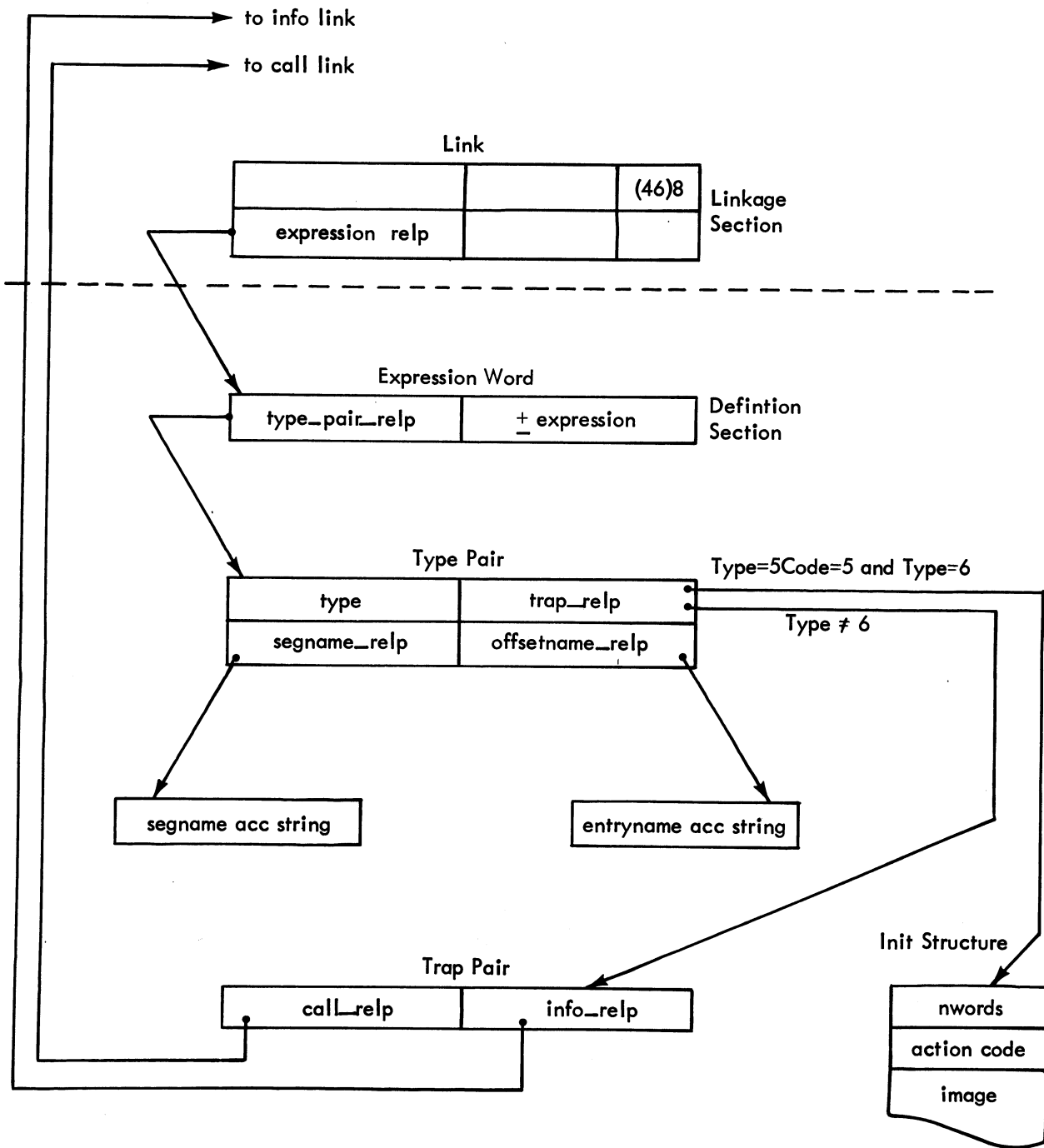


Figure 1-2. Structure of a Link

STRUCTURE OF THE SYMBOL SECTION

The symbol section consists of one or more symbol blocks threaded together to form a single list. A symbol block has two main functions: to document the circumstances under which the object segment was created, and to serve as a repository for information (relocation information, compiler's symbol tree, etc.) that does not belong in any of the other sections.

The symbol section must contain at least one symbol block, describing the circumstances under which the object segment was created. A symbol section can contain more than one symbol block. An example of multiple symbol blocks is the case of a bound segment where in addition to the symbol block describing the segment's creation by the binder, there is also a symbol block for each of the component object segments.

A symbol block consists of a fixed length header and a variable length area pointed to by the header. The contents of this area depend on the symbol block. For example, a compiler's symbol block can contain a symbol tree, and the binder's symbol block contains the bind map.

Symbol Block Header

All symbol blocks have a standard fixed-format header, although not all items in the header have meaning for all symbol blocks. The description of a particular symbol block lists items that have meaning for that symbol block. The header has the following format:

```
dcl 1 symbol_block_header          aligned,
  2 decl_vers                     fixed bin initial(1),
  2 identifier                     char(8) aligned,
  2 gen_version_number             fixed bin,
  2 gen_creation_time             fixed bin(71),
  2 object_creation_time          fixed bin(71),
  2 generator                     char(8) aligned,
  2 gen_version_name_relp         bit(18) unaligned,
  2 gen_version_name_length       bit(18) unaligned,
  2 access_name_relp             bit(18) unaligned,
  2 access_name_length           bit(18) unaligned,
  2 comment_relp                 bit(18) unaligned,
  2 comment_length               bit(18) unaligned,
  2 text_boundary                bit(18) unaligned,
  2 stat_boundary                bit(18) unaligned,
  2 source_map_relp              bit(18) unaligned,
  2 area_relp                    bit(18) unaligned,
  2 section_relp                 bit(18) unaligned,
  2 block_size                   bit(18) unaligned,
  2 next_block_thread            bit(18) unaligned,
  2 text_relocation_relp         bit(18) unaligned,
  2 def_relocation_relp          bit(18) unaligned,
  2 link_relocation_relp         bit(18) unaligned,
  2 symbol_relocation_relp       bit(18) unaligned,
  2 default_truncate            bit(18) unaligned,
  2 optional_truncate            bit(18) unaligned;
```

where:

1. decl_vers is the version number of the structure.
2. identifier is a symbolic name identifying the type of symbol block.

3. `gen_version_number` is a code designating the version of the generator that created this object segment. A generator's version number is normally changed when the generator or its output is significantly modified.
4. `gen_creation_time` is a calendar clock reading specifying the date and time when this generator was created.
5. `object_creation_time` is a calendar clock reading specifying the date and time when this symbol block was generated.
6. `generator` is the name of the processor that generated this symbol block.
7. `gen_version_relp` is an offset (relative to the base of the symbol block) to an aligned string describing the version of the generator. For example:


```
"PL/I Compiler Version 7.3
of Wednesday, July 28, 1971"
```

The integer part of the version number embedded in the string must be identical to the number stored in `gen_version_number`.
8. `gen_version_name_length` is the length of the aligned string describing the version of the generator.
9. `access_name_relp` is an offset (relative to the base of the symbol block) to an aligned string containing the access identification (i.e., the value returned by the `get_group_id` subroutine described in the MPM Subroutines) of the user for whom this symbol block was created.
10. `access_name_length` is the length of the aligned string containing the access identification of the user for whom the symbol block was created.
11. `comment_relp` is an offset (relative to the base of the symbol block) to an aligned string containing generator-dependent symbolic information. For example, a compiler might store diagnostic messages concerning nonfatal errors encountered while generating the object segment. A value of "0"b indicates no comment.
12. `comment_length` is the length of the aligned string containing generator-dependent symbolic information.
13. `text_boundary` is a number indicating the boundary on which the text section must begin. For example, a value of 32 would indicate that the text section must begin on a 0 mod 32 word boundary. This value must be a multiple of 2. It is used by the binder to determine where to locate the text section of this object segment.
14. `stat_boundary` is the same as `text_boundary` except that it applies to the internal static area of the linkage section of this object segment.
15. `source_map_relp` is an offset (relative to the base of the symbol block) to the source map (see "Source Map" below).
16. `area_relp` is an offset (relative to the base of the symbol block) to the variable-length area of the symbol block. The contents of this area depend on the symbol block.

- | | |
|----------------------------|---|
| 17. section_relp | is an offset (relative to base of the symbol block) to the base of the symbol section; that is, the negative of the offset of the symbol block in the symbol section. |
| 18. block_size | is the size of the symbol block (including the header) in words. |
| 19. next_block_thread | is a thread (relative to the base of the symbol section) to the next symbol block. This item is "0"b for the last block. |
| 20. text_relocation_relp | is an offset (relative to the base of the symbol block) to text section relocation information (see "Relocation Information" below). |
| 21. def_relocation_relp | is an offset (relative to the base of the symbol block) to definition section relocation information. |
| 22. link_relocation_relp | is an offset (relative to the base of the symbol block) to linkage section relocation information. |
| 23. symbol_relocation_relp | is an offset (relative to the base of the symbol block) to symbol section relocation information. |
| 24. default_truncate | is an offset (relative to the base of the symbol block) starting from which the binder systematically truncates control information (such as relocation bits) from the symbol section, while still maintaining such information as the symbol tree. |
| 25. optional_truncate | is an offset (relative to this base of the symbol block) starting from which the binder can optionally truncate nonessential parts of the symbol tree in order to achieve maximum reduction in the size of a bound object segment. |

Source Map

The source map is a structure that uniquely identifies the source segments used to generate the object segment. It has the following format:

```

dcl 1 source_map      aligned,
    2 decl_vers      fixed bin initial(1),
    2 size           fixed bin,
    2 map (size)     aligned,
        3 pathname_relp  bit(18) unaligned,
        3 pathname_length bit(18) unaligned,
        3 uid           bit(36) aligned,
        3 dtm          fixed bin(71);

```

where:

- | | |
|--------------|---|
| 1. decl_vers | is the version number of the structure. |
| 2. size | is the number of entries in the map array; that is, the number of source segments used to generate this object segment. |

3. `pathname_relp` is an offset (relative to the base of the symbol block) to an aligned string containing the absolute pathname of this source segment.
4. `pathname_length` is the length of the above string.
5. `uid` is the unique identifier of this source segment at the time the object segment was generated.
6. `dtm` is the date-time-modified value of this source segment at the time the object segment was created.

Relocation Information

Relocation information, designating all instances of relative addressing within a given section of the object segment, enables the relocation of the section (as in the case of binding). A variable-length prefix coding scheme is used, where there is a logical relocation item for each halfword of a given section. If the halfword is an absolute value (nonrelocatable), that item is a single bit whose value is 0. Otherwise, the item is a string of either 5 or 15 bits whose first bit is set to "1"b. The relocation information is concatenated to form a single string that can only be accessed sequentially. If the next bit is a zero, it is a single-bit absolute relocation item; otherwise, it is either a 5- or a 15-bit item depending upon the relocation codes defined below.

There are four distinct blocks of relocation information, one for each of the four object segment sections: text, definition, linkage and symbol; these relocation blocks are known as `rel_text`, `rel_def`, `rel_link` and `rel_symbol`, respectively.

The relocation blocks reside within the symbol block of the generator that produced the object segment. The correspondence between the packed relocation items and the halfwords in a given section is determined by matching the sequence of items with a sequence of halfwords, from left-to-right and from word-to-word by increasing value of address.

The relocation block pointed to from the symbol block header (e.g., `rel_text`) is structured as follows:

```
dcl 1 relinfo      aligned,
    2 decl_vers    fixed bin initial(2),
    2 n_bits       fixed bin,
    2 relbits      bit(0 refer(n_bits)) aligned;
```

where:

1. `decl_vers` is the version number of the structure.
2. `n_bits` is the length (in bits) of the string of relocation bits.
3. `relbits` is the string of relocation bits.

Following is a tabulation of the possible codes and their corresponding relocation types, followed by a description of each relocation type.

"0"b	-	absolute
"10000"b	-	text
"10001"b	-	negative text
"10010"b	-	link 18
"10011"b	-	negative link 18
"10100"b	-	link 15
"10101"b	-	definition
"10110"b	-	symbol
"10111"b	-	negative symbol
"11000"b	-	internal storage 18
"11001"b	-	internal storage 15
"11010"b	-	self relative
"11011"b	-	unused
"11100"b	-	unused
"11101"b	-	unused
"11110"b	-	expanded absolute
"11111"b	-	escape

where:

1. absolute does not relocate.
2. text uses text section relocation counter.
3. negative text uses text section relocation counter. The reason for having distinct relocation codes for negative quantities is that special coding might be necessary to convert the 18-bit field in question into its correct fixed binary form.
4. link 18 uses linkage section relocation counter on the entire 18-bit halfword. This, as well as the negative link 18 and the link 15 relocation codes apply only to the array of links in the linkage section (i.e., by definition, usage of these relocation codes implies external reference through a link).
5. negative link 18 is the same as link 18 above.
6. link 15 uses linkage section relocation counter on the low-order 15 bits of the halfword. This relocation code can only be used in conjunction with an instruction featuring a base/offset address field.
7. definition indicates that the halfword contains an address that is relative to the base of the definition section.
8. symbol uses symbol section relocation counter.
9. negative symbol is the same as symbol above.
10. internal storage 18 uses internal storage relocation counter on the entire 18-bit halfword.
11. internal storage 15 uses internal storage relocation counter on the low-order 15 bits of the halfword.
12. self relative indicates that the halfword contains a relocatable address that is referenced using a location counter modifier; the instruction is self-relocating.

13. expanded absolute allows the definition of a block of absolute relocated halfwords, for efficiency reasons. It has been established that a major part of an object program has the absolute relocation code. The five bits of relocation code are immediately followed by a fixed length 10-bit field that is a count of the number of contiguous halfwords all having an absolute relocation. Use of the expanded absolute code can be economically justified only if the number of contiguous absolute halfwords exceeds 15.
14. escape reserved for possible future use.

STRUCTURE OF THE OBJECT MAP

The object map contains information used to locate the various sections of an object segment. The map itself can be located immediately before or immediately after any one of the five sections. Translators normally place it immediately after the symbol section. The last word of the object segment (as defined by the bit count of the object segment) must contain a left-justified 18-bit offset (relative to the base of the object segment) to the object map. The object map has the following format:

```
dcl 1 object_map      aligned,
    2 decl_vers      fixed bin init(2),
    2 identifier     char(8) aligned,
    2 text_relp      bit(18) unaligned,
    2 text_length    bit(18) unaligned,
    2 def_relp       bit(18) unaligned,
    2 def_length     bit(18) unaligned,
    2 link_relp      bit(18) unaligned,
    2 link_length    bit(18) unaligned,
    2 static_relp    bit(18) unaligned,
    2 static_length  bit(18) unaligned,
    2 symb_relp      bit(18) unaligned,
    2 symb_length    bit(18) unaligned,
    2 bmap_relp      bit(18) unaligned,
    2 bmap_length    bit(18) unaligned,
    2 entry_bound    bit(18) unaligned,
    2 text_link_relp bit(18) unaligned,
    2 format         aligned,
    3 bound          bit(1) unaligned,
    3 relocatable    bit(1) unaligned,
    3 procedure      bit(1) unaligned,
    3 standard       bit(1) unaligned,
    3 separate_static bit(1) unaligned,
    3 links_in_text  bit(1) unaligned,
    3 unused         bit(30) unaligned;
```

where:

1. decl_vers is the version number of the structure.
2. identifier is the constant "obj_map".
3. text_relp is an offset (relative to the base of the object segment) to the base of the text section.
4. text_length is the length (in words) of the text section.
5. def_relp is an offset (relative to the base of the object segment) to the base of the definition section.

- 6. `def_length` is the length (in words) of the definition section.
- 7. `link_relp` is an offset (relative to the base of the object segment) to the base of the linkage section.
- 8. `link_length` is the length (in words) of the linkage section.
- 9. `static_relp` is an offset (relative to the base of the object segment) to the base of the static section.
- 10. `static_length` is the length (in words) of the static section.
- 11. `symb_relp` is an offset (relative to the base of the object segment) to the base of the symbol section.
- 12. `symb_length` is the length (in words) of the symbol section.
- 13. `bmap_relp` is an offset (relative to the base of the object segment) to the base of the break map section.
- 14. `bmap_length` is the length (in words) of the break map section.
- 15. `entry_bound` is the offset of the end of the entry transfer vector if the object segment is to be a gate.
- 16. `text_link_relp` is the offset of the first text-embedded link if `links_in_text` equals "1".
- 17. `bound` indicates if the object segment is a bound segment.
"1" the object segment is a bound segment
"0" the object segment is not a bound segment
- 18. `relocatable` indicates if the object segment is relocatable; that is, if it contains relocation information. This information (if present) must be stored in the segment's first symbol block. See "Structure of the Symbol Section" above.
"1" the object segment is relocatable
"0" the object segment is not relocatable
- 19. `procedure` indicates whether this is an executable object segment.
"1" this is an executable object segment
"0" this is not an executable object segment
- 20. `standard` indicates whether the object segment is in standard format.
"1" the object segment is in standard format
"0" the object segment is not in standard format
- 21. `separate_static` indicates whether the static section is separate from the linkage section.
"1" the static section is separate from the linkage section
"0" the static section is not separate from the linkage section
- 22. `links_in_text` indicates whether the object segment contains text-embedded links.
"1" the object segment contains text-embedded links
"0" the object segment does not contain text-embedded links
- 23. `unused` is reserved for future use and must be "0".

GENERATED CODE CONVENTIONS

The following discussion specifies those portions of generated code that must conform to a system-wide standard. For a description of the various relocation codes see "Structure of the Symbol Section" above.

Text Section

Those parts of the text section that must conform to a system-wide standard are:

- entry sequence
- text relocation codes.

ENTRY SEQUENCE

The entry sequence must fulfill two requirements:

1. The location preceding the entry point (i.e., entry point minus 1) must contain a left adjusted 18-bit relative pointer to the definition of that entry point within the definition section
2. The entry sequence executed within that entry point must store an ITS pointer to that entry point in the entry_ptr field in the stack frame header (as described in the stack frame include file). The procedure's current stack frame can then be used to determine the address of the entry point at which it was invoked. That entry's symbolic name can be reconstructed through use of its definition pointer. (See "Entry Sequence" earlier in this section.)

TEXT RELOCATION CODES

The following list defines those relocation codes that can be generated in conjunction with the text section. These can be generated only within the scope of the restrictions specified.

absolute	no restriction
text	no restriction
negative text	no restriction
link 18	can only be a direct (i.e., unindexed) reference to a link.
link 15	can only appear within the address field of a pointer-register/offset type instruction (bit 29 = "1"b). The first two bits of the modifier field of the instruction cannot be "10"b. If the instruction uses indexing, the first two bits of the modifier must be "11"b. Also the following instruction codes cannot have this relocation code: STBA (551)8 STBQ (552)8 STCA (751)8 STCQ (752)8

definition	the offset to be relocated must be that of the beginning of a definition (relative to the beginning of the definition section).
symbol	no restriction
internal storage 18	no restriction
internal storage 15	can only apply to the left half of a word. If the word is an instruction, the first two bits of the modifier must not be "10"b.
self relative	no restriction
expanded absolute	no restriction.

The restrictions imposed upon the link 15 and internal storage 15 relocation codes stem from the fact that these relocation codes apply to pointer-register/offset type address fields encountered in the address portion of machine instructions. Since the effective value of such an address is computed by the hardware at execution time, certain hardware restrictions are imposed on instructions containing them. When the Multics binder processes these instructions, it often resolves them into simple-address format and has to further modify information in the opcode (right-hand) portion of the instruction word. Therefore, these relocation codes must only be specified in a context that is comprehensible to the Multics processor.

Definition Section

Those parts of the definition section that must conform to a system-wide standard are:

- general structure
- definition relocation codes
- implicit definitions

DEFINITION RELOCATION CODES

absolute	no restriction
text	no restriction
link 18	no restriction
definition	no restriction
symbol	no restriction
internal storage 18	no restriction
self relative	no restriction
expanded absolute	no restriction

IMPLICIT DEFINITIONS

All generated object segments must feature the following implicit definition:

symbol_table defines the base of the symbol block generated by the current language processor, relative to the base of the symbol section.

Linkage Section

Those parts of the linkage section that must conform to a system-wide standard are:

internal storage
links
linkage relocation codes

INTERNAL STORAGE

The internal storage is a repository for items of the internal static storage class. It may contain data items only; it cannot contain any executable code.

LINKS

The link area can only contain a set of links. The links must be considered as distinct unrelated items, and no structure (e.g., array) of links can be assumed. They must be accessed explicitly and individually through an unindexed internal reference featuring the link 18 or the link 15 relocation codes. The order of links will not necessarily be preserved by the binder.

LINKAGE RELOCATION CODES

Only the linkage section header and the links can have relocation codes associated with them (the internal storage area has associated with it a single expanded absolute relocation item). They are:

absolute	no restriction; mandatory for the internal storage area
text	no restriction
link 18	no restriction
negative link 18	no restriction
definition	no restriction
internal storage 18	no restriction
expanded absolute	no restriction

Static Section

The static section does not have relocation codes associated with it. Absolute relocation is assumed. See "Internal Storage Area" above.

Symbol Section

The symbol section can contain information related to some other section (such as a symbol tree defining addresses of symbolic items), and therefore can have relocation codes associated with it. They are:

absolute	no restriction
text	no restriction
link 18	no restriction
definition	no restriction
symbol	no restriction
negative symbol	no restriction
internal storage 18	no restriction
self relative	no restriction
expanded absolute	no restriction

STRUCTURE OF BOUND SEGMENTS

A bound segment consists of several object segments that have been combined so that all internal intersegment references are automatically prelinked and to reduce the combined size by minimizing page breakage. The component segments are not simply concatenated; the binder breaks them apart and creates an object segment with single text, definition, static, linkage, and symbol sections as illustrated in Figure 1-3 below. (When the static section is separate, it is located before the linkage header rather than between the linkage header and the links.) As explained below, the definition section and link array are completely reconstructed while the text, internal static, and symbol sections are the corresponding concatenations of the component segments' text, internal static, and symbol sections with relocation adjustments. (See "Structure of the Symbol Section" above.) If all of the components' static sections are separate (i.e., not in linkage), the bound segment has a separate static section; otherwise, all component static sections are placed in the bound segment's linkage section.

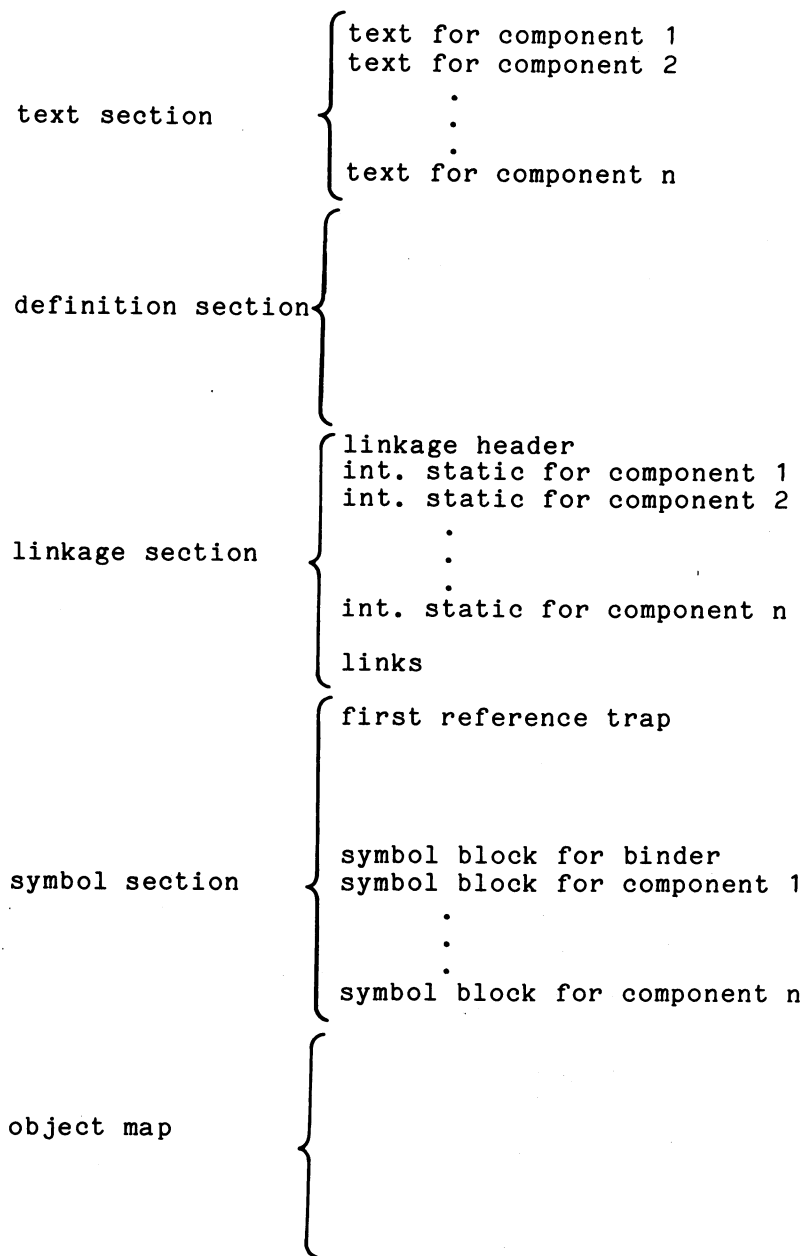


Figure 1-3. Structure of a Bound Segment

Internal Link Resolution

The primary distinction between bound and unbound groups of segments occurs in the manner in which they reference external items and are themselves referenced. Most references by one component to another component in the same bound segment are prelinked; i.e., the link references are converted to direct text-to-text references and the associated links are not regenerated. The remaining external links are combined so that for the whole bound segment there is only one link for each different target. Prelinking enables some component segments to lose their identity in cases where the bound segment itself is the main logical entity, having been coded as separate segments for ease of coding and debugging. Definitions for external entries that are no longer necessary, i.e., have become completely internal, can be omitted from the bound segment (see the bind command described in MPM Commands).

Definition Section

The definition section of a bound segment is generally more elaborate than that of an unbound object segment because it reflects both the combination and deletion of definitions. There is a definition block for each component. It contains the retained definitions and the segment names associated with the component. This organization allows definitions for multiple entries with the same name to be distinguished. The first definition block is for the binder and contains a definition for bind_map, discussed below.

Binder Symbol Block

The symbol block of the binder has a standard header if all of the components are standard object segments. The symbol block can be located using the bind_map definition. Most of the items in the header are adequately explained under "Structure of the Symbol Section" above; however, some have special meaning for bound segments. The format of a standard symbol block header is repeated below for reference, followed by the explanations specific to the binder's symbol block.

```
dcl 1 symbol_block_header      aligned,
  2 decl_vers                  fixed bin initial(1),
  2 identifier                  char(8) aligned,
  2 gen_version_number          fixed bin,
  2 gen_creation_time           fixed bin(71),
  2 object_creation_time        fixed bin(71),
  2 generator                   char(8) aligned,
  2 gen_version_name_relp        bit(18) unaligned,
  2 gen_version_name_length      bit(18) unaligned,
  2 access_name_relp             bit(18) unaligned,
  2 access_name_length           bit(18) unaligned,
  2 comment_relp                 bit(18) unaligned,
  2 comment_length               bit(18) unaligned,
  2 text_boundary                bit(18) unaligned,
  2 stat_boundary                bit(18) unaligned,
  2 source_map_relp              bit(18) unaligned,
  2 area_relp                    bit(18) unaligned,
  2 section_relp                 bit(18) unaligned,
  2 block_size                   bit(18) unaligned,
```

```

2 next_block_thread      bit(18) unaligned,
2 text_relocation_relp   bit(18) unaligned,
2 def_relocation_relp    bit(18) unaligned,
2 link_relocation_relp   bit(18) unaligned,
2 symbol_relocation_relp bit(18) unaligned,
2 default_truncate       bit(18) unaligned,
2 optional_truncate      bit(18) unaligned;

```

where:

2. identifier is the string "bind_map".
6. generator is the string "binder".
11. comment_relp is always "0"b.
16. area_relp is an offset (relative to the base of the symbol block) to the beginning of the bind map. (See "Bind Map" below.)

Bound segments currently are not relocatable, so none of the relocation relative pointers or truncation offsets have any meaning.

Bind Map

The bind map is part of the symbol block produced by the binder and describes the relocation values assigned to the various sections of the bound component object segments. It consists of a variable length structure followed by an area in which variable length symbolic information is stored. The bind map structure has the following format:

```

dcl 1 bindmap based          aligned,
  2 decl_vers              fixed bin initial(1),
  2 n_components          fixed bin,
  2 component(0 refer(n_components)) aligned,
    3 name_relp           bit(18)unaligned,
    3 name_length        bit(18) unaligned,
    3 generator_name     char(8) aligned,
    3 text_relp          bit(18) unaligned,
    3 text_length       bit(18) unaligned,
    3 static_relp       bit(18) unaligned,
    3 static_length    bit(18) unaligned,
    3 symbol_relp      bit(18) unaligned,
    3 symbol_length    bit(18) unaligned,
    3 defblock_relp    bit(18) unaligned,
    3 number_of_blocks bit(18) unaligned,
  2 bindfile_name        aligned,
    3 bindfile_name_relp bit(18)unaligned,
    3 bindfile_name_length bit(18)unaligned,
  2 bindfile_date_updated char(24),
  2 bindfile_date_modified char(24);

```

where:

1. decl_vers is a constant designating the format of this structure; this constant is modified whenever the structure is, allowing system tools to easily differentiate between several incompatible versions of a single structure.
2. n_components is the number of component object segments bound within this bound segment.

3. component is a variable-length array featuring one entry per bound component object segment.
4. name_relp is the offset (relative to the base of the bind map structure) of the symbolic name of the bound component. This is the name under which the component object was identified within the archive file used as the binder's input (i.e., the name corresponding to the object's objectname entry in the bindfile).
5. name_length is the length (in characters) of the component's name.
6. generator_name is the name of the translator that created this component object segment.
7. text_relp is the offset (relative to the base of the bound segment) of the component's text section.
8. text_length is the length (in words) of the component's text section.
9. static_relp is the offset (relative to the base of the static section) of the component's internal static.
10. static_length is the length of the component's internal static.
11. symbol_relp is an offset (relative to the base of the symbol section) to the component's symbol section.
12. symbol_length is the length of the component's symbol section.
13. defblock_relp if nonzero, this is a pointer (relative to the base of the definition section) to the component's definition block (first class-3 segname definition of that component's definition block).
14. number_of_blocks is the number of symbol blocks in the component's symbol section.
15. bindfile_name_relp is the offset (relative to the base of the symbol section) of the symbolic name of the bindfile.
16. bindfile_name_length is the length (in characters) of the bindfile name.
17. bindfile_date_updated is the date, in symbolic form, that the bindfile was updated in the archive (of object segments) used as input by the binder.
18. bindfile_date_modified is the date, in symbolic form, that the bindfile was last modified before being put into the binder's object archive.



SECTION II

STANDARD EXECUTION ENVIRONMENT

STANDARD STACK AND LINK AREA FORMATS

Because of the linkage mechanism, stack manipulations, and the complexity of the Multics hardware, a series of Multics execution environment standards have been adopted. All standard translators (including assemblers) adhere to these standards as do all supervisor and standard storage system procedures. Furthermore, they assume that other procedures do so as well.

Multics Stack

The normal mode of execution in a standard Multics process uses a stack segment. There is one stack segment for each ring. The stack for a given ring has the entryname `stack_R`, where R is the ring number, and is located in the process directory. Each stack contains a "header" followed by as many "stack frames" as are required by the executing procedures. A stack header contains pointers to special code and data that are initialized when the stack is created. Some of these pointers are variable and change during process execution. They are included in the stack header so that they can always be retrieved without supervisor intervention (for efficiency). The actual format of the stack header is described under "Stack Header" below.

Stack frames begin at a location specified in the stack header, are variable in length, and contain both control information and data for dynamically active procedures. In general, a stack frame is allocated by the procedure to which it belongs when that procedure is invoked. The stack frames are threaded to each other with forward and backward pointers, making it an easy task to trace the stack in either direction. The stack usage described below is critical to normal Multics operation; any deviations from the stated discipline can result in unexpected behavior.

Stack Header

The stack header contains pointers (on a per-ring basis) to information about the process, to operator segments, and to code sequences that can be used to invoke the standard call, push, pop, and return functions (described below). Figure 2-1 gives the format of the stack header. The following descriptions are based on that figure and on the following PL/I declaration.

+0	Reserved			Old Lot Pointer	Combined Static Pointer
+8	Combined Linkage Pointer	Max Lot Size	Current Lot Size	System Storage Pointer	User Storage Pointer
+16	Null Pointer	Stack Begin Pointer		Stack End Pointer	Lot Pointer
+24	Signal Pointer	BAR Mode Stack Pointer		PL/I Operators Pointer	Call Operator Pointer
+32	Push Operator Pointer	Return Operator Pointer		Short Return Operator Ptr	Entry Operator Pointer
+40	Translator Operator Pointer	Internal Static Offset Table Pointer		System Condition Table Pointer	Unwinding Procedure Pointer
+48	*system Link Info Pointer				
+56	Reserved				
+64					

Figure 2-1. Stack Header Format

```
dcl 1 stack_header based
  2 pad1(4)                aligned,
  2 old_lot_ptr            fixed bin,
  2 combined_stat_ptr     ptr,
  2 clr_ptr               ptr,
  2 max_lot_size         fixed bin(17) unaligned,
  2 pad2                 bit(18) unaligned,
  2 cur_lot_size         fixed bin(17) unaligned,
  2 pad3                 bit(18) unaligned,
  2 system_storage_ptr   ptr,
  2 user_storage_ptr     ptr,
  2 null_ptr             ptr,
  2 stack_begin_ptr     ptr,
  2 stack_end_ptr       ptr,
  2 lot_ptr              ptr,
  2 signal_ptr           ptr,
  2 bar_mode_sp_ptr      ptr,
  2 pl1_operators_ptr    ptr,
  2 call_op_ptr          ptr,
```

```

2 push_op_ptr          ptr,
2 return_op_ptr        ptr,
2 short_return_op_ptr ptr,
2 entry_op_ptr         ptr,
2 trans_op_tv_ptr     ptr,
2 isot_ptr             ptr,
2 sct_ptr              ptr,
2 unwinder_ptr         ptr,
2 sys_link_info_ptr   ptr,
2 pad4(14)             fixed bin;

```

where:

1. pad1 is unused.
2. old_lot_ptr is a pointer to the linkage offset table (LOT) for the current ring. This field is obsolete.
3. combined_stat_ptr is a pointer to the area in which separate static sections are allocated.
4. clr_ptr is a pointer to the area in which linkage sections are allocated.
5. max_lot_size is the maximum number of words (entries) that the LOT and internal static offset table (ISOT) can have.
6. pad2 is unused.
7. cur_lot_size is the current number of words (entries) in the LOT and ISOT.
8. pad3 is unused.
9. system_storage_ptr is a pointer to the area used for system storage, which includes command storage and the *system link name table.
10. user_storage_ptr is a pointer to the area used for user storage, which includes FORTRAN common and PL/I external static variables whose names do not include "\$".
11. null_ptr contains a null pointer value. In some circumstances, the stack header can be treated as a stack frame. When this is done, the null pointer field occupies the same location as the previous stack frame pointer of the stack frame. (See "Multics Stack Frame" below.) A null pointer indicates that there is no stack frame prior to the current one.
12. stack_begin_ptr is a pointer to the first stack frame on the stack. The first stack frame does not necessarily begin at the end of the stack header. Other information, such as the linkage offset table, can be located between the stack header and the first stack frame.
13. stack_end_ptr is a pointer to the first unused word after the last stack frame. It points to the location where the next stack frame is placed on this stack (if one is needed). A stack frame must be a multiple of 16 words; thus, both of the above pointers point to 0 (mod 16) word boundaries.

14. lot_ptr is a pointer to the linkage offset table (LOT) for the current ring. The LOT contains packed pointers to the dynamic linkage sections known in the ring in which the LOT exists. The linkage offset table is described below under "Linkage Offset Table."
15. signal_ptr is a pointer to the signalling procedure to be invoked when a condition is raised in the current ring.
16. bar_mode_sp_ptr is a pointer to the stack frame in effect when BAR mode was entered. (This is needed because typical BAR mode programs can change the word offset of the stack frame pointer register.)
17. pl1_operators_ptr is a pointer to the standard operator segment used by PL/I. It is used by PL/I and FORTRAN object code to locate the appropriate operator segment.
18. call_op_ptr is a pointer to the Multics standard call operator used by ALM procedures. It is used to invoke another procedure in the standard way.
19. push_op_ptr is a pointer to the Multics standard push operator that is used by ALM programs when allocating a new stack frame. All push operations performed on a Multics stack should use either this or an equivalent operator; otherwise results are unpredictable. (The push operation was formerly called save.)
20. return_op_ptr is a pointer to the Multics standard return operator used by ALM procedures. It assumes that a push has been performed by the invoking ALM procedure and pops the stack prior to returning control to the caller of the ALM procedure.
21. short_return_op_ptr is a pointer to the Multics standard short return operator used by ALM procedures. It is invoked by a procedure that has not performed a push to return control to its caller.
22. entry_op_ptr is a pointer to the Multics standard entry operator. The entry operator does little more than find a pointer to the invoker's linkage section.
23. trans_op_tv_ptr points to a vector of pointers to special language operators; this table can be expanded to accommodate new languages without causing a change in the stack header.
24. isot_ptr is a pointer to the internal static offset table (ISOT). The ISOT contains packed pointers to the dynamic internal static sections known in the ring in which the ISOT exists.
25. sct_ptr is a pointer to the system condition table (SCT) used by system code in handling certain events.
26. unwinder_ptr is a pointer to the unwinding procedure to be invoked when a nonlocal goto is executed in the current ring.

27. sys_link_info_ptr is a pointer to the *system link name table.
28. pad4 is unused.

The call, push, return, short return, and entry operators are invoked by the object code generated by the ALM assembler. Other translators that intend to use the standard call/push/return strategy should either use these operators or an operator segment with a set of operators consistent with these. For a detailed description of what the operators do and how to invoke them, see "Subroutine Calling Sequences" later in this section.

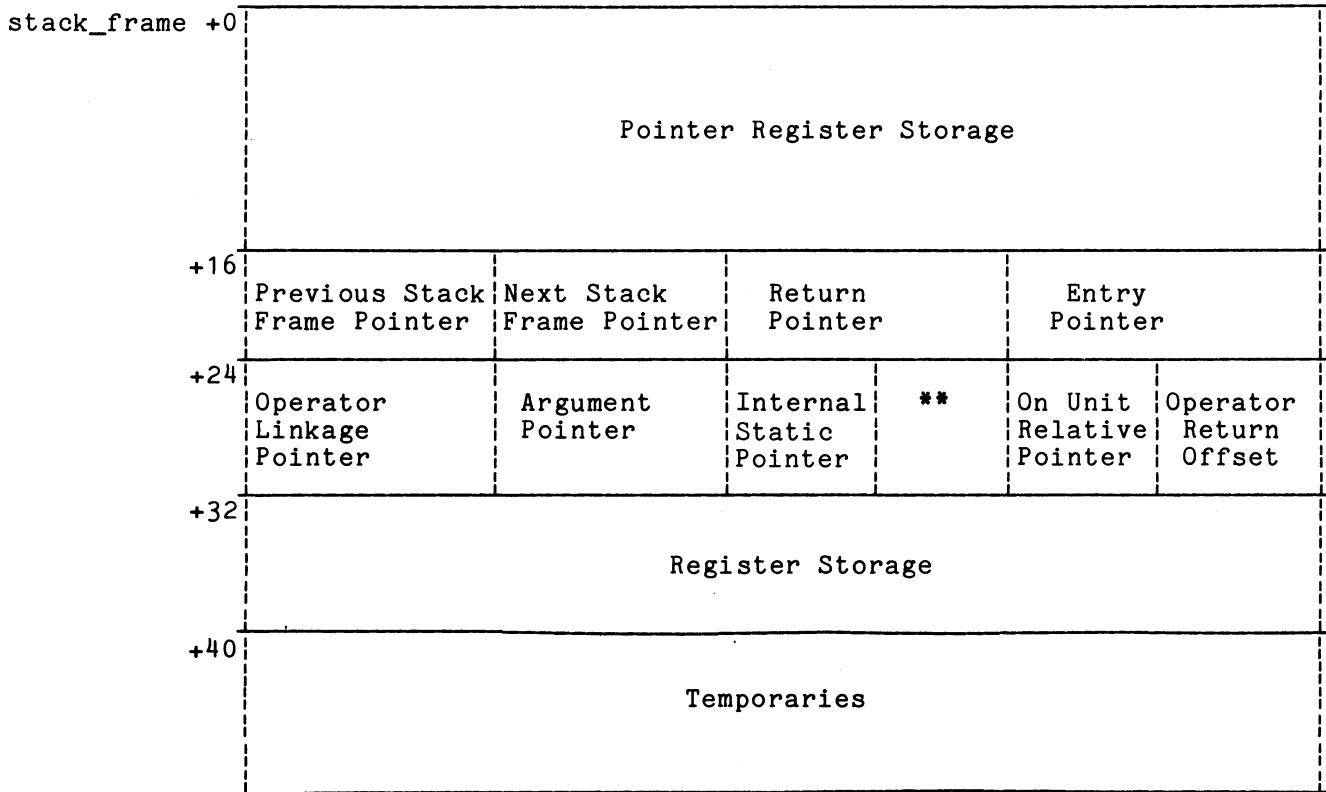
The PL/I and FORTRAN compilers use slightly different operators that perform equivalent and compatible functions. All supported translators, however, depend on the effects generated by these operators.

Multics Stack Frame

The format given below for a standard Multics stack frame must be strictly followed because several critical procedures of the Multics system depend on it. A bad stack segment or stack frame can easily lead to process termination, looping, and other undesirable effects.

In the discussion that follows, the "owner" of a stack frame is the procedure that created it (with a push operation). Some programs (generally ALM programs) never perform a push and hence do not own a stack frame. If a procedure that does not own a stack frame is executing, it can neither call another procedure nor use stack temporaries; all stack information refers to the program that called such a program.

Figure 2-2 illustrates the detailed structure of a stack frame. The following descriptions are based on that diagram and on the following PL/I declaration.



** Reserved

Figure 2-2. Stack Frame Format

```

dcl 1 stack_frame          based (sp) aligned,
    2 prs(16)              fixed bin,
    2 prev_stack_frame_ptr ptr,
    2 next_stack_frame_ptr ptr,
    2 return_ptr           ptr,
    2 entry_ptr            ptr,
    2 operator_link_ptr   ptr,
    2 argument_ptr        ptr,
    2 static_ptr          ptr unaligned,
    2 reserved            fixed bin,
    2 on_unit_rel_ptrs(2) bit(18) unaligned,
    2 translator_id       bit(18) unaligned,
    2 operator_return_offset bit(18) unaligned,
    2 regs(8)             fixed bin;

```

where:

1. prs is used to save pointer registers of the calling program when the ALM call operator is invoked.
2. prev_stack_frame_ptr is a pointer to the base of the stack frame of the procedure that called the procedure owning the current stack frame. This pointer may or may not point to a stack frame in the same stack segment.

3. `next_stack_frame_ptr` is a pointer to the base of the next stack frame. For the last stack frame on a stack, the pointer points to the next available area in the stack where a procedure can lay down a stack frame; i.e., it has the same value as the `stack_end_ptr` in the stack header. The previous stack frame pointers and the next stack frame pointers form threads through all active frames on the stack. These two threads are used by debugging tools to search and trace the stack as well as by the call/push/return mechanism.
4. `return_ptr` is a pointer to the location to which a return can be made in the procedure that owns the given frame. This pointer is undefined if the procedure has never made an external call, and points to the return location associated with the last external call if the given procedure has been returned to and is currently executing.
5. `entry_ptr` is a pointer to the procedure entry point that was called and that owns the stack frame. The pointer points to a standard entry point. See "Structure of the Text Section" in Section I.
6. `operator_link_ptr` is usually the operator pointer being used by the procedure that owns the given stack frame. For ALM programs, this points to the linkage section of the procedure.
7. `argument_ptr` is a pointer to the argument list passed to the procedure that owns the given stack frame.
8. `static_ptr` is a pointer to the internal static storage for the procedure owning the stack frame.
9. `reserved` is reserved for future use.
10. `on_unit_rel_ptrs` is a pair of relative pointers to on unit information contained within the stack frame. This on unit information is valid only if bit 29 of the second word of `prev_stack_frame_ptr` is a 1. (This bit is automatically set to 0 when a push is performed by the procedure that owns the stack frame.) The first of the `on_unit_rel_ptrs` is a pointer (relative to the stack frame base) to a list of enabled conditions. The second of the `on_unit_rel_ptrs` is obsolete.
11. `translator_id` is a coded number indicating the translator used to generate the object code of the owner of the stack frame.
12. `operator_return_offset` contains a return location for certain `pl1_operators_` functions. If it is nonzero, it is a relative pointer to the return location in the compiled program (return from `pl1_operators_`). If it is zero, a dedicated register (known by `pl1_operators_`) contains the return location.
13. `regs` is used to save arithmetic registers of the calling program when the ALM call operator is invoked.

Two major areas of a stack frame not explicitly defined above are the first 16 words and words 32 through 39. The contents of these areas is not always defined or meaningful, although they have a well-defined purpose for ALM programs and are used internally by the PL/I and FORTRAN programs. The procedure owning the stack frame can use these areas as it sees fit.

Linkage Offset Table

As described above, each stack header contains a pointer to the linkage offset table (LOT) for the current ring. The LOT is an array, indexed by text segment number, of packed pointers to the linkage sections for the procedure segments known in the current ring.

The structure of the LOT is defined by the following PL/I declaration:

```
dcl 1 lot based (lot_ptr)           aligned,  
    2 linkage_ptr (0: stack_header.cur_lot_size-1) ptr unaligned;
```

where linkage_ptr is the array of linkage section pointers.

If one of the slots in the linkage_ptr array contains all 0's, the segment number associated with the slot either does not correspond to a known segment or corresponds to a segment that does not have a linkage section allocated.

Internal Static Offset Table

The stack header in each ring contains a pointer to the internal static offset table (ISOT) for the current ring. The ISOT is an array, indexed by text segment number, of packed pointers to the internal static sections for the corresponding procedure segments known in the current ring. Since the ISOT always immediately follows the LOT, the isot_ptr is redundant but is retained for efficiency.

The internal static pointers are identical to the linkage section pointers unless the corresponding object segment was generated with separate static. If the static is separate, i.e., not allocated in the linkage section, the internal static pointer either points to the allocated static or contains a value that causes an "isot fault" if referenced.

The structure of the ISOT is defined by the following PL/I declaration:

```
dcl 1 isot based (isot_ptr)         aligned,  
    2 static_ptr (0: stack_header.cur_lot_size-1) ptr unaligned;
```

where static_ptr is the array of static/linkage section pointers.

SUBROUTINE CALLING SEQUENCES

The Multics standard call and return conventions are described in the following paragraphs. For information about the format of stack segments and stack frames, see "Standard Stack and Linkage Area Formats" above.

The call and return from one procedure to another can be broken down into seven separate steps. Operators to perform these steps have been provided in the standard operator segment named `pl1_operators_` (for PL/I, FORTRAN, and ALM procedures). These operators are invoked when appropriate by the object code generated by these translators.

The steps involved in a call and return and the associated operators are listed below.

1. A procedure call, i.e., a transfer of control and passing of an argument list pointer to the called procedure (call).
2. Generation of a linkage (and internal static) pointer for the called procedure (entry).
3. Creation of a stack frame for the called procedure (push).
4. Storage of standard items to be saved in the stack frame of the called procedure (entry and push).
5. Release of the stack frame of the called procedure just prior to returning (return).
6. Reestablishment of the execution environment of the calling procedure (return and `short_return`).
7. Return of control to the calling procedure (return and `short_return`).

Preparation of the argument list, although necessary, was not listed above because the operators need know nothing about the format of an argument list. See "Argument List Format" later in this section.

The following description is based on the operators used by ALM procedures. The operators used by PL/I and FORTRAN procedures are basically the same but differ at a detailed level due to: (1) slight changes in the execution environment when PL/I and FORTRAN programs are running; and (2) simplification and combination of operators made possible by the execution environment of PL/I. The PL/I and FORTRAN operators are not described here other than to define a minimum execution environment that must be established when returning to a PL/I or FORTRAN program.

(The following description is given in terms of Honeywell hardware.)

Call Operator

The call operator transfers control to the called procedure. This operator is invoked in two ways from ALM procedures. The first is a result of the call pseudo-op, which invokes the call operator after saving the machine registers in the calling program's stack frame and loading pointer register 0 with a pointer to the argument list to be passed to the called procedure. Upon return to the calling program, these saved values are restored into the hardware registers by the calling procedure. The second way that ALM procedures can invoke the call operator is through the short_call pseudo-op. This is used when the calling procedure does not need all of the machine registers saved and restored across the call. The ALM procedure can selectively save whatever registers are needed.

Neither the call nor the short_call pseudo-ops (nor the PL/I and FORTRAN equivalents) require or expect the machine registers to be restored by the called procedure. In fact, only the pointer registers 0 (operator segment pointer) and 6 (stack frame pointer) are ever guaranteed to be restored across a call. It is up to the calling procedure to save and restore any other machine registers that are needed.

Entry Operator

The entry operator used by ALM programs performs two functions. It generates a pointer to the linkage section of the called procedure (which it leaves in pointer register 4) and it stores a pointer to the entry in what will be the stack frame of the called procedure (if the procedure ever creates a stack frame for itself). At the time the entry operator is invoked, a new stack frame has not yet been established. Indeed, the called procedure may never create one. However, it is certainly possible to know where the stack frame will go if and when it is created and this knowledge is used to store the entry pointer.

The entry operator is invoked by an ALM procedure that transfers to a label in another procedure that has been declared as an entry through the entry pseudo-op. The transfer is made to a standard entry structure the first executable word of which is (PR7 is assumed to point to the base of the current stack segment):

```
tsp2 7|entry_op,*
```

The operator returns to the instruction after the tsp2 instruction, which may or may not be another transfer instruction. (A link to the entry, when snapped, points to the tsp2 instruction.) See "Structure of the Text Section" in Section I.

Some ALM programs may not require a linkage pointer. Such programs can declare the label to which control should be transferred with a segdef pseudo-op. This causes the appropriate definition and linkage information to be generated so that other procedures can find the entry point. When called, the transfer is straight to the code at the label and the normal entry structure is not generated or used. No linkage pointer is found and no entry pointer is saved. This technique is recommended only where speed of execution is of utmost importance since it avoids calculation of useful diagnostic information.

Push Operator

The push operator used by ALM procedures is invoked as a result of the push pseudo-op that is used to create a stack frame for the called procedure. In addition to creating a stack frame, several pointers are saved in the new stack frame. They are:

1. Argument pointer
2. Linkage pointer (and internal static pointer)
3. Previous stack frame pointer
4. Next stack frame pointer

If the called procedure is defined as an entry (rather than segdef), the entry pointer has already been saved in the new stack frame.

The push pseudo-op must be invoked if the called procedure makes further calls itself or uses temporary storage. Due to their manner of execution, PL/I and FORTRAN procedures combine the entry and push operators into a single operator.

The push operator and the return operators are managers of the stack frames and the stack segment in general. The push operator establishes the forward and backward stack frame threads and updates the stack end pointer in the stack header appropriately. The return operators use these threads and also update the stack end pointer as needed. Any program that wishes to duplicate these functions must do so in a way that is compatible with the procedures outlined in this discussion and those described above under the heading "Standard Stack and Linkage Area Formats".

Return Operator

The return operator is invoked by ALM procedures that have specified the return pseudo-op. The return operator pops the stack, reestablishes the minimum execution environment, and returns control to the calling procedure. The only registers restored are pointer registers 0 and 6, as mentioned above.

Short Return Operator

The short_return operator is invoked by ALM procedures that have specified the short_return pseudo-op. The short_return operator differs from the return operator in that the stack frame is not popped. This return is used by ALM procedures that did not perform a push.

Pseudo-op Code Sequences

The following code sequences are generated by the assembler for the specified pseudo-op.

OBJECT	CODE	OPERATORS
call:		
	spr1 6 0	
	sreg 6 32	
	epp0 arglist	
	epp2 entrypoint	
	tsp4 7 call_op,*	
		spr14 6 return_ptr
		sti 6 return_ptr+1
		epp4 6 lp_ptr,*
		call6 2 0
	lpri 6 0	
	lreg 6 32	
short_call:		
	epp2 entrypoint	
	tsp4 7 call_op,*	
		(as above)
	epp4 sp lp_ptr,*	
return:		
	tra 7 return_op,*	
		spr16 7 stack_end_ptr
		epp6 6 prev_sp,*
		epp7 6 0
		epp0 6 op_ptr,*
		ldi 6 return_ptr+1
		rtcd 6 return_ptr
short_return:		
	tra 7 short_return_op,*	
		epp7 6 0
		epp0 6 op_ptr,*
		ldi 6 return_ptr+1
		rtcd 6 return_ptr
entry:		
	tsp2 7 entry_op,*	
		epp2 2 -1
		epp4 7 stack_end_ptr,*
		spr12 4 entry_ptr
		epaq 2 0
		lprp5 7 isot_ptr,*au
		sprp5 4 static_ptr
		lprp4 7 lot_ptr,*au
		tra 2 1
	tra executable_code	

push:

eax7	stack_frame_size		
tsp2	7 push_op,*		
		spri2	7 stack_end_ptr,*
		epp2	7 stack_end_ptr,*
		spri6	2 prev_sp
		spri0	2 arg_ptr
		spri4	2 lp_ptr
		epp6	2 0
		epp2	6 0,7
		spri2	7 stack_end_ptr
		spri2	6 next_sp
		eax7	1
		stx7	6 translator_id
		tra	6 0,*

Register Usage Conventions

The following conventions, used in the standard environment, should be followed by any user-written translator.

1. The only registers that are restored across a call are the pointer registers:

0 (ap) operator segment pointer

6 (sp) stack frame pointer

The operator segment pointer is restored correctly only if it is saved at some time prior to the call (e.g., at entry time).

2. The code generated by the ALM assembler assumes that pointer register 4 (lp) always points to the linkage section for the executing procedure and that pointer register 7(sb) always points to the stack header.
3. Pointer register 7 is assumed to be pointing to the base of the stack when control is passed to a called procedure.

Argument List Format

When a standard call is performed, the argument pointer (pointer register 0) is set to point at the argument list to be used by the called procedure. The argument list is a sequence of pointers and control information about the arguments. The argument list header contains a count of the number of arguments, a count of the number of descriptors, and a code specifying whether the argument list contains an extra stack frame pointer. The format of the argument list is shown in Figure 2-3.

The argument list must begin on an even word boundary. The pointers in the argument list need not be ITS pointers; however, they must be pointers through which the hardware can perform indirect addressing. Packed (unaligned) pointers cannot be used.

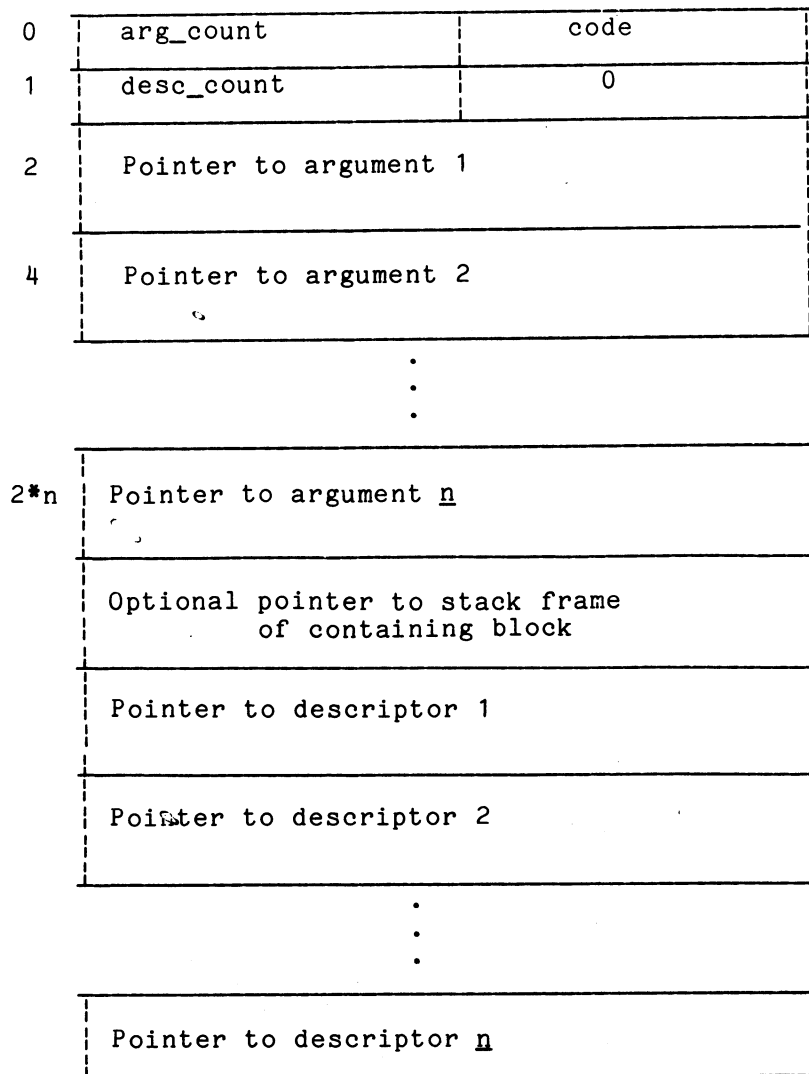


Figure 2-3. Standard Argument List

where:

- n is the number of arguments passed to the called procedure.
- arg_count is in the left half of word 0; it is two times the number of arguments passed.
- code is in the right half of word 0; it is 4 for normal intersegment calls and 10 (octal) for calling sequences that contain an extra stack frame pointer. This pointer occupies the two words following the last argument pointer. It is present for calls to PL/I internal procedures and for calls made through PL/I entry variables.
- desc_count is in the left half of word 1; it is two times the number of descriptors passed. If this number is nonzero, it must be the same as arg_count.

An argument pointer points directly to an argument. A descriptor pointer points to the descriptor associated with the argument.

The format of an argument descriptor is described by the following PL/I declaration:

```
dcl 1 descriptor    aligned,
  (2 flag          bit(1),
   2 type          bit(6),
   2 packed        bit(1),
   2 number_dims   bit(4),
   2 size          bit(24)) unaligned;
```

where:

1. flag always has the value "1"b and is used to tell this descriptor format from an earlier format. (Shown as 1 in the descriptor below.)
2. type is the data type according to the following encoding:
 - 1 real fixed binary short
 - 2 real fixed binary long
 - 3 real floating binary short
 - 4 real floating binary long
 - 5 complex fixed binary short
 - 6 complex fixed binary long
 - 7 complex floating binary short
 - 8 complex floating binary long
 - 9 real fixed decimal
 - 10 real floating decimal
 - 11 complex fixed decimal
 - 12 complex floating decimal
 - 13 pointer
 - 14 offset
 - 15 label
 - 16 entry
 - 17 structure
 - 18 area
 - 19 bit string
 - 20 varying bit string
 - 21 character string
 - 22 varying character string
 - 23 file
3. packed has the value "1"b if the data item is packed. (Shown as "p" in the typical descriptor below.)

4. number_dims is the number of dimensions in an array. (Shown as "m" in the descriptor below.) The array bounds and multipliers follow the basic descriptors in the following manner:

1	type	p	m	size	basic descriptor
lower bound					descriptive information
upper bound					for the mth
multiplier					(rightmost) dimension
⋮					
lower bound					descriptive information
upper bound					for the first
multiplier					(leftmost) dimension

If the data is packed, the multipliers give the element separation in bits; otherwise, they give the element separation in words.

5. size is the size (in bits, characters, or words) of string or area data, the number of structure elements for structure data, or the scale and precision (as two 12-bit fields) for arithmetic data. For arithmetic data, the scale is recorded in the leftmost 12 bits and the precision is recorded in the rightmost 12 bits. The scale is a 2's complement, signed value.

The descriptor of a structure is immediately followed by descriptors of each of its members. The example below shows a declaration (assuming that each element of C or D occupies one word) and its related descriptor.

```
dcl 1 S,
    2 A,
    2 B (5),
    3 C,
    3 D;
```

	basic descriptor of S
	basic descriptor of A
	basic descriptor of B
1	lower bound of B
5	upper bound of B
2	element separation of B
	basic descriptor of C
1	lower bound of C
5	upper bound of C
2	element separation of C
	basic descriptor of D
1	lower bound of D
5	upper bound of D
2	element separation of D

Members of dimensioned structures are arrays, and their descriptor contains copies of the bounds of the containing structure.



SECTION III

SUBSYSTEM PROGRAMMING ENVIRONMENT

WRITING A PROCESS OVERSEER

Almost every feature of the standard Multics system interface can be replaced by providing a specially tailored process overseer procedure in place of the standard version. The standard Multics process overseer procedure, `process_overseer_`, is the initial procedure assigned to a user unless the project administrator specifies otherwise by an `initproc` or `Initproc` statement in the project master file (PMF). (See the Multics Administrators' Manual Project Administrator, Order No. AK51.) If a user has the `v_process_overseer` attribute, he may specify a different initial procedure when he logs in by using the `-process_overseer (-po)` control argument as in the following example:

```
login Smith -po >udd>AEC>special_overseer_
```

If Smith does not have the `v_process_overseer` attribute, the system refuses the login.

Process Initialization

When a process is created for a user when he logs in or in response to either a `new_proc` command (described in the MPM Commands) or process termination signal, the new process initializes itself, sets the default search rules, and then calls one of the following three procedures in the user's initial ring:

```
user_real_init_admin_   for an interactive process
absentee_real_init_admin_ for an absentee process
daemon_real_init_admin_ for a system daemon process
```

These procedures first perform several initialization tasks and then call the user's process overseer procedure, expecting that the process overseer will not return. A return is treated as an error, and a report is made to the system that the process cannot be initialized.

In order to initialize the process, several items of information must be passed to the process by the system control process. The system places this information in a special per-process segment, called the process initialization table (PIT), that resides in the process directory. The user process may read the contents of the PIT, but may not modify it. The `user_info_` subroutine (described in the MPM Subroutines) is used to extract information from the PIT.

Before calling the process overseer, `user_real_init_admin_` attaches the I/O switch named `user_i/o` (through an I/O system module named in the PIT) to the target (also specified in the PIT). It then attaches the I/O switches named `user_output`, `user_input`, and `error_output` as synonyms of `user_i/o`. The I/O module used for an interactive process is `tty_`, the Multics terminal device I/O module. (This module is described in the MPM Subroutines.)

For an absentee process, the Multics absentee I/O module, `abs_io_`, is used. When an absentee process is being created, `absentee_real_init_admin_` obtains the arguments to the absentee process; it then makes them available to the `abs_io_` I/O module and informs this module of the locations of the input and output segments. If a CPU time limit has been specified for the absentee process, `absentee_real_init_admin_` also starts a timer with this limit value; the process is logged out when this value is reached.

The final action taken by the appropriate `init_admin_` procedure is to locate the process overseer procedure named in the PIT and to call it. If the process overseer cannot be located or accessed, the appropriate `init_admin_` procedure signals an error to the system control process, and the user is logged out with the message "Process cannot be initialized".

Process Overseer Functions

If an unclaimed signal reaches the appropriate `init_admin_` procedure, the user process is terminated on the assumption that the process could not be initialized. Therefore, one of the first things that the process overseer procedure does is establish an appropriate handler for all conditions that could be specified. The standard system process overseer does this by executing:

```
call condition_ ("any_other", standard_default_handler_);
```

The `standard_default_handler_` procedure is invoked on all signals not intercepted by any subsequently established condition handler. In general, the `standard_default_handler_` procedure either performs some default action (such as inserting a pagemark into the stream when an endpage condition is signalled) and restarts execution, or else it prints a standard error message and calls the current listener.

A process overseer procedure may perform many other actions besides those executed by the system version. For example, initialization of special per-project accounting procedures may be accomplished at this point or requests issued for an additional password or any other administrative information required by a project.

The system process overseer terminates processing by calling the standard listener in the following manner:

```
call listen_ (initial_command_line);
```

The initial command line used by the system process overseer is:

```
exec_com home_dir>start_up start_type proc_type
```

where:

1. start_type is either login or new_proc, depending on which of these was invoked to create the process.
2. proc_type is either interactive or absentee.

These arguments can be used by the start_up.ec segment as described in connection with the exec_com command in the MPM Commands.

The command line given above assumes that the no_start_up flag is off and that the segment named start_up.ec can be found in the user's home directory. The no_start_up flag is off unless the project administrator has given the user the no_start_up attribute and the user has included the proper control argument (-no_start_up or -ns) in his login line.

If no start_up.ec segment is provided, or if one is provided but the no_start_up flag is on, the standard Multics process overseer checks the brief switch in the PIT. If this switch is off, and if the process was not created in response to a new_proc command or process termination signal, the process overseer prints the contents of the message_of_the_day segment located in the directory named >system_control_1.

The standard process overseer does not expect the listener to return. If it does, the appropriate init_admin procedure is recalled and the process is logged out with the message Process cannot be initialized.

Handling of Quit Signals

A quit signal is indicated by pressing the appropriate key, such as ATTN or BRK, on the terminal in use. When a terminal is first attached for interactive processing, quit signals from the terminal are disabled. A user quit signal issued at this time causes the flushing of terminal output buffers, but the quit condition is not raised in the user ring. The recognition of quit signals is enabled when the following call is made:

```
call iox_$control (iox_$user_io, "quit_enable", null(), status);
```

If a project administrator wishes to replace the standard user environment with his own programs, he must find an appropriate place for the quit_enable order, after the mechanism for handling quit signals has been established.



SECTION IV

IMPLEMENTATION TO INPUT/OUTPUT MODULES

This section contains information applicable to writing I/O modules. It describes the format and function of I/O control blocks, provides a list of implementation rules, and describes the use of certain `iox_` subroutine entry points necessary in I/O module construction. These entry points are described in more detail in Section VII. For descriptions of the other `iox_` entry points, refer to the MPM Subroutines.

Some instances in which a user might wish to create a new I/O module are given below.

1. Pseudo Device or File. An I/O module could be used to simulate I/O to/from a device or file. For example, it might provide a sequence of random numbers in response to an input request. The `discard_system` I/O module (described in the MPM Subroutines) is an example of this sort of module.
2. New File Type. An I/O module could be used to support a new type of file in the storage system, such as a file in which records have multiple keys.
3. Reinterpreting a File. An I/O module could be designed to overlay a new structure (relative to the standard file types) on a standard type of file. For example, an unstructured file might be interpreted as a sequential file by considering 80 characters as a record.
4. Monitoring a Switch. An I/O module could be designed to pass operations along to another module while monitoring them in some way (e.g., by copying input data to a file).
5. Unusual Devices. Working through the `tty_` I/O module (described in the MPM Subroutines) in the raw mode, another I/O module might transmit data to/from a device that is not a standard Multics device type (as regards character codes, etc.).

The last three items listed illustrate a common arrangement. The user attaches an I/O switch, `x`, using an I/O module, `A`. To implement the attachment, module `A` attaches another switch, `y`, using another I/O module, `B`. When the user calls module `A` through the switch `x`, module `A` in turn calls module `B` through the switch `y`. Any nonsystem I/O module that performs true I/O works in this way, because it (or some module that it calls) must call a system I/O module. There are system I/O routines at a more primitive level than the I/O modules, but user-written I/O modules must not call these routines.

I/O CONTROL BLOCKS

Each I/O switch has an associated I/O control block that is created the first time a call to `iox_$find_iocb` requests a pointer to the control block. The control block remains in existence for the life of the process unless explicitly destroyed by a call to `iox_$destroy_iocb`.

The principal components of an I/O control block are pointer variables and entry variables whose values describe the attachment and opening of the I/O switch. There is one entry variable for each I/O operation with the exception of the attach operation. To perform an I/O operation through the switch, the corresponding entry value in the control block is called. For example, if `iocb_ptr` is a pointer to an I/O control block, the call:

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

results in the call:

```
call iocb_ptr->iocb.put_chars (iocb_ptr, buff_ptr, buff_len, code);
```

Certain system routines are allowed to make the latter call directly, without going through the `iox_` subroutine; all other routines must call the `iox_` subroutine.

I/O Control Block Structure

The declaration given below describes the first part of an I/O control block. Only those few I/O system programs that use the remainder of the I/O control block declare the entire block. Thus, all references to I/O control blocks here refer only to the first part of the control block. For example, the statement "no other changes are made to the control block" means that no other changes are made to the first part of the control block, and so on. The I/O system might make changes to the remainder of the block, but these are of interest only to the I/O system. For full details on the entry variables, see the descriptions of the corresponding entries in the `iox_` subroutine in the MPM Subroutines.

```
dcl 1 iocb                aligned,
  2 iocb_version          fixed bin init(1),
  2 name                  char(32),
  2 actual_iocb_ptr      ptr,
  2 attach_descrip_ptr  ptr,
  2 attach_data_ptr     ptr,
  2 open_descrip_ptr    ptr,
  2 open_data_ptr       ptr,
  2 reserved              bit(72),
  2 detach_iocb          entry (ptr, fixed bin(35)),
  2 open                  entry (ptr, fixed bin, bit(1) aligned,
                             fixed bin(35)),
  2 close                entry (ptr, fixed bin(35)),
  2 get_line              entry (ptr, ptr, fixed bin(21), fixed bin(21),
                             fixed bin(35)),
  2 get_chars            entry (ptr, ptr, fixed bin(21), fixed bin(35)),
  2 put_chars            entry (ptr, ptr, fixed bin(21), fixed bin(35)),
  2 modes                 entry (ptr, char(*), char(*), fixed bin(35)),
  2 position              entry (ptr, fixed bin, fixed bin(21),
                             fixed bin(35)),
```

```

2 control          entry (ptr, char(*), ptr, fixed bin(35)),
2 read_record     entry (ptr, ptr, fixed bin(21), fixed bin(21),
                  fixed bin(35)),
2 write_record    entry (ptr, ptr, fixed bin(21), fixed bin(35)),
2 rewrite_record  entry (ptr, ptr, fixed bin(21), fixed bin(35)),
2 delete_record   entry (ptr, fixed bin(35)),
2 seek_key        entry (ptr, char(256) varying, fixed bin(21),
                  fixed bin(35)),
2 read_key         entry (ptr, char(256) varying, fixed bin(21),
                  fixed bin(35)),
2 read_length     entry (ptr, fixed bin(21), fixed bin(35));

```

Attach Pointers

If the I/O switch is detached, the value of `iocb.attach_descrip_ptr` is null. If the I/O switch is attached, the value is a pointer to the following structure:

```

dcl 1 attach_descrip based aligned,
   2 length                fixed bin(17),
   2 string                 char (0 refer (length));

```

The value of `attach_descrip.string` is the attach description. See "Multics Input/Output System" in Section IV of the MPM Reference Guide for details on the attach description.

If the I/O switch is detached, the value of `iocb.attach_data_ptr` is null. If the I/O switch is attached, the value may be null, or it may be a pointer to data used by the I/O module that attached the switch. To determine whether the I/O switch is attached or not, the value of `iocb.attach_descrip_ptr` should be examined, if it is null, the switch is attached.

Open Pointers

If the I/O switch is closed (whether attached or detached), the value of `iocb.open_descrip_ptr` is null. If the switch is open, the value is a pointer to the following structure:

```

dcl 1 open_descrip based aligned,
   2 length                fixed bin(17),
   2 string                 char (0 refer (length));

```

The value of `open_descrip.string` is the open description. It has the following form:

```
mode -info-
```

where:

1. `mode` is one of the opening modes (e.g., `stream_input`) listed below. The modes and their corresponding numbers are:

```
1  stream_input
2  stream_output
3  stream_input_output
4  sequential_input
5  sequential_output
6  sequential_input_output
7  sequential_update
8  keyed_sequential_input
9  keyed_sequential_output
10 keyed_sequential_update
11 direct_input
12 direct_output
13 direct_update
```

2. `info` is other information about the opening. If `info` occurs in the string, it is preceded by one blank character.

If the I/O switch is closed, the value of `iocb.open_data_ptr` is null. If the I/O switch is open, the value may be null, or it may be a pointer to data used by the I/O module that opened the switch.

Entry Variables

The value of each entry variable in an I/O control block is an entry point in an external procedure. When the I/O switch is in a state that supports a particular operation, the value of the corresponding entry variable is an entry point that performs the operation. When the I/O switch is in a state that does not support the operation, the value of the entry variable is an entry point that returns an appropriate error code.

Synonyms

When an I/O switch named `x` is attached as a synonym for an I/O switch named `y`, the values of all entry variables in the I/O control block for `x` are identical to those in the I/O control block for `y` with the exception of `iocb.detach`. Thus a call:

```
call iocbx_ptr->iocb.op(iocbx_ptr,...);
```

immediately goes to the correct routine.

The values of `iocb.open_descrip_ptr` and `iocb.open_data_ptr` for `x` are also the same as those for `y`. Thus, the I/O routine has access to its open data (if any) through the I/O control block pointed to by `iocbx_ptr`.

The value of `iocb.actual_iocb_ptr` for `x` is a pointer to the control block for the switch that is the ultimate target of a chain of synonyms. (When the switch `x` is not attached as synonym, this pointer points to the control block for `x` itself.) I/O modules use this pointer to access the ultimate I/O control block whose contents are to be changed, for example, when a switch is opened. The I/O system then propagates the changes to other control blocks as required by synonym attachments.

WRITING AN I/O MODULE

The information presented in the following paragraphs pertains to the design and programming of an I/O module. In particular, conventions are given that must be followed if the I/O module is to interface properly with the I/O system. The reader should be familiar with the material presented under the headings "Multics Input/Output System" and "File Input/Output" in Section IV of the MPM Reference Guide, the `iox_` subroutine in the MPM Subroutines, and under "I/O Control Blocks" above.

Design Considerations

Before programming begins on an I/O module, the functions it is to perform should be clearly specified. In particular, the designer should list the opening modes to be supported and consider the meaning of each I/O operation supported for those modes. (See "Open Pointers" above for a list of opening modes.) The specifications in the description of the `iox_` subroutine must be related to the particular I/O module (e.g., what `seek_key` means for the `discard_` I/O module).

An I/O module contains routines to perform attach, open, close, and detach operations and the operations supported by the opening modes. Typically, though not necessarily, all routines are in one object segment. If the module is a bound segment, only the attach entry need be retained as an external entry. Other routines are accessed through entry variables in I/O control blocks.

An I/O module may have several routines that perform the same function but in different situations (e.g., one `get_line` routine for `stream_input` openings, another for `stream_input_output` openings). Whenever the situation changes (e.g., at opening), the module stores the appropriate entry values in the I/O control block.

Implementation Rules

The following rules apply to the implementation of all I/O operations. Additional rules that are specific to a particular operation are given later. In the rules, `iocb` is a based variable declared as described under "I/O Control Blocks" above, and `iocb_ptr` is an argument of the operation in question.

1. Except for `attach`, the usage (entry declaration and parameters) of a routine that implements an I/O operation is the same as the usage of the corresponding entry in the `iox_` subroutine. See the MPM Subroutines for details on the `iox_` subroutine.
2. Except for `attach` and `detach`, the actual I/O control block to which an operation applies (i.e., the control block attached by the called I/O module) must be referenced using the value of `iocb_ptr->iocb.actual_iocb_ptr`. It is incorrect to use just `iocb_ptr`, and it is incorrect to remember the location of the control block from a previous call (e.g., by storing it in a data structure pointed to by `iocb.open_data_ptr`).
3. On entry to an I/O module, the value of `iocb_ptr->iocb.open_data_ptr` always equals the value of:

```
iocb_ptr->iocb.actual_iocb_ptr->iocb.open_data_ptr
```

The value of `ptr->iocb.open_descrip_ptr` always equals the value of:

```
iocb_ptr->iocb.actual_iocb_ptr->iocb.open_descrip_ptr
```

Thus, the data structures related to an opening may be accessed without going through `iocb.actual_iocb_ptr`.

4. If an I/O operation changes any values in an I/O control block, it must be the actual I/O control block (Rule 1 above); and, before returning, the operation must execute the call:

```
call iox_$propagate (p);
```

where `p` points to the changed control block. The routine `iox_$propagate` reflects changes to other control blocks attached as synonyms. It also makes certain adjustments to the entry variables in the control block when the I/O switch is attached, opened, closed, or detached.

5. All I/O operations must be external procedures.

Attach Operation

The name of the routine that performs the attach operation is derived by concatenating the word "attach" to the name of the I/O module (e.g., discard_attach is the name of the attach routine for the discard_ I/O module). Each attach routine has the following usage:

```
declare module_nameattach entry (ptr, (*)char(*) varying, bit(1) aligned,  
    fixed bin(35));
```

```
call module_nameattach (iocb_ptr, option_array, com_err_switch, code);
```

where:

1. iocb_ptr points to the control block of the I/O switch to be attached. (Input)
2. option_array contains the options in the attach description. If there are no options, its bounds are (0:0). Otherwise, its bounds are (1:n) where n is the number of options. (Input)
3. com_err_switch indicates whether the attach routine should call the com_err_ subroutine (described in the MPM Subroutines) when an error is detected. (Input)
"1"b yes
"0"b no
4. code is a standard system status code. (Output)

The following rules apply to coding an attach routine:

1. If the I/O switch is already attached (i.e., if iocb_ptr->iocb.attach_descrip_ptr is not null), return the code error_table_\$not_detached; do not make the attachment.
2. If, for any reason, the switch cannot be attached, return an appropriate nonzero code and do not modify the control block. Call the com_err_ subroutine if, and only if, com_err_switch is "1"b. If the attachment can be made, follow the remaining rules and return with code set to 0.
3. Set iocb_ptr->iocb.open and iocb_ptr->iocb.detach_iocb to the appropriate open and detach routines. In addition, set iocb_ptr->attach_descrip_ptr to point to a structure as described in "I/O Control Blocks" above. The attach description in this structure must be fabricated from the options in the argument option array, and there may be some modification of options, e.g., expanding a pathname.
4. If desired, set iocb_ptr->iocb.attach_data_ptr, iocb_ptr->iocb.modes, and iocb_ptr->iocb.control. Make no other modifications to the control block.

Open Operation

An open operation is performed only when the actual I/O switch is attached (through the I/O module containing the routine) but not open. The following rules apply to coding an open routine:

1. If, for any reason, the opening cannot be performed, return an appropriate code and do not modify the I/O control block. If the opening can be performed, follow the remaining rules and return with code set to 0.
2. Set `iocb_ptr->iocb.actual_iocb_ptr->iocb.op` (where `op` is any operation listed under "Open Pointers" above) to an appropriate routine. This applies for each operation allowed for the specified opening mode.
3. If either the modes operation or the control operation is enabled with the I/O switch attached but not open, set `iocb_ptr->iocb.actual_iocb_ptr->iocb.op` (where `op` is modes or control) to `iox_$err_no_operation`.
4. Set `open_descrip_ptr` to point to a structure as described in "I/O Control Blocks" above.
5. If desired, set `iocb_ptr->iocb.actual_iocb_ptr->iocb.open_data_ptr`. Do not make any other modifications to the control block.

Close Operation

A close operation is performed only when the actual I/O switch is open, the opening having been made by the I/O module containing the close routine. The following rules apply to coding a close routine:

1. Set the following to the appropriate open and detach routines:

```
iocb_ptr->iocb.actual_iocb_ptr->iocb.open  
iocb_ptr->iocb.actual_iocb_ptr->iocb.detach_iocb
```


Set `iocb_ptr->iocb.actual_iocb_ptr->iocb.open_descrip_ptr` to null.
2. If either the modes operation or the control operation is enabled with the switch open, set `iocb_ptr->iocb.actual_iocb_ptr->iocb.op`, where `op` is modes or control. Unless the operation is enabled with the switch closed, set the entry variable to `iox_$err_no_operation`.
3. Do not make any other modifications to the control block.
4. The close routine should set the bit counts on modified segments of a file, free any storage allocated for buffers, etc., and in general, clean things up.
5. The close routine must not return without closing the switch.

Detach Operation

A detach operation is performed only when the actual I/O switch is attached but not open, the attachment having been made by the I/O module containing the detach routine. The following rules apply to coding detach routines:

1. Set `iocb_ptr->iocb.attach_descrip_ptr` to null.
2. Do not make any other modifications to the control block.
3. The detach routine must not return without detaching the switch.

Modes and Control Operations

These operations can be accepted with the I/O switch attached but closed; however, it is generally better practice to accept them only when the switch is open.

If the control operation is supported, it must return the code `error_table_$no_operation` when given an invalid order. In this situation, the state of the I/O switch must not be changed.

If the modes operation is supported, it must return the code `error_table_$bad_mode` when given an invalid mode.

Other Operations

Routines for the other operations are called only when the actual I/O switch is attached and open in a mode for which the operation is allowed, the opening and attachment having been made by the I/O module containing the routine. In coding these routines, make only the following modifications to the I/O control block of the actual I/O switch.

1. Reset `iocb_ptr->iocb.actual_iocb_ptr->iocb.open_data_ptr`.
2. Reset an entry variable set by the open routine, e.g., to switch from one `put_chars` routine to another.
3. Close the switch in an error situation. In this case, the rules above for the close operation must be followed.



SECTION V

REFERENCE TO COMMANDS AND SUBROUTINES BY FUNCTION

COMMAND REPERTOIRE

The Multics commands described in this manual are organized by function into the following categories:

Debugging and Performance Monitoring Facilities
Language Translators, Compilers, Assemblers, and Interpreters
Object Segment Manipulation
Storage System, Access Control
Storage System, Directory Manipulation
Storage System, Mailbox Manipulation
Storage System, Segment Manipulation

Detailed descriptions of these commands, arranged alphabetically rather than functionally, are given in Section VI of this document. In addition, many of the commands have online descriptions, which the user may obtain by invoking the help command (described in the MPM Commands).

See "Reference to Commands By Function" in Section I of the MPM Commands for the functional grouping of the commands described in that manual.

Debugging and Performance Monitoring Facilities

area_status	displays information about an area
create_area	creates an area and initializes it
delete_external_variables	deletes specified variables managed by the system
display_component_name	converts bound segment offset into referenced component object segment offset
list_external_variables	prints information about variables managed by the system
list_temp_segments	lists segments in temporary segment pool
print_linkage_usage	prints block storage usage for combined linkage regions
reset_external_variables	reinitializes system managed variables
set_system_storage	establishes an area as the storage region for normal system allocations
set_user_storage	establishes an area as the storage region for normal user allocations

Language Translators, Compilers, Assemblers, and Interpreters

alm	invokes ALM assembler
alm_abs	invokes ALM assembler in absentee job
error_table_compiler	compiles table of status codes and messages from ASCII source segments

Object Segment Manipulation

print_bind_map	prints bind map of object segment
print_link_info	prints information about object segments

Storage System, Access Control

set_ring_brackets	changes ring brackets of segment
-------------------	----------------------------------

Storage System, Directory Manipulation

copy_names	copies names from one segment to another
move_names	moves names from one segment to another
set_max_length	specifies maximum length of nondirectory segment

Storage System, Mailbox Manipulation

mbx_add_name	adds alternate names to mailbox
mbx_create	creates mailbox
mbx_delete	deletes mailbox
mbx_delete_acl	deletes entries from mailbox ACL
mbx_delete_name	deletes name from mailbox
mbx_list_acl	lists ACL of mailbox
mbx_rename	replaces one name with another on mailbox
mbx_set_acl	adds and changes entries on mailbox ACL
mbx_set_max_length	sets maximum length of a mailbox segment

Storage System, Segment Manipulation

archive_sort	sorts components of archive segment
reorder_archive	orders components of archive segment

SUBROUTINE REPERTOIRE

The Multics subroutines described in this manual are organized by function into the following categories:

- Clock and Timer Procedures
- Command Environment Utility Procedures
- Condition Mechanism
- Data Type Conversion Procedures

Error Handling Procedures
 Input/Output System Procedures
 Miscellaneous Procedures
 Object Segment Manipulation
 Process Synchronization
 Storage System, Access Control and Rings of Protection
 Storage System, Address Space
 Storage System, Directory and Segment Manipulation
 Storage System, Utility Procedures

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

Detailed descriptions of these subroutines, arranged alphabetically rather than functionally, are given in Section VII of this document.

Many of the functions provided by these subroutines are also available as part of the runtime facilities of Multics-supported programming languages; users are encouraged to use the language-related facilities wherever possible.

See "Introduction to Standard Subroutines" in Section I of the MPM Subroutines for the functional grouping of the subroutines described in that manual.

Clock and Timer Procedures

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

Command Environment Utility Procedures

check_star_name_	verifies formation of entrynames according to star name rules
cu_	command utility programs provide functions needed by command and subsystem writers
get_default_wdir_	returns pathname of user's current default working directory
get_definition_	returns pointer to specified definition within an object segment
get_entry_name_	returns associated name of externally defined location or entry point in segment
get_equal_name_	constructs target name by substituting from entryname into equal name
get_system_free_area_	returns pointer to system free area for calling ring

Condition Mechanism

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

find_condition_info_	returns information about condition when signal occurs
prepare_mc_restart_	checks machine conditions for restartability, and permits modifications to them for user changes to process execution, before condition handler returns
signal_	signals occurrence of given condition
unwinder_	performs nonlocal goto on Multics stack

Data Type Conversion Procedures

ascii_to_ebcdic_	performs conversion from ASCII to EBCDIC
assign_	assigns specified source value to specified target performing required conversion
cv_bin_	converts binary representation of integer to 12-character ASCII string
cv_entry_	converts a virtual entry to an entry value
cv_hex_	returns the fixed binary representation of an ASCII hexadecimal integer
cv_oct_	converts ASCII representation of octal integer to fixed binary representation of that number
cv_ptr_	converts a virtual pointer to a pointer value
ebcdic_to_ascii_	performs conversion from EBCDIC to ASCII

Error Handling Procedures

active_fnc_err_	prints formatted error message and signals active_function_error condition
convert_status_code_	returns short and long status messages for given status code
sub_err_	reports errors detected by other subroutines

Input/Output System Procedures

convert_dial_message_	controls dialed terminals
dial_manager_	interfaces the answering service dial facility
dprint_	adds segment print or punch request to specified queue
iod_info_	extracts information from I/O daemon tables for commands and subroutines submitting I/O daemon requests
iox_	provides interfaces for controlling the data structures of the I/O system
vfile_	supports I/O from/to segments and multisegment files in the storage system

Miscellaneous Procedures

decode_descriptor_	extracts information from argument descriptors
get_privileges_	returns process' access privileges
sys_info	is a wired-down, per-system data base
system_info_	provides user with information on system parameters

Object Segment Manipulation

object_info_	prints structural and identifying information extracted from object segment
stu_	retrieves information from object segment's (PL/I or FORTRAN) runtime symbol table section
tssi_	simplifies use of storage system by language translators

Process Synchronization

hcs_\$wakeup	sends interprocess communication wakeup to blocked process over specified event channel
ipc_	user interface to Multics interprocess communication facility

Storage System, Access Control and Rings of Protection

aim_check_	determines relationship between two access attributes
convert_aim_attributes_	converts representation of process'/segment's access authorization/class into character string of defined form
get_privileges_	returns process' access privileges
get_ring_	returns number of current protection ring
hcs_\$add_dir_inacl_entries	} adds specified access modes to initial ACL for segments or directories
hcs_\$add_inacl_entries	
hcs_\$delete_dir_inacl_entries	} deletes specified entries from initial ACL for segments or directories
hcs_\$delete_inacl_entries	
hcs_\$get_dir_ring_brackets	} returns ring brackets for specified segment or subdirectory
hcs_\$get_ring_brackets	
hcs_\$list_dir_inacl	} returns all or part of initial ACL for segments or directories
hcs_\$list_inacl	
hcs_\$replace_dir_inacl	} replaces initial ACL with user-provided one for segments or directories
hcs_\$replace_inacl	
hcs_\$set_dir_ring_brackets	} sets ring brackets for specified segment or directory
hcs_\$set_ring_brackets	
read_allowed_	} determines if AIM allows specified operations on object given process' authorization and object's access class
read_write_allowed_	
write_allowed_	

Storage System, Address Space

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

Storage System, Directory and Segment Manipulation

hcs_\$del_dir_tree	deletes subdirectory's contents
hcs_\$get_author	returns author of segment, directory, or link
hcs_\$get_bc_author	returns bit-count author of a segment or directory
hcs_\$get_max_length	returns maximum length of segment,
hcs_\$get_max_length_seg	
hcs_\$get_safety_sw	returns safety switch value of directory or segment
hcs_\$get_safety_sw_seg	
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_entry_bound	sets entry point bound of segment
hcs_\$set_entry_bound_seg	
hcs_\$set_max_length	sets maximum length of segment
hcs_\$set_max_length_seg	
hcs_\$set_safety_sw	sets safety switch of segment
hcs_\$set_safety_sw_seg	
hcs_\$star_	returns storage system type and all names that match entryname according to star name rules

Storage System, Utility Procedures

area_info_	returns information about an area
define_area_	initializes a region of storage as an area
get_default_wdir_	returns pathname of user's current default working directory
get_definition_	returns pointer to specified definition within an object segment
get_entry_name_	returns associated name of externally defined location or entry point in segment
get_equal_name_	constructs target name by substituting from entryname into equal name
match_star_name_	compares entryname with star name
msf_manager_	provides the means for multisegment files to create, access, and delete components, truncate the file and control access
release_area_	cleans up an area
tssi_	simplifies use of storage system by language translators

SECTION VI

COMMANDS

COMMAND DESCRIPTION FORMAT

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

Name

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

Usage

This part of the command description first shows a single line that demonstrates the proper format to use when invoking the command and then explains each element in the line. The following conventions apply in the usage line.

1. Optional arguments are enclosed in braces (e.g., {path}, {User_ids}). All other arguments are required.
2. Control arguments are identified in the usage line with a leading hyphen (e.g., {-control_args}) simply as a reminder that all control arguments must be preceded by a hyphen in the actual invocation of the command.
3. To indicate that a command accepts more than one of a specific argument, an "s" is added to the argument name (e.g., paths, {paths}, {-control_args}).

NOTE: Keep in mind the difference between a plural argument name that is enclosed in braces (i.e., optional) and one that is not (i.e., required). If the plural argument is enclosed in braces, clearly no argument of that type need be given. However, if there are no braces, at least one argument of that type must be given. Thus "paths" in a usage line could also be written as:

```
path1 {path2 ... pathn}
```

The convention of using "paths" rather than the above is merely a method of saving space.

4. Different arguments that must be given in pairs are numbered (e.g., xxx₁ yyy₁ {... xxx_n yyy_n}).
5. To indicate that the same generic argument must be given in pairs, the arguments are given letters and numbers (e.g., pathA₁ pathB₁ {... pathA_n pathB_n}).
6. To indicate one of a group of the same arguments, an "i" is added to the argument name (e.g., path_i, User_{idi}).

To illustrate these conventions, consider the following usage line:

```
command {paths} {-control_args}
```

The lines below are just a few examples of valid invocations of this command:

```
command
command path path
command path -control_arg
command -control_arg -control_arg
command path path path -control_arg -control_arg -control_arg
```

In many cases, the control arguments take values. For simplicity, common values are indicated as follows:

STR	any character string; individual command descriptions indicate any restrictions (e.g., must be chosen from specified list; must not exceed 136 characters).
N	number; individual command descriptions indicate whether it is octal or decimal and any other restrictions (e.g., cannot be greater than 4).
DT	date-time character string in a form acceptable to the convert_date_to_binary_ subroutine described in the MPM Subroutines.
path	pathname of an entry; unless otherwise indicated, it may be either a relative or an absolute pathname.

The lines below are samples of control arguments that take values:

```
-access_name STR, -an STR
-ring N, -rg N
-date DT, -dt DT
-home_dir path, -hd path
```

Notes

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

Examples

The examples show different valid invocations of the command. An exclamation mark (!) is printed at the beginning of each user-typed line. This is done only to distinguish user-typed lines from system-typed lines. The results of each example command line are either shown or explained.

Other Headings

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

Name: alm

ALM is the standard Multics assembly language. It is commonly used for privileged supervisor code, higher level support operators and utility packages, and data bases. It is occasionally used for efficiency or for hardware features not accessible in higher level languages; however, its routine use is discouraged.

The alm command invokes the ALM assembler to translate a segment containing the text of an assembly language program into a Multics standard object segment. A listing segment can also be produced. These segments are placed in the user's current working directory.

The ALM language is described briefly in this command description. The Multics Processor Manual, Order No. AL39, fully describes the instruction set.

Usage

```
alm path {-control_args}
```

where:

1. path
is the pathname of an ALM source segment that is to be translated by the ALM assembler. If path does not have a suffix of alm, one is assumed. However, the suffix must be the last component of the name of the source segment.
2. control_args
are optional arguments that can only appear after the path argument. The control arguments are:
 - list, -ls
produces an assembly listing segment.
 - no_symbols
suppresses the listing of a cross-reference table in the listing segment. This cross-reference table is included by default in the listing segment when the -list control argument is given.
 - brief, -bf
prevents errors from being printed on the terminal. Any errors are flagged in the listing (if one has been requested).

Notes

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

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

If the user specifies the -list control argument, the alm command creates a listing segment in the working directory and gives it a name consisting of the entryname portion of the source segment with the suffix list rather than alm (e.g., a source segment named prt_conv_.alm would have a listing segment named prt_conv_.list). The ACL is as described for the object segment except that the user is given rw access to the newly created segment. Previous copies of the object segment and the listing segment are replaced by the new segments created by the compilation.

The assembler is serially reusable and sharable, but cannot be reentered once translation has begun; that is, it cannot be interrupted during execution, invoked again, then restarted in its previous invocation.

Error Conditions

Errors arising in the command interface, such as inability to locate the source segment, are reported in the normal Multics manner. Some conditions can arise within the assembler that are considered malfunctions in the assembler; these are reported by a line printed on the terminal and also in the listing. Any of the above cases is immediately fatal to the translation.

Errors detected in the source program, such as undefined symbols, are reported by placing one-letter error flags at the left margin of the erroneous line in the listing segment. Any line so flagged is also printed on the user's terminal, unless the -brief control argument is in effect. Flag letters and their meanings are given below.

- B mnemonic used belongs to obsolete (Honeywell Model 645) processor instruction set
- E malformed expression in arithmetic field
- F error in formation of pseudo-operation operand field
- M reference to a multiply defined symbol
- N unimplemented or obsolete pseudo-operation
- O unrecognized opcode
- P phase error; location counter at this statement has changed between passes, possibly due to misuse of org pseudo-operation
- R expression produces an invalid relocation type
- S error in the definition of a symbol
- T undefined modifier (tag field)

- U reference to an undefined symbol
- 7 digit 8 or 9 appears in an octal field

The errors B, E, M, O, P, and U are considered fatal. If any of them occurs, the standard Multics "Translation failed" error message is reported after completion of the translation.

ALM Language

An ALM source program is a sequence of statements separated by newline characters or semicolons. The last statement must be the end pseudo-operation.

Fields must be separated by white space, which is defined to include space, tab, new page, and percent characters.

A name is a sequence of uppercase and lowercase letters, digits, underscores, and periods. A name must begin with a letter, period, or underscore and cannot be longer than 31 characters.

Labels

Each statement can begin with any number of names, each followed immediately by a colon. Any such names are defined as labels, with the current value of the location counter. A label on a pseudo-operation that changes location counters or forces even alignment (such as org or its) might not refer to the expected location. White space is optional. It can appear before, after, or between labels, but not before the colon.

Opcode

The first field after any labels is the opcode. It can be any instruction mnemonic or any one of the pseudo-operations listed later in this description under "Pseudo-operations." The opcode can be omitted, and any labels are still defined. White space can appear before the opcode, but is not required.

Operand

Following the opcode, and separated from it by mandatory white space, is the operand field. For instructions, the operand defines the address, pointer register, and tag (modifier) of the instruction. For each pseudo-operation, the operand field is described under "Pseudo-operations." The operand field can be omitted in an instruction. Those pseudo-operations that use their operands generally do not permit the operand field to be omitted.

Comments

Since the assembler ignores any text following the end of the operand field, this space is commonly used for comments. In those pseudo-operations that do not use the operand field, all text following the opcode is ignored and can be used for comments. Also, a quote character (") in any field introduces a comment that extends to the end of the statement. (The only exceptions are the acc, aci, and bci pseudo-operations, for which the quote character can be used to delimit literal character strings.) The semicolon ends a statement and therefore ends a comment as well.

Instruction Operands

The operand field of an instruction can be of several distinct formats. Most common is the direct specification of pointer register, address, and tag (modifier). This consists of three subfields, any of which can be omitted. The first subfield specifies a pointer register by number, user-defined name, or predefined name (pr0, pr1, pr2, pr3, pr4, pr5, pr6, pr7). The subfield ends with a vertical bar. If the pointer register and vertical bar are omitted, no pointer register is used in the instruction. The second subfield is any arithmetic expression, relocatable or absolute. This is the address part of the instruction, and its default is zero. Arithmetic expressions are defined below under "Arithmetic Expressions." The last subfield is the modifier or tag. It is separated from the preceding subfields by a comma. If the tag subfield and comma are omitted, no instruction modification is used. (This is an all zero modifier.) Valid modifiers are defined below under "Modifiers."

Other formats of instruction operands are used to imply pointer registers. If a symbolic name defined by temp, tempd, or temp8 is used in the address subfield (it can be used in an arithmetic expression), then pointer register 6 is used if no pointer register is specified explicitly. This form can have a tag subfield.

Similarly, if an external expression is used in the address subfield, then pointer register 4 is implied; this causes a reference through a link. The pointer register subfield may not be specified explicitly. If a modifier subfield is specified, it is taken as part of the external expression; the instruction has an implicit n* modifier to go through the link pair. External expressions are defined below under "External Expressions."

A literal operand begins with an equal sign followed by a literal expression. The literal expression can be enclosed in parentheses. It has no pointer register but can have a tag subfield. A literal reference normally causes the instruction to refer to a word in a literal pool that contains the value of the literal expression. However, if the modifier du or dl is used, the value of the literal is placed directly in the instruction address field. Literal expressions are defined below under "Literal Expressions."

Special Instruction Formats

Certain instructions assembled by the ALM assembler do not follow the standard opcode-operand format as described above. These instructions fall into three basic classes: the repeat instructions, special treatment of the index and pointer register instructions, and EIS instructions. Each of these special cases is described below.

REPEAT INSTRUCTIONS

The repeat instructions are used to repeat either one or a pair of instructions until specified termination conditions are met. There are two basic forms:

```
rpt tally,delta,term1,term2,...,termn
```

generates the machine rpt instruction as described in the Multics Processor Manual. Both tally and delta are absolute arithmetic expressions. The term_i specify the termination conditions as the names of corresponding conditional transfer instructions. This same format can be used with the rpt, rpd, rpda, and rpdb pseudo-operations.

```
rptx ,delta
```

generates the machine rpt instruction with a bit set to indicate that the tally and termination conditions are to be taken from index register 0. This format can be used with rplx and rpx.

INDEX REGISTER INSTRUCTIONS

The opcodes for manipulation of the index registers have the general form opx_n, where n specifies the index register to be used in the operation. ALM allows the more general form:

```
opx index,operand
```

which assembles opx_n, where index is an absolute arithmetic expression whose value is n. This format can be used for all index register instructions.

POINTER REGISTER INSTRUCTIONS

As with the index register instructions, the opcodes for the manipulation of the pointer registers have the general form `oprn`, where `n` specifies the pointer register to be used. ALM extends this form to allow:

`opr pointer,operand`

which assembles as `oprn`, where `n` is found as follows: If `pointer` is a built-in pointer name (`pr0`, `pr1`, etc.), that register is selected; otherwise, `pointer` must be an absolute arithmetic expression whose value is `n`. This format can be used with all pointer register instructions except `spri`.

EIS MULTIWORD INSTRUCTIONS

An EIS multiword instruction consists of an operation code word, followed by one or more descriptor words. The descriptor words can be assembled by using the "desc" pseudo-operations listed under "Pseudo-Operations" below. The operation code word has the following general form:

`eisop (MF1),(MF2),keyword1(octexpression),keyword2`

where:

1. `MF1, MF2` are EIS modification fields as described in "EIS Modifiers" below.
2. `keyword1` can be either `fill`, `bool`, or `mask`.
3. `octexpression` is a logical expression that specifies the bits to be placed in the appropriate parts of the instruction.
4. `keyword2` can be `round`, `enablefault`, or `ascii`; these cause single option bits in the instruction to be set.

Keywords can appear in any order, before or after an MF field. This format can be used for all Multics EIS multiword instructions.

EIS SINGLEWORD INSTRUCTIONS

The Multics processor contains a set of 10 instructions that may be used to alter the contents of an address register. These instructions have the following general form:

opcode pr|offset,modifier

where:

1. pr selects the address register that is to be modified by the instruction.
2. offset is a value whose interpretation is dependent upon the opcode used.
3. modifier must be one of the register modifiers (au, ql, x0, etc.).

These instructions have two modes of operation depending on the setting of bit 29 in the instruction. If bit 29 is 1, the current contents of the selected address register are used in determining its new contents; if bit 29 is 0, the contents of the word and bit offset portions of the selected address register are assumed to be zero at the start of the instruction (this results in a load operation into the selected address register). ALM normally sets bit 29 to 1, unless the opcode ends in "x" (e.g., awdx is an awd instruction with bit 29 set to 0). This format can be used with a4bd, a6bd, a9bd, abd, awd, s4bd, s6bd, s9bd, sbd, and swd.

Examples of Instruction Statements

Six examples of instruction statements are shown below. A brief description of each example follows the sample statements.

xlab:	lda	pr0 2,*	" Example 1.
	eax7	xlab-1	
	rccl	<sys_info> [clock_*]	" Example 2.
	segref	sys_info,time_delta	" Example 3.
	adl	time_delta+1	
	temp	nexti	" Example 4.
	lx10	nexti,*	
	link	goto,<unwinder_> [unwinder_]	" Example 5.
	tra	pr4 goto,*	
	ana	=o777777,du	" Example 6.
	ada	=v36/list_end-1	

Example 1 shows direct specification of address, pointer register, and tag fields. In the second instruction, no pointer register is specified, and the symbol xlab is not external, so no pointer register is used.

Example 2 shows an explicit link reference. Indirection is specified for the link as the item at clock_ (in sys_info) is merely a pointer to the final operand.

Example 3 uses an external expression as the operand of the adl instruction. In this particular case, the operand itself is in sys_info.

Example 4 uses a stack temporary. Since the word is directly addressable using pr6, the modifier specified is used in the instruction.

Example 5 shows a directly specified operand that refers to an external entity. It is necessary in this case to specify the pointer register and modifier fields, unlike segref.

Example 6 uses two literal operands. Only the second instruction causes the literal value to be stored in the literal pool.

Arithmetic Expressions

An arithmetic expression consists of names (other than external names) and decimal numbers joined by the ordinary operators + - * /. Parentheses can be used with their normal meaning.

An asterisk in an expression, when not used as an operator, has the value of the current location counter.

All intermediate and final results of the expression must be absolute or relocatable with respect to a single location counter. A relocatable expression cannot be multiplied or divided.

Logical Expressions

A logical expression is composed of octal constants and absolute symbols combined with the Boolean operators + (OR), - (XOR), * (AND), and ^ (NOT). Parentheses can be used with their normal meaning.

External Expressions

An external expression refers symbolically to some other segment. It consists of an external name or explicit link reference, an optional arithmetic expression added or subtracted, and an optional modifier subfield. An external name is one defined by the segref pseudo-operation. An explicit link reference must begin with a segment name enclosed in "angle brackets" (the less-than and greater-than characters) and followed by a vertical bar. This can optionally be followed by an entryname in square brackets. For example:

```
<segname>|[entryname]  
<segname>|0,5*
```

A segment name of *text, *link, or *static indicates a reference to this procedure's text, linkage, or static sections.

A link pair is constructed for each combination of segment name, entryname, arithmetic expression, and tag that is referenced.

Literal Expressions

A literal reference causes the instruction to refer to a word in a literal pool that contains the value specified. However, the du and dl modifiers cause the value to be stored directly in the address field of the instruction. The various formats of literals are described in the following paragraphs.

A decimal literal can be signed. If it contains a decimal point or exponent, it is floating point. If the exponent begins with "d" instead of "e", it is double precision. A binary scale factor beginning with "b" indicates fixed point and forces conversion from floating point.

An octal literal begins with an "o" followed by up to 12 octal digits.

ASCII literals can occur in two forms. One form begins with a decimal number between 1 and 32 followed by "a" followed by the number of data characters specified by the integer preceding the "a", which can cross statement delimiters. The other form begins with "a" followed by up to four data characters, which can be delimited by the newline character.

A GBCD literal begins with "h" followed by up to six data characters, which can be delimited by the newline character. Translation is performed to the 6-bit character code.

An ITS (ITP) literal begins with "its" ("itp") followed by a parenthesized list containing the same operands accepted by the its (itp) pseudo-operation. The value is the same as that created by the pseudo-operation.

A variable-field literal begins with "v" followed by any number of decimal, octal, and ASCII subfields as in the vfd pseudo-operation. It must be enclosed in parentheses if a modifier subfield is to be used.

Modifiers

These specify indirection, index register address modification, immediate operands, and miscellaneous tally word operations. They can be specified as 2-digit octal numbers (particularly useful for instructions like stba) or symbolically using the mnemonics described here.

Simple register modification is specified by using any of the register designators listed below. It causes the contents of the selected register to be added to the effective address.

<u>Designators</u>		<u>Register</u>
x0	0	index register 0
x1	1	index register 1
x2	2	index register 2
x3	3	index register 3
x4	4	index register 4
x5	5	index register 5
x6	6	index register 6
x7	7	index register 7
n	none	(no modification)
au		A bits 0-17
al		A bits 18-35 or 0-35
qu		Q bits 0-17
ql		Q bits 18-35 or 0-35
ic		instruction counter

In addition to the above, any symbol that is not otherwise a valid modifier (e.g., au, ql, x7) may be used as a modifier to designate an index register. Thus,

```
equ    regc,3  
lda    sp|0,*regc
```

is equivalent to:

```
lda    sp|0,*3
```

Register-then-indirect modification is specified by using any of the register designators followed by an asterisk. If the asterisk is used alone, it is equivalent to the n* modifier. The register is added to the effective address, then the address and modifier fields of the word addressed are used in determining the final effective address. Indirect cycles continue as long as the indirect words contain an indirect modifier.

Indirect-then-register modification is specified by placing an asterisk before any one of the register designators listed above.

Direct modifiers are du and dl. They cause an immediate operand word to be fabricated from the address field of the instruction. For dl, the 18 address bits are right-justified in the effective operand word; for du they are left-justified. In either case, the remaining 18 bits of the effective operand are filled with 0's.

Segment addressing modifiers are its and itp; they can only occur in an indirect word pair on a double-word boundary. The addressing modifier its causes the address field of the even word to replace the segment number of the effective address, then continues the indirect cycle with the odd word of the pair. Nearly all indirection in Multics uses ITS pairs. For itp, see the Multics Processor Manual.

Tally modifiers i, ci, sc, scr, ad, sd, id, di, idc, and dic control incrementing and decrementing of the address and tally fields in the indirect word. They are difficult to use in Multics because the indirect word and the data must be in the same segment.

Fault tag modifiers f1, f2, and f3 cause distinct hardware faults whenever they are encountered. The modifier f2 is reserved for use in the Multics dynamic linking mechanism; the other modifiers result in the signalling of the conditions "fault_tag_1" and "fault_tag_3".

EIS Modifiers

An EIS modifier appears in the first word of an EIS multiword instruction. It affects the interpretation of operand descriptors in subsequent words of the instruction. No check is made by ALM to determine whether the modifier specified is consistent with the operand descriptor specified elsewhere.

An EIS modifier consists of one or more subfields separated by commas. Each subfield contains either a keyword as listed below, a register designator, or a logical expression. The values of the subfields are OR'ed together to produce the result.

<u>Keyword</u>	<u>Meaning</u>
pr	Descriptor contains a pointer register reference.
id	Descriptor is an indirect word pointing to the true descriptor.
rl	Descriptor length field names a register containing data length.

Separate Static Object Segments

If a separate static object segment is desired, a join pseudo-operation specifying static should exist in the program.

Pseudo-operations

The pseudo-operations are listed below in alphabetical order.

- acc /string/,expression**
assembles the ASCII string <string> into as many contiguous words as are required (up to 42). The delimiting character (/ above) can be any non-white-space character. The quoted string can contain newline and semicolon characters. The length of the string is placed in the first character position in acc format. If present, expression defines the length of the string; otherwise, the length is the actual length of the quoted string. If the given string is shorter than the defined length, it is padded on the right with blanks.
- aci /string/,expression**
is similar to acc, but no length is stored. The first character position contains the first character in aci format.
- arg operand**
assembles exactly like an instruction with a zero opcode. Any form of instruction operand can be used.
- bci /string/,expression**
is similar to aci, but uses GBCD 6-bit character codes and GBCD blanks for padding.
- bfs name,expression**
reserves a block of expression words with name defined as the address of the first word after the block reserved.
- bool name,expression**
defines the symbol name with the logical value expression. See the definition of logical expressions above under "Logical Expressions."
- bss name,expression**
defines the symbol name as the address of a block of expression words at the current location. The name can be omitted, in which case the storage is still reserved.
- call routine(arglist)**
calls out to the procedure routine using the argument list at arglist. Both routine and arglist can be any valid instruction operand, including tags. If arglist and the parentheses are omitted, an empty argument list is created. All registers are saved and restored by call.
- dec number1,number2,...,number_n**
assembles the decimal integers number1, number2, through number_n into consecutive words.

desc4a address(offset),length
desc6a address(offset),length
desc9a address(offset),length

generates one of the operand descriptors of an EIS multiword instruction. The address is any arithmetic expression, possibly preceded by a pointer register subfield as in an instruction operand. The offset is an absolute arithmetic expression giving the offset (in characters) to the first bit of data. It can be omitted if the parentheses are also omitted. The length is either a built-in index register name (al, au, ql, x0, etc.) or an absolute arithmetic expression for the data length field of the descriptor. The character size (in bits) is specified as part of the pseudo-operation name.

desc4fl address(offset),length,scale
desc4ls address(offset),length,scale
desc4ns address(offset),length,scale
desc4ts address(offset),length,scale

generates an operand descriptor for a decimal string. The scale is an absolute arithmetic expression for a decimal scaling factor to be applied to the operand. It can be omitted, and is ignored in a floating-point operand. Data format is specified in the pseudo-operation name: desc4fl indicates floating point, desc4ls indicates leading sign fixed point, desc4ns indicates unsigned fixed point, and desc4ts indicates trailing sign fixed point. Nine-bit digits can be specified by using desc9fl, desc9ls, desc9ns, and desc9ts.

descb address(offset),length

generates an operand descriptor for a bit string. Both offset and length are in bits.

eight

(see the even pseudo-operation)

end

terminates the source segment.

entry name1,name2,...,namen

generates entry sequences for labels name1, name2, through namen and makes the externally-defined symbols name1, name2, through namen refer to the entry sequence code rather than directly to the labels. The entry sequence performs such functions as initializing base register pr4 to point to the linkage section, which is necessary to make external symbolic references (link, segref, explicit links). The entry sequence can use (alter) base register pr2, index registers 0 and 7, and the A and Q registers. It requires pr6 and pr7 to be properly set (as they normally are).

equ name,expression

defines the symbol name with the arithmetic value expression.

even

inserts padding (nop) to a specified word boundary.

firstref extexpression1(extexpression2)

calls the procedure extexpression1 with the argument pointer extexpression2 the first time (in a process) that this object segment is linked to by an external symbol. If extexpression2 and the parentheses are omitted, an empty argument list is supplied. The expressions are any external expressions, including tags.

getlp
sets the pointer register pr4 to point to the linkage section. This can be used with segdef to simulate the effect of entry. This operator can use pointer register pr2, index registers 0 and 7, and the A and Q registers, and requires pr6 and pr7 to be set properly.

include segmentname
inserts the text of the segment segmentname.incl.alm immediately after this statement. A standard include library search is done to find the include file. See "System Libraries and Search Rules" in Section III of the MPM Reference Guide.

inhibit off
instruct assembler to turn off the interrupt inhibit bit in subsequent instructions. This mode continues until the inhibit on pseudo-operation is used.

inhibit on
instructs assembler to turn on the interrupt inhibit bit (bit 28) in subsequent instructions. This mode continues until the inhibit off pseudo-operation is used.

itp prno,offset,tag
generates an ITP pointer referencing the pointer register prno.

its segno,offset,tag
generates an ITS pointer to the segment segno, word offset <offset>, with optional modifier tag. If the current location is not even, a word of padding (nop) is inserted. Such padding causes any labels on the statement to be incorrectly defined.

join /text/name1,name2,.../link/name3,name4,.../static/name5,name6,....
appends the location counters name1, name2, etc., to the text section, appends the location counters name3, name4, etc., to the linkage section and appends the location counters name5, name6, etc., to the static section. Any number of names can appear. Each name must have been previously referred to in a use statement. Any location counters not joined are appended to the text section. If both link and static are specified in join pseudo-operations, then a warning is printed on the terminal.

link name,extexpression
defines the symbol name with the value equal to the offset from lp to the link pair generated for the external expression extexpression. An external expression can include a tag subfield. The name is not an external symbol, so an instruction should refer to this link by:
pr4|name,*

mod <expression>
inserts padding (nop) to an <expression> word boundary.

name objectname
specifies again the object segment name as it appears in the object segment. By default, the storage system name is used.

null
is ignored. This pseudo-operation is used for comments.

oct number1,number2,...,numbern
is like dec, with octal integer constants.

odd
(see the even pseudo-operation)

org expression
sets the location counter to the value of the absolute arithmetic expression <expression>. The expression can only use symbols previously defined.

push expression
creates a new stack frame for this procedure, containing expression words. If expression is omitted (the usual case), the frame is just large enough to contain all cells reserved by temp, tempd, and temp8.

rem
(see the null pseudo-operation)

return
is used to return from a procedure that has performed a push.

segdef name1,name2,...,namen
makes the labels name1, name2, through namen available to the linker for referencing from outside programs, using the symbolic names name1, name2, through namen. Such incoming references go directly to the labels name1, name2 through namen so the segdef pseudo-operation is usually used for defining external static data. For program entry points, the entry pseudo-operation is usually used.

segref segname,name1,name2,...,namen
defines the symbols name1, name2, through namen as external symbols referencing the entry points name1, name2, through namen in segment segname. This defines a symbol with an implicit base register reference.

set name,expression
assigns the arithmetic value expression to the symbol name. Its value can be reset in other set statements.

shortcall routine
calls out to routine using the argument list pointed to by pr0. Only pr4 and pr6 are preserved by shortcall.

shortreturn
is used to return from a procedure that has not performed a push.

sixtyfour
(see the even pseudo-operation)

temp name1(n1),name2(n2),...,namen(nn)
defines the symbols name1, name2, through namen to reference unique stack temporaries of n1, n2, through nn words each. Each ni is an absolute arithmetic expression and can be omitted (the parentheses should also be omitted). The default is one word per namei.

temp8 name1(n1),name2(n2),...,namen(nn)
is similar to temp, except that 8-word units are allocated, each on an 8-word boundary.

tempd name1(n1),name2(n2),...,namen(nn)
is similar to temp, except that n1 (n2 through nn) double words are allocated, each on a double-word boundary.

use name
assembles subsequent code into the location counter name. The default location counter is ".text."

vfd T₁L₁/expression₁,T₂L₂/expression₂,...,T_nL_n/expression_n
is variable format data. Each expression_i is of type T_i and is stored in the next L_i bits of storage. As many words are used as required. Individual items can cross word boundaries and exceed 36 bits in length. Type is indicated by the letters "a" (ASCII constant) or "o" (logical expression) or none (arithmetic expression). Regardless of type, the low-order L_i bits of data are used, padded if needed on the left. The T_i can appear either before or after L_i.

Restrictions: The total length of the variable format data cannot exceed 128 words. A relocatable expression cannot be stored in a field less than 18 bits long, and it must end on either bit 17 or bit 35 of a word.

zero expression₁,expression₂
assembles expression₁ into the left 18 bits of a word and expression₂ into the right 18 bits. Both subfields default to zero.

Name: alm_abs, aa

The alm_abs command submits an absentee request to perform ALM assemblies. The absentee process for which alm_abs submits a request assembles the segments named and dprints and deletes each listing segment if it exists. If the -output_file control argument is not specified, an output segment, path.absout, is created in the user's working directory. (If more than one path is specified, the first is used.) If the segment to be assembled cannot be found, no absentee request is submitted.

Usage

alm_abs paths {alm_arg} {-dp_args} {-control_args}

where:

1. paths
are pathnames of segments to be assembled.
2. alm_arg
can be the -list control argument accepted by the alm command (described earlier in this document).
3. dp_args
can be one or more control arguments (except -delete) accepted by the dprint command. (See the MPM Commands for a description of the dprint command.)
4. control_args
can be one or more of the following control arguments:
 - queue N, -q N
specifies in which priority queue the request is to be placed ($N \leq 3$). The default queue is 3. The listing segment is also dprinted in queue N.
 - hold
specifies that alm_abs should not dprint or delete the listing segment.
 - output_file path, -of path
specifies that absentee output is to go to segment path where path is a pathname.

Notes

Control arguments and segment pathnames can be mixed freely and can appear anywhere on the command line after the command. All control arguments apply to all segment pathnames. If an unrecognizable control argument is given, the absentee request is not submitted.

Unpredictable results can occur if two absentee requests are submitted that could simultaneously attempt to assemble the same segment or write into the same absout segment.

When performing several assemblies, it is more efficient to give several segment pathnames in one command rather than several commands. With one command, only one process is set up. The links that need to be snapped when setting up a process and when invoking the assembler need be snapped only once.

archive_sort

archive_sort

Name: archive_sort, as

The archive_sort command is used to sort the components of an archive segment. The components are sorted into ascending order by name using the standard ASCII collating sequence. The original archive segment is replaced by the sorted archive. For more information on archives and reordering them, see the archive command in the MPM Commands and the reorder_archive command in this document.

Usage

archive_sort paths

where paths are the pathnames of the archive segments to be sorted. The user need not supply the archive suffix.

Notes

There may be no more than 1000 components in an archive segment that is to be sorted.

Storage system errors encountered while attempting to move the temporary sorted copy of the archive segment back into the user's original segment result in diagnostic messages and preservation of the sorted copy in the user's process directory. If the original archive segment is protected, the user is interrogated to determine whether it should be overwritten.

Name: `area_status`

The `area_status` command is used to display certain information about an area.

Usage

```
area_status area_name {-control_args}
```

where:

1. `area_name`
is a pathname specifying the segment containing the area to be looked at.
2. `control_args`
can be chosen from the following:
 - trace
displays a trace of all free and used blocks in the area.
 - offset N, -ofs N
specifies that the area begins at offset N (octal) in the given segment.
 - long, -lg
dumps the contents of each block in both octal and ASCII format.

Note

If the area has internal format errors, these are reported. The command does not report anything about (old) buddy system areas except that the area is in an obsolete format.

Name: `copy_names`

The `copy_names` command copies all names of one entry (directory, segment, multisegment file, or link) to another. All names are left on the original entry. The two entries cannot reside in the same directory because name duplication is not allowed in the same directory. To move the alternate names see the `move_names` command in this document.

Usage

```
copy_names from_path1 {to_path1 ... from_pathn to_pathn}
```

where:

1. `from_pathi`
is the pathname of the entry whose names are to be copied.
2. `to_pathi`
is the pathname of the entry to which all names on `from_pathi` are to be copied. If this argument is omitted, the working directory is assumed.

Note

The equal convention may be used.

Name: create_area

The create_area command creates an area and initializes it with user-specified area management control information.

Usage

create_area virtual_ptr {-control_args}

where:

1. virtual_ptr
is a virtual pointer to the area to be created. The syntax of virtual pointers is described in the cv_ptr_subroutine description. If the segment already exists, the specified portion is still initialized as an area.
2. control_args
can be chosen from the following:
 - no_freeing
allows the area management mechanism to use a faster allocation strategy that never frees.
 - dont_free
is used during debugging to disable the free mechanism. This does not affect the allocation strategy.
 - zero_on_alloc
instructs the area management mechanism to clear blocks at allocation time.
 - zero_on_free
instructs the area management mechanism to clear blocks at free time.
 - extend
causes the area to be extensible, i.e., span more than one segment. This feature should be used only for perprocess, temporary areas.
 - size N
specifies the octal size, in words, of the area being created or of the first component, if extensible. If this control argument is omitted, the default size of the area is the maximum size allowable for a segment.
 - id STR
specifies a string to be used in constructing the names of the components of extensible areas.

delete_external_variables

delete_external_variables

Name: delete_external_variables

The delete_external_variables command deletes from the user's name space specified variables managed by the system for the user. All links to those variables are unsnapped and their storage is freed.

Usage

delete_external_variables names {-control_arg}

where:

1. names
are the names of the external variables, separated by spaces, to be deleted.
2. control_arg
is -unlabeled_common (or -uc) to indicate unlabeled (or blank) common.

display_component_name

display_component_name

Name: display_component_name, dcn

The display_component_name command converts an offset within a bound segment (e.g., bound_zilch_|23017) into an offset within the referenced component object (e.g., comp|1527). This command is especially useful when it is necessary to convert an offset within a bound segment (as displayed by a stack trace) into an offset corresponding to a compilation listing.

Usage

display_component_name path offsets

where:

1. path is the pathname of a bound object segment.
2. offsets are octal offsets within the text of the bound object segment specified by the path argument.

Example

The command line:

display_component_name bound_zilch_ 17523 64251

might respond with the following lines:

```
17523    component5|1057
64251    component7|63
```

Name: error_table_compiler, etc

The error_table_compiler command compiles a table of status codes and associated messages from symbolic ASCII source segments. The output is in a format suitable for the ALM assembler to produce a standard status code table.

Usage

error_table_compiler error_table

where error_table specifies a source segment in the format described below. An et suffix is added to the source segment name. The output segment is named error_table.alm. This segment must then be assembled by the ALM assembler prior to using it.

Notes

Each status code is defined by a statement in the source segment that specifies the name, short message, and long message associated with a status code. Any number of names may be given to a status code; each name must be 30 characters or less. Blanks and newline characters in the name are ignored. Each name is delimited by a colon (:).

The short message is eight characters or less in length. Blanks and newline characters in the short message are ignored. The short message is terminated by a comma (,). The short message (but not the terminating comma) may be omitted; in this case, the short message is set to the first eight characters of the name.

The long message is 100 characters or less in length. Leading blanks, newline characters, and blanks following a newline character are ignored in the long message. The long message is terminated by a semicolon (;). Comments that begin with the characters /* and end with the characters */ are ignored.

The syntax of a statement is:

```
name1: ... namen: short_message, long_message;
```

An error table source segment is composed of a series of statements of the above format, terminated by an end statement. The format of the end statement is:

```
end;
```

There is a special statement that should not be used except when compiling the hardcore system error table. This statement causes a special nondynamic initialization of status codes in that segment, optimizing the system error table slightly. This statement can appear anywhere in the source before the end statement. The format of this statement is:

```
system;
```

See the "List of System Status Codes and Meanings" in Section VII of the MPM Reference Guide for a list of system error table status codes.

Example

The comment syntax is similar to PL/I in the following example:

```
/* This is a sample error table compiler source segment. */

too_few_arguments:    toofew,There were too few arguments.;

could_not_access_data:  noprivlg,The user is not sufficiently
privileged to access required data;

fatal: disaster:      disaster,There was a disastrous error in the data
base;

end;
```

Each status code in the table produced by error_table_compiler should be referenced as a fixed binary(35) quantity, known externally:

```
declare user_errors$disaster fixed bin(35) external,
        code fixed bin(35);

call data_base_manager (info, code);
if code = user_errors$disaster /* this is bad */
then call kill_subsystem;
```

list_external_variables

list_external_variables

Name: list_external_variables

The list_external_variables command prints information about variables managed by the system for the user, including FORTRAN common and PL/I external static variables whose names do not contain dollar signs. The default information is the location and size of each specified variable.

Usage

list_external_variables names {-control_args}

where:

1. names
are names of external variables, separated by spaces.
2. control_args
can be chosen from the following:
 - unlabeled_common, -uc
is the name for unlabeled (or blank) common.
 - long, -lg
prints how and when the variables were allocated.
 - all, -a
prints information for each variable the system is managing.
 - no_header, -nhe
suppresses the header.

Name: list_temp_segments

The list_temp_segments command lists the segments currently in the temporary segment pool associated with the user's process. This pool is managed by the get_temp_segments_ and release_temp_segments_ subroutines (described in the MPM Subroutines).

Usage

```
list_temp_segments {names} {-control_arg}
```

where:

1. names is a list of names identifying the programs whose temp segments are to be listed.
2. control_arg is -all (or -a) to list all temporary segments. If the command is issued with no control argument, it lists only those temporary segments currently assigned to some program.

Examples

To list all the segments currently in the pool, type:

```
! list_temp_segments -all
      5 Segments, 2 Free
!BBBCdfghgffkkkl.temp.0246 work
!BBBCdffdddfdfkl.temp.0247 work
!BBBCddffdfhfh.hhh.temp.0253 (free)
!BBBCdgdgfhgfsf.temp.0254 (free)
!BBBCvdfvgvvgvvv.temp.0321 editor
```

To list the segments currently in use, type:

```
! list_temp_segments
      3 Segments
!BBBCdfghgffkkkl.temp.0246 work
!BBBCdffdddfdfkl.temp.0247 work
!BBBCvdfvgvvgvvv.temp.0321 editor
```

list_temp_segments

list_temp_segments

To list segments used by the program named editor, type:

! list_temp_segments editor

1 segment

!BBBCvdvfgvdgvvv.temp.0321 editor

Name: mbx_add_name, mban

The mbx_add_name command adds an alternate name to the existing name(s) of a mailbox.

Usage

mbx_add_name path names

where:

1. path is the pathname of a mailbox. The atar convention is allowed.
2. names are names to be added to a mailbox. The equal convention is allowed.

Notes

If path does not have the mbx suffix, one is assumed.

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

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

Example

The command line:

```
mban >udd>m>Gillis>**.private ==.pv
```

adds to every mailbox in >udd>m>Gillis whose name ends in ".private.mbx" a similar name ending in ".pv.mbx".

mbx_create

mbx_create

Name: mbx_create, mber

The mbx_create command creates a mailbox with a specified name in a specified directory.

Usage

mbx_create paths

where paths are the pathnames of mailboxes to be created.

Notes

If path_i does not have the mbx suffix, one is assumed.

The user must have modify and append permission on the directory in which he is creating a mailbox.

If the creation of a mailbox would introduce a duplication of names within the directory, and if the old mailbox has only one name, the user is interrogated as to whether he wishes the old mailbox to be deleted. If the user answers "no", no action is taken. If the old mailbox has multiple names, the conflicting name is removed and a message to that effect is issued to the user.

The extended access placed on a new mailbox is:

```
adros      user who created the mailbox
ao         *.SysDaemon.*
ao         *.*.*
```

For more information on extended access, see the mail command in the MPM Commands and mbx_set_acl in this document.

Example

The command line:

```
mber Green Jones.home >udd>Multics>Gillis>Gillis
```

creates the mailboxes Green.mbx and Jones.home.mbx in the working directory and creates the mailbox Gillis.mbx in the directory >udd>Multics>Gillis.

Name: `mbx_delete`, `mbdl`

The `mbx_delete` command deletes the specified mailboxes.

Usage

`mbx_delete paths`

where `paths` are the pathnames of mailboxes to be deleted. The star convention is allowed.

Notes

If `pathi` does not have the `mbx` suffix, one is assumed.

The user must have modify permission on the containing directory and delete extended access on the mailbox. If delete access is lacking, the user is asked whether he wants the mailbox deleted. If the user answers "yes", delete access is forced. If he answers "no", no action is taken.

For more information on extended access, see the `mail` command in the MPM Commands and `mbx_set_acl` in this document.

Examples

The command line:

```
mbdl **
```

deletes all mailboxes in the working directory.

The command line:

```
mbdl Green >udd>Multics>Gillis>Jones
```

deletes the mailbox `Green.mbx` from the working directory and the mailbox `Jones.mbx` from the directory `>udd>Multics>Gillis`.

Name: mbx_delete_acl, mdba

The `mbx_delete_acl` command deletes entries from the access control list (ACL) of a given mailbox.

Usage

```
mbx_delete_acl path {access_names}
```

where:

1. `path` is the pathname of a mailbox. The star convention is allowed.
2. `access_names` are access control names of the form `Person_id.Project_id.tag`. If all three components are present, the ACL entry with that name is deleted. If one or more components is missing, all ACL entries with matching names are deleted. (The matching strategy is described below under "Notes.") If no access control name is specified, the user's `Person_id` and current `Project_id` are assumed.

Notes

If `path` does not have the `mbx` suffix, one is assumed.

The user must have modify permission on the containing directory.

ACL entries for `*.SysDaemon.*` and `*.*.*` cannot be deleted. To deny them access to a mailbox, set the access to null giving `*.SysDaemon` and `*.*.*` as the `access_names` arguments.

The matching strategy for access control names is as follows:

1. A literal component name, including `"*"`, matches only a component of the same name.
2. A missing component name not delimited by a period is taken to be a literal `"*"` (e.g., `*.Multics` is treated as `*.Multics.*`). Missing components on the left must be delimited by periods.
3. A missing component name delimited by a period matches any component name.

Some examples of `access_names` and which ACL entries they match are:

<code>*.*.*</code>	matches only the ACL entry <code>"*.*.*"</code> .
<code>Multics</code>	matches only the ACL entry <code>"Multics.*.*"</code> . (The absence of a leading period makes <code>Multics</code> the first component.)
<code>.Multics.</code>	matches every ACL entry with middle component of <code>Multics</code> .
<code>..</code>	matches every ACL entry.
<code>.</code>	matches every ACL entry with a last component of <code>"*"</code> .
<code>""</code>	(null string) matches every entry ending in <code>"*.*.*"</code> .

Example

The command line:

```
mbda Green .Multics Jones
```

deletes from the ACL of the mailbox `Green.mbx` all entries whose name ends in `".Multics.*"` and the specific entry `"Jones.*.*"`. If no ACL entries exist for one of the specified access names (e.g., ending in `".Multics.*"` from above example), an error message is printed.

mbx_delete_name

mbx_delete_name

Name: mbx_delete_name, mbdn

The mbx_delete_name command removes a specified name from a specified mailbox.

Usage

mbx_delete_name paths

where paths are the pathnames of mailboxes. The star convention is allowed.

Notes

If path_i does not have the mbx suffix, one is assumed.

The user must have modify permission on the containing directory.

The entryname portion of path_i is the name to be removed. If removing the name would leave no names on the mailbox, the user is asked if he wants the mailbox to be deleted. If he answers "no", no action is taken with respect to that entryname.

Example

The command line:

```
mbdn **.private >udd>Multics>Gillis>Jones
```

removes from the mailboxes in the working directory all names ending in ".private.mbx", and removes the name Jones.mbx from the mailbox Jones.mbx in the directory >udd>Multics>Gillis.

Name: mbx_list_acl, mbla

The mbx_list_acl command lists all or part of the access control list (ACL) of a given mailbox.

Usage

mbx_list_acl path {access_names}

where:

1. path is the pathname of a mailbox. The star convention is allowed.
2. access_names are access control names of the form Person_id.Project_id.tag. If all three components are present, the ACL entry with that name is listed. If one or more components is missing, all ACL entries with matching names are listed. The matching strategy is described under "Notes" in the description of the mbx_delete_acl command in this document. If no access control name is specified, or if the access control name is -all or -a, the entire ACL is listed.

Note

If path does not have the mbx suffix, one is assumed.

Example

The command line:

mbla Green *.* Jones Gillis...

lists, from the ACL of Green.mbx, the specific entries "*.*" and "Jones.*" and all entries with a first component of Gillis. If no ACL entry with a first component of Gillis exists, an error message is printed.

mbx_rename

mbx_rename

Name: mbrn, mbrn

The mbrn command replaces a given name on a mailbox with a different name, without affecting any other names the mailbox has.

Usage

```
mbx_rename path1 name1 {... pathn namen}
```

where:

1. path_i is the pathname of a mailbox. The entryname portion is the name to be replaced. The star convention is allowed.
2. name_i is the new name to be placed on the mailbox. The equal convention is allowed.

Notes

If path_i does not have the mbrn suffix, one is assumed.

The user must have modify permission on the directory specified by path_i.

Since two entries in a directory cannot have the same entryname, special action is taken by this command if name_i already exists in the directory specified by path_i. If the mailbox having the entryname name_i has an additional name, entryname name_i is removed and the user is informed of this action by a message printed on his terminal. If the mailbox having the entryname name_i has only one name, the user is asked if that mailbox is to be deleted. If the user answers "no", the renaming operation does not take place.

Example

The command line:

```
mbrn *.private ==.public >udd>m>Joe>Normal Urgent
```

replaces all mailbox names ending in private.mbx in the working directory with similar names ending in public.mbx and renames the mailbox Normal.mbx in the directory >udd>m>Joe to Urgent.mbx.

Name: mbx_set_acl, mbsa

The mbx_set_acl command changes and adds entries to the access control list (ACL) of a given mailbox.

Usage

```
mbx_set_acl path mode1 {access_name1 ... moden} access_namen
```

where:

1. path is the pathname of a mailbox. The star convention is allowed.
2. mode_i is a valid access mode. It can consist of any or all of the letters adros (see "Notes" below) or it can be "n", "null" or "" to specify null access.
3. access_name_i is an access control name of the form Person_id.Project_id.tag. If all three components are present, the ACL entry with that name is changed; if no entry with that name exists, one is added. If one or more components is missing, all ACL entries with names that match the access control name are changed. The matching strategy is described under "Notes" in the description of the mbx_delete_acl command in this document. If no access control name is specified, the user's Person_id and current Project_id are assumed.

Notes

If path does not have the mbx suffix, one is assumed.

The user must have modify permission on the containing directory.

Access on a newly created mailbox is automatically set to adros for the user who created it, ao for *.SysDaemon.*, and ao for *.*.*. The extended access modes for mailboxes are:

add	a	add a message
delete	d	delete any message
read	r	read any message
own	o	read or delete only your own messages; that is, those sent by you
status	s	find out how many messages are in the mailbox

Example

The command line:

```
mbsa Green adros Klein.. null Jones.Multics a *.*.*
```

manipulates the ACL of Green.mbx so that all previously existing entries with a first component of Klein have adros access, Jones.Multics.* has null access and *.*.* has "a" access. If no ACL entry exists with a first component of Klein, an error message is printed.

Name: `mbx_set_max_length`, `mbsml`

The `mbx_set_max_length` command sets the maximum length of a mailbox. The mailbox must be empty for this command to work.

Usage

`mbx_set_max_length path length {-control_args}`

where:

1. `path`
is the pathname of a mailbox. If the suffix `mbx` is missing, it is assumed. The star convention is allowed.
2. `length`
is the maximum length in words. This number must be greater than zero. If it is not a multiple of 1024 words, it is rounded to the next higher multiple of 1024 with a warning.
3. `control_args`
can be chosen from the following list of control arguments:
 - decimal, -dc
length is a decimal number. (This is the default.)
 - octal, -oc
length is an octal number.
 - brief, -bf
suppress the warning that length has been rounded to the next higher multiple of 1024 words.

Name: move_names

The move_names command moves all the alternate names from one entry (directory, segment, multisegment file, or link) to another. The name used to designate the entry is not moved. To copy the alternate names, see the copy_names command in this document.

Usage

`move_names from_path1 {to_path1 ... from_pathn to_pathn}`

where:

1. from_pathi
is the pathname of the entry whose alternate names are to be moved.
2. to_pathi
is the pathname of the entry to which alternate names on from_pathi are to be moved. If to_path is omitted, the working directory is assumed.

Note

The equal convention may be used.

Name: `print_bind_map`

The `print_bind_map` command displays all or part of the bind map of an object segment generated by version number 4 or subsequent versions of the binder.

Usage

```
print_bind_map path {components} {-control_args}
```

where:

1. `path`
is the pathname of a bound object segment.
2. `components`
are the optional names of one or more components of this bound object and/or the bindfile name. Only the lines corresponding to these components are displayed. A component name must contain one or more nonnumeric characters. If it is purely numerical, it is assumed to be an octal offset within the bound segment and the lines corresponding to the component residing at that offset are displayed. A numerical component name can be specified by preceding it with the `-name` control argument (see below). If no component names are specified, the entire bind map is displayed.
3. `control_args`
may be chosen from the following list:
 - `-long, -lg`
prints the components' relocation values (also printed in the default brief mode), compilation times, and source languages.
 - `-name STR, -nm STR`
is used to indicate that STR is really a component name, even though it appears to be an octal offset.
 - `-no_header, -nhe`
omits all headers, printing only lines concerning the components themselves.

Name: print_link_info, pli

The print_link_info command prints selected items of information for the specified object segments.

Usage

```
print_link_info paths {-control_args}
```

where:

1. paths
are the pathnames of object segments.
2. control_args
can be chosen from the following list. (See "Notes" below.)
 - length, -ln
print only the lengths of the sections in path_i.
 - entry, -et
print only a listing of the path_i external definitions, giving their symbolic names and their relative addresses within the segment.
 - link, -lk
print only an alphabetically sorted listing of all the external symbols referenced by path_i.
 - long
prints more information when the header is printed. Additional information includes a listing of source programs used to generate the object segment, the contents of the "comment" field of the symbol header (often containing compiler options), and any unusual values in the symbol header.
 - header, -he
prints the header (The header is not printed by default, if the -length, -entry, or -link control argument is specified.)
 - no_header
suppresses printing of the header.

Note

Control arguments can appear anywhere on the command line and apply to all pathnames.

Example

```
! print_link_info program -long -length
```

```
program 07/30/76 1554.2 edt Fri
```

```
Object Segment >udd>Work>Wilson>program  
Created on 07/30/76 0010.1 edt Fri  
by Wilson.Work.a
```

```
using Experimental PL/I Compiler of Thursday, July 26, 1976 at 21:38
```

```
Translator: PL/I  
Comment: map table optimize
```

```
Source:  
07/30/76 0010.1 edt Fri >user_dir_dir>work>Wilson>s>s>program.pl1  
12/15/75 1338.1 edt Mon >library_dir_dir>include>linkdcl.incl.pl1  
06/30/75 1657.7 edt Mon >library_dir_dir>include>object_info.incl.pl1  
10/06/72 1206.8 edt Fri >library_dir_dir>include>source_map.incl.pl1  
05/18/72 1512.4 edt Thu >library_dir_dir>include>symbol_block.incl.pl1  
01/17/73 1551.4 edt Wed >library_dir_dir>include>pl1_symbol_block.incl.pl1
```

```
Attributes: relocatable,procedure,standard
```

	Object	Text	Defs	Link	Symb	Static
Start	0	0	3450	3620	3656	3630
Length	11110	3450	150	36	5215	0

<ready>

Also printed is:

```
Severity, if it is nonzero.  
Entrybound, if it is nonzero.  
Text Boundary, if it is not 2.  
Static Boundary, if it is not 2.
```

print_linkage_usage

print_linkage_usage

Name: print_linkage_usage, plu

The print_linkage_usage command lists the locations and size of linkage and static sections allocated for the current ring. This information is useful for debugging purposes or for analysis of how a process uses its linkage segments.

A linkage section is associated with every procedure segment and every data segment that has definitions.

Usage

print_linkage_usage

Note

For standard procedure segments, the information printed includes the name of the segment, its segment number, the offset of its linkage section, and the size (in words) of both its linkage section and its internal static storage.

Name: reorder_archive

The reorder_archive command provides a convenient way of reordering the contents of an archive segment, eliminating the need to extract, order, and replace the entire contents of an archive. This command places specified components at the beginning of the archive, leaving any unspecified components in their original order at the end of the archive. For information on archives and how they can be sorted, see the archive command in the MPM Commands and the archive_sort command in this document.

Usage

```
reorder_archive {-control_arg1} path1 ... {-control_argn} pathn
```

where:

1. control_argi
may be chosen from the following:
 - console_input, -ci
indicates the command is to be driven from terminal input. (This is the default.)
 - file_input, -fi
indicates the command is to be driven from a driving list. (See "Notes" below.)
2. pathi
is the pathname of the archive segment to be reordered. If pathi does not have the archive suffix, one is assumed.

Notes

If no control arguments are specified, the -console_input control argument is assumed.

When the command is invoked with the -console_input control argument or with no control arguments, the message "input for archive_name" is printed where archive_name is the name of the archive segment to be reordered. Component names are then typed in the order desired, separated by linefeeds. A period (.) on a line by itself terminates input. The two-character line ".*" causes the command to print an asterisk (*). This feature can be used to make sure there are no typing errors before typing a period (.). The two-character line ".q" causes the command to terminate without reordering the archive.

The driving list (-file_input control argument) must have the name name.order where name.archive is the name of the archive segment to be reordered. The order segment must be in the working directory. It consists of a list of component names in the order desired, separated by linefeeds. No period (.) is necessary to terminate the list. Any errors in the list (name not found in the archive segment, name duplication) cause the command to terminate without altering the archive.

reorder_archive

reorder_archive

A temporary segment named ra_temp.archive is created in the user's process directory. This temporary segment is created once per process, and is truncated after it is copied into the directory specified by path_i. If the command cannot copy the temporary segment, it attempts to save it and rename it with the name of the archive specified.

The reorder_archive command does not operate upon archive segments containing more than 1000 components.

Name: reset_external_variables

The reset_external_variables command reinitializes system-managed variables to the values they had when they were allocated.

Usage

reset_external_variables names {-control_arg}

where:

1. names
are the names of the external variables, separated by spaces, to be reinitialized.
2. control_arg
is -unlabeled_common (or -uc) to indicate unlabeled (or block) common.

Note

A variable cannot be reset if the segment containing the initialization information is terminated after the variable is allocated.

set_max_length

set_max_length

Name: set_max_length, sml

The set_max_length command allows the maximum length of a nondirectory segment to be set. The maximum length is the maximum size the segment can attain. Currently, maximum length must be a multiple of 1024 words (one page).

Usage

set_max_length path length {-control_args}

where:

1. path
is the pathname of the segment whose maximum length is to be set. If path is a link, the maximum length of the target segment of the link is set. The star convention can be used.
2. length
is the new maximum length expressed in words. If this length is not a multiple of 1024 words, it is converted to the next higher multiple of 1024 words.
3. control_args
can be chosen from the following list of control arguments and can appear in any position:
 - decimal, -dc
says that length is a decimal number. (This is the default.)
 - octal, -oc
says that length is an octal number.
 - brief, -bf
suppresses a warning message that the length argument has been converted to the next multiple of 1024 words.

Notes

If the new maximum length is less than the current length of the segment, the user is asked if the segment should be truncated to the maximum length. If the user answers "yes", the truncation takes place and the maximum length of the segment is set. If the user answers "no", no action is taken.

The user must have modify permission on the directory containing the segment in order to change its maximum length.

set_max_length

set_max_length

Examples

The command line:

```
set_max_length report -oc 10000
```

sets the maximum length of the segment named report in the working directory to four pages.

The command line:

```
set_max_length *.archive 16384
```

sets the maximum length of all two-component segments with a second component of archive in the working directory to 16 pages.

Name: set_ring_brackets, srb

The set_ring_brackets command allows a user to modify the ring brackets of a specified segment.

Usage

set_ring_brackets path {ring_numbers}

where:

1. path
is the relative or absolute pathname of the segment whose ring brackets are to be modified.
2. ring_numbers
are the numbers that represent the three ring brackets (rb1 rb2 rb3) of the segment. The ring brackets must be in the allowable range 0 through 7 and must have the ordering:
$$rb1 \leq rb2 \leq rb3$$

If rb1, rb2, and rb3 are omitted, they are set to the user's current validation level.

rb1
is the number to be used as the first ring bracket of the segment. If rb1 is omitted, rb2 and rb3 cannot be given and rb1, rb2, and rb3 are set to the user's current validation level.

rb2
is the number to be used as the second ring bracket of the segment. If rb2 is omitted, rb3 cannot be given and is set, by default, to rb1.

rb3
is the number to be used as the third ring bracket of the segment. If rb3 is omitted, it is set to rb2.

Note

The user's process must have a validation level less than or equal to rb1. Ring brackets and validation levels are discussed in "Intraprocess Access Control" in Section VI of the MPM Reference Guide.

Name: set_system_storage

The set_system_storage command establishes an area as the storage region in which normal system allocations are performed.

Usage

```
set_system_storage {virtual_ptr -control_arg}
```

where:

1. virtual_ptr
is a virtual pointer to an initialized area. The syntax of virtual pointers is described in the cv_ptr_subroutine description. This argument must be specified only if the -system control argument is not supplied.
2. control_arg
is -system to specify the area used for linkage sections. This control argument must be specified only if virtual_ptr is not specified.

Notes

To initialize or create an area, refer to the description of the create_area command.

The area must be set up as either zero_on_free or zero_on_alloc.

It is recommended that the area specified be extensible.

Examples

The command line:

```
set_system_storage free_$free_
```

places objects in the segment whose reference name is free_ at the offset whose entry point name is free_.

set_system_storage

set_system_storage

The command line:

```
set_system_storage my_seg$
```

uses the segment whose reference name is my_seg. The area is assumed to be at an offset of 0 in the segment. The segment must already exist with the reference name my_seg and must be initialized as an area.

The command line:

```
set_system_storage my_seg
```

uses the segment whose (relative) pathname is my_seg. The segment must already exist.

Name: set_user_storage

The set_user_storage command establishes an area as the storage region in which normal user allocations are performed. These allocations include FORTRAN common blocks and PL/I external variables whose names do not contain dollar signs.

Usage

```
set_user_storage {virtual_ptr -control_arg}
```

where:

1. virtual_ptr
is a virtual pointer to an initialized area. The syntax of virtual pointers is described in the cv_ptr_ subroutine description. This argument must be specified only if the -system control argument is not specified.
2. control_arg
is -system to specify the area used for linkage sections. This control argument must be specified only if virtual_ptr is not specified.

Notes

To initialize or create an area, refer to the description of the create_area command.

The area must be set up as either zero_on_free or zero_on_alloc.

It is recommended that the area specified be extensible.

Examples

The command line:

```
set_user_storage free_$free_
```

places objects in the segment whose reference name is free_ at the offset whose entry point name is free_.

set_user_storage

set_user_storage

The command line:

set_user_storage my_seg\$

uses the segment whose reference name is my_seg. The area is assumed to be at an offset of 0 in the segment. The segment must already exist with the reference name my_seg and must be initialized as an area.

The command line:

set_user_storage my_seg

uses the segment whose (relative) pathname is my_seg. The segment must already exist.

SECTION VII

SUBROUTINE DESCRIPTIONS

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

Name

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

Entry

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

Usage

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

Notes

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

Other Headings

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

Status Codes

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

Treatment of Links

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

Name: 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 "List of System Conditions and Default Handlers" in Section VI of the MPM Reference Guide for further information.)

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

Usage

```
declare active_fnc_err_ entry options (variable);  
call active_fnc_err_ (code, caller, control_string, arg1, ..., argn);
```

where:

1. code is a standard status code (fixed bin(35)). (Input)
2. caller is the name (char(*)) of the calling procedure. It can be either varying or nonvarying. (Input)
3. control_string is an ioa_ subroutine control string (char(*)). (The ioa_ subroutine is described in the MPM Subroutines.) This argument is optional. See "Note" below. (Input)
4. arg_i are ioa_ subroutine arguments to be substituted into control_string. These arguments are optional. (However, they can only be used if the control_string argument is given first.) See "Note" below. (Input)

Note

The error message prepared by the active_fnc_err_ subroutine has the format:

```
caller: system_message user_message
```

where:

1. caller is the caller argument described above and should be the name of the procedure detecting the error.
2. 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.
3. 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.

Name: aim_check_

The aim_check_ subroutine determines 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);
```

where:

1. acc_att_i are access attributes. (Input)
2. 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);
```

where:

1. acc_att_i are access attributes. (Input)
2. 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);
```

where:

1. acc_att*i* are access attributes. (Input)
2. 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

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);
```

where:

1. info_ptr points to the structure described below. (Input)
2. code is a system status code. (Output)

Notes

The structure pointed to by info_ptr is described by the following PL/I declaration:

```
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 area_ptr          ptr,
    2 allocated_blocks  fixed bin,
    2 free_blocks       fixed bin,
    2 allocated_words   fixed bin (30),
    2 free_words        fixed bin (30);
```

where:

1. version is set by the caller and should be 1.
2. control are control bits describing the format and type of the area.
3. extend indicates whether the area is extensible.
"1"b yes
"0"b no

4. zero_on_alloc indicates whether blocks are cleared (set to all zeros) at allocation time.
"1"b yes
"0"b no
5. zero_on_free indicates whether blocks are cleared (set to all zeros) at free time.
"1"b yes
"0"b no
6. dont_free indicates whether free requests are disabled (for debugging).
"1"b yes
"0"b no
7. no_freeing indicates whether the allocation method assumes no freeing will be done.
"1"b yes
"0"b no
8. system indicates whether the area is managed by the system.
"1"b yes
"0"b no
9. mbz is not used and must be zeros.
10. owner is the name of the program that created the area if the area is extensible.
11. n_components is the number of components in the area.
12. size is the total number of words in the area.
13. version_of_area is 0 for (old) buddy system areas and 1 for standard areas.
14. area_ptr is filled in by the caller and can point to any component of the area.
15. allocated_blocks is the number of allocated blocks in the area.
16. 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).
17. allocated_words is the number of allocated words in the area.
18. 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.

The above structure is defined by the system include file, area_info.incl.pl1.

ascii_to_ebcdic_

ascii_to_ebcdic_

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 $0 \leq \text{octal_value} \leq 177$.

Entry: ascii_to_ebcdic_

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);
```

where:

1. `ascii_in` is a string of ASCII characters to be converted. (Input)
2. `ebcdic_out` is the EBCDIC equivalent of the input string. (Output)

Entry: ascii_to_ebcdic_\$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_$table char(128) external static;
```

ISOMORPHIC ASCII/EBCDIC CONVERSION TABLE

ASCII		EBCDIC	
GRAPHIC	OCTAL	HEXADECIMAL	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	+
,	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

GRAPHIC OCTAL HEXADECIMAL GRAPHIC

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
Y	131	E8	Y
Z	132	E9	Z
[133	AD	[(see "Notes")
\	134	E0	\
]	135	BD] (see "Notes")
^	136	5F	logical NOT
~	137	6D	~
a	140	79	a
b	141	81	b
c	142	82	c
d	143	83	d
e	144	84	e
f	145	85	f
g	146	86	g
h	147	87	h
i	150	88	i
j	151	89	j
k	152	91	k
l	153	92	l
m	154	93	m
	155	94	

GRAPHIC	OCTAL	HEXADECIMAL	GRAPHIC
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	}
~	176	A1	~
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 mapping discussed in "Punched Card Codes" in Section V of the MPM Reference Guide. The characters that differ when mapped are: [] \ and NL (newline).

Name: assign_

The assign_ subroutine assigns a specified source value to a specified target. Any PL/I arithmetic or string data type can be assigned to any other arithmetic or string data type; conversion is done according to the rules of PL/I. This subroutine uses rounding in the conversion when the target is floating point and 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);
```

where:

1. target_ptr points to the target of the assignment; it can contain a bit offset. (Input)
2. target_type specifies the type of the target; its value is $2*M+P$ where M is the Multics standard data type code (see "Multics Standard Data Type Formats" in Section V of the MPM Reference Guide) and P is 0 if the target is unpacked and 1 if the target is packed. (Input)
3. 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 fixed bin(17) unaligned fields; the left half of the word is the signed scale and the right half of the word is the precision. (Input)
4. source_ptr points at the source of the assignment; it can contain a bit offset. (Input)
5. source_type specifies the source type using the same format as target_type. (Input)
6. source_length is the string length or arithmetic scale and precision of the source using the same format as target_length. (Input)

check_star_name_

check_star_name_

Name: check_star_name_

The check_star_name_ subroutine validates an entryname to ensure that it has been formed according to the rules for constructing star names. For more information on star names, see "Constructing and Interpreting Names" in Section I of the MPM Commands. It also returns a nonstandard status code that indicates whether the entryname is a star name and whether it is a star name that matches every entryname.

Entry: check_star_name_\$path

This entry point accepts an absolute pathname as its input and validates the final entryname in that path.

Usage

```
declare check_star_name_$path entry (char(*), fixed bin(35));
call check_star_name_$path (path, code);
```

where:

1. path is the pathname whose final entryname is to be validated. Trailing spaces in the pathname character string are ignored. (Input)
 2. code is a status code. (Output) It may have the following values:

0	the entryname is valid and is not a star name (does not contain asterisks or question marks).
1	the entryname is valid and is a star name (does contain asterisks or question marks).
2	the entryname is valid and is a star name that matches every entryname (either **, or *.* , or *.*).
- error_table_\$badstar the entryname is invalid. It violates one or more of the rules for constructing star names.

Entry: check_star_name_\$entry

This entry point accepts the entryname to be validated as input.

check_star_name_

check_star_name_

Usage

```
declare check_star_name_$entry entry (char(*), fixed bin(35));  
call check_star_name_$entry (entryname, code);
```

where:

1. entryname is the entryname to be validated. Trailing spaces in the entryname character string are ignored. (Input)
2. code is as described above. (Output)

Notes

The procedure for obtaining a list of directory entries that match a given star name is explained in the description of the hcs_\$star_ subroutine in this document.

The procedure comparing an entryname with a given star name is explained in the description of the match_star_name_ subroutine in this document.

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, alm, and cput. Some conditions do not have messages and others cause special actions to be taken (such as finish). These are described in "Notes" below. For more information on conditions, see "Multics Condition Mechanism" in Section VI of the MPM Reference Guide.

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);
```

where:

1. area_ptr is a pointer to the area in which the message is to be allocated, if the message is to be returned. For safety, the area size should be at least 300 words. If the message is to be printed, the pointer is null. (Input)
2. m_ptr points to the allocated message if area_ptr is not null; otherwise it is not set. (Output)
3. 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)
4. 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
5. mc_ptr if not null, points to machine conditions describing the state of the processor at the time the condition was raised. (Input)
6. cond_name is the name of the condition being raised. (Input)
7. wc_ptr is usually null; but when mc_ptr points to machine conditions from ring 0, wc_ptr points to alternate machine conditions. (Input)
8. info_ptr if not null, points to the information structure described in the find_condition_info_ subroutine in this document. (Input)

Notes

The following conditions cause a return with no message:

command_error
command_question
stringsize

The finish condition does not usually cause a message to be printed; it does, however, cause all I/O blocks to be closed, so that no input/output may be done upon return.

continue_to_signal_

continue_to_signal_

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);
```

where code is a standard status code and is nonzero if the signalling procedure's frame is not found. (Output)

convert_aim_attributes_

convert_aim_attributes_

Name: convert_aim_attributes_

The convert_aim_attributes_ subroutine 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.

This subroutine is generally used by ring 0 auditing programs to format an access authorization or class into a character string suitable for logging in the syserr log.

Usage

```
declare convert_aim_attributes_ entry (bit(72) aligned, char(32) aligned);
call convert_aim_attributes_ (aim_bits, aim_chars);
```

where:

1. aim_bits is the binary representation to be converted. (Input)
2. 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)".



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 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 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_ I/O system.

Usage

```
declare convert_dial_message_$return_io_module entry (fixed bin(71),
char(*), char(*), fixed bin, 1 aligned like status_flags, fixed
bin(35));

call convert_dial_message_$return_io_module (message, channel_name,
io_module, n_dialed, flags, code);
```

where:

1. message is the event message to be decoded. (Input)
2. channel_name is the name of the channel that has dialed-up or hung-up. (Output)
3. io_module is the name of the iox_ I/O module to be used with the assigned device. (Output)
4. n_dialed is the number of terminals currently dialed to the process or -1. (Output)
5. 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;
```
6. code is a system status code. (Output)

convert_status_code_

convert_status_code_

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.

Usage

```
declare convert_status_code_ entry (fixed bin(35), char(8) aligned,  
char(100) aligned);
```

```
call convert_status_code_ (code, shortinfo, longinfo);
```

where:

1. code is a system status code. (Input)
2. shortinfo is a short status message corresponding to code. (Output)
3. longinfo is a long status message corresponding to code; the message is padded on the right with blanks. (Output)

Note

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.

—
cu_
—

—
cu_
—

Name: cu_

The cu_ (command utility) subroutine contains a number of useful command utility programs that provide functions not directly available in the PL/I language. Three of the entry points, cu_\$arg_count, cu_\$arg_ptr, and cu_\$cp, are described in the MPM Subroutines; the rest are described below.

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.

Usage

```
declare cu_$ready_proc entry;
call cu_$ready_proc ();
    or
declare 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);
```

where:

1. mode.ready_sw is the static ready switch that specifies whether the ready procedure should print a ready message. (Input)
"1"b print ready message
"0"b do not print ready message
2. mode.mbz is reserved for future use and must be "0"b. (Input)

Note

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 (see below). 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. (See the MPM Commands for descriptions of the ready, ready_on, and ready_off commands.)

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);
```

where ready_entry is the current ready procedure. (Output)

Entry: cu_\$set_ready_procedure

This entry point allows the user to change the ready procedure of his process.

Usage

```
declare cu_$set_ready_procedure entry (entry);  
call cu_$set_ready_procedure (ready_entry);
```

where ready_entry is the (external) procedure entry point that is to become the new ready procedure of the process. (Input)

Entry: cu_\$get_ready_mode

This entry point returns the value of the static ready mode.

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);
```

where:

1. 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
2. mode.mbz is reserved for future use and must be "0"b. (Output)

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);
```

where:

1. 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
2. mode.mbz is reserved for future use and must be "0"b. (Input)

Entry: cu_\$arg_list_ptr

It is sometimes desirable to design a PL/I procedure to accept a variable number of arguments of varying data types (e.g., the ioa_ subroutine described in the MPM Subroutines). 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 PL/I pointer to its caller's argument list.

Usage

```
declare cu_$arg_list_ptr entry (ptr);  
call cu_$arg_list_ptr (arg_list_ptr);
```

where arg_list_ptr is a pointer to the caller's argument list. (Output)

Note

A description of the argument list and its structure is found under "Argument List Format" in Section II of this document.

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_$arg_ptr_rel entry (fixed bin, ptr, fixed bin, fixed bin(35),  
ptr);  
call cu_$arg_ptr_rel (arg_no, arg_ptr, arg_len, code, arg_list_ptr);
```

where:

1. arg_no is an integer specifying the number of the desired argument. (Input)
2. arg_ptr is a pointer to the unaligned character-string argument specified by arg_no. (Output)
3. arg_len is the length (in characters) of the argument specified by arg_no. (Output)

4. code is a standard status code. (Output) It can be one of the following:
- 0 normal return
- error_table_\$noarg argument specified by arg_no does not exist (if error_table_\$noarg is returned, the values of arg_ptr and arg_len are undefined)
5. arg_list_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_\$af_arg_count

This entry point assumes it has been called by an active function. It returns to its caller the number of arguments passed to the caller by its caller, not including the active function return argument. If the caller has not been invoked as an active function, a 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).

Usage

```
declare cu_$af_arg_count entry (fixed bin, fixed bin(35));
call cu_$af_arg_count (nargs, code);
```

where:

1. nargs is the number of input arguments passed to the caller. (Output)
2. code is a standard status code. (Output) It can be one of the following:
 - 0 caller was called as an active function
 - error_table_\$nodescr no argument descriptors were passed to the caller or an incorrect argument list header was encountered
 - error_table_\$not_act_fnc the caller was not invoked as an active function

Note

This entry point and the two following entry points, cu_\$af_arg_ptr and cu_\$af_return_arg, have been provided so that active functions need not have knowledge of the mechanism for returning arguments.

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, except that it verifies that the caller was invoked as an active function, and does not allow the return argument to be accessed. That is, if the return argument happens to be the *i*th argument in the actual argument list, and the caller asks cu_\$af_arg_ptr for the *i*th argument, it returns the (i+1)st argument (if any). If the (i+1)st argument does not exist, a "no argument" standard status code is returned. In practice, the return argument is always the last one, but use of this entry and the following entry allows the active function to be independent of the position of the return argument in the argument list. (See "Note" under cu_\$af_arg_count above.)

Usage

```
declare cu_$af_arg_ptr entry (fixed bin, ptr, fixed bin, fixed bin(35));  
call cu_$af_arg_ptr (arg_no, arg_ptr, arg_len, code);
```

where:

1. arg_no is the number of the desired argument. (Input)
2. arg_ptr is a pointer to the unaligned character-string argument specified by arg_no. It is set to the null value if any error is encountered. (Output)
3. arg_len is the length (in characters) of the argument specified by arg_no. It is set to 0 if any error is encountered. (Output)
4. code is a standard status code and is the same as for cu_\$af_arg_count, except that error_table_\$noarg can also be returned, meaning that the input argument requested by arg_no was not present. (Output)

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 "Note" below. It is provided to permit writing of active functions that accept an arbitrary number of arguments. (See "Note" under cu_\$af_arg_count above.)

Usage

```
declare cu_$af_return_arg entry (fixed bin, ptr, fixed bin, fixed bin(35));
declare return_string char (max_length) varying based (rtn_string_ptr);
call cu_$af_return_arg (nargs, rtn_string_ptr, max_length, code);
```

where:

1. nargs is the same as in the cu_\$af_arg_count entry point above.
(Output)
2. rtn_string_ptr is a pointer to the varying string return argument of the active function. (Output)
3. max_length is the maximum length of the varying string pointed to by rtn_string_ptr. (Output)
4. code is the same as in the cu_\$af_arg_count entry point above.
(Output)

Note

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 its maximum length. It declares the based varying string, return_string, as described above. It then assigns its return value to return_string.

Entry: cu_\$stack_frame_ptr

The cu_\$stack_frame_ptr entry point returns a pointer to its caller's stack frame.

Usage

```
declare cu_$stack_frame_ptr entry (ptr);
call cu_$stack_frame_ptr (stack_ptr);
```

where stack_ptr is a pointer to the caller's stack frame. (Output)

Entry: cu_\$stack_frame_size

The cu_\$stack_frame_size entry point returns the size (in words) of its caller's stack frame.

Usage

```
declare cu_$stack_frame_size entry (fixed bin);
call cu_$stack_frame_size (size);
```

where size is the size (in words) of the caller's stack frame. (Output)

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);
```

where:

1. proc_entry is the (external) procedure entry point to be called. (Input)
2. a_ptr is a pointer to the argument list to be passed to the called procedure. (Input)

Entry: cu_\$set_command_processor

Some standard Multics commands (e.g., edm, described in the MPM Commands) permit the user to escape from them to execute other commands. In this case, the escapable command passes the execute request line to the command processor. To allow use of these escapable standard commands in a closed subsystem environment, instead of calling the command processor directly, the cu_\$cp (command processor) entry point in cu_ is called. (See the MPM Subroutines for a description of the cu_\$cp entry point.) 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 procedure of his own. This mechanism can be used to ensure that the subsystem remains in full control and still allows subsystem users the use of many standard commands.

Usage

```
declare cu_$set_command_processor entry (entry);  
call cu_$set_command_processor (proc_entry);
```

where proc_entry is the (external) procedure entry point to which control is passed upon receiving a call to cu_\$cp. (Input)

Entry: cu_\$get_command_processor

This entry point returns to the caller the entry value of the procedure currently being invoked in a call to cu_\$cp.

Usage

```
declare cu_$get_command_processor entry (entry);  
call cu_$get_command_processor (proc_entry);
```

where proc_entry is the procedure entry point to which control is passed upon receiving a call to cu_\$cp. (Output)

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 in this document). The standard error handlers attempt to print an understandable diagnostic and call a procedure to reenter command level. Reentering command level is done for a user through a call to get_to_cl_\$unclaimed_signal. However, in order to allow use of the standard error handling procedures in a closed subsystem environment, the error handlers do not call get_to_cl_\$unclaimed_signal directly but call the cu_\$cl (command level) entry point in cu_. This procedure 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 get_to_cl_\$unclaimed_signal on a call to cu_\$cl.

Usage

```
declare cu_$set_cl_intermediary entry (entry);  
call cu_$set_cl_intermediary (proc_entry);
```

where proc_entry is the (external) procedure entry to be called by the standard error handlers after printing a diagnostic message. (Input)

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. If no such procedure has been specified (the normal case), control is passed to get_to_cl_\$unclaimed_signal, which reenters command level.

Usage

```
declare cu_$cl entry;  
call cu_$cl ();
```

There are no arguments.

—
cu_
—

—
cu_
—

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);
```

where proc_entry is the procedure entry being called by the standard error handlers after printing a diagnostic message. (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);
```

where level specifies the new protection validation level and must be greater than or equal to the current ring number. (Input)

Entry: cu_\$level_get

The 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);
```

where level is the current validation level. (Output)

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);
```

where:

1. entry_value is the entry value to be decoded. (Input)
2. ep_ptr is the entry point pointer. (Output)
3. env_ptr is the environment pointer. (Output)

This page intentionally left blank.

cv_bin_

cv_bin_

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);
```

where:

1. n is the binary integer to be converted. (Input)
2. string is the ASCII equivalent of n. (Output)
3. 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);
```

where:

1. n is the binary integer to be converted. (Input)
2. string is the ASCII equivalent of n. (Output)

cv_bin_

cv_bin_

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);
```

where:

1. n is the binary integer to be converted. (Input)
2. string is the ASCII equivalent of n. (Output)

Note

If the character-string representation of the number exceeds 12 characters, then only the low-order 12 digits are returned.

Name: cv_entry_

The cv_entry_ subroutine 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);
```

where:

1. ventry is the virtual entry to be converted. (Input) See "Virtual Entries" below for more information.
2. referencing_ptr is a pointer to a segment in the referencing directory. (Input) 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.
3. code is a standard status code. (Output)
4. entry_value is the entry value that results from the conversion. (Output)

Virtual Entries

The cv_entry_ subroutine converts virtual entries that contain one or two components -- a segment identifier and an optional offset into the segment. Altogether, seven 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.

<u>Virtual Entry</u>	<u>Interpretation</u>
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 segment identified by path.
path	same as path [entry path].
ref_name\$entry_pt	entry at word identified by entry point entry_pt in segment 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.

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. Refer to the description of this entry point in the MPM Subroutines for more information.

The cv_entry_ subroutine returns an entry value that may be used in a call to cu_\$generate_call. The cu_\$decode_entry_value subroutine may be called if an entry pointer is required, rather than an entry variable. For pointers not used as entry pointers, use the cv_ptr_ subroutine to convert a virtual pointer.

This page intentionally left blank.

Name: cv_hex_

The cv_hex_ subroutine 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.

Usage

```
declare cv_hex_ entry (char(*)) returns (fixed bin(35));  
a = cv_hex_ (string);
```

where:

1. string is the string to be converted. It must be nonvarying. (Input)
2. a is the result of the conversion. (Output)

Entry: cv_hex_check_

This entry point differs from the cv_hex_ entry point only in that a code is returned indicating the possibility of a conversion error.

Usage

```
declare cv_hex_check_ entry (char(*), fixed bin(35));  
returns (fixed bin(35));  
a = cv_hex_check_ (string, code);
```

where:

1. string is the string to be converted. It must be nonvarying. (Input)
2. code is a status code that equals 0 if no error occurred; otherwise, it is the index of the character that terminated the conversion. (Output)
3. a is the result of the conversion. (Output)

cv_oct_

cv_oct_

Name: cv_oct_

The cv_oct_ subroutine takes an ASCII representation of an octal integer and returns the fixed binary(35) representation of that number. It can be called as shown because the segment has multiple names.

Usage

```
declare cv_oct_ entry (char(*)) returns (fixed bin(35));  
a = cv_oct_ (string);
```

where:

1. string is the string to be converted. (Input)
2. a is the result of the conversion. (Output)

Entry: cv_oct_check_

This entry point differs from the cv_oct_ entry point only in that a code is returned indicating the possibility of a conversion error. It can be called as shown because the segment has multiple names.

Usage

```
declare cv_oct_check_ entry (char(*), fixed bin(35)) returns (fixed  
bin(35));  
a = cv_oct_check_ (string, code);
```

where:

1. string is the string to be converted. It must be nonvarying. (Input)
2. code is a status code that equals 0 if no error occurred; otherwise it is the index of the character that terminated the conversion. (Output)
3. a is the result of the conversion. (Output)

Name: cv_ptr_

The cv_ptr_ subroutine 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);
```

where:

1. vptr is the virtual pointer to be converted. (Input) See "Virtual Pointers" below for more information.
2. code is a standard status code. (Output)
3. 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);
```

where ptr_value is the pointer returned by the previous call to cv_ptr_. (Input)

Notes

Pointers returned by the cv_ptr_ subroutine cannot be used as entry pointers in calls to cu_\$gen_call or cu_\$make_entry_value. The cv_ptr_ subroutine 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_ subroutine 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_ subroutine converts virtual pointers that contain one or two components -- a segment identifier and an optional offset into the segment. Altogether, fourteen 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. B is a decimal bit offset within the word. It may have a value from 0 to 35 inclusive.

<u>Virtual Pointer</u>	<u>Interpretation</u>
path W(B)	points to octal word W, decimal bit B of segment identified by absolute or relative pathname path.
path W	same as path W(0).
path	same as path 0(0).
path	same as path 0(0).
path entry_pt	points to word identified by entry point entry_pt in segment identified by path.
ref_name\$entry_pt	points to word identified by entry point entry_pt in segment whose reference name is ref_name.
ref_name\$W(B)	points to octal word W, decimal bit B of segment whose reference name is ref_name.
ref_name\$W	same as ref_name\$W(0).
ref_name\$	same as ref_name\$0(0).
segno W(B)	points to octal word W, decimal bit B of 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 word identified by entry point entry_pt in segment whose octal segment number is segno.

A null pointer is represented by the virtual pointer 77777|1, by -1|1, or by -1.

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 and FORTRAN. For a list of the type codes used, see "Argument List Format" in Section II of this manual.

Usage

```
declare decode_descriptor_ entry (ptr, fixed bin, fixed bin, bit(1)
    aligned, fixed bin, fixed bin, fixed bin);
call decode_descriptor_ (ptr, n, type, packed, ndims, size, scale);
```

where:

1. `ptr` points either directly at the descriptor to be decoded or at the argument list in which the descriptor appears. (Input)
2. `n` controls which descriptor is decoded. If `n` is 0, `ptr` points at the descriptor to be decoded; otherwise, `ptr` points at the argument list header and the `n`th descriptor is decoded. (Input)
3. `type` is the data type specified by the descriptor. Type codes appearing in an old form of descriptor are mapped into the new codes. (Output)
 - 0 is returned if an invalid type code is found in the old format descriptor
 - 1 is returned if descriptors are not present in the argument list or if the `n`th descriptor does not exist
4. `packed` describes how the data is stored. (Output)
 - new format descriptors
 - "1"b data is packed
 - "0"b data is not packed
 - old format descriptors
 - "1"b data is a string
 - "0"b data is not a string
5. `ndims` indicates either the number of dimensions of the descriptor array or whether the descriptor is an array or a scalar. (Output)
 - new format descriptor
 - `n` descriptor is an array of `n` dimensions
 - 0 descriptor is a scalar
 - old format descriptor
 - 1 descriptor is an array
 - 0 descriptor is a scalar

6. size . is the arithmetic precision, string size, or number of structure elements of the data of the new format descriptor. This value is 0 if an old form of descriptor specifies a structure. (Output)
7. scale is the scale of an arithmetic value for a new format descriptor. This value is 0 for an old form of descriptor. (Output)

define_area_

define_area_

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.

Usage

```
declare define_area_ entry (ptr, fixed bin (35));
call define_area_ (info_ptr, code);
```

where:

1. info_ptr points to the information structure described below.
(Input)
2. code is a system status code. (Output)

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:

```
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 area_ptr          ptr,
    2 allocated_blocks  fixed bin,
    2 free_blocks       fixed bin,
    2 allocated_words   fixed bin (30),
    2 free_words        fixed bin (30);
```

where:

1. version is to be filled in by the caller and should be 1.
2. control are control flags for enabling or disabling features of the area management mechanism.

3. extend indicates whether the area is extensible. This feature should only be used for perprocess, temporary areas.
"1"b yes
"0"b no
4. zero_on_alloc indicates whether blocks are cleared (set to all zeros) at allocation time.
"1"b yes
"0"b no
5. zero_on_free indicates whether blocks are cleared (set to all zeros) at free time.
"1"b yes
"0"b no
6. dont_free indicates whether the free requests are disabled thereby not allowing reuse of storage within the area.
"1"b yes
"0"b no
7. 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
8. system is used only by system code and indicates that the area is managed by the system.
"1"b yes
"0"b no
9. pad is not used and must be all zeros.
10. owner is the name of the program requesting that the area be defined. This is used for extensible areas only and is needed by the temporary segment manager.
11. n_components is the number of components in the area. (This item is not used by the define_area_ subroutine.)
12. size is the size, in words, of the area being defined.
13. version_of_area is 1 for current areas and 0 for old-style areas. (This item is not used by the define_area_ subroutine.)
14. area_ptr 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 area_ptr is set as a returned value. If area_ptr is initially nonnull, it must point to a 0 mod 2 address.
15. allocated_blocks is the number of allocated blocks in the entire area. (This item is not used by the define_area_ subroutine.)
16. 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.)

define_area_

define_area_

17. allocated_words is the number of allocated words in the entire area. (This item is not used by the define_area_ subroutine.)
18. free_words is the number of free words in the entire area. (This item is not used by the define_area_ subroutine.)

The above structure is defined by the system include file, area_info.incl.pl1.

Note

See the release_area_ subroutine for a description of how to free up segments acquired via this interface.

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. For more information, see the description of the dial command in the MPM Commands.

Entry: dial_manager_\$allow_dials

This entry point requests that the answering service allow terminals to dial to the calling process. The caller must set dial_manager_arg.dial_qualifier to an alphanumeric string from 1 to 22 characters in length. This string is used by the dial command to differentiate multiple processes with the same Person_id and Project_id. The caller must also set dial_manager_arg.dial_channel to an event-wait channel in the caller's process. The answering service sends notices of dial connections and hangups over this channel. After the dial_manager_\$allow_dials entry point has been called, the event channel may be changed to an event-call channel. The user program receiving the wakeup should call the convert_dial_message_ subroutine to decode the event message.

Usage

```
declare dial_manager_$allow_dials entry (ptr, fixed bin(35));
call dial_manager_$allow_dials (request_ptr, code);
```

where:

1. request_ptr is a pointer to the dial_manager_arg structure described in "Notes" below. (Input)
2. 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 telephone number and, if the channel is successfully dialed, that the channel be assigned to the requesting process. The caller must set dial_manager_arg.dial_qualifier to the telephone number to be dialed. Nonnumeric characters in the telephone number are ignored. 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, or he may set it to null, in which case the answering service chooses a channel.

Usage

```
declare dial_manager_$dial_out entry (ptr, fixed bin(35));
call dial_manager_$dial_out (request_ptr, code);
```

where:

1. request_ptr is a pointer to the dial_manager_arg structure described in "Notes" below. (Input)
2. 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 entry point.

Usage

```
declare dial_manager_$shutoff_dials (ptr, fixed bin(35));
call dial_manager_$shutoff_dials (request_ptr, code);
```

where the arguments are the same as for the dial_manager_\$allow_dials entry point.

Entry: dial_manager_\$privileged_attach

This entry point allows a privileged process to attach any terminal that is in the channel master file, and is not already in use. 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. This must be the same name as specified by the channel master file.

Usage

```
declare dial_manager_$privileged_attach entry (ptr, fixed bin(35));
call dial_manager_$privileged_attach (request_ptr, code);
```

where the arguments are the same as for the dial_manager_\$allow_dials entry point.

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 same information should be passed to this entry point as to the dial_manager_\$dial_out entry point. The caller must set dial_manager_arg.channel_name to the name of the channel being used; channel_name cannot be null.

Usage

```
declare dial_manager_$terminate_dial_out entry (ptr, fixed bin(35));
call dial_manager_$terminate_dial_out (request_ptr, code);
```

where:

1. request_ptr is a pointer to the dial_manager_arg structure described in "Notes" below.
2. code is a standard status code. (Output)

Notes

The first argument in all of the calls (request_ptr) is a pointer to the dial_manager_arg structure. This structure is used to pass a variety of information to the dial_manager_ subroutine. It has the following declaration:

```
dcl 1 dial_manager_arg aligned,
    2 version          fixed bin initial (1),
    2 dial_qualifier   char(22),
    2 dial_channel     fixed bin(71),
    2 channel_name     char(32);
```

where:

1. version indicates the version of the structure that is being used. Currently this must be 1.
2. dial_qualifier is the telephone number to be called for calls to the dial_manager_\$dial_out entry point. Notice that nonnumeric characters are ignored, so the user need not remove them from a telephone number string. This is the dial qualifier for calls to the dial_manager_\$allow_dials entry point.

3. dial_channel is an interprocess communication channel used to receive messages from the answering service. Notice that the channel should be the same for all calls used in the same session.
4. channel_name is used for calls in the dial_manager_\$terminate_dial_out subroutine to indicate which channel should be disconnected. In calls to the dial_manager_\$dial_out subroutine, it must be either a null string (in which case the answering service attempts to assign any available auto call channel) or a specific channel to be used for the auto call attempt.

This page intentionally left blank.

Name: dprint_

The dprint_ subroutine is the interface for the dprint and dpunch commands (described in the MPM Commands). It causes a request to print or punch a segment to be added to the specified queue.

Usage

```
declare dprint_ entry (char(*), char(*), ptr, fixed bin(35));
call dprint_ (dir_name, entryname, arg_ptr, code);
```

where:

1. dir_name is the pathname of the containing directory. (Input)
2. entryname is the entryname of the segment to be printed or punched. It can also be the name of a multisegment file or a link that points to a segment or multisegment file. (Input)
3. arg_ptr is a pointer to the argument structure described in "Notes" below. If no argument structure is supplied, arg_ptr should be null. (Input)
4. code is a status code. (Output)

Notes

The dprint_ subroutine uses the structure described below 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 mbz1                   bit(30) unaligned,
  2 mbz2(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);
```

where:

1. version is the version number of the structure. The current version number is 4.
2. copies is the number of copies requested. (The default is 1.)
3. delete indicates whether the segment is to be deleted after printing or punching.
1 deletes the segment
0 does not delete the segment (default)
4. queue is the priority queue in which the request is placed. (The default is 3.)
5. pt_pch indicates whether the request is for printing or punching.
1 print request (default)
2 punch request
6. notify indicates whether the requestor is to be notified when the request is completed. This feature is not implemented at present.
1 notifies the requestor
0 does not notify the requestor (default)
7. 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. (The default is the null string.)
8. output_module indicates the I/O module to be used in executing the request.
1 indicates printing (default)
2 indicates 7-punching
3 indicates Multics card code (mcc) punching
4 indicates "raw" punching
9. dest 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. (The default is the null string.)
10. nep indicates whether no-endpage mode is used.
"1"b yes
"0"b no (default)
11. 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 (default)

12. 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 (default)
13. truncate indicates whether truncate mode is used.
"1"b yes
"0"b no (default)
14. center_top_label indicates whether the top label should be centered.
"1"b yes
"0"b no (default)
15. center_bottom_label indicates whether the bottom label should be centered.
"1"b yes
"0"b no (default)
16. mbz1 is not used and should be set to (30)"0"b.
17. mbz2 is not used and should be set to 0's.
18. forms is not used.
19. lmargin indicates the left margin position. (The default is 0.)
20. line_lth indicates the line length. (The default is -1, which implies maximum line length.)
21. class indicates the request type (formerly called device class). (The default is "printer" if pt_pch is 1 and "punch" if pt_pch is 2.)
22. page_lth indicates the page length, i.e., the number of lines per logical page. (The default is -1, which implies the physical page length.)
23. top_label is a label to be placed at the top of every page. (The default is the null string.)
24. bottom_label is a label to be placed at the bottom of every page. (The default is the null string.)

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 $00 \leq \text{hex_value} \leq \text{FF}$ (octal value in the range $000 \leq \text{oct_value} \leq 377$).

Entry: ebcdic_to_ascii_

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);
```

where:

1. ebcdic_in is the string of EBCDIC characters to be converted. (Input)
2. ascii_out is the ASCII equivalent of the input string. (Output)

Entry: ebcdic_to_ascii_\$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_$table char(256) external static;
```

ebcdic_to_ascii_

ebcdic_to_ascii_

Note

EBCDIC to ASCII conversion cannot be performed by the PL/I translate builtin because the EBCDIC string would be scanned under a 7-bit mask. 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: find_condition_info_

The find_condition_info_ 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, cond_info_ptr, code);
```

where:

1. stack_ptr is a pointer to a stack frame being used when a condition occurred. It is normally the result of a call to locate the condition frame; if null, the most recent condition frame is used. (Input)
2. cond_info_ptr is a pointer to the following structure in which information is returned. (Input)

```
dcl 1 cond_info          aligned,  
    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            aligned,  
    3 crawlout        bit(1) unaligned,  
    3 mbz1             bit(35) unaligned,  
    2 mbz2             bit(36) aligned,  
    2 user_loc_ptr     ptr,  
    2 mbz(4)           bit(36) aligned;
```

where:

mc_ptr if not null, points to the machine conditions. Machine conditions are described under "Multics Condition Mechanism" in Section VI of the MPM Reference Guide.

version is the version number of this structure (currently this number is 1).

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 "List of System Conditions and Default Handlers" in Section VI of the MPM Reference Guide.

`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

`mbz1` is currently unused and should be set to "0"b.

`mbz2` 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.

`mbz` is currently unused and should be set to "0"b.

3. `code` is a standard system 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)

get_default_wdir_

get_default_wdir_

Name: get_default_wdir_

The get_default_wdir_ subroutine returns the pathname of the user's current default working directory.

Usage

```
declare get_default_wdir_ entry returns (char(168) aligned);  
default_wdir = get_default_wdir_ ();
```

where 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);
```

where:

1. 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)
2. segname is the name of the object segment. (Input)
3. entryname is the name of the desired entry point. (Input)
4. def_ptr is a pointer to the definition for the entry point. (Output)
5. code is a standard status code. If the entry point is found, code is 0. (Output)

This page intentionally left blank.

get_entry_name_

get_entry_name_

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);
```

where:

1. entry_ptr is a pointer to a procedure entry point. (Input)
2. symbolname is the name corresponding to the location specified by entry_ptr. The maximum length is 256 characters. (Output)
3. 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)
4. lang is the language in which the segment or component pointed to by entry_ptr was compiled. (Output)
5. code is a standard status code. (Output)

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 Section I of MPM Commands 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);
```

where:

1. entryname is the entryname from which the target is to be constructed. Trailing blanks in the entryname character string are ignored. (Input)
2. equal_name is the equal name from which the target is to be constructed. Trailing blanks in the equal name character string are ignored. (Input)
3. target_name is the target name that is constructed. (Output)
4. code is a standard status code. (Output) It can be one of the following:

0	the target name was constructed properly
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_\$longeq1	the target name to be constructed is longer than 32 characters

get_equal_name_

get_equal_name_

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.

get_privileges_

get_privileges_

Name: get_privileges_

The get_privileges_ subroutine returns the access privileges of the process. (See "Access Control" in Section III of the MPM Reference Guide for more information on access privileges.)

Usage

```
declare get_privileges_ entry returns (bit(36) aligned);  
privilege_string = get_privileges_ ();
```

where 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,  
     2 dir,  
     2 seg,  
     2 soos,  
     2 ring1) bit(1),  
     2 mbz    bit(31);
```

where:

1. 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
2. dir indicates whether the AIM restrictions for accessing any directory are bypassed for the calling process.
"1"b yes
"0"b no
3. seg indicates whether the AIM restrictions for accessing any segment are bypassed for the calling process.
"1"b yes
"0"b no
4. 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

5. 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
6. mbz is unused and must be "0"b.

get_ring_

get_ring_

Name: get_ring_

The get_ring_ subroutine 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--Rings" in Section III of the MPM Reference Guide.

Usage

```
declare get_ring_ entry returns (fixed bin(3));  
ring_no = get_ring_ ();
```

where ring_no is the number of the ring in which the caller is executing.
(Output)

get_system_free_area_

get_system_free_area_

Name: get_system_free_area_

The get_system_free_area_ subroutine returns a pointer to the system free area for the ring in which it was called (namely system_free_k_ where k is the current ring). Allocations by system programs are performed in this area.

Usage

```
declare get_system_free_area_ entry returns (ptr);  
area_ptr = get_system_free_area_ ();
```

where area_ptr points to the system free area. (Output)

Name: hcs_\$add_dir_inacl_entries

The hcs_\$add_dir_inacl_entries 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);
```

where:

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

Notes

The following structure is used for dir_acl:

```
dcl 1 dir_acl (acl_count) aligned based (acl_ptr),  
    2 access_name char(32),  
    2 dir_modes bit(36),  
    2 status_code fixed bin(35);
```

where:

1. access_name is the access name (in the form Person_id.Project_id.tag) that identifies the processes to which this initial ACL entry applies.
2. dir_modes contains the directory modes for this access name. The first three bits correspond to the modes status, modify, and append. The remaining bits must be 0's. For example, status permission is expressed as "100"b.

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

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

The hcs_\$add_inacl_entries 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);
```

where:

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

Notes

The following structure is used for segment_acl:

```
dcl 1 segment_acl (acl_count)      aligned based (acl_ptr),
    2 access_name                  char(32),
    2 modes                        bit(36),
    2 zero_pad                     bit(36),
    2 status_code                  fixed bin(35);
```

where:

1. access_name is the access name (in the form Person_id.Project_id.tag) that identifies the processes to which this initial ACL entry applies.
2. modes contains the modes for this access name. The first three bits correspond to the modes read, execute, and write. The remaining bits must be 0's. For example, rw access is expressed as "101"b.

3. zero_pad must contain the value zero. (This field is for use with extended access and may only be used by the system.)
4. status_code is a storage system status code for this initial ACL entry only.

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_\$del_dir_tree

The hcs_\$del_dir_tree 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 in the MPM Subroutines.

Usage

```
declare hcs_$del_dir_tree entry (char(*), char(*), fixed bin(35));  
call hcs_$del_dir_tree (dir_name, entryname, code);
```

where:

1. dir_name is the pathname of the containing directory. (Input)
2. entryname is the entryname of the directory. (Input)
3. code is a storage system status code. (Output)

Notes

The user must have status and modify permission on the subdirectory and the safety switch must be off in that directory. 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--Rings" in Section III of the MPM Reference Guide.

Name: `hcs_$delete_dir_inacl_entries`

The `hcs_$delete_dir_inacl_entries` entry point is used to delete specified entries from an initial access control list (initial ACL) for new directories created for the specified ring within the specified directory. The `delete_acl` structure used by this subroutine is described in the `hcs_$delete_inacl_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);
```

where:

1. `dir_name` is the pathname of the containing directory. (Input)
2. `entryname` is the entryname of the directory. (Input)
3. `acl_ptr` points to the user-filled `delete_acl` structure as described in the `hcs_$delete_inacl_entries` entry point. (Input)
4. `acl_count` is the number of initial ACL entries in the `delete_acl` structure. (Input)
5. `ring` is the ring number of the initial ACL. (Input)
6. `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` 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` structure cannot be matched to one existing on the initial ACL, then the `status_code` of that initial ACL entry in the `delete_acl` structure is set to `error_table_$user_not_found`. Processing continues to the end of the `delete_acl` structure and `code` is returned as 0.

Name: hcs_\$delete_inacl_entries

The hcs_\$delete_inacl_entries 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.

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);
```

where:

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

Notes

The following is the delete_acl structure:

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

where:

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

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

hcs_\$delete_inacl_entries

hcs_\$delete_inacl_entries

If an access_name in the delete_acl structure cannot be matched to one existing on the initial ACL, then the status_code of that initial ACL entry in the delete_acl structure is set to error_table_\$user_not_found. Processing continues to the end of the delete_acl structure and code is returned as 0.

Name: hcs_\$get_author

The hcs_\$get_author 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);
```

where:

1. dir_name is the pathname of the containing directory. The pathname can have a maximum length of 168 characters. (Input)
2. entryname is the entryname of the segment, directory, multisegment file, or link. It can have a maximum length of 32 characters. (Input)
3. 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
4. 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. An error is not detected if the string, author, is too short to hold the author. (Output)
5. code is a storage system status code. (Output)

Note

The user must have status permission on the containing directory.

hcs_\$get_bc_author

hcs_\$get_bc_author

Name: hcs_\$get_bc_author

The hcs_\$get_bc_author 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);
```

where:

1. dir_name is the pathname of the containing directory. (Input)
2. entryname is the entryname of the segment or directory. (Input)
3. bc_author is the bit count author of the segment or directory in the form of Person_id.Project_id.tag. An error is not detected if the string, bc_author, is too short to hold the bit count author. (Output)
4. code is a storage system status code. (Output)

Note

The user must have status permission on the containing directory.

hcs_\$get_dir_ring_brackets

hcs_\$get_dir_ring_brackets

Name: hcs_\$get_dir_ring_brackets

The hcs_\$get_dir_ring_brackets entry point, 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);
```

where:

1. dir_name is the pathname of the containing directory. (Input)
2. entryname is the entryname of the subdirectory. (Input)
3. drb is a two-element array that contains the directory's ring brackets. The first element contains the level required for modify and append permission; the second element contains the level required for status permission. (Output)
4. code is a storage system status code. (Output)

Notes

The user must have status permission on the containing directory.

Ring brackets are discussed in "Intraprocess Access Control--Rings" in Section III of the MPM Reference Guide.

Name: hcs_\$get_max_length

The hcs_\$get_max_length 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);
```

where:

1. dir_name is the pathname of the containing directory. (Input)
2. entryname is the entryname of the segment. (Input)
3. max_length is the maximum length of the segment in words. (Output)
4. code is a storage system status code. (Output)

Note

The user must have status permission on the directory containing the segment or nonnull access to the segment.

Name: hcs_\$get_max_length_seg

The hcs_\$get_max_length_seg 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);
```

where:

1. seg_ptr is a pointer to the segment whose maximum length is to be returned. (Input)
2. max_length is the maximum length of the segment in words. (Output)
3. code is a storage system status code. (Output)

Note

The user must have status permission on the directory containing the segment or nonnull access to the segment.

Name: `hcs_$get_ring_brackets`

The `hcs_$get_ring_brackets` 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);
```

where:

1. `dir_name` is the pathname of the containing directory. (Input)
2. `entryname` is the entryname of the segment. (Input)
3. `rb` is a three-element array that contains the segment's ring brackets. Ring brackets and validation levels are discussed in "Intraprocess Access Control--Rings" in Section III of the MPM Reference Guide. (Output)
4. `code` is a storage system status code. (Output)

Note

The user must have status permission on the containing directory.

hcs_\$get_safety_sw

hcs_\$get_safety_sw

Name: hcs_\$get_safety_sw

The hcs_\$get_safety_sw 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);
```

where:

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

Note

The user must have status permission on the containing directory or must have nonnull access to the segment.

hcs_\$get_safety_sw_seg

hcs_\$get_safety_sw_seg

Name: hcs_\$get_safety_sw_seg

The hcs_\$get_safety_sw_seg 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);
```

where:

1. seg_ptr is a pointer to the segment whose safety switch is to be examined. (Input)
2. 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
3. code is a storage system status code. (Output)

Note

The user must have status permission on the directory containing the segment or must have nonnull access to the segment.

hcs_\$get_search_rules

hcs_\$get_search_rules

Name: hcs_\$get_search_rules

The hcs_\$get_search_rules 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);
```

where search_rules_ptr is a pointer to a user-supplied search rules structure. See "Note" below. (Input)

Note

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;
```

where:

1. number is the number of search rules in the array.
2. 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.)

hcs_\$get_system_search_rules

hcs_\$get_system_search_rules

Name: hcs_\$get_system_search_rules

The hcs_\$get_system_search_rules 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);
```

where:

1. search_rules_ptr is a pointer to the structure described below. (Input)
2. 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 (10),
      3 name          char(168),
      3 flag          bit(36);
```

where:

1. ntags is the number of tags.
2. nrules is the number of rules.
3. tags is an array of keywords.
4. tags.name is the keyword.
5. tags.flag is a bit field with one bit on.
6. rules is an array of directory names.
7. rules.name is the directory absolute path.
8. rules.flag is a bit field with bits on for every tag that selects this directory.

This page intentionally left blank.

Name: hcs_\$initiate_search_rules

The hcs_\$initiate_search_rules entry point provides the user with a subroutine interface for specifying the search rules that he wants to use in his process. (For a description of the set_search_rules command, see the MPM Commands.)

Usage

```
declare hcs_$initiate_search_rules entry (ptr, fixed bin(35));
call hcs_$initiate_search_rules (search_rules_ptr, code);
```

where:

1. search_rules_ptr is a pointer to a structure containing the new search rules. See "Notes" below. (Input)
2. code is a storage system status code. (Output)

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;
```

where:

1. number is the number of search rules contained in the array. The current maximum number of search rules the user can define is 21.
2. names are the names of the search rules. They can be absolute pathnames of directories or keywords.

Two types of search rules are permitted: absolute pathnames of directories to be searched or keywords. The keywords are:

1. initiated_segments search for the already initiated segments.
2. referencing_dir search the containing directory of the segment making the reference.
3. working_dir search the working directory.
4. process_dir search the process directory.
5. home_dir search the home directory.

6. `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.
7. `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.

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 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_dir_inacl`

The `hcs_$list_dir_inacl` entry point is used either to list the entire initial access control list (initial ACL) for new directories created for the specified ring within the specified directory or to return the access modes for specified initial ACL entries. The `dir_acl` structure described in the `hcs_$add_dir_inacl_entries` entry point is used by this entry point.

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);
```

where:

1. `dir_name` is the pathname of the containing directory. (Input)
2. `entryname` is the entryname of the directory. (Input)
3. `area_ptr` points to an area into which the list of initial ACL entries, which makes up the entire initial ACL of the directory, is allocated. 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). (Input)
4. `area_ret_ptr` points to the start of the allocated list of initial ACL entries. (Output)
5. `acl_ptr` if `area_ptr` is null, then `acl_ptr` points to an initial ACL structure, `dir_acl`, into which mode information is placed for the access names specified in that same structure. (Input)
6. `acl_count` is the number of entries in the ACL structure. (Input or Output)
Input is the number of entries in the initial ACL structure identified by `acl_ptr`
Output is the number of entries in the `dir_acl` structure allocated in the area pointed to by `area_ptr`, if `area_ptr` is not null
7. `ring` is the ring number of the initial ACL. (Input)
8. `code` is a storage system status code. (Output)

Note

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

Name: `hcs_$list_inacl`

The `hcs_$list_inacl` entry point is used either to list the entire initial access control list (initial ACL) for new segments created for the specified ring within the specified directory or to return the access modes for specified initial ACL entries. The `segment_acl` structure used by this entry point is described in the `hcs_$add_inacl_entries` entry point.

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);
```

where:

1. `dir_name` is the pathname of the containing directory. (Input)
2. `entryname` is the entryname of the directory. (Input)
3. `area_ptr` points to an area into which the list of initial ACL entries, which makes up the entire initial ACL of the directory, is allocated. 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). (Input)
4. `area_ret_ptr` points to the start of the allocated list of initial ACL entries. (Output)
5. `acl_ptr` if `area_ptr` is null, then `acl_ptr` points to an initial ACL structure, `segment_acl`, into which mode information is to be placed for the access names specified in that same structure. (Input)
6. `acl_count` is the number of entries in the 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` structure allocated in the area pointed to by `area_ptr`, if `area_ptr` is not null
7. `ring` is the ring number of the initial ACL. (Input)
8. `code` is a storage system status code. (Output)

hcs_\$list_inacl

hcs_\$list_inacl

Note

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` 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.

hcs_\$quota_move

hcs_\$quota_move

Name: hcs_\$quota_move

The hcs_\$quota_move 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);
```

where:

1. dir_name is the pathname of the containing directory. (Input)
2. entryname is the entryname of the directory. (Input)
3. quota_change is the number of records of secondary storage quota to be moved between the superior directory and the inferior directory. (See "Notes" below.) (Input)
4. 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.

hcs_\$quota_read

hcs_\$quota_read

Name: hcs_\$quota_read

The hcs_\$quota_read 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);
```

where:

1. dir_name is the pathname of the directory for which quota information is desired. (Input)
2. quota is the segment record quota in the directory. (Output)
3. trp is the time-record product (trp) charged to the directory. This double-precision number is in units of record-seconds. (Output)
4. 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, described in the MPM Subroutines), that the trp was last updated. (Output)
5. sons_lvid is the logical volume ID for segments contained in this directory. (Output)
6. tacc_sw is the terminal account switch. The setting of this switch determines how charges are made. (Output)
"1"b records are charged against the quota in this directory
"0"b records are charged against the quota in the first superior directory with a terminal account
7. used is the number of records used by segments in this directory and by segments in nonterminal inferior directories. (Output)
8. code is a storage system status code. (Output)

Note

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_\$replace_dir_inacl

The hcs_\$replace_dir_inacl 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 structure described in the hcs_\$add_dir_inacl_entries entry point is used by this entry point.

Usage

```
declare hcs_$replace_dir_inacl entry (char(*), char(*), ptr, fixed bin,
    bit(1) aligned, fixed bin(3), fixed bin(35));
```

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

where:

1. dir_name is the pathname of the containing directory. (Input)
2. entryname is the entryname of the directory. (Input)
3. acl_ptr points to a user-supplied dir_acl structure that is to replace the current initial ACL. (Input)
4. acl_count contains the number of entries in the dir_acl structure. (Input)
5. no_sysdaemon_sw is a switch that indicates whether the sma *.SysDaemon.* entry is put on the initial ACL after the existing initial ACL is deleted and before the user-supplied dir_acl entries are added. (Input)
"0"b adds sma *.SysDaemon.* entry
"1"b replaces the existing initial ACL with only the user-supplied dir_acl
6. ring is the ring number of the initial ACL. (Input)
7. code is a storage system status code. (Output)

Note

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 entries is performed top to bottom, allowing later entries to overwrite previous ones if the access_name in the dir_acl structure is identical.

Name: hcs_\$replace_inacl

The hcs_\$replace_inacl 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 structure described in the hcs_\$add_inacl_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);
```

where:

1. dir_name is the pathname of the containing directory. (Input)
2. entryname is the entryname of the directory. (Input)
3. acl_ptr points to the user-supplied segment_acl structure that is to replace the current initial ACL. (Input)
4. acl_count contains the number of entries in the segment_acl structure. (Input)
5. 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 entries are added. (Input)
"0"b adds rw *.SysDaemon.* entry
"1"b replaces the existing initial ACL with only the user-supplied segment_acl
6. ring is the ring number of the initial ACL. (Input)
7. code is a storage system status code. (Output)

Note

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 entries is performed top to bottom, allowing later entries to overwrite previous ones if the access_name in the segment_acl structure is identical.

Name: hcs_\$set_dir_ring_brackets

The hcs_\$set_dir_ring_brackets 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);
```

where:

1. dir_name is the pathname of the containing directory. (Input)
2. entryname is the entryname of the subdirectory. (Input)
3. drb is a two-element array specifying the ring brackets of the directory. The first element contains the level required for modify and append permission; the second element contains the level required for status permission. (Input)
4. 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--Rings" in the MPM Reference Guide.

hcs_\$set_entry_bound

hcs_\$set_entry_bound

Name: hcs_\$set_entry_bound

The hcs_\$set_entry_bound 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);
```

where:

1. dir_name is the pathname of the containing directory. (Input)
2. entryname is the entryname of the segment. (Input)
3. entry_bound is the new value in words for the entry point bound of the segment. If the value of entry_bound is 0, then the mechanism is disabled. (Input)
4. code is a storage system status code. (See "Notes" below.) (Output)

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

The hcs_\$set_entry_bound_seg 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);
```

where:

1. seg_ptr is a pointer to the segment whose entry point bound is to be changed. (Input)
2. entry_bound is the new value in words for the entry point bound of the segment. If the value of entry_bound is 0, then the mechanism is disabled. (Input)
3. code is a storage system status code. (See "Notes" below.) (Output)

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.

hcs_\$set_max_length

hcs_\$set_max_length

Name: hcs_\$set_max_length

The hcs_\$set_max_length entry point, given a directory name, 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);
```

where:

1. dir_name is the pathname of the containing directory. (Input)
2. entryname is the entryname of the segment. (Input)
3. max_length is the new value in words for the maximum length of the segment. (Input)
4. code is a storage system status code. (See "Notes" below.) (Output)

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 16 words, and if max_length is not a multiple of 16 words, it is set to the next multiple of 16 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 MPM Reference Guide.

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

The hcs_\$set_max_length_seg 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);
```

where:

1. seg_ptr is the pointer to the segment whose maximum length is to be changed. (Input)
2. max_length is the new value in words for the maximum length of the segment. (Input)
3. code is a storage system status code. (See "Notes" below.) (Output)

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 16 words, and if max_length is not a multiple of 16 words, it is set to the next multiple of 16 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 MPM Reference Guide.

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

The hcs_\$set_ring_brackets 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);
```

where:

1. dir_name is the pathname of the containing directory. (Input)
2. entryname is the entryname of the segment. (Input)
3. rb is a three-element array specifying the ring brackets of the segment; see "Notes" below. (Input)
4. 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--Rings" in Section III of the MPM Reference Guide.

Name: `hcs_$set_safety_sw`

The `hcs_$set_safety_sw` 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 Section III of the MPM Reference Guide 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);
```

where:

1. `dir_name` is the pathname of the containing directory. (Input)
2. `entryname` is the entryname of the segment or directory. (Input)
3. `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
4. `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.

hcs_\$set_safety_sw_seg

hcs_\$set_safety_sw_seg

Name: hcs_\$set_safety_sw_seg

The hcs_\$set_safety_sw_seg entry point, given a pointer to a segment, sets the safety switch of the segment. See "Segment, Directory, and Link Attributes" in Section III of the MPM Reference Guide 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);
```

where:

1. seg_ptr is the pointer to the segment. (Input)
2. 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
3. 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_

The hcs_\$star_ entry point is the star convention handler for the storage system. (See "Constructing and Interpreting Names" in Section III of MPM Commands.) 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.

Status permission is required on the directory to be searched.

The main entry point returns the storage system type and all names that match the given entryname. (See the hcs_\$star_dir_list_ and hcs_\$star_list_ entry points below to obtain 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.)

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, select_sw, area_ptr, entry_count,
    entry_ptr, n_ptr, code);
```

where:

1. dir_name is the pathname of the containing directory. (Input)
2. star_name is the entryname that can contain asterisks or question marks. (Input)
3. select_sw indicates what information is to be returned. (Input) It can be:
 - 1 information is returned about link entries only
 - 2 information is returned about segment and directory entries only
 - 3 information is returned about segment, directory, and link entries
4. area_ptr is a pointer to the area in which information is to be returned. If the pointer is null, entry_count is set to the total number of selected entries. See "Notes" below. (Input)
5. entry_count is a count of the number of entries that match the entryname. (Output)

hcs_\$star_

hcs_\$star_

6. entry_ptr is a pointer to the allocated structure in which information on each entry is returned. (Output)
7. n_ptr is a pointer to the allocated array of all the entrynames in this directory that match star_name. See "Notes" below. (Output)
8. code is a storage system status code. See "Status Codes" below. (Output)

Notes

Even if area_ptr is null, entry_count is set to the total number of entries in the directory that match star_name. The setting of select_sw determines whether 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.

The entry information structure is as follows:

```
dcl 1 entries (ecount)          aligned based (entry_ptr),
    2 type                      bit(2),
    2 nnames                     fixed bin(15),
    2 nindex                     fixed bin(17)) unaligned;
```

where:

1. type specifies the storage system type of entry:
0 ("00"b) link
1 ("01"b) segment
2 ("10"b) directory
2. nnames specifies the number of names for this entry that match star_name.
3. nindex specifies the offset in the array of names (pointed to by n_ptr) for the first name returned for this entry.

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, is as follows:

```
declare names (total_names) char(32) aligned based (n_ptr);
```

where total_names is the total number of names returned.

The user must provide an area large enough for the hcs_\$star_ entry point to store the requested information.

Entry: 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(3), ptr,
    fixed bin, fixed bin, ptr, ptr, fixed bin(35));

call hcs_$star_dir_list_ (dir_name, star_name, select_sw, area_ptr,
    branch_count, link_count, entry_ptr, n_ptr, code);
```

where the arguments are exactly the same as those for the hcs_\$star_list_ entry point below.

Notes

The notes for hcs_\$star_list_ also apply to this entry.

The layouts of these structures are identical to those used by hcs_\$star_list_. Only the meanings of two elements differ: dtem and bit_count.

```
dcl 1 branches (count)          aligned based (entry_ptr),
    (2 type                      bit(2),
     2 nnames                    fixed bin(15),
     2 nindex                   fixed bin(17),
     2 dtem                      bit(36),
     2 pad1                     bit(36),
     2 mode                     bit(5),
     2 raw_mode                 bit(5),
     2 master_dir              bit(1),
     2 bit_count               fixed bin(24)) unaligned;
```

where:

1. type specifies the storage system type of entry:
0 ("00"b) link
1 ("01"b) segment
2 ("10"b) directory
2. nnames specifies the number of names for this entry that match star_name.

3. nindex specifies the offset in the array of names (pointed to by n_ptr) for the first name returned for this entry.
4. dtem is the date and time the directory entry for the segment or directory was last modified.
5. pad1 is unused space in this structure.
6. mode is the current user's access mode to the segment or directory. See "Access Modes" below.
7. raw_mode is the current user's access mode before ring brackets and access isolation are considered.
8. master_dir specifies whether entry is a master directory:
"1"b yes
"0"b no
9. bit_count is the bit count of the segment or directory.

The structure used if the entry is a link is identical to the one used by hcs_\$star_list_ and identical information is returned by both entries for links.

Entry: 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_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, select_sw, area_ptr,  
    branch_count, link_count, entry_ptr, n_ptr, code);
```

where:

1. dir_name is the pathname of the containing directory. (Input)
2. star_name is the entryname that can contain asterisks or question marks. (Input)

3. `select_sw` indicates what information is to be returned. (Input) It can be:
- 1 information is returned about link entries only
 - 2 information is returned about segment and directory entries only
 - 3 information is returned about segment, directory, and link entries
 - 5 information is returned about link entries only, including the pathname associated with each link entry
 - 7 information is returned about segment, directory, and link entries, including the pathname associated with each link entry
4. `area_ptr` is a pointer to the area in which information is to be returned. If the pointer is null, `branch_count` and `link_count` are set to the total number of selected entries. See "Notes" below. (Input)
5. `branch_count` is a count of the number of segments and directories that match the entryname. (Output)
6. `link_count` is a count of the number of links that match the entryname. (Output)
7. `entry_ptr` is a pointer to the allocated structure in which information on each entry is returned. (Output)
8. `n_ptr` is a pointer to the allocated array in which selected entrynames and pathnames associated with link entries are stored. (Output)
9. `code` is a storage system status code. See "Status Codes" below. (Output)

Notes

Even if `area_ptr` is null, `branch_count` and `link_count` may be set. If information on segments and directories is requested, `branch_count` is set to the total number of segments and directories that match `star_name`. If information on links is requested, `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. The number of structures allocated is `count`, which is equal to `branch_count` plus `link_count`. Each element in the structure array may be either of the structures described below (the links structure for links or the branches 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 is used if the entry is a segment or a directory:

```
dcl 1 branches (count)          aligned based (entry_ptr),
  (2 type                       bit(2),
   2 nnames                     fixed bin(15),
   2 nindex                     fixed bin(17),
   2 dtcm                       bit(36),
   2 dtu                        bit(36),
   2 mode                       bit(5),
   2 raw_mode                   bit(5),
   2 master_dir                 bit(1),
   2 records                    fixed bin(24)) unaligned;
```

where:

1. `type` specifies the storage system type of entry:
0 ("00"b) link
1 ("01"b) segment
2 ("10"b) directory
2. `nnames` specifies the number of names for this entry that match `star_name`.
3. `nindex` specifies the offset in the array of names (pointed to by `n_ptr`) for the first name returned for this entry.
4. `dtcm` is the date and time the contents of the segment or directory were last modified.
5. `dtu` is the date and time the segment or directory was last used.
6. `mode` is the current user's access mode to the segment or directory.
7. `raw_mode` is the current user's access mode before ring brackets and access isolation are considered.
8. `master_dir` specifies whether entry is a master directory:
"1"b yes
"0"b no
9. `records` is the number of 1024-word records of secondary storage that have been assigned to the segment or directory.

The following structure is used if the entry is a link:

```
dcl 1 links (count)          aligned based (entry_ptr),
  (2 type                       bit(2),
   2 nnames                     fixed bin(15),
   2 nindex                     fixed bin(17),
   2 dtem                       bit(36),
   2 dtd                        bit(36),
   2 pathname_len              fixed bin(17),
   2 pathname_index           fixed bin(17)) unaligned;
```

hcs_\$star_

hcs_\$star_

where:

1. type is the same as above.
2. nnames is the same as above.
3. nindex is the same as above.
4. dtem is the date and time the link was last modified.
5. dtd is the date and time the link was last dumped.
6. pathname_len is the number of significant characters in the pathname associated with the link.
7. pathname_index is the index in the array of names for the link pathname.

If the pathname associated with each link was requested, the pathname is placed in the names array and occupies six units of this array. 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.

This page intentionally left blank.

Name: `hcs_$wakeup`

The `hcs_$wakeup` entry point sends an interprocess communication wakeup signal to a specified process over a specified event channel. If that process had previously called the `ipc_$block` entry point, it would be awakened. See the `ipc_` subroutine description in this document.

Usage

```
declare hcs_$wakeup entry (bit(36), fixed bin(71), fixed bin(71), fixed
    bin(35));
```

```
call hcs_$wakeup (process_id, channel_id, message, code);
```

where:

1. `process_id` is the process identifier of the target process. (Input)
2. `channel_id` is the identifier of the event channel over which the wakeup is to be sent. (Input)
3. `message` is the event message to be interpreted by the target process. (Input)
4. `code` is a system status code. (Output) It can be one of the following:

0	no error
1	signalling was correctly done, but the target process was in the stopped state
2	an input argument was incorrect, so signalling was aborted
3	the target process was not found (e.g., <code>process_id</code> was incorrect or the target process had been destroyed), so signalling was aborted

`error_table_$invalid_channel` the channel identifier was not valid

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_\$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 in the MPM Commands 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);
```

where:

1. request_type is the name of a request type defined in the I/O daemon tables. (Input)
2. generic_type is the name of the generic type of the above request type. (Output)
3. 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_\$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);
```

where:

1. request_type is the name of a request type defined in the I/O daemon tables. (Input)
2. access_name is the driver access name for the above request type. (Output)
3. code is a standard status code. If the specified request type is not found, the code error_table_\$id_not_found is returned. (Output)

Name: iox_

This subroutine performs I/O operations and some related functions. Most of the iox_ subroutine entry points are described in the MPM Subroutines.

The following entry points are probably not of interest to every user. However, they are needed by any user who is writing his own I/O module. For information on I/O module construction and the use of certain iox_ entry points, see "Writing an I/O Module" in Section IV.

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);
```

where:

1. iocb_ptr points to the I/O control block to be freed. (Input)
2. code is an I/O system status code. (Output)

Entries: iox_\$err_no_operation, iox_\$err_not_open, iox_\$err_not_closed,
iox_\$err_not_attached

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_open, error_table_\$not_closed, or error_table_\$not_attached. 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 "Writing an I/O Module" in Section IV for instructions on when to assign this entry point to such an entry variable.

Usage

```
declare iox_$err_no_operation entry options (variable);
```

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);
```

where:

1. n is the number of the I/O control block. (Input)
2. iocb_ptr is a pointer to the control block. (Output)
3. code is an I/O system status code. (Output)

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 (switchname, iocb_ptr, code);
```

where:

1. switchname is the name of the I/O switch. (Input)
2. iocb_ptr is a pointer to the control block. (Output)
3. code is an I/O system status code. (Output)

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. See "Writing an I/O Module" in Section IV for instructions on when to call iox_\$propagate.

Usage

```
declare iox_$propagate entry (ptr);
```

```
call iox_$propagate (iocb_ptr);
```

where iocb_ptr is a pointer to the control block. (Input)

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 that 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 "Intraprocess Access Control" in Section VI of the MPM Reference Guide).

The hcs_\$wakeup entry point (described in this document) is used to wake up a blocked process for a specified event.

Entry: 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);
```

where:

1. channel_id is the identifier of the event channel. (Output)
2. code is a nonstandard status code; see "Status Code Values" later in this description. (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);
```

where:

1. channel_id is the identifier of the event channel. (Input)
2. code is a nonstandard status code; see "Status Code Values" later in this description. (Output)

Entry: ipc_\$dcl_ev_call_chn

This entry point changes an event-wait channel into an event-call channel.

Usage

```
declare ipc_$dcl_ev_call_chn entry (fixed bin(71), entry, ptr, fixed bin,
fixed bin(35));
call ipc_$dcl_ev_call_chn (channel_id, procedure, data_ptr, priority,
code);
```

where:

1. channel_id is the identifier of the event channel. (Input)
2. procedure is the procedure entry point invoked when an event occurs on the specified channel. The procedure entry point should not be an internal procedure. (Input)
3. data_ptr is a pointer to a region where data to be passed to and interpreted by that procedure entry point is placed. (Input)
4. 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)
5. code is a nonstandard status code; see "Status Code Values" later in this description. (Output)

4. 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)
5. code is a nonstandard status code; see "Status Code Values" later in this description. (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);
```

where:

1. channel_id is the identifier of the event channel. (Input)
2. code is a nonstandard status code; see "Status Code Values" later in this description. (Output)

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);
```

where:

1. channel_id is the identifier of the event channel. (Input)
2. code is a nonstandard status code; see "Status Code Values" later in this description. (Output)

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);
```

where:

1. channel_id is the identifier of the event channel. (Input)
2. code is a nonstandard status code; see "Status Code Values" later in this description. (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);
```

where:

1. channel_id is the identifier of the event channel. (Input)
2. code is a nonstandard status code; see "Status Code Values" later in this description. (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);
```

where code is a nonstandard status code; see "Status Code Values" later in this description.

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 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);
```

where code is a nonstandard status code; see "Status Code Values" later in this description.

Entry: ipc_\$mask_ev_calls

This entry point causes the ipc_\$block entry point (see below) to completely ignore event-call channels occurring in the user's ring at the time of this call.

Usage

```
declare ipc_$mask_ev_calls entry (fixed bin(35));  
call ipc_$mask_ev_calls (code);
```

where code is a nonstandard status code; see "Status Code Values" later in this description.

Entry: ipc_\$unmask_ev_calls

This entry point reverses the effect of the ipc_\$mask_ev_calls entry point.

Usage

```
declare ipc_$unmask_ev_calls entry (fixed bin(35));
call ipc_$unmask_ev_calls (code);
```

where code is a nonstandard status code; see "Status Code Values" later in this description.

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 (wait_list_ptr, info_ptr, code);
```

where:

1. wait_list_ptr is a pointer to the base of a structure that specifies the channels on which events are being awaited. (Input)

```
dcl 1 wait_list          based,
    2 nchan              fixed bin,
    2 channel_id (nchan) fixed bin(71);
```

where:

nchan is the number of channels.

channel_id is an array of channel identifiers selecting the channels to wait on.

2. info_ptr is a pointer to the base of 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). (Input)

```
dcl 1 event_info,
    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          bit(18) unaligned,
2 channel_index fixed bin;

```

where:

- 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 wait_list structure above.
3. code is a nonstandard status code; see "Status Code Values" later in this description. (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, info_ptr, code);

```

where:

1. channel_id is the identifier of the event channel. (Input)
2. ev_occurred indicates whether an event occurred on the specified channel. (Output)
 0 no event occurred
 1 an event occurred
3. info_ptr is as above. (Input)
4. code is a nonstandard status code; see "Status Code Values" below. (Output)

Status Code Values

All of the entry points described above return a value from 0 to 5 for the code argument. The values mean the following:

- | | |
|---|--|
| 0 | no error. |
| 1 | ring violation; e.g., the event channel resides in a ring that is not accessible from the caller's ring. |
| 2 | the table that contains the event channels for a given ring was not found. |
| 3 | the specified event channel was not found. |
| 4 | a logical error in using the ipc_ subroutine was encountered; e.g., waiting on an event-call channel. |
| 5 | a bad argument was passed to the ipc_ subroutine; e.g., a zero-value event channel identifier. |

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_\$dcl_event_call_channel entry point. The procedure is called with one argument, a pointer to the following structure:

```
dcl 1 event_info based,
  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            bit(18) unaligned,
  2 data_ptr        ptr;
```

where:

1. channel_id is the identifier of the event channel.
2. message is an event message as specified to the hcs_\$wakeup entry point.
3. sender is the process identifier of the sending process.
4. dev_signal indicates whether the event occurred as the result of an I/O interrupt.
 "1"b yes
 "0"b no
5. ring is the sender's validation level.
6. data_ptr points to further data to be used by the called procedure.

Name: match_star_name_

The match_star_name_ subroutine implements the Multics storage system star convention by comparing an entryname with a name containing stars or question marks (called a star name). Refer to "Constructing and Interpreting Names" in Section I of the MPM Commands for a description of the star convention and a definition of acceptable star name formats.

Usage

```
declare match_star_name_ entry (char(*), char(*), fixed bin(35));
call match_star_name_ (entryname, star_name, code);
```

where:

1. entryname is the entryname to be compared with the star name. Trailing spaces in the entryname are ignored. (Input)
2. star_name is the star name with which entryname is compared. Trailing spaces in the star name are ignored. (Input)
3. code is a standard status code. (Output) It can be:
0 the entryname matches the star name
error_table_\$nomatch the entryname does not match the star name
error_table_\$badstar the star name does not have an acceptable format

Notes

Refer to the description of the hcs_\$star_ entry point in this document to see how to list the directory entries that match a given star name.

Refer to the description of the check_star_name_ subroutine in this document to see how to validate a star name.

Name: msf_manager_

The msf_manager_ subroutine is designed to provide 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. 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.

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_\$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);
```

where:

1. dir_name is the pathname of the containing directory. (Input)
2. entryname is the entryname of the multisegment file. (Input)

3. fcb_ptr is a pointer to the file control block. (Output)
4. code is a storage system status code. The code error_table_\$dirseg is returned when an attempt is made to open a directory. (Output)

Entry: msf_manager_\$get_ptr

The msf_manager_\$get_ptr entry point returns a pointer to a specified component in the 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 multisegment file. This change does not affect a previously returned pointer to component 0.

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);
```

where:

1. fcb_ptr is a pointer to the file control block. (Input)
2. component is the number of the component desired. (Input)
3. 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
4. seg_ptr is a pointer to the specified component in the file, or null (if there is an error). (Output)
5. bc is the bit count of the component. (Output)
6. code is a storage system status code. (Output) It may be one of the following:

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

```
error_table_$segknown if the specified segment is already known
```

Entry: msf_manager_\$adjust

The msf_manager_\$adjust entry point optionally sets the bit count, truncates, and terminates the components of a multisegment file. It must be told the number of the last component and its bit count. The bit counts of all components with numbers less than the given component are set to sys_info\$max_seg_size*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);
```

where:

1. fcb_ptr is a pointer to the file control block. (Input)
2. component is the number of the intended last component. (Input)
3. bc is the bit count to be placed on the last component. (Input)
4. 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
```

5. 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);
```

where fcb_ptr is the pointer to the file control block. (Input)

Entry: msf_manager_\$acl_list

This entry point returns the access control list (ACL) of a multisegment file.

Usage

```
declare msf_manager_$acl_list entry (ptr, ptr, ptr, fixed bin, ptr, fixed  
bin(35));  
call msf_manager_$acl_list (fcb_ptr, area_ptr, area_ret_ptr, acl_ptr,  
acl_count, code);
```

where:

1. fcb_ptr is a pointer to the file control block. (Input)
2. 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 (see below). (Input)
3. area_ret_ptr points to the start of the allocated list of ACL entries. (Output)
4. acl_ptr if area_ptr is null, then acl_ptr points to an ACL structure, segment_acl, (described in "Notes" below) into which mode information is placed for the access names specified in that same structure. (Input)
5. acl_count is the number of entries in the segment_acl structure. (Input or Output)
Input is the number of entries in the ACL structure identified by acl_ptr
Output is the number of entries in the segment_acl structure allocated in the area pointed to by area_ptr, if area_ptr is not null
6. code is a storage system status code. (Output)

Notes

The following is the segment_acl structure:

```
dcl 1 segment_acl (acl_count)      aligned based (acl_ptr),
  2 access_name                    char(32),
  2 modes                          bit(36),
  2 zero_pad                       bit(36),
  2 status_code                    fixed bin(35);
```

where:

1. access_name is the access name (in the form Person_id.Project_id.tag) that identifies the process to which this ACL entry applies.
2. modes contains the modes for this access name. The first three bits correspond to the modes read, execute, and write. The remaining bits must be 0's. For example, rw access is expressed as "101"b.
3. zero_pad must contain the value zero. (This field is for use with extended access and may only be used by the system.)
4. status_code is a storage system status code for this ACL entry only.

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 structure either has status_code set to 0 and contains the multisegment file's mode 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);
```

where:

1. fcb_ptr is a pointer to the file control block. (Input)
2. acl_ptr points to the user-supplied segment_acl structure (described in the msf_manager_\$acl_list entry point above) that is to replace the current ACL. (Input)

3. `acl_count` is the number of entries in the `segment_acl` structure. (Input)
4. `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` entries are added. (Input)
"0"b adds `rw *.SysDaemon.*` entry
"1"b replaces the existing ACL with only the user-supplied `segment_acl`
5. `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` entries is performed top to bottom, allowing a later entry to overwrite a previous one if the `access_name` in the `segment_acl` structure is identical.

Entry: `msf_manager_$acl_add`

This entry point adds the specified access modes to the ACL of the 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);
```

where:

1. `fcb_ptr` is a pointer to the file control block. (Input)
2. `acl_ptr` points to the user-supplied `segment_acl` structure (described in the `msf_manager_$acl_list` entry point above). (Input)
3. `acl_count` is the number of ACL entries in the `segment_acl` structure. (Input)
4. `code` is a storage system status code. (Output)

Note

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.

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);
```

where:

1. fcb_ptr is a pointer to the file control block. (Input)
2. acl_ptr points to a user-supplied delete_acl structure. See "Notes" below. (Input)
3. acl_count is the number of ACL entries in the delete_acl structure. (Input)
4. code is a storage system status code. (Output)

Notes

The delete_acl structure is as follows:

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

where:

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

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.

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 information structure described below. This structure can be found in the object_info.incl.pl1 include file.

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);
```

where:

1. seg_ptr is a pointer to the base of the object segment. (Input)
2. bc is the bit count of the object segment. (Input)
3. info_ptr is a pointer to the info structure in which the object information is returned. See "Information Structure" later in this description. (Input)
4. 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 (described in this document).

Usage

```
declare object_info_$display entry (ptr, fixed bin(24), ptr, fixed
    bin(35));
```

```
call object_info_$display (seg_ptr, bc, info_ptr, code);
```

where seg_ptr, bc, info_ptr, and code are as 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);
```

where seg_ptr, bc, info_ptr, and code are the same as in the object_info_\$brief entry point above.

Information Structure

The information structure is as follows:

```
dcl 1 object_info          aligned,
  2 version_number        fixed bin,
  2 text_ptr              ptr,
  2 def_ptr               ptr,
  2 link_ptr              ptr,
  2 stat_ptr              ptr,
  2 symb_ptr              ptr,
  2 bmap_ptr              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 pad                bit(28) unaligned,
    2 entry_bound        fixed bin,
    2 textlink_ptr       ptr,
```

/*This is the limit of the \$brief info structure.*/

```
    2 compiler           char(8) aligned,
    2 compile_time       fixed bin(71),
    2 access_name        char(32) aligned,
    2 cvers              aligned,
    3 offset            bit(18) unaligned,
    3 length            bit(18) unaligned,
    2 comment,          bit(18) unaligned,
    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.*/

where:

1. version_number is the version number of the structure (currently this number is 2). This value is input.
2. text_ptr is a pointer to the base of the text section.
3. def_ptr is a pointer to the base of the definition section.
4. link_ptr is a pointer to the base of the linkage section.
5. stat_ptr is a pointer to the base of the static section.
6. symb_ptr is a pointer to the base of the symbol section.
7. bmap_ptr is a pointer to the break map.
8. tlng is the length (in words) of the text section.
9. dlng is the length (in words) of the definition section.
10. llng is the length (in words) of the linkage section.

11. `ilng` is the length (in words) of the static section.
12. `slng` is the length (in words) of the symbol section.
13. `blng` is the length (in words) of the break map.
14. `old_format` indicates the format of the segment.
"1"b old format
"0"b new format
15. `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
16. `relocatable` indicates whether the object is relocatable.
"1"b the object is relocatable
"0"b the object is not relocatable
17. `procedure` indicates whether the segment is a procedure.
"1"b it is a procedure
"0"b it is nonexecutable data
18. `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
19. `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
20. `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
21. `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
22. `pad` is currently unused.
23. `entry_bound` is the entry bound if this is a gate procedure.
24. `textlink_ptr` 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.
25. `compiler` is the name of the compiler that generated this object segment.
26. `compile_time` is the date and time this object was generated.
27. `access_name` is the access identifier (in the form `Person_id.Project_id.tag`) of the user in whose behalf this object was generated.

28. `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.
29. `cvers.length` is the length (in characters) of the compiler version string.
30. `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.
31. `comment.length` is the length (in characters) of the comment string.
32. `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.
33. `rel_text` is a pointer to the object's text section relocation information.
34. `rel_def` is a pointer to the object's definition section relocation information.
35. `rel_link` is a pointer to the object's linkage section relocation information.
36. `rel_static` is a pointer to the object's static section relocation information.
37. `rel_symbol` is a pointer to the object's symbol section relocation information.
38. `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.
39. `static_boundary` is analogous to `text_boundary` for internal static.
40. `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).
41. `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).

This is the limit of the info structure for the `object_info_$long` entry point.

Name: prepare_mc_restart_

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).

The prepare_mc_restart_ subroutine is provided to check machine conditions for restartability, and to make modifications to the machine conditions (to accomplish user modifications to process execution) before a condition handler returns.

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);
```

where:

1. mc_ptr is a pointer to the machine conditions. (Input)
2. 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_\$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);
```

where:

1. mc_ptr is as above. (Input)
2. new_ins is the desired substitute machine instruction. (Input)
3. code is as above. (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);
```

where:

1. mc_ptr is the same as in the prepare_mc_restart_\$retry entry point above. (Input)
2. newp is used in replacing the instruction counter in the machine conditions. (Input)
3. code is the same as in the prepare_mc_restart_\$retry entry point above. (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 "Multics Condition Mechanism" in Section VI of the MPM Reference Guide.

For all entry points in the `prepare_mc_restart_` subroutine, the following codes can be returned:

<code>error_table_\$badarg</code>	an invalid <code>mc_ptr</code> was provided
<code>error_table_\$no_restart</code>	the machine conditions cannot be restarted
<code>error_table_\$bad_ptr</code>	the restart location is not accessible
<code>error_table_\$useless_restart</code>	the same error will occur again if restart is attempted

Name: read_allowed_

The read_allowed_ subroutine determines whether a subject of specified authorization has access (with respect to the access isolation mechanism) to read an object of specified access class.

Usage

```
declare read_allowed_entry (bit(72) aligned, bit(72) aligned) returns
    (bit(1) aligned);

returned_bit = read_allowed_ (authorization, access_class);
```

where:

1. authorization is the authorization of the subject. (Input)
2. access_class is the access class of the object. (Input)
3. returned_bit indicates whether the subject is allowed to read the object.
(Output)
"1"b read is allowed
"0"b read is not allowed

read_write_allowed_

read_write_allowed_

Name: read_write_allowed_

The read_write_allowed_ subroutine determines whether a subject of specified authorization has access (with respect to the access isolation mechanism) to read and write an object of specified access class.

Usage

```
declare read_write_allowed_ entry (bit(72) aligned, bit(72) aligned)
      returns (bit(1) aligned);
```

```
returned_bit = read_write_allowed_ (authorization, access_class);
```

where:

1. authorization is the authorization of the subject. (Input)
2. access_class is the access class of the object. (Input)
3. 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

release_area_

release_area_

Name: release_area_

The release_area_ subroutine is used to clean 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 any case, segments acquired to extend the area are released to the free pool of temporary segments.

Usage

```
declare release_area_ entry (ptr);  
call release_area_ (area_ptr);
```

where area_ptr points to the area to be released. (Input)

This page intentionally left blank.

signal_

signal_

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 "Multics Condition Mechanism" in Section VI of the MPM Reference Guide.

Usage

```
declare signal_ entry (char(*), ptr, ptr);  
call signal_ (name, mc_ptr, info_ptr);
```

where:

1. name is the name of the condition to be signalled. (Input)
2. mc_ptr points 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)
3. info_ptr points to information relating to the condition being raised. The structure of the information is dependent upon the condition being signalled; however, conditions raised with the same name should provide the information in the same structure. 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 "List of System Conditions and Default Handlers" in Section VI of the MPM Reference Guide. This argument is intended for use in signalling conditions other than hardware faults. This argument is also optional. (Input)

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: 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. (See the pl1, fortran, and cobol commands in the MPM Commands.) 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_\$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);
```

where:

1. seg_ptr points to any location in the object segment. (Input)
2. 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)
3. bc is the bit count of the object segment; if 0, the stu_\$find_header entry point determines the bit count itself. (Input)
4. header_ptr points to the symbol table header if it is found or is null if the header is not found. (Output)

Note

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, described in the MPM Subroutines).

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);
```

where:

1. header_ptr points to a symbol table header. (Input)
2. 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)
3. 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_\$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);
```

where:

1. stack_ptr points to an active stack frame. (Input)
2. 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)

3. `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)
4. `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_$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);
```

where:

1. `block_ptr` points to the `runtime_block` node in which the search is to begin. (Input)
2. `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)
3. `found_ptr` is set to point to the `runtime_block` node in which the specified identifier is found. (Output)
4. `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

5. 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_\$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);
```

where:

1. v is an encoded value from a runtime_symbol node, e.g., runtime_symbol.size. (Input)
2. 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 above. (Input)
3. 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)
4. 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)

5. `text_ptr` is a pointer to the base of the object segment that contains the specified block. If `text_ptr` is null, the `stu_$decode_runtime_value` entry point attempts to obtain the text pointer, if it is needed, from the active stack frame or the `block_ptr`; decoding fails if a pointer to the object segment is needed and `stack_ptr`, `block_ptr`, and `text_ptr` are all null. (Input)
6. `ref_ptr` is the value of the pointer to be used as locator qualifier if the variable that corresponds to the `runtime_symbol` node that contains the encoded value is based. The value of `ref_ptr` can often be determined by means of the `stu_$get_implicit_qualifier` entry point described below. (Input)
7. `code` is a status code. (Output) It is:
- 0 if the encoded value was successfully decoded
 - 1 if the value could not be decoded
8. `value` is the decoded value if the value of `code` is 0. (Output)

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);
```

where:

1. `block_ptr` points to the `runtime_block` node that corresponds to the procedure or begin block in which the based variable is declared. (Input)
2. `symbol_ptr` points to the `runtime_symbol` node that corresponds to the based variable. (Input)
3. `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)

4. `link_ptr` is a pointer to the linkage section of the specified block. If `link_ptr` is null, the `stu_$get_implicit_qualifier` entry point attempts to obtain the linkage pointer, if it is needed, from the active stack frame; the implicit qualifier cannot be determined if a pointer to the linkage section is needed and `stack_ptr` and `link_ptr` are both null. (Input)
5. `text_ptr` is a pointer to the base of the object segment that contains the specified block. If `text_ptr` is null, the `stu_$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)
6. `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:

1. The based variable was declared without an implicit qualifier, e.g.,
`dcl a based;`
2. Determining the implicit qualifier involves evaluating an expression, for example, the based variable was declared as:
`dcl a based(p(i));`
3. 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_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);
```

where:

1. 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)
2. symbol_ptr points to the runtime_symbol node that corresponds to the symbol whose address is to be determined. (Input)
3. 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)
4. 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)
5. 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)
6. 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)
7. 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)
8. 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_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);
```

where:

1. block_ptr points to the runtime_block node that corresponds to the block in which the instruction offset exists. (Input)
2. offset is the offset of an instruction in the text segment. (Input)
3. 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)
4. num is set to the number of words generated for the specified source line. (Output)
5. 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_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);
```

where:

1. head_ptr is a pointer to the symbol section header of a standard object segment. (Input)
2. offset is the offset of an instruction in the text section. (Input)
3. 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)
4. num is set to the number of words in the object code generated for the specified source line. (Output)
5. 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);
```

where:

1. block_ptr points to the runtime_block node. (Input)
2. line_no specifies the source line number, which must be in the main source file. (Input)
3. 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_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);
```

where:

1. head_ptr is a pointer to the symbol section header of a standard object segment. (Input)
2. offset is the offset of an instruction in the text section. (Input)

3. `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)
4. `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)
5. `line_offset` is set to the number of characters that precede the first character of the source for the specified statement. (Output)
6. `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)
7. `file` is the number of the source file in which the source for the desired statements is contained (see "Source Map" in Section I of this manual). (Output)

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);
```

where:

1. `head_ptr` is a pointer to the symbol section header of a standard object segment. (Input)
2. `line_no` is the line number of a statement in the main source file. (Input)
3. `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);
```

where:

1. head_ptr is a pointer to the symbol section header of a standard object segment. (Input)
2. 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)
3. 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)
4. 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);
```

where:

1. block_ptr points to the runtime_block node that corresponds to the procedure or begin block in which the offset variable is declared. (Input)
2. symbol_ptr points to the runtime_symbol node that corresponds to the offset variable. (Input)

3. data_ptr points to the offset value to be converted to a pointer. (Input)
4. 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)
5. 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)
6. 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)
7. 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);
```

where:

1. block_ptr is as above. (Input)
2. symbol_ptr is as above. (Input)
3. data_ptr points at the pointer value to be converted to an offset. This pointer value must be an unpacked pointer value. (Input)
4. stack_ptr is as above. (Input)
5. link_ptr is as above. (Input)

- 6. text_ptr is as above. (Input)
- 7. 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);

where:

- 1. value is the remote format value to be decoded. (Input)
- 2. stack_ptr is a pointer to the active stack frame of the block that contains the format being decoded. (Input)
- 3. ref_ptr is the pointer value to be used if the format value being decoded requires pointer qualification. (Input)
- 4. format is set to the format value if decoding is successful. (Output)
- 5. code is a status code. (Output) It is:
 - 0 if decoding is successful
 - 1 if decoding is not successful

Example

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, 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;

```


stu_

stu_

```
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;
```

This page intentionally left blank.

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);
```

where:

1. code is a standard system status code (declared fixed bin(35)) describing the reason for calling the sub_err_ subroutine. (Input)
2. 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)
3. flags describe how and whether restart may be attempted. The flags argument should be declared as a nonvarying character string. (Input) One of the following values is permitted:
h halt at command level after printing message; resume if start command is invoked (described in the MPM Commands).
c continue after printing message.
s stop; attempt to restart raises the illegal_return condition.
4. info_ptr is a pointer (declared as an aligned pointer) to optional information specific to the situation. The standard system environment does not use this pointer, but it is provided for the convenience of other environments. (Input)
5. retval is a return value from the environment to which the error was reported. 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)

sub_err_

sub_err_

6. `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 in the MPM Subroutines. (Input)
7. `ioa_args` are any arguments required for conversion by the `ctl_string` argument. (Input)

Operation

The `sub_err_` subroutine proceeds as follows: the structure described below is filled in from the arguments to the `sub_err_` subroutine and `signal_` 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 `c` 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 `s` is specified. If the value `s` 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` and `default_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 `software_info_ptr` that points to a structure with the following declaration.

```
dcl 1 info          aligned based (software_info_ptr),
    2 length        fixed bin,
    2 version        fixed bin,
    2 action_flags  aligned,
    3 cant_restart  bit(1) unal,
    3 default_restart bit(1) unal,
    3 pad           bit(34) unal,
    2 info_string   char(256) varying,
    2 code          fixed bin(35),
    2 retval        fixed bin(35),
    2 name          char(32),
    2 info_ptr      ptr;
```

where:

1. length is the size of the structure in words.
2. version is the version number of the structure. Currently, the version is 2.
3. cant_restart indicates if the condition cannot be restarted.
"1"b yes
"0"b no
4. default_restart indicates if the standard environment prints the message and continues execution without calling the listener.
"1"b yes
"0"b no
5. pad is padding.
6. info_string is the converted message from the ctl_string and ioa_args arguments.
7. code is a standard system status code.
8. retval is the return value. The standard environment sets this value to zero.
9. name is the name of the module encountering the condition.
10. info_ptr is a pointer to additional information associated with the condition.

The handler should check info.name and 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: sys_info

The sys_info data base is a wired-down, per-system data base. It is accessible in all rings but can be modified only in ring 0. It contains many system parameters and constants. All references to it are made through externally defined variables.

Usage

```
dcl ( sys_info$clock_          fixed bin,
      1 sys_info$ips_mask_data aligned,
      2 count                  fixed bin,
      2 array                  (4 refer(count)),
      3 mask                   bit(35) aligned,
      3 name                   char(4) aligned,
      sys_info$page_size       fixed bin(35),
      sys_info$max_seg_size    fixed bin(35),
      sys_info$default_stack_length fixed bin(35),
      sys_info$access_class_ceiling bit(22),
      sys_info$time_correction_constant fixed bin(71),
      sys_info$maxlinks        fixed bin,
      sys_info$time_delta      fixed bin,
      sys_info$time_of_bootload fixed bin(71),
      sys_info$time_zone       char(3)) external;
```

where:

1. clock_ is the port number of the system controller containing the clock.
2. ips_mask_data is the array that specifies the number and mapping of interprocess signal (IPS) masks. "count" is the current number of valid IPS names and "array" specifies the mapping between IPS name and IPS mask for each IPS signal.
3. count is the current number of valid IPS names.
4. array specifies the mapping between IPS name and IPS mask for each IPS signal.
5. page_size is the page size in words.
6. max_seg_size is the maximum segment size in words.
7. default_stack_length is the default stack maximum size in words.
8. access_class_ceiling is the maximum access class.
9. time_correction_constant is the correction from Greenwich mean time (GMT) in microseconds.
10. maxlinks is the maximum depth to which the system chases a link without finding a branch.
11. time_delta is the difference between this time zone and GMT in hours.

sys_info

sys_info

12. time_of_bootload

is the clock reading at the time of bootload.

13. time_zone

is the name of the time zone (e.g., EST).

system_info_

system_info_

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_\$installation_id

This entry point returns the 32-character installation identifier typed in the header of the who command (described in the MPM Commands) and at dial-up time.

Usage

```
declare system_info_$installation_id entry (char(*));  
call system_info_$installation_id (id);
```

where id is the installation identifier. (Output)

Entry: system_info_\$sysid

This entry point returns the eight-character system identifier 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);
```

where sys is the system identifier that identifies the current version of the system. (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);
```

where:

1. c is the company or institution name (a maximum of 64 characters). (Output)
2. d is the department or division name (a maximum of 64 characters). (Output)
3. cc is the company name, double spaced (a maximum of 120 characters). (Output)
4. dd is the department name, double spaced (a maximum of 120 characters). (Output)

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);
```

where:

1. mn is the maximum number of users. (Output)
2. nn is the current number of users. (Output)
3. mu is the maximum number of load units (times 10). (Output)
4. nu is the current number of load units (times 10). (Output)

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);
```

where tu is the time the system came up. (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 the system will return, if this data is available.

Usage

```
declare system_info_$next_shutdown entry (fixed bin(71), char(*), fixed  
bin(71));  
call system_info_$next_shutdown (td, rsn, tn);
```

where:

1. td is the time of the next scheduled shutdown. If none is scheduled, this is 0. (Output)
2. rsn is the reason for the next shutdown (a maximum of 32 characters). If it is not known, it is blank. (Output)
3. 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);
```

where:

1. cpu is the CPU-hour rate per shift. (Output)
2. log is the connect-hour rate per shift. (Output)
3. prc is the process-hour rate per shift. (Output)
4. cor is the page-second rate for main memory per shift. (Output)
5. dsk is the page-second rate for secondary storage. (Output)
6. reg is the registration fee per user per month. (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);
```

where:

1. ndev is the number of devices with prices. (Output)
2. dev_ptr points to an array where device prices are stored. (Input)

Note

In the above entry point, the user must provide the following array (in his storage) for device prices:

```
dcl 1 dvt(16)          based (devp) aligned,
    2 device_id        char(8),
    2 device_price     (0:7) float bin;
```

where:

1. dvt is the user structure. Only the first ndev of the 16 will be filled in.
2. device_id is the name of the device.
3. device_price is the per-hour price by shifts for the device.

Entry: system_info_\$shift_table

This entry point returns a table that defines when each shift begins and ends.

Usage

```
declare system_info_$shift_table entry ((336) fixed bin);
call system_info_$shift_table (st);
```

where st is a table with one entry for each half hour, beginning with 0000 Monday. The table gives the shift number for that half-hour period. Shifts can be from 0 to 7. (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);
```

where:

1. cpurate is the price per CPU hour for absentee queues 1 to 4. (Output)
2. realrate is the memory unit rate for absentee queues 1 to 4. (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);
```

where rp is the price per 1000 units (a unit is 700 bits) 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);
```

where:

1. time is the clock time of the last shutdown. (Output)
2. erfno is the ERF number of the last crash, or blank. (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);
```

where ceil is the access ceiling. (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);
```

where:

1. long is an array of the long level names. (Output)
2. short is an array of the short level names. (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.

system_info_

system_info_

Usage

```
declare system_info_$category_names entry (dim(18) char(32), dim(18)
      char(8));
call system_info_$category_names (long, short);
```

where long and short are the same as for the system_info_\$level_names entry point.

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.

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. These 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 conditions alarm and cput. 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.

1. 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.

2. routine is a procedure entry point that is called when the timer goes off. The routine is called as follows:

```
declare routine entry (ptr, char(*));
```

```
call routine (mc_ptr, name);
```

where:

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: alarm for a real-time timer and cput for a CPU 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.

3. `time` is the time (fixed binary(71)) at which the wakeup or call is desired.
4. `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 (described in the MPM Subroutines). Absolute CPU time is total virtual time used by the the process, i.e., the time returned by the `cpu_time_and_paging_` subroutine (described in the MPM Subroutines). 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

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

```
declare 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."

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

```
declare timer_manager_$alarm_call entry (fixed bin(71), bit(2), entry);  
call timer_manager_$alarm_call (time, flags, routine);
```

Entry: timer_manager_\$alarm_call_inhibit

This entry point sets up a real-time timer that calls the handler routine specified when the timer goes off while all interrupts are inhibited. When the handler routine returns from the interrupt, interrupts are reenabled. If the handler routine does not return from the interrupt, interrupts are not reenabled and the user process may malfunction.

Usage

```
declare timer_manager_$alarm_call_inhibit entry (fixed bin(71), bit(2),
    entry);
call timer_manager_$alarm_call_inhibit (time, flags, routine);
```

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___". (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

```
declare timer_manager_$cpu_call entry (fixed bin(71), bit(2), entry);
call timer_manager_$cpu_call (time, flags, routine);
```

Entry: timer_manager_\$cpu_call_inhibit

This entry point sets up a CPU timer that calls the handler routine specified with all interrupts inhibited when the timer goes off. When the handler routine returns from the interrupt, interrupts are reenabled. If the handler routine does not return from the interrupt, interrupts are not reenabled and the user process may malfunction.

Usage

```
declare timer_manager_$cpu_call_inhibit entry (fixed bin(71), bit(2),
entry);
call timer_manager_$cpu_call_inhibit (time, flags, routine);
```

Entry: timer_manager_\$cpu_wakeup

This entry 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_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);
```

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.

timer_manager_

timer_manager_

Usage

```
declare timer_manager_$reset_cpu_wakeup entry (fixed bin(71));
call timer_manager_$reset_cpu_wakeup (channel);
```

Entry: timer_manager_\$reset_alarm_call

This entry point turns off all real-time timers that call the routine specified when they go off.

Usage

```
declare timer_manager_$reset_alarm_call entry (entry);
call timer_manager_$reset_alarm_call (routine);
```

Entry: timer_manager_\$reset_alarm_wakeup

This entry 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);
```

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);
```

where:

1. dir_name is the pathname of the containing directory. (Input)
2. entryname is the entryname of the segment. (Input)
3. seg_ptr is a pointer to the segment, or is null if an error is encountered. (Output)
4. aclinfo_ptr is a pointer to ACL information (if any) needed by the tssi_\$finish_segment entry point. (Output)
5. 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 in this document), with the original segment considered as component 0.

tssi_

tssi_

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);
```

where:

1. dir_name is the pathname of the containing directory. (Input)
2. entryname is the entryname of the multisegment file. (Input)
3. seg_ptr is a pointer to component 0 of the file. (Output)
4. aclinfo_ptr is a pointer to ACL information (if any) needed by the tssi_\$finish_file entry point. (Output)
5. fcb_ptr is a pointer to the file control block needed by the msf_manager_subroutine. (Output)
6. 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. 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.

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);
```

where:

1. seg_ptr is a pointer to the segment. (Input)
2. bc is the bit count of the segment. (Input)
3. mode is the access mode to be put on the segment. (Input)
"110"b re access
"101"b rw access
4. aclinfo_ptr is a pointer to the saved ACL information returned by the tssi_\$get_segment entry point. (Input)
5. 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);
```

where:

1. fcb_ptr is a pointer to the file control block returned by the tssi_\$get_file entry point. (Input)
2. component is the highest-numbered component in the file. (Input)
3. bc is the bit count of the highest-numbered component. (Input)
4. mode is the access mode to be put on the multisegment file. (Input)
5. aclinfo_ptr is a pointer to the saved ACL information returned by the tssi_\$get_file entry point. (Input)
6. 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 "Nonlocal Transfers and Cleanup Procedures" in Section VI of the MPM Reference Guide.) 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).

tssi_

tssi_

Usage

```
declare tssi_$clean_up_segment entry (ptr);
call tssi_$clean_up_segment (aclinfo_ptr);
```

where 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);
```

where:

1. fcb_ptr is a pointer to the file control block returned by the tssi_\$get_file entry point. (Input)
2. aclinfo_ptr is a pointer to the saved ACL information returned by the tssi_\$get_segment entry point. (Input)

Name: tty_

The control operations described below can be used to change various aspects of the behavior of the user's terminal. For a description of the more commonly used control operations, as well as other types of operation supported by the tty_ I/O module, see the description of the module in MPM Subroutines.

For the control operations described below whose names begin with "set", with the exception of set_editing_chars, the ring 0 typewriter DIM does not copy the user's table, but simply copies the pointer supplied by the user. The user must therefore neither destroy nor modify such a table after making one of these calls.

set_delay

sets the numbers of delay characters associated with the output of carriage-motion characters. The info_ptr points to the following structure:

```
dcl 1 delay      based aligned,
    2 version    fixed bin,
    2 default    fixed bin,
    2 vert_nl    fixed bin,
    2 horz_nl    float bin,
    2 const_tab  fixed bin,
    2 var_tab    float bin,
    2 backspace  fixed bin,
    2 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. If it is not zero, the remainder of the structure is ignored.

vert_nl is the number of delay characters to be output for all newlines to allow for the linefeed. If it is negative, it is the complement of the minimum number of characters that must be transmitted between two linefeeds (for a device such as a GE 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. The formula for calculating the number of delay characters to be output following a newline is:

$$\text{ndelays} = \text{vert_nl} + \text{fixed} (\text{horz_nl} * \text{column})$$

const_tab is the constant portion of the number of delays associated with any horizontal tab character.

`var_tab` is the number of additional delays associated with a horizontal tab for each column traversed. The formula for calculating the number of delays to be output following a horizontal tab is:

$$\text{ndelays} = \text{const_tab} + \text{fixed} (\text{var_tab} * \text{n_columns})$$

`backspace` is the number of delays to be output following a backspace character. If it is negative, it is the complement of 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 GE TermiNet 300 which 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.

`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` (above), which is filled in as a result of the call (except for the version number, which must be supplied by the caller).

`set_editing_chars`

changes the characters used for editing input. The `info_ptr` points to the following structure:

```
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. (Version 1 is used by network software.)

`erase` is the erase character.

`kill` is the kill character.

The following rules apply to editing characters:

1. The two editing characters may not be the same.
2. No carriage-movement character (carriage return, newline, horizontal tab, backspace, vertical tab, or formfeed) may be used for either of the editing functions.
3. NUL and space may not be used for either editing function.
4. If either of the editing characters is an ASCII control character, it will not have the desired effect unless `ctl_char` mode is on.

get_editing_chars

is used to find out what input editing characters are in effect. The `info_ptr` points to the structure described above 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).

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:

```
dcl 1 translation_info aligned,
    2 version          fixed bin,
    2 default          fixed bin,
    2 table            aligned,
    3 entries          (0:127) char (1) 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. If it is not zero, the remainder of the structure is ignored.

The 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 last 64 to characters in upper shift.

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` (above). 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_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:

```
dcl 1 conversion_info aligned,
    2 version          fixed bin,
    2 default          fixed bin,
    2 table            aligned,
    3 entries          (0:127) fixed bin (8) unaligned;
```

where:

version is as above.

default is as above.

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:

- 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)

set_output_conversion

provides a table to 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:

- 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)
- 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.

```

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 above, which is filled in as a result of the call (except for the version number, which must be supplied by the caller). In the case of translation tables, if the specified table does not exist (no translation is required), the status code error_table_\$no_table is returned.

set_special

provides a table that specifies sequences to be substituted for certain output characters, and characters which are to be interpreted as parts of escape sequences on input. Output sequences are of the following form:

```

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 ($0 \leq \text{count} \leq 3$). 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:

```

dcl 1 special_chars  aligned based,
      2 version      fixed bin,
      2 default      fixed bin,
      2 nl_seq       aligned like c_chars,
      2 cr_seq       aligned like c_chars,
      2 bs_seq       aligned like c_chars,
      2 tab_seq      aligned like c_chars,
      2 vt_seq       aligned like c_chars,
      2 ff_seq       aligned like c_chars,
      2 printer_on   aligned like c_chars,
      2 printer_off  aligned like c_chars,
      2 red_ribbon_shift aligned like c_chars,
      2 black_ribbon_shift aligned like c_chars,
      2 end_of_page  aligned like c_chars,

      2 escape_length fixed bin,
      2 not_edited_escapes (0 refer (escape_length)) like c_chars,
      2 edited_escapes   (0 refer (escape_length)) like c_chars,

      2 input_escapes aligned,
      3 len             fixed bin (8) unaligned,
      3 str             char (1 refer (input_escapes.len)) unaligned,
      2 input_results  aligned,
      3 pad            bit (9) unaligned,
      3 str            char (1 refer (input_escapes.len)) unaligned;

```

where:

- version** is the version number of this structure. It must be 1.
- default** is as above.
- nl_seq** is the output character sequence to be substituted for a newline character.
- 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.
- 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 blanks are substituted.
- tab_seq** is the output character sequence to be substituted for a horizontal tab. If count is zero, the appropriate number of blanks 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.
- 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.

`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` (above).

NOTE: `nl_seq.count` should generally be nonzero, as should either `cr_seq.count` or `bs_seq.count`.

`get_special` is used to obtain the contents of the `special_chars` table currently in use. The `info_ptr` points to the following structure:

```
dcl 1 get_special_info 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)

Notes

To assist the user in determining how to alter the tables described above, a summary is given below of the sequence of operations performed on input and output strings in ring 0.

INPUT PROCESSING

1. 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.
2. Canonicalization
The input string is rearranged (if necessary) into canonical form as described in Section III of the MPM Reference Guide.
3. Editing
Erase and kill editing is carried out, using the `editing_chars` string described above.

4. 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

1. Capitalization
Lowercase letters are replaced by uppercase for terminals in "capo" mode; uppercase letters are prefixed by escape characters if appropriate.
2. 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.
3. 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.

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);
```

where tag is a nonlocal label variable. (Input)

This page intentionally left blank.

Name: vfile_

The majority of the vfile_ I/O module documentation is in Section III of the MPM Subroutines. The information given here describes additional order calls for users of indexed files. These orders allow a greater degree of control in the areas of synchronization and separate record/index manipulation. They implement various features of indexed files that require somewhat more knowledge of internal file structure than is expected of most users.

min_block_size

The min_block_size operation 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);
```

where:

1. 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)
2. 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. The minimum nonnull record capacity is, therefore, 24 bytes.
2. The size of an allocated block is always an integral number of full words, i.e., a multiple of four bytes.

The formula below gives the allocation size, `block_words`, used for a `write_record` or `rewrite_record` operation with a given buffer length, `buff_len`:

$$\text{block_words} = 0 \text{ (no allocation if and only if } \text{buff_len} \text{ and } \text{min_residue} \text{ and } \text{min_capacity} \text{ all are equal to 0)}$$

otherwise,

$$\text{block_words} = \max(8, (\max(\text{buff_len} + \text{min_residue}, \text{min_capacity}) + 3) / 4)$$

record_status

The `record_status` operation returns information about a specified record in an indexed file, and optionally permits the user to manipulate the record's lock and/or to allocate an empty record.

An argument is provided that permits the user to entirely avoid using the index in accessing and creating records (see "Notes" below).

The I/O switch must be open and attached to an indexed file. The next record position is not altered or used by this operation. The current record position is always set to the record referenced.

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 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 mbz1        bit(32) unal,
    2 record_len      fixed bin(21),
    2 max_rec_len     fixed bin(21),
    2 record_ptr      ptr,
    2 descriptor      fixed bin(35),
    2 mbz2            fixed bin;
```

```
dcl rs_info_version_1 static internal fixed bin init(1);
```

where:

1. version is provided for compatibility with possible future versions of this info structure. The user should set this argument to rs_info_version_1. (Input)
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 (documented in the MPM Subroutines). (Input)
"1"b yes
"0"b no

Possible error codes are:
error_table_\$invalid_lock_reset
error_table_\$locked_by_this_process
error_table_\$lock_wait_time_exceeded
error_table_\$no_room_for_lock
The last code is returned if the allocated record block is too small to contain a lock. (See "Records Locks" below).
3. 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.)
4. 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.
5. 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.

6. mbz1, mbz2 must be set to zero by the user. (Input)
7. record_len gives the record's length in bytes. (Output) If create_sw equals "1"b, this argument is input.
8. max_rec_len if create_sw equals "1"b this argument is input and overrides any minimum block size specification that may currently be in effect (see min_block_size order above). 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. (Output)
9. record_ptr points to the first byte of the allocated record, or is set to null if no allocated record exists. (Output)
10. descriptor is a process independent locator for the specified record. This value is used as an input argument when locate_sw equals "1"b and create_sw equals "0"b. (Output) The actual structure of each descriptor 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. Notice 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.

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 no key associated with the current record. This situation may also arise after using the delete_key operation; if so, a subsequent rewrite_record or delete_record operation 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 control operations 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, pains must be taken 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.

get_key

The get_key operation returns both the key and the record descriptor for the next record in an indexed file.

The I/O 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 undefined, 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.

For this order, the info_ptr argument must point to a structure of the following form:

```
dcl 1 get_key_info based (info_ptr),
    2 mbz          fixed bin,
    2 descriptor   fixed bin(35),
    2 key_length   fixed bin,
    2 key_string   char(0 refer(get_key_info.key_length));
```

where:

1. descriptor is the record locator for the next record. This value may be used as an input argument to the control operations add_key, delete_key, reassign_key, and record_status (see "Notes" below). (Output)
2. key_length is the length of the key at the next record position. (Output)
3. key_string is the next record's key. (Output)
4. mbz must be set to zero by the user. (Input)

Notes

The interpretation of the descriptor argument as a record locator is not mandatory, since the `add_key` and `reassign_key` operations 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` operation.

`add_key`

The `add_key` operation 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 operation. 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" below.)

This operation, as well as 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 of the following form:

```
dcl 1 add_key_info      based(info_ptr),
  2 flags              aligned,
  3 input_key          bit(1) unal,
  3 input_descrip      bit(1) unal,
  3 mbz                bit(34) unal,
  2 descriptor         fixed bin(35),
  2 key_length         fixed bin,
  2 key_string         char(0 refer(add_key_info.key_length));
```

where:

1. `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.

2. `input_descrip` 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.
3. `descriptor` is used only if the variable `input_descrip` is set to "1"b. 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` control operation. Descriptors are used by operation that reference the contents or length of a record, in order to obtain the record's address. (Input)
4. `key_length` is the length of the `key_string`. Keys must be between 0 and 256 characters, inclusive. (Input)
5. `key_string` is used only if `add_key_info.input_key` is set to "1"b. It defines the key to be added to the index with the appropriate record descriptor. (Input)
6. `mbz` must be set to zero by the user. (Input)

delete_key

The `delete_key` operation 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.

For this order, the `info_ptr` argument may be null or may point to a structure of the following form:

```
dcl 1 delete_key_info like add_key_info based (info_ptr);
```

where:

1. `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.

2. `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.
3. `descriptor` is used only if `delete_key_info.input_descrip` equals "1"b. The entry that is deleted is the first whose descriptor matches this value, among those entries with the specified key. (Input)
4. `key_length` same as in the `add_key_info` structure above. (Input)
5. `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)

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.

reassign_key

The `reassign_key` operation 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.

For this order, the `info_ptr` argument must point to a structure of the following form:

```
dcl 1 reassign_key_info      based(info_ptr),
  2 flags                    aligned,
  3 input_key                bit(1) unal,
  3 input_old_descrip       bit(1) unal,
  3 input_new_descrip       bit(1) unal,
  3 mbz                      bit(33) unal,
  2 old_descrip              fixed bin(35),
  2 new_descrip              fixed bin(35),
  2 key_length               fixed bin,
  2 key_string               char(0 refer(reassign_key_info.key_length));
```

where:

1. `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.
2. `input_old_descrip` 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_$no_record` is returned.
 - "1"b indicates that the `old_descrip` argument defines the descriptor portion of the index entry to be changed.
3. `input_new_descrip` indicates whether the new descriptor is given in the info structure. (Input)
 - "0"b indicates that the specified index entry is to be reassigned to the current record. If the current record is undefined, the code `error_table_$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.
4. `old_descrip` is used only if `reassign_key_info.input_old_descrip` equals "1"b. The entry that is reassigned is the first whose descriptor matches this value, among those index entries with the specified key. (Input)
5. `new_descrip` is used only if `reassign_key_info.input_new_descrip` equals "1"b. This value replaces the old descriptor of the specified index entry. (Input)
6. `key_length` same as in the `add_key_info` structure above. (Input)
7. `key_string` if `reassign_key_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)

`set_file_lock`

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 operation 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_wait_time_exceeded
error_table_$lock_not_locked
error_table_$locked_by_other_process
```

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.

error_status

The error_status order is accepted when the I/O switch is open and attached to an indexed or sequential file. The operation returns information about the most recent attempt to position beyond either 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,
    2 type fixed,
    2 requested fixed(34),
    2 received fixed(34);
```

where:

1. version must be set to one by the user. (Input)
2. type indicates the type of error that has occurred. (Output)
0 no errors
1 attempt to position beyond end or beginning of file.
3. requested gives the value of the position skip argument that led to the most recent error. (Output)
4. received gives the actual number of records successfully skipped before encountering end or beginning of file (negative if backwards skip). (Output)

Record Locks

This feature pertains only to indexed files. 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.

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 operations that reference the length or contents of an existing record (e.g., `seek_key`, but not `seek_head`) also check the record's lock (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 operation 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` operation on a record locked by another process has no effect other than to return the code `error_table_$record_busy` (file is unchanged). If the lock is invalid, these operations return the code `error_table_$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 case, the operation is successful.

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`.

Control Operations from Command Level

All control orders can be performed using the `io_call` command. The general format is:

```
io_call control switchname order -optional_args-
```

where:

1. `order` is any of the control operations supported by `vfile_`, or is one of the abbreviations listed below.
2. `optional_args` are required for certain orders as indicated below.

<u>order, abbreviation</u>	<u>arguments</u>	<u>output</u>
min_block_size, mb	-min_res- -min_cap-	Nothing is printed.
<p>where both arguments are integers that may optionally be omitted, in which case they default to zero.</p>		
record_status, rs, rsb	-flags- -recl- -maxl- -descrip-	If abbreviation rsb is used, nothing is printed; otherwise the record_length, max_rec_len, record_ptr, and record descriptor (in octal) are printed.

where:

1. flags is a string of four bits, corresponding to the switch settings for lock_sw, unlock_sw, create_sw, and locate_sw. This argument defaults to "0000"b if not given.
2. recl is an integer that must be given when flags.create_sw is set. This determines the new record length.
3. maxl is an optionally supplied integer that may be given along with recl to specify a maximum record length. This defaults to recl if not given.
4. descrip is an octal record descriptor required when flags.locate_sw is set and flags.create_sw is not set

get_key_, gk	No arguments	The next record's key and descriptor (octal) are printed.
add_key, ak	flags -key- -descrip-	Nothing is printed.

where:

1. flags is a string of two bits corresponding to the switch settings for input_key and input_descrip. This argument is required.
2. key is a character string that must be given if flags.input_key is set.
3. descrip is an octal descriptor that must be supplied if flags.input_descrip is set.

delete_key, dk	-flags- -key- -descrip-	Nothing is printed.
----------------	-------------------------------	---------------------

where args are the same as for add_key. Optionally, if no arguments are given, the operation is equivalent to a delete_key order with no info structure (null info_ptr).

reassign_key, rk	flags -key- -old_desc- -new_desc-	Nothing is printed.
------------------	--	---------------------

where:

1. flags is a string of three bits corresponding to the switch settings input_key, input_old_desc, input_new_desc. This argument is required.
2. key is a character string that must be given if flags.input_key is set.
3. old_descrip is an octal number required if flags.input_old_desc is set.
4. new_descrip is an octal number required if flags.input_new_desc is set.

set_file_lock, sf set_lock_flag Nothing is printed.

where set_lock_flag is a string of two bits, that must be specified.

error_status, er no arguments The requested and received counts are printed for the most recent skip error.

write_allowed_

write_allowed_

Name: write_allowed_

The write_allowed_ subroutine determines whether a subject of specified authorization has access (with respect to the access isolation mechanism) to write an object of specified access class.

Usage

```
declare write_allowed_ entry (bit(72) aligned, bit(72) aligned) returns
    (bit(1) aligned);
returned_bit = write_allowed_ (authorization, access_class);
```

where:

1. authorization is the authorization of the subject. (Input)
2. access_class is the access class of the object. (Input)
3. returned_bit indicates whether the subject is allowed to write the object. (Output)
"1"b write is allowed
"0"b write is not allowed

APPENDIX A

APPROVED CONTROL ARGUMENTS

Many Multics commands take control argument strings, i.e., an argument whose first character is a minus sign (-). These character strings should be standardized as much as possible, not only for the convenience of the general user but also for those programmers writing their own commands. Two different lists of control arguments are presented on the following pages. Table A-1 consists of general purpose control arguments, which are already used by several system commands and may be expected to cover most situations. Programmers writing their own commands (and system programmers) should use items from this list whenever possible. Table A-2 consists of more specialized control arguments, which cover a more limited range of situations.

NOTE: Currently, not all Multics commands conform to the information provided in these lists.

Table A-1. Approved Standard Control Arguments

-absentee	-as
-access_class	-acc
-access_name	-an
-account	
-acl	
-address	-addr
-admin	-am
-after	
-alarm	-al
-all	-a
-arguments	-ag
-ascending	-asc
-assignments	-asm
-author	-at
-authorization	-auth
-bcd	
-before	
-block	-bk
-branch	-br
-brief	-bf
-brief_table	-bftb
-call	
-category	-cat
-character	-ch
-check	-ck
-comment	-com
-console_input	-ci
-copy	-cp
-count	-ct
-date	-dt
-date_time_contents_modified	-dtcm
-date_time_entry_modified	-dtem
-date_time_used	-dtu
-debug	-db
-decimal	-dc

Table A-1 (cont). Approved Standard Control Arguments

-delete	-dl
-delimiter	-dm
-density	-den
-depth	-dh
-descending	-dsc
-destination	-ds
-device	-dv
-directory	-dr
-entry	-et
-every	-ev
-exclude	-ex
-execute	
-field	-fl
-file	-f
-first	-ft
-force	
-from	-fm
-gen_type	-gt
-header	-he
-hold	-hd
-home_dir	-hdr
-indent	-ind
-input_file	-if
-input_switch	-isw
-io_switch	-iosw
-label	-lbl
-last	-lt
-length	-ln
-level	
-limit	-li
-line_length	-ll
-lines	
-link	-lk
-link_path	-lp
-list	-ls
-logical_volume	-lv
-long	-lg
-map	
-mask	
-match	
-mode	-md
-model	
-modes	
-multisegment_file	-msf
-name	-nm
-nl	
-nnl	
-no_address	-naddr
-no_header	-nhe
-no_offset	-nofs
-no_pagination	-npgn
-no_restore	-nr
-no_update	-nud
-number	-nb
-octal	-oc
-off	
-offset	-ofs
-on	
-optimize	-ot
-ordered_field	-ofl
-outer_module	-om
-output_file	-of
-output_switch	-osw
-owner	-ow
-page	-pg
-page_length	-pl
-parameter	-pm

Table A-1 (cont). Approved Standard Control Arguments

-pass	
-pathname	-pn
-primary	-pri
-print	-pr
-profile	-pf
-project	-pj
-queue	-q
-quota	
-record	-rec
-repeat	-rpt
-replace	-rp
-request_type	-rqt
-reset	-rs
-restart	-rt
-reverse	-rv
-ring	-rg
-ring_brackets	-rb
-search	-srh
-section	-scn
-segment	-sm
-severity	-sv
-short	-sh
-sort	
-source	-sc
-start	-sr
-stop	-sp
-subscriptrange	-subrg
-subsystem	-ss
-symbols	-sb
-system	-sys
-table	-tb
-tabs	
-terminal_type	-ttp
-time	-tm
-title	
-to	
-total	-tt
-track	-tk
-truncate	-tc
-type	-tp
-unique	-uq
-update	-ud
-volume	-vol
-wait	-wt
-working_dir	-wd

Table A-2. Approved Special Control Arguments

-7punch	-7p
-access_label	-albl
-append	-app
-attached	-att
-attachments	-atm
-ball	-bl
-bottom_label	-blbl
-bottom_up	-bu
-card	
-change_default_auth	-cda
-change_default_project	-cdp
-change_password	-cpw
-compile	
-continue	-ctu
-convert	
-cput	
-detach	-det
-dprint	-dp
-dpunch	-dpn
-file_input	-fi
-flush	
-format	-fmt
-generate_password	-gpw
-go	
-govern	-gv
-hyphenate	-hph
-in	
-input_description	-ids
-interactive	
-interrupt	-int
-invisible	-iv
-keyed	
-library	-lib
-lmargin	-lm
-lower_case	-lc
-mcc	
-meter	-mt
-no_canonicalize	-ncan
-no_endpage	-nep
-no_label	-nlbl
-no_preempt	-np
-no_print_off	-nprf
-no_start_up	-ns
-no_symbols	
-no_warning	-nw
-nogo	
-out	
-output_description	-ods
-print_off	-prf
-process_overseer	-po
-raw	
-realt	
-remove	-rm
-retain	-ret
-retain_data	-retd
-return_value	-rtv
-set_bc	
-set_nl	
-single	-sg
-sleep	
-status	-st
-stop_proc	-spp
-subtotal	-stt
-tape	
-tape7	
-tape9	
-temp_dir	-td

Table A-2 (cont). Approved Special Control Arguments

-template	-tmp
-timers	
-top_label	-tlbl
-trace	
-train	-tn
-use_bc	
-use_nl	
-watch	

This page intentionally left blank.

INDEX

A

- absentee
 - arguments to absentee process 3-2
 - assembly
 - alm_abs 6-19
 - CPU time limit 3-2
 - absentee_real_init_admin_
 - see process: creation
- absolute
 - see relocation information
- abs_io_ 3-2
- acc format string 1-5
- access control
 - access isolation mechanism (AIM)
 - aim_check_\$equal 7-5
 - aim_check_\$greater 7-5
 - aim_check_\$greater_or_equal 7-6
 - convert_aim_attributes_ 7-17
 - get_privileges_ 7-48
 - read_allowed_ 7-123
 - read_write_allowed_ 7-160
 - system_info_\$access_ceiling 7-148
 - system_info_\$category_names 7-148
 - system_info_\$level_names 7-148
 - write_allowed_ 7-160
 - add ACL entry
 - mbx_set_acl 6-34
 - msf_manager_\$acl_add 7-113
 - delete ACL entry
 - mbx_delete_acl 6-29
 - msf_manager_\$acl_delete 7-114
 - initial ACL for new directories
 - hcs_\$add_dir_inacl_entries 7-52
 - hcs_\$delete_dir_inacl_entries 7-57
 - hcs_\$list_dir_inacl 7-71
 - hcs_\$replace_dir_inacl 7-77
 - initial ACL for new segments
 - hcs_\$add_inacl_entries 7-54
 - hcs_\$delete_inacl_entries 7-58
 - hcs_\$list_inacl 7-73
 - hcs_\$replace_inacl 7-78
- access control (continued)
 - list ACL
 - mbx_list_acl 6-32
 - msf_manager_\$acl_list 7-111
 - mailbox
 - mbx_delete_acl 6-29
 - mbx_list_acl 6-32
 - mbx_set_acl 6-34
 - multisegment file ACL
 - msf_manager_\$acl_add 7-113
 - msf_manager_\$acl_delete 7-114
 - msf_manager_\$acl_list 7-111
 - msf_manager_\$acl_replace 7-112
 - privileges
 - get_privileges_ 7-48
 - replace ACL
 - hcs_\$replace_dir_inacl 7-77
 - hcs_\$replace_inacl 7-78
 - msf_manager_\$acl_replace 7-112
 - rings
 - get_ring_ 7-50
 - hcs_\$get_dir_ring_brackets 7-62
 - hcs_\$get_ring_brackets 7-65
 - hcs_\$set_dir_ring_brackets 7-79
 - hcs_\$set_ring_brackets 7-84
 - set_ring_brackets 6-44
- access isolation mechanism (AIM)
 - access checking
 - aim_check_\$equal 7-5
 - aim_check_\$greater 7-5
 - aim_check_\$greater_or_equal 7-6
 - access classes
 - system_info_\$access_ceiling 7-148
 - system_info_\$category_names 7-148
 - system_info_\$level_names 7-148
 - effective access
 - read_allowed_ 7-123
 - read_write_allowed_ 7-160
 - write_allowed_ 7-160
 - privileges
 - get_privileges_ 7-48
- accounting
 - obtaining usage
 - system_info_\$abs_prices 7-146
 - system_info_\$device_prices 7-145
 - system_info_\$io_prices 7-147
 - system_info_\$prices 7-144
 - shift definition table
 - system_info_\$shift_table 7-146

(continued)

- accounting (continued)
 - storage quota
 - hcs_\$quota_move 7-75
 - hcs_\$quota_read 7-76
- ACL
 - see access control
- active function
 - argument list
 - cu_\$af_arg_count 7-23
 - cu_\$af_arg_ptr 7-24
 - cu_\$af_return_arg 7-24
 - error messages
 - active_fnc_err_ 7-3
- active_function_error
 - see condition: raising
- administrative access control
 - see access isolation mechanism (AIM)
- AIM
 - see access isolation mechanism (AIM)
- alarm
 - see time
- allocation
 - area
 - get_system_free_area_ 7-51
- alrm
 - see condition: alrm
- any_other
 - see condition: any_other
- ap
 - see register: pointer register 0 (PRO)
- archive
 - sorting
 - archive_sort 6-21
 - reorder_archive 6-40
- area
 - free storage
 - get_system_free_area_ 7-51
- argument list
 - absentee process 3-2
 - active function
 - cu_\$af_arg_count 7-23
 - cu_\$af_arg_ptr 7-24
 - cu_\$af_return_arg 7-24
 - descriptor 2-15
 - examining
 - cu_\$arg_list_ptr 7-22
 - cu_\$arg_ptr_rel 7-22
 - decode_descriptor_ 7-35
 - format 2-13
 - generating
 - cu_\$gen_call 7-26
 - cu_\$ptr_call 7-26
 - header 2-13

- argument_ptr
 - see stack: frame
- arithmetic
 - conversion
 - assign_ 7-11
- ASCII
 - see character set
- assembly
 - alm 6-3
 - alm_abs 6-19
- attach operation
 - see I/O: operations
- ATTN
 - see quit
- author
 - see segment: author
- authorization
 - see access isolation mechanism (AIM)
- automatic storage 2-1

B

- bar_mode_sp
 - see stack: header
- bind map
 - see bound segment: bind map
- binding
 - see bound segment
- bit count
 - see multisegment file: bit count
- bit count author
 - see segment: bit count author
- block
 - see interprocess communication
- bound segment 1-27
 - bind map 1-30
 - binder symbol block 1-29
 - definition section 1-29
 - display_component_name 6-23
 - internal references 1-29
 - print_bind_map 6-37
 - structure 1-27
- break
 - see quit: handling
- break map 1-2
- break point 1-2

brief mode
 login 3-3

bulk I/O
 offline
 dprint_ 7-37
 iod_info_\$driver_access_name 7-94
 iod_info_\$generic_type 7-94
 system_info_\$io_prices 7-147

byte
 see character set

C

call
 generating 2-10
 cu_\$gen_call 7-26
 cu_\$ptr_call 7-26
 short 2-10

call operator
 see operator

call_op_ptr
 see stack: header

character set
 ascii_to_ebcdic_ 7-7
 ebcdic_to_ascii_ 7-40

cleanup
 bit count
 msf_manager_\$adjust 7-110
 I/O
 iox_\$destroy_iocb 7-96
 releasing stack frame
 unwinder_ 7-159
 translator temporary segment
 tssi_\$clean_up_file 7-158
 tssi_\$clean_up_segment 7-157

clock
 see time

close operation
 see I/O: operations

closing
 multisegment file
 msf_manager_\$close 7-110

combined linkage region (CLR) 2-8
 print_linkage_usage 6-39

command environment
 active function
 cu_\$af_arg_count 7-23
 cu_\$af_arg_ptr 7-24
 cu_\$af_return_arg 7-24

command environment (continued)
 argument list
 cu_\$arg_list_ptr 7-22
 cu_\$arg_ptr_rel 7-22
 command processor
 cu_\$get_cp 7-27
 cu_\$set_cp 7-27
 entry value
 cu_\$decode_entry_value 7-30
 cu_\$make_entry_value 7-30
 get_entry_name_ 7-45
 generating call
 cu_\$gen_call 7-26
 cu_\$ptr_call 7-26
 listener
 cu_\$cl 7-28
 cu_\$get_cl 7-29
 cu_\$set_cl 7-28
 ready message
 cu_\$get_ready_mode 7-20
 cu_\$get_ready_proc 7-20
 cu_\$ready_proc 7-19
 cu_\$set_ready_mode 7-21
 cu_\$set_ready_proc 7-20
 stack frame
 cu_\$stack_frame_ptr 7-25
 cu_\$stack_frame_size 7-25
 system free storage
 get_system_free_area_ 7-51
 validation level
 cu_\$level_get 7-30
 cu_\$level_set 7-29

comparison
 star name
 match_star_name_ 7-107

component
 bound segment
 display_component_name 6-23

com_err_ 4-7

condition 3-2
 alarm 3-2
 any_other 3-2
 cput 3-2
 error messages
 active_fnc_err_ 7-3
 condition_interpreter_ 7-14
 convert_status_code_ 7-18
 error_table_compiler 6-24
 find_condition_info_ 7-42
 handling
 continue_to_signal_ 7-125
 find_condition_info_ 7-42
 unwinder_ 7-159
 machine conditions
 prepare_mc_restart_\$replace 7-121
 prepare_mc_restart_\$retry 7-120
 prepare_mc_restart_\$tra 7-121
 raising
 active_fnc_err_ 7-3
 signal_ 7-125

control operation
 see I/O: operations

convention
 equal
 get_equal_name_ 7-46
 star
 check_star_name_\$entry 7-12
 check_star_name_\$path 7-12
 hcs_\$star_ 7-87
 hcs_\$star_list_ 7-89
 match_star_name_ 7-107
 conversion
 access control
 convert_aim_attributes_ 7-17
 argument
 decode_descriptor_ 7-35
 bound segment offset
 display_component_name 6-23
 character set
 ascii_to_ebcdic_ 7-7
 ebcdic_to_ascii_ 7-40
 entry value
 cu_\$decode_entry_value 7-30
 cu_\$make_entry_value 7-30
 equal name
 get_equal_name_ 7-46
 general
 assign_ 7-11
 location to name
 get_entry_name_ 7-45
 locator
 stu_\$offset_to_pointer 7-137
 stu_\$pointer_to_offset 7-138
 numeric to string
 assign_ 7-11
 cv_bin_ 7-32
 cv_bin_\$dec 7-32
 cv_bin_\$oct 7-33
 string to numeric
 assign_ 7-11
 cv_oct_ 7-34
 cv_oct_check 7-34
 copying
 names
 copy_names 6-22
 move_names 6-36
 cost-saving features
 absentee
 system_info_\$abs_prices 7-146
 combining object segments 1-27
 program tuning
 print_linkage_usage 6-39
 CPU
 time limit for absentee process 3-2
 usage
 system_info_\$abs_prices 7-146
 system_info_\$prices 7-144
 timer_manager_\$cpu_call 7-152
 timer_manager_
 \$cpu_call_inhibit 7-153
 timer_manager_\$cpu_wakeup 7-153
 timer_manager_
 \$reset_cpu_call 7-153
 timer_manager_
 \$reset_cpu_wakeup 7-153

cput
 see condition: cput
 create-if-not-found link 1-11
 creating
 data segment 1-12
 mailbox
 mbx_create 6-27
 multiple names
 mbx_add_name 6-26
 creation time of object segment
 1-2, 1-17

D

daemon
 offline I/O
 dprint_ 7-37
 iod_info_\$driver_access_name 7-94
 iod_info_\$generic_type 7-94
 system_info_\$io_prices 7-147
 daemon_real_init_admin_
 see process: creation
 data segment 1-1
 creating 1-12
 data-directed I/O 1-2
 debugging 1-2
 bound segment
 display_component_name 6-23
 print_bind_map 6-37
 error messages
 active_fnc_err_ 7-3
 condition_interpreter_ 7-14
 convert_status_code_ 7-18
 linkage section
 print_linkage_usage 6-39
 stack trace
 cu_\$stack_frame_ptr 7-25
 find_condition_info_ 7-42
 utilities
 stu_\$decode_runtime_value 7-129
 stu_\$find_block 7-127
 stu_\$find_header 7-126
 stu_\$find_runtime_symbol 7-128
 stu_\$get_implicit_qualifier 7-130
 stu_\$get_line 7-135
 stu_\$get_line_no 7-133
 stu_\$get_location 7-135
 stu_\$get_runtime_address 7-131
 stu_\$get_runtime_block 7-127
 stu_\$get_runtime_line_no 7-134
 stu_\$get_runtime_location 7-136
 stu_\$get_statement_map 7-137
 stu_\$offset_to_pointer 7-137
 stu_\$pointer_to_offset 7-138
 stu_\$remote_format 7-139

- default error handling
 - condition_interpreter_ 7-14
 - continue_to_signal_ 7-125
 - find_condition_info_ 7-42
 - signal_ 7-125
- default working directory
 - get_default_wdir_ 7-44
- definition
 - class three 1-9
 - format 1-7
 - implicit 1-26
- definition section 1-1, 1-5
 - header 1-7, 1-25
 - see also relocation information
- delete_record operation
 - see I/O: operations
- deleting
 - ACL entries
 - hcs_\$delete_dir_inacl_entries 7-57
 - hcs_\$delete_inacl_entries 7-58
 - mbx_delete_acl 6-29
 - msf_manager_\$acl_delete 7-114
 - directory
 - hcs_\$del_dir_tree 7-56
 - initial ACL entries
 - hcs_\$delete_dir_inacl_entries 7-57
 - hcs_\$delete_inacl_entries 7-58
 - mailbox
 - mbx_delete 6-28
 - mailbox ACL
 - mbx_delete_acl 6-29
 - multiple names
 - mbx_delete_name 6-31
 - search rules
 - hcs_\$initiate_search_rules 7-69
- descriptor
 - argument 2-15
 - decode_descriptor_ 7-35
- detach operation
 - see I/O: operations
- detach_iocb operation
 - see I/O: operations
- directory
 - access control
 - hcs_\$get_dir_ring_brackets 7-62
 - hcs_\$set_dir_ring_brackets 7-79
 - ACL for new directories
 - hcs_\$add_dir_inacl_entries 7-52
 - hcs_\$delete_dir_inacl_entries 7-57
 - hcs_\$list_dir_inacl 7-71
 - hcs_\$replace_dir_inacl 7-77
 - author
 - hcs_\$get_author 7-60
 - default working
 - get_default_wdir_ 7-44
 - deleting
 - hcs_\$del_dir_tree 7-56
 - home 3-3

- directory (continued)
 - listing contents
 - hcs_\$star_ 7-87
 - hcs_\$star_list_ 7-89
 - name manipulation
 - copy_names 6-22
 - move_names 6-36
 - quota
 - hcs_\$quota_move 7-75
 - hcs_\$quota_read 7-76
 - safety switch
 - hcs_\$get_safety_sw 7-66
 - hcs_\$set_safety_sw 7-85
 - working
 - get_default_wdir_ 7-44
- discretionary access control
 - see access control
- dynamic linking
 - see linking (interprocedure)

E

- EBCDIC
 - see character set
- entry operator 2-10
- entry point
 - bound
 - see segment: entry point bound
 - name
 - get_entry_name_ 7-45
 - status
 - print_link_info 6-38
- entry sequence 1-3, 1-24
- entry value
 - conversion to pointer
 - cu_\$decode_entry_value 7-30
 - generation from pointer
 - cu_\$make_entry_value 7-30
- entryname
 - conversion from equal name
 - get_equal_name_ 7-46
 - match with star name
 - match_star_name_ 7-107
- entry_op_ptr
 - see stack: header
- entry_ptr
 - see stack: frame
- equal convention
 - get_equal_name_ 7-46
- error handling
 - active_fnc_err_ 7-3
 - condition_interpreter_ 7-14
 (continued)

error handling (continued)
convert_status_code_ 7-18
error_table_compiler 6-24
find_condition_info_ 7-42
signal_ 7-125

error_output
see I/O: attachment

event channel
see interprocess communication

exec_com 3-3

expression word 1-10

external symbol 1-5, 2-10
elimination by binder 1-29
name
get_entry_name_ 7-45

F

fault
see condition

fault tag two (FT2) 1-14

fcbl
see file control block

file (multisegment)
see multisegment file

file control block
msf_manager_\$close 7-110
msf_manager_\$open 7-108
tssi_\$clean_up_file 7-158
tssi_\$finish_file 7-157
tssi_\$get_file 7-155

first reference trap 1-15

free storage
get_system_free_area_ 7-51

G

gate
entry point transfer vector 1-4

get_chars operation
see I/O: operations

get_line operation
see I/O: operations

H

handling
see condition

hardware
faults
see condition
machine language
alm 6-3
alm_abs 6-19
registers
see register

home directory 3-3

I

I/O
attach description 4-3, 4-7
attachment 3-2
cleanup
iox_\$destroy_iocb 7-96
control block 4-4
iox_\$find_iocb_n 7-97
iox_\$look_iocb 7-97
structure 4-2
data-directed 1-2
error messages
active_fnc_err_ 7-3
condition_interpreter_ 7-14
iox_\$err_not_attached 7-96
iox_\$err_not_closed 7-96
iox_\$err_not_open 7-96
iox_\$err_no_operation 7-96
offline (daemon)
dprint_ 7-37
iod_info_\$driver_access_name 7-94
iod_info_\$generic_type 7-94
system_info_\$io_prices 7-147
open data 4-4
open description 4-3, 4-8
operations 4-2, 4-7, 4-8, 4-9
synonym attachment 4-4, 4-6
iox_\$propagate 7-98

information
see status

initial command line 3-3

initial procedure
see process overseer

initproc
see process overseer

internal static offset table
(ISOT) 2-8

internal static storage
see static storage

internal storage 1-2, 1-14, 1-26

interprocess communication

event channel creation
ipc_\$create_ev_chn 7-99
ipc_\$dcl_event_call_channel 7-100
ipc_\$decl_ev_wait_chn 7-101
event channel deletion
ipc_\$delete_ev_chn 7-100
event channel management
ipc_\$cutoff 7-102
ipc_\$drain_chn 7-101
ipc_\$mask_ev_calls 7-103
ipc_\$read_ev_chn 7-105
ipc_\$reconnect 7-102
ipc_\$set_call_prior 7-103
ipc_\$set_wait_prior 7-102
ipc_\$unmask_ev_calls 7-104
request wakeup
timer_manager_\$alarm_wakeup 7-152
timer_manager_\$cpu_wakeup 7-153
timer_manager_
\$reset_alarm_wakeup 7-154
timer_manager_
\$reset_cpu_wakeup 7-153
send wakeup
hcs_\$wakeup 7-93
wait for wakeup
ipc_\$block 7-104
timer_manager_\$sleep 7-151

interrupt

communication
see interprocess communication
suspend execution
see condition

intersegment reference

see linking (interprocedure)

interuser communication

mailbox commands
mbx_add_name 6-26
mbx_create 6-27
mbx_delete 6-28
mbx_delete_acl 6-29
mbx_delete_name 6-31
mbx_list_acl 6-32
mbx_rename 6-33
mbx_set_acl 6-34
see also interprocess communication

iocb

see I/O: control block

iox_\$destroy_iocb 4-2

iox_\$find_iocb 4-2

iox_\$propagate

see I/O: synonym attachment

ISOT

see internal static offset
table (ISOT)

isot_ptr

see stack: header

L

languages

absentee compilation
alm_abs 6-19
assemblers
alm 6-3
command language
see command environment
status table
error_table_compiler 6-24
translator utilities
tssi_\$clean_up_file 7-158
tssi_\$clean_up_segment 7-157
tssi_\$finish_segment 7-156
tssi_\$get_file 7-155
tssi_\$get_segment 7-155

length of segment

see segment: length

libraries

search rules
hcs_\$get_search_rules 7-68
hcs_\$initiate_search_rules 7-69

link

author
hcs_\$get_author 7-60
name manipulation
copy_names 6-22
move_names 6-36
status
hcs_\$star_ 7-87
hcs_\$star_list_ 7-89

linkage offset table (LOT) 2-7

linkage section 1-2, 1-13, 1-26, 2-7
header 1-13

linking (interprocedure) 1-1
1-14, 1-26

elimination by binder 1-29
linkage pointer
generation by entry operator 2-10
printing linkage section map
print_linkage_usage 6-39
printing object segment linkage
print_link_info 6-38
self-referencing link 1-10
type 6 link
initialization structure 1-13

listener

called by
standard_default_handler_ 3-2
cu_\$cl 7-28
cu_\$get_cl 7-29
cu_\$set_cl 7-28

- listing
 - directory contents
 - hcs_\$star_ 7-87
 - hcs_\$star_list_ 7-89
 - initial ACL for new directories
 - hcs_\$list_dir_inacl 7-71
 - initial ACL for new segments
 - hcs_\$list_inacl 7-73
 - mailbox ACL
 - mbx_list_acl 6-32
 - multisegment file ACL
 - msf_manager_\$acl_list 7-111

- listing segment
 - translator interface
 - tssi_\$clean_up_file 7-158
 - tssi_\$clean_up_segment 7-157
 - tssi_\$finish_file 7-157
 - tssi_\$finish_segment 7-156
 - tssi_\$get_file 7-155
 - tssi_\$get_segment 7-155

login 3-1

login responder
see process overseer

LOT
see linkage offset table (LOT)

lot_ptr
see stack: header

lp
see register: pointer register
4 (PR4)

M

machine conditions

- examining
 - find_condition_info_ 7-42
- restarting
 - prepare_mc_restart_\$replace 7-121
 - prepare_mc_restart_\$retry 7-120
 - prepare_mc_restart_\$tra 7-121

machine language 1-2

- alm 6-3
- alm_abs 6-19

mailbox

- access control
 - mbx_delete_acl 6-29
 - mbx_list_acl 6-32
 - mbx_set_acl 6-34
- creating
 - mbx_create 6-27
- deleting
 - mbx_delete 6-28
- name manipulation
 - mbx_add_name 6-26
 - mbx_delete_name 6-31
 - mbx_rename 6-33

messages

- error
 - active_fnc_err_ 7-3
 - condition_interpreter_ 7-14
 - convert_status_code_ 7-18
 - error_table_compiler 6-24
 - find_condition_info_ 7-42
- message of the day 3-3
- ready
 - cu_\$get_ready_mode 7-20
 - cu_\$get_ready_proc 7-20
 - cu_\$ready_proc 7-19
 - cu_\$set_ready_mode 7-21
 - cu_\$set_ready_proc 7-20

mode
see access control

modes operation
see I/O: operations

moving

- names
 - copy_names 6-22
 - move_names 6-36

multiple names

- creating
 - mbx_add_name 6-26
- deleting
 - mbx_delete_name 6-31
- moving
 - copy_names 6-22
 - move_names 6-36

multisegment file

- access control
 - msf_manager_\$acl_add 7-113
 - msf_manager_\$acl_delete 7-114
 - msf_manager_\$acl_list 7-111
 - msf_manager_\$acl_replace 7-112
- bit count
 - msf_manager_\$adjust 7-110
 - msf_manager_\$open 7-108
- closing
 - msf_manager_\$close 7-110
- finding next component
 - msf_manager_\$get_ptr 7-109
- opening
 - msf_manager_\$open 7-108

N

name

- access class
 - system_info_\$category_names 7-148
 - system_info_\$level_names 7-148
- copying
 - copy_names 6-22
 - move_names 6-36
- equal
 - get_equal_name_ 7-46

(continued)

name (continued)
 external symbol (entry point)
 get_entry_name_ 7-45
 print_link_info 6-38
 installation
 system_info_\$installation_id 7-142
 system_info_\$titles 7-142
 mailbox
 mbx_add_name 6-26
 mbx_delete_name 6-31
 mbx_rename 6-33
 moving
 copy_names 6-22
 move_names 6-36
 reference 1-11
 star
 check_star_name_\$entry 7-12
 check_star_name_\$path 7-12
 hcs_\$star_ 7-87
 hcs_\$star_list_ 7-89
 match_star_name_ 7-107
 system version
 system_info_\$sysid 7-142
 new_proc 3-1
 next_stack_frame_ptr
 see stack: frame
 nonlocal transfer
 unwinder_ 7-159
 no_start_up attribute 3-3

O

object map 1-2, 1-22
 object segment 1-1, 1-19
 creation time 1-17
 format 1-1
 status
 display_component_name 6-23
 object_info_\$brief 7-115
 object_info_\$display 7-115
 object_info_\$long 7-116
 print_bind_map 6-37
 print_link_info 6-38
 symbol table
 stu_\$decode_runtime_value 7-129
 stu_\$find_block 7-127
 stu_\$find_header 7-126
 stu_\$find_runtime_symbol 7-128
 stu_\$get_implicit_qualifier 7-130
 stu_\$get_line 7-135
 stu_\$get_line_no 7-133
 stu_\$get_location 7-135
 stu_\$get_runtime_address 7-131
 stu_\$get_runtime_block 7-127
 stu_\$get_runtime_line_no 7-134
 stu_\$get_runtime_location 7-136
 stu_\$get_statement_map 7-137
 stu_\$offset_to_pointer 7-137

object segment (continued)
 stu_\$pointer_to_offset 7-138
 stu_\$remove_format 7-139
 translator interface
 tssi_\$clean_up_file 7-158
 tssi_\$clean_up_segment 7-157
 tssi_\$finish_file 7-157
 tssi_\$finish_segment 7-156
 tssi_\$get_file 7-155
 tssi_\$get_segment 7-155
 on_unit_rel_ptr
 see stack: frame
 open operation
 see I/O: operations
 opening
 multisegment file
 msf_manager_\$open 7-108
 operator 2-10
 call 2-10
 entry 2-10
 push 2-11
 return 2-11
 short return 2-11
 short_call 2-10
 operator_link_ptr
 see stack: frame
 operator_return_offset
 see stack: frame
 ordering archive components
 see sorting

P

parameter
 descriptor 1-3
 per-process data
 linkage section 1-2
 stack 2-1
 PIT
 see process initialization
 table (PIT)
 pl1_operators_ptr
 see stack: header
 pointer register
 see register
 position operation
 see I/O: operations
 prelinking
 bound segment 1-29

prev_stack_frame_ptr
 see stack: frame

prices
 system_info_\$abs_prices 7-146
 system_info_\$device_prices 7-145
 system_info_\$io_prices 7-147
 system_info_\$prices 7-144

printing
 offline
 dprint_ 7-37
 iod_info_\$driver_access_name 7-94
 iod_info_\$generic_type 7-94
 system_info_\$io_prices 7-147

privileges
 see access isolation mechanism (AIM)

procedure segment
 see object segment

process
 access privileges
 get_privileges_ 7-48
 creation 3-1
 synchronization
 see interprocess communication

process initialization table (PIT) 3-1

process overseer 3-1

protection rings
 see rings

punched cards
 offline output
 dprint_ 7-37
 iod_info_\$driver_access_name 7-94
 iod_info_\$generic_type 7-94
 system_info_\$io_prices 7-147

pure procedure
 see object segment

push_op_ptr
 see stack: header

put_chars operation
 see I/O: operations

Q

queue
 absentee
 alm_abs 6-19
 I/O daemon
 dprint_ 7-37
 system_info_\$io_prices 7-147

quit 3-3
 abort execution
 signal_ 7-125
 unwinder_ 7-159
 enabling 3-3
 handling 3-3
 continue_to_signal_ 7-125
 find_condition_info_ 7-42

quit_enable order
 see quit: enabling

quota
 storage
 hcs_\$quota_move 7-75
 hcs_\$quota_read 7-76

R

ready messages
 cu_\$get_ready_mode 7-20
 cu_\$get_ready_proc 7-20
 cu_\$ready_proc 7-19
 cu_\$set_ready_mode 7-21
 cu_\$set_ready_proc 7-20

read_key operation
 see I/O: operations

read_length operation
 see I/O: operations

read_record operation
 see I/O: operations

reference name 1-11

referencing_dir
 hcs_\$get_search_rules 7-68
 hcs_\$initiate_search_rules 7-69

register
 machine conditions
 condition_interpreter_ 7-14
 find_condition_info_ 7-42
 prepare_mc_restart_\$replace 7-121
 prepare_mc_restart_\$retry 7-120
 prepare_mc_restart_\$tra 7-121
 pointer register 0 (PRO)
 operator segment pointer
 2-10, 2-13
 pointer register 4 (PR4)
 linkage pointer 2-10, 2-13
 pointer register 6 (PR6)
 stack frame pointer 2-13
 pointer register 7 (PR7)
 stack base pointer 2-13
 saving registers 2-10, 2-11

relocation information 1-2, 1-20, 1-25
 text section 1-24

- renaming
 - mailbox
 - mbx_rename 6-33
- restarting
 - fault
 - prepare_mc_restart_\$replace 7-121
 - prepare_mc_restart_\$retry 7-120
 - prepare_mc_restart_\$tra 7-121
- return_op_ptr
 - see stack: header
- return_ptr
 - see stack: frame
- rewrite_record operation
 - see I/O: operations
- rings 2-1
 - access control (ring brackets)
 - hcs_\$get_dir_ring_brackets 7-62
 - hcs_\$get_ring_brackets 7-65
 - hcs_\$set_dir_ring_brackets 7-79
 - hcs_\$set_ring_brackets 7-84
 - set_ring_brackets 6-44
 - execution
 - get_ring_ 7-50
 - hcs_\$get_dir_ring_brackets 7-62
 - hcs_\$get_ring_brackets 7-65
 - validation level
 - cu_\$level_get 7-30
 - cu_\$level_set 7-29
- runtime operators
 - see operator
- runtime symbol table
 - see symbol table

S

- safety switch
 - see segment: safety switch
- sb
 - see register: pointer register 7 (PR7)
- sct_ptr
 - see stack: header
- search rules
 - home_dir 3-3
 - obtaining
 - hcs_\$get_search_rules 7-68
 - setting
 - hcs_\$initiate_search_rules 7-69
- seek_key operation
 - see I/O: operations
- segdef
 - see external symbol

- segment
 - access control
 - hcs_\$get_ring_brackets 7-65
 - hcs_\$set_ring_brackets 7-84
 - set_ring_brackets 6-44
 - ACL for new segments
 - hcs_\$add_inacl_entries 7-54
 - hcs_\$delete_inacl_entries 7-58
 - hcs_\$list_inacl 7-73
 - hcs_\$replace_inacl 7-78
 - author
 - hcs_\$get_author 7-60
 - bit count author
 - hcs_\$get_bc_author 7-61
 - entry point bound 1-4
 - hcs_\$set_entry_bound 7-80
 - hcs_\$set_entry_bound_seg 7-81
 - length
 - print_link_info 6-38
 - maximum length
 - hcs_\$get_max_length 7-63
 - hcs_\$get_max_length_seg 7-64
 - hcs_\$set_max_length 7-82
 - hcs_\$set_max_length_seg 7-83
 - name manipulation
 - copy_names 6-22
 - move_names 6-36
 - safety switch
 - hcs_\$get_safety_sw 7-66
 - hcs_\$get_safety_sw_seg 7-67
 - hcs_\$set_safety_sw 7-85
 - hcs_\$set_safety_sw_seg 7-86
- self-referencing link 1-10
- self-relative
 - see relocation information
- seven-punch cards
 - dprint_ 7-37
- shift
 - accounting
 - system_info_\$shift_table 7-146
- short_return_op_ptr
 - see stack: header
- shutdown
 - last system shutdown
 - system_info_\$last_shutdown 7-147
 - next system shutdown
 - system_info_\$next_shutdown 7-144
- signal_ptr
 - see stack: header
- sorting
 - archive
 - archive_sort 6-21
 - reorder_archive 6-40
- source map
 - see object segment

sp
 see register: pointer register
 6 (PR6)

special directories
 default working directory
 get_default_wdir_ 7-44
 home directory 3-3
 search rules
 hcs_\$get_search_rules 7-68
 hcs_\$initiate_search_rules 7-69
 working directory
 get_default_wdir_ 7-44

stack 2-1
 examining frame
 stu_\$decode_runtime_value 7-129
 stu_\$find_runtime_symbol 7-128
 stu_\$get_implicit_qualifier 7-130
 stu_\$get_runtime_address 7-131
 stu_\$get_runtime_block 7-127
 stu_\$offset_to_pointer 7-137
 stu_\$pointer_to_offset 7-138
 stu_\$remote_format 7-139
 frame 1-24, 2-1, 2-5, 2-11
 creating 2-11
 frame pointer
 cu_\$stack_frame_ptr 7-25
 frame size
 cu_\$stack_frame_size 7-25
 header 2-1, 2-2, 2-11
 releasing frame
 unwinder_ 7-159

stack_begin_ptr
 see stack: header

stack_end_ptr
 see stack: header

standard_default_handler_ 3-2

star convention
 check_star_name_\$entry 7-12
 check_star_name_\$path 7-12
 get_equal_name_ 7-46
 hcs_\$star_ 7-87
 hcs_\$star_list_ 7-89
 match_star_name_ 7-107

start_up.ec 3-3

statement map
 stu_\$get_statement_map 7-137

static section 1-2, 1-13, 1-27
 print_linkage_usage 6-39

static storage 1-2

static_ptr
 see stack: frame

status
 access control
 msf_manager_\$acl_list 7-111

status (continued)
 directory
 hcs_\$get_author 7-60
 hcs_\$get_dir_ring_brackets 7-62
 hcs_\$get_safety_sw 7-66
 hcs_\$quota_read 7-76
 directory contents
 hcs_\$star_ 7-87
 hcs_\$star_list_ 7-89
 messages
 active_fnc_err_ 7-3
 condition_interpreter_ 7-14
 convert_status_code_ 7-18
 error_table_compiler 6-24
 object segment
 object_info_\$brief 7-115
 object_info_\$display 7-115
 object_info_\$long 7-116
 segment
 hcs_\$get_author 7-60
 hcs_\$get_bc_author 7-61
 hcs_\$get_max_length 7-63
 hcs_\$get_max_length_seg 7-64
 hcs_\$get_ring_brackets 7-65
 hcs_\$get_safety_sw 7-66
 hcs_\$get_safety_sw_seg 7-67
 system information
 system_info_\$abs_prices 7-146
 system_info_\$device_prices 7-145
 system_info_\$installation_id 7-142
 system_info_\$io_prices 7-147
 system_info_\$last_shutdown 7-147
 system_info_\$next_shutdown 7-144
 system_info_\$prices 7-144
 system_info_\$shift_table 7-146
 system_info_\$sysid 7-142
 system_info_\$timeup 7-143
 system_info_\$titles 7-142
 system_info_\$users 7-143

storage
 automatic 2-1
 based
 get_system_free_area_ 7-51
 static 2-8

storage quota
 hcs_\$quota_move 7-75
 hcs_\$quota_read 7-76

subroutine
 calling 2-9
 cu_\$gen_call 7-26
 cu_\$ptr_call 7-26

subsystem 3-1

symbol block 1-2
 binder 1-29
 header 1-17

symbol section 1-17, 1-27

symbol table
 using
 stu_\$decode_runtime_value 7-129
 stu_\$find_block 7-127
 (continued)

symbol table (continued)
 stu_\$find_header 7-126
 stu_\$find_runtime_symbol 7-128
 stu_\$get_implicit_qualifier 7-130
 stu_\$get_line 7-135
 stu_\$get_line_no 7-133
 stu_\$get_location 7-135
 stu_\$get_runtime_address 7-131
 stu_\$get_runtime_block 7-127
 stu_\$get_runtime_line_no 7-134
 stu_\$get_runtime_location 7-136
 stu_\$get_statement_map 7-137
 stu_\$offset_to_pointer 7-137
 stu_\$pointer_to_offset 7-138
 stu_\$remote_format 7-139

symbol tree 1-2

symbolic offset 1-11

symbol_table
 see definition: implicit

synonym
 see I/O: synonym attachment

system information
 see status: system information

T

temporary storage
 see stack: frame

text section 1-1

time
 last system shutdown
 system_info_\$last_shutdown 7-147
 next system shutdown
 system_info_\$next_shutdown 7-144
 process CPU usage 3-2
 system_info_\$abs_prices 7-146
 system_info_\$prices 7-144
 timer_manager_\$cpu_call 7-152
 timer_manager_
 \$cpu_call_inhibit 7-153
 timer_manager_\$cpu_wakeup 7-153
 timer_manager_
 \$reset_cpu_call 7-153
 timer_manager_
 \$reset_cpu_wakeup 7-153
real time 3-2
 system_info_\$prices 7-144
 timer_manager_\$alarm_call 7-151
 timer_manager_
 \$alarm_call_inhibit 7-152
 timer_manager_\$alarm_wakeup 7-152
 timer_manager_
 \$reset_alarm_call 7-154
 timer_manager_
 \$reset_alarm_wakeup 7-154
system startup time
 system_info_\$timeup 7-143

translators
 storage system
 tssi_\$clean_up_file 7-158
 tssi_\$clean_up_segment 7-157
 tssi_\$finish_file 7-157
 tssi_\$finish_segment 7-156
 tssi_\$get_file 7-155
 tssi_\$get_segment 7-155
 version number 1-2, 1-17

trans_op_tv_ptr
 see stack: header

trap pair 1-12

tty_ 3-2

type 6 link
 initialization structure 1-13

type pair 1-10

U

unclaimed signal
 see condition: any_other

unsnapped link 1-14

unwinder_ptr
 see stack: header

user
 parameters 3-1
 get_privileges_ 7-48

user_i/o
 see I/O: attachment

user_info_ 3-1

user_input
 see I/O: attachment

user_output
 see I/O: attachment

user_real_init_admin_
 see process: creation

V

validation level
 directory
 hcs_\$get_dir_ring_brackets 7-62
 hcs_\$set_dir_ring_brackets 7-79
 obtaining
 cu_\$level_get 7-30
(continued)

validation level (continued)

segment

hcs_\$get_ring_brackets 7-65

hcs_\$set_ring_brackets 7-84

setting

cu_\$level_set 7-29

version of translator 1-2

v_process_overseer attribute

see process overseer

working directory

default

get_default_wdir_ 7-44

search rules

hcs_\$get_search_rules 7-68

hcs_\$initiate_search_rules 7-69

write_record operation

see I/O: operations

W

wakeup

process CPU usage

timer_manager_\$cpu_wakeup 7-153

timer_manager_

\$reset_cpu_wakeup 7-153

real time

timer_manager_\$alarm_wakeup 7-152

timer_manager_

\$reset_alarm_wakeup 7-154

see also interprocess communication

HONEYWELL INFORMATION SYSTEMS
Technical Publications Remarks Form

TITLE

SERIES 60 (LEVEL 68)
MULTICS PROGRAMMERS' MANUAL
SUBSYSTEM WRITERS' GUIDE

ORDER NO.

AK92, REV. 1

DATED

SEPTEMBER 1975

ERRORS IN PUBLICATION

[Empty box for errors in publication]

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

[Empty box for suggestions for improvement to publication]



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

FROM: NAME _____

DATE _____

TITLE _____

COMPANY _____

ADDRESS _____

CUT ALONG LINE

PLEASE FOLD AND TAPE -

NOTE: U. S. Postal Service will not deliver stapled forms

FIRST CLASS
PERMIT NO. 39531
WALTHAM, MA
02154

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

Postage Will Be Paid By:

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTENTION: PUBLICATIONS, MS 486

Honeywell

FOLD ALONG LINE

FOLD ALONG LINE

FOLD ALONG LINE