Honeywell

MULTICS PROGRAMMERS' MANUAL SUBSYSTEM WRITERS' GUIDE ADDENDUM C

SERIES 60 (LEVEL 68)

SOFTWARE

SUBJECT:

Additions and Changes to the Reference Guide for Advanced Multics Users, Writing Their Own Subsystems

SPECIAL INSTRUCTIONS:

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

Reference Guide	Order No. AG91
Commands and Active Functions	Order No. AG92
Subroutines	Order No. AG93
Subsystem Writers' Guide	Order No. AK92
Peripheral Input/Output	Order No. AX49

This is the third addendum to AK92, Revision 1, dated September 1975.

Insert the attached pages into the manual according to the collating instructions on the back of this cover.

The major change to Section I is the addition of a "Definition Hash Table."

Section II has changes to the stack header.

Section IV has one major addition called "Performing Control Operations from Command Level."

In order to consolidate subroutine descriptions into one manual, cu_, iox_, tty_, and vfile_ have been moved to the MPM Subroutines. The subroutine stu_ has been taken out of this manual and will be in the next addendum for the Multics System Programming Tools, Order No. AZ03. The items listed below are new to this manual and do not contain change bars:

delete_volume_quota	get_lock_id_
set_ttt_path	hcs_$get_link_target
component_info_	hcs_$get_user_effmode
cross_ring_	mhcs_$get_seg_usage
cross_ring_io_$allow_cross	pl1_io_
cv_dec_	suffixed_name_
cv_dec_check_	ttt_info_
dump_segment_	

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

Section VIII is a new section that has been created for data base information. It contains a new data base, time_table_$zones, and also sys_info, which was moved out of the subroutine section.

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

SOFTWARE SUPPORTED:

Multics Software Release 6.0

DATE:

November 1977

ORDER NUMBER:

AK92C, Rev. 1

20624
1578
Printed in U.S.A.

COLLATING INSTRUCTIONS

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

Remove	Insert
iii through ix	iii through ix
1-5 through 1-8	1-5 through 1-8
	1-12.1, 1-12.2 1-12.3, blank
1-13 through 1-16	1-13 through 1-16
1-21 through 1-24	1-21 through 1-24
1-31	1-31
2-1 through 2-8	2-1 through 2-8
3-3	3-3
4-1 through 4-9	4-1 through 4-11
5-1 through 5-6	5-1 through 5-7
6-3, 6-4	6-3, 6-4
6-11, 6-12	6-11, 6-12 6-12.1, blank
6-15 through 6-19	6-15 through 6-19 6-19.1 through 6-19.11
	6-26.1, blank
6-35, 6-36	6-35, 6-36
	6-56.1, blank
7-3 through 7-160	7-3 through 7-190
	8-1 through 8-5

File No.: 1L13

11/77

AK92C

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 sybsystem changes when these interfaces change. Thus, the sybsystem 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 knowledgable 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

ILLUSTRATIONS

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 $\underline{n}$ words of a gate segment with $\underline{n}$ entry points must be an entry point transfer vector. That is, the kth word ($0 \leq k \leq \underline{n}-1$) must be a transfer instruction to the kth 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 kth entry point is the offset of the kth transfer instruction (i.e., word k of the segment) rather than the offset of the code_sequence item of the kth 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 $\underline{n}$ 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.

Optionally, the definition section can contain a definition hash table. If present, the hash table is used by the linker to expedite the search for a definition.

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 Section IV in 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. Unused bits of the last word of the string must be zero. Such a structure is referred to as an acc string.

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

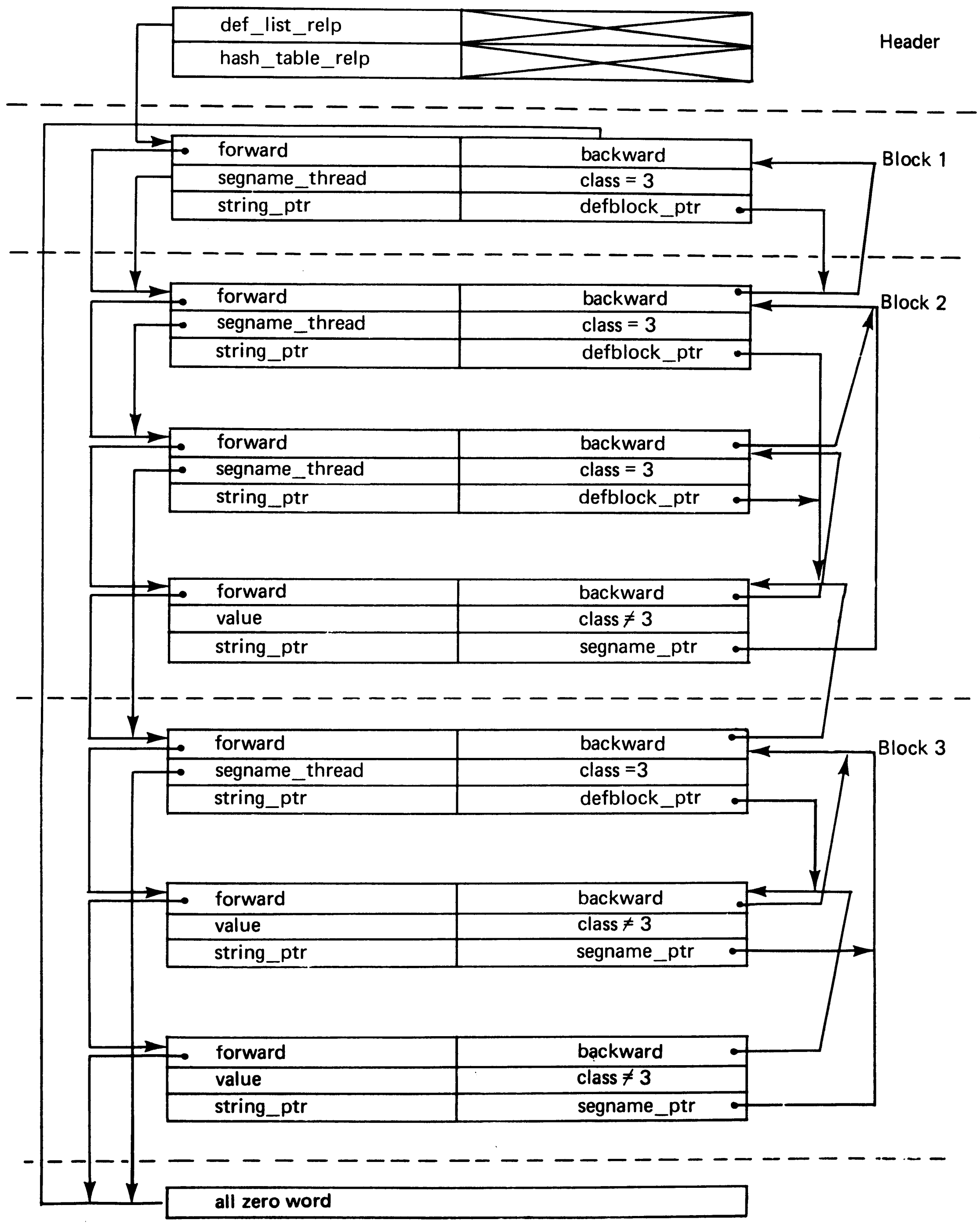


Figure 1-1. Sample Definition List

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 hash_table_relp   bit(18) unaligned,
    2 unused            bit(18) 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. hash_table_relp — is a relative pointer to the beginning of the definition hash table. If no definition hash table is present, this pointer must be "0"b.

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

4. 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_relp 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_relp — 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"b.

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"b.

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:

   myself|0+expression,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.

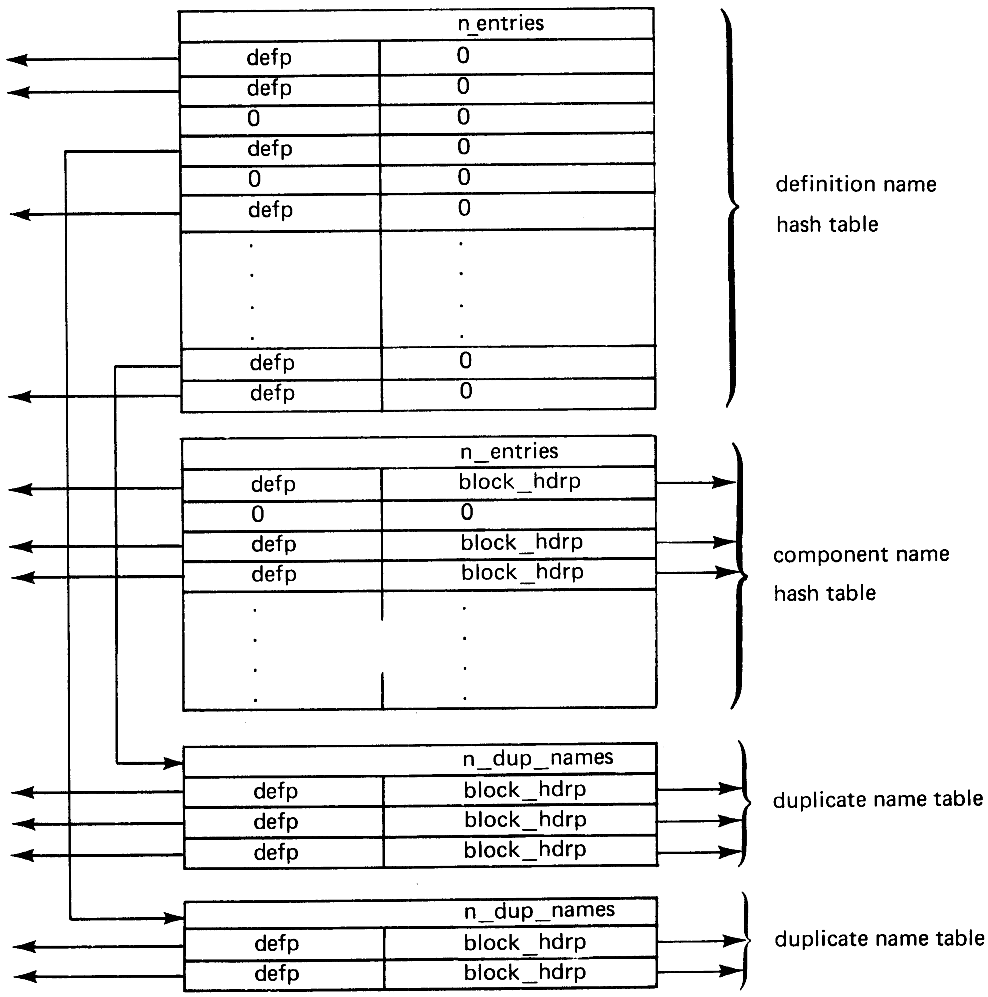


Figure 1-2. Definition Hash Table

Definition Hash Table

A definition hash table may be present in the definition section of an object segment. In its basic form, the definition hash table contains an array of pointers to definitions. The definition hashing algorithm selects a particular pointer. If the selected pointer does not point to the desired definition, a linear search is then performed until the appropriate definition is found or a zero pointer is encountered. The initial hash code is generated by taking the remainder of the first word of the definition name (the count and first three characters of the "acc" format string) divided by the size of the hash table. The hash table size is such that it is never more than 80% full.

In bound segments, different components may contain definitions with identical names. In this case, a second hash table is required in order to resolve ambiguities. In addition to this second hash table, a duplicate name table must be provided for each duplicated definition name.

The format of the tables described above is shown in Figure 1-2 and is described below:

The definition name hash table is pointed to by a relative pointer in the definition section header. It must contain one nonzero entry for each non-class-3 definition name.

```
dcl 1 defht      aligned,
    2 n_entries  fixed bin,
    2 table      (n refer (defht.n_entries)),
      (3 defp    bit(18),
       3 unused  bit(18)) unal;
```

where:

1. n_entries — is the number of elements in the hash table.

2. defp — is an array of pointers to non-class-3 definitions. In the case of a duplicated definition name, a particular defp does not point directly to a definition, but rather to a duplicate name table (see below).

A component name hash table is present only if duplicated definition names are present in a bound segment. It must immediately follow the definition hash table. There is one entry in this hash table for each bound segment component name and synonym (i.e., for each class-3 definition).

```
dcl 1 compht            aligned,
    2 n_entries         fixed bin,
    2 table             (nrefer (compht.n_entries)),
      (3 defp           bit(18),
       3 block_hdrp     bit(18)) unaligned;
```

where:

1. n_entries — is the number of elements in the component name hash table.

2. table — contains one nonzero element for each class-3 definition.

3. defp — is a relative pointer to a class-3 definition.

4. block_hdrp — is a relative pointer to the first class-3 definition of the definition block containing the definition pointed to by defp.

A duplicate name table must be supplied for each duplicated definition name. Each table has one entry for each instance of the duplicated name. The definition searching algorithm can determine whether the relative pointer retrieved from the definition hash table points to a definition or to a duplicate name table by examining the left half of the first word pointed to. A definition never contains a zero forward_thread, while a duplicate name table is never nonzero in the left half of the first word.

```
dcl 1 dupt              aligned,
    2 n_dup_names       fixed bin,
    2 table             (n refer (dupt.n_dup_names)),
      (3 defp           bit(18),
       3 block_hdrp     bit(18)) unaligned;
```

where:

1. n_dup_names — is the number of instances of a given duplicated name.

2. table — contains one element for each instance of the duplicated name.

3. defp — is a pointer to a non-class-3 definition.

4. block_hdrp — is a pointer to the first class-3 definition of the definition block containing the non-class-3 definition.

Definition searching with a definition hash table is done by first searching for the definition name. If no duplicate name table is encountered, no ambiguity exists and the correct definition is quickly found. If a duplicate name table is encountered, the component name hash table must be searched. Then, a linear search is done on the duplicate name table to match a block_hdrp with the block_hdrp in the component name hash table.

STRUCTURE OF THE STATIC SECTION

The static section is unstructured.

This page intentionally left blank.

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 ignore1           bit(6) unaligned,
    2 run_unit_depth    fixed bin(5) 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. run_unit_depth — must be 0 in a generated (unsnapped) link. When the link is snapped, this field is filled in with the number of the current run unit level.

4. tag — is a constant (46)8 that represents the hardware fault tag 2 and distinctly identifies an unsnapped link. The snapped link (ITS pair) has a distinct (43)8 tag. See "Simulated Fault" in Section VII of the MPM Reference Guide.

5. expression_relp — is an offset (relative to the base of the definition section) to the expression word for this link.

6. ignore2 — is reserved for future use and must be "0"b.

7. 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 Section IV in 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.

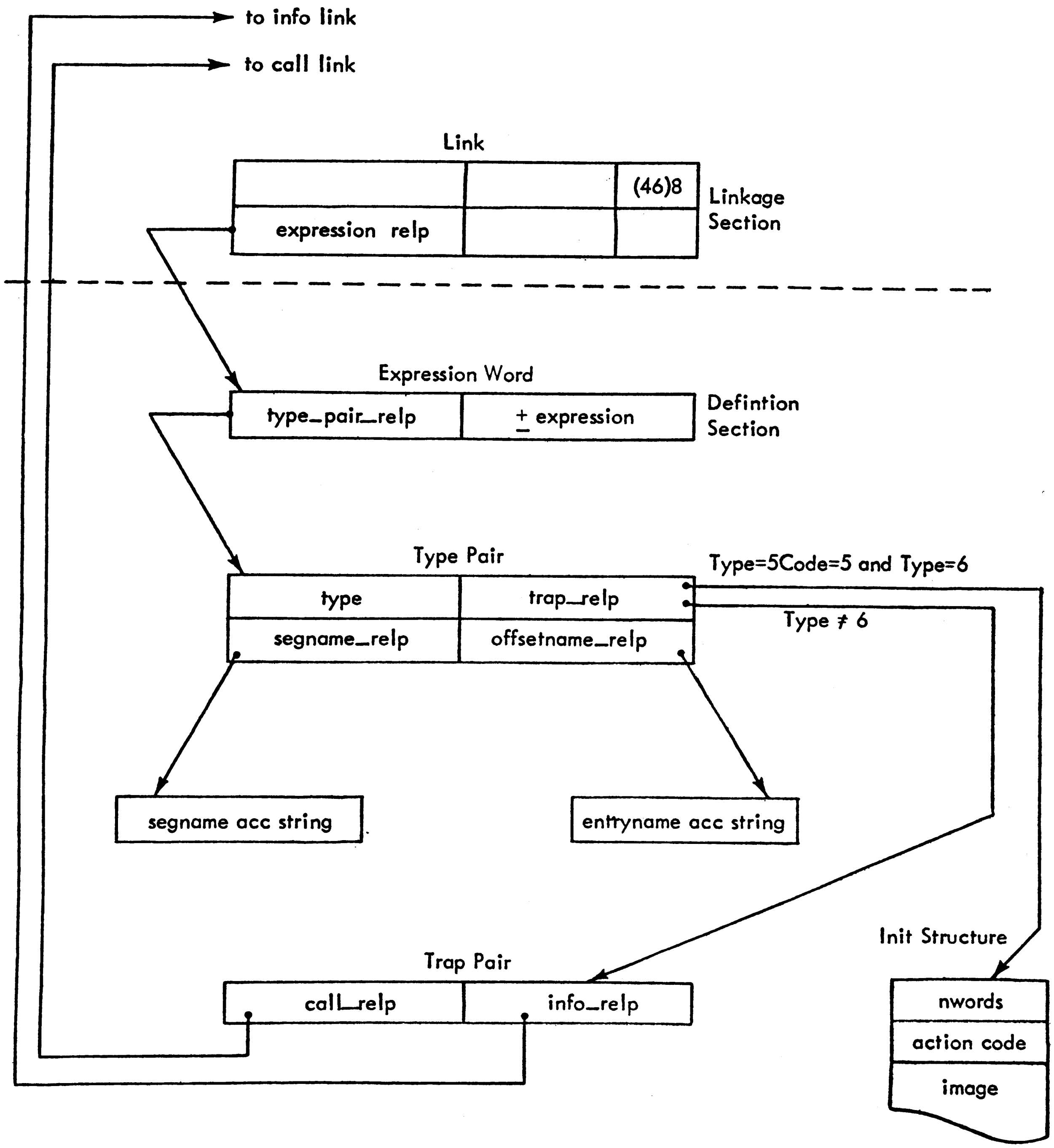


Figure 1-2. Structure of a Link

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 perprocess_static  bit(1) unaligned,
      3 unused             bit(29) 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"b.

17. bound — indicates if the object segment is a bound segment.
    - "1"b the object segment is a bound segment
    - "0"b 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"b the object segment is relocatable
    - "0"b the object segment is not relocatable

19. procedure — indicates whether this is an executable object segment.
    - "1"b this is an executable object segment
    - "0"b this is not an executable object segment

20. standard — indicates whether the object segment is in standard format.
    - "1"b the object segment is in standard format
    - "0"b the object segment is not in standard format

21. separate_static — indicates whether the static section is separate from the linkage section.
    - "1"b the static section is separate from the linkage section
    - "0"b the static section is not separate from the linkage section

22. 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

23. perprocess_static — indicates whether the static section should be reinitialized for a run unit.
    - "1"b static section is used as is
    - "0"b static section is per run unit

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

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.

text section	text for component 1 text for component 2 . . . text for component n
definition section	
linkage section	linkage header int. static for component 1 int. static for component 2 . . . int. static for component n links
symbol section	first reference trap symbol block for binder symbol block for component 1 . . . symbol block for component n
object map	

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 binder's symbol block) 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 binder's symbol block) 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.

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.

Offset					
+0	Reserved			Old Lot Pointer	Combined Static Pointer
+8	Combined Linkage Pointer	Max Lot Size	Run Unit Depth	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	Reference Name Table Pointer		Event Channel Table Pointer	Assign Linkage Pointer
+56	Reserved				
+64					

Figure 2-1. Stack Header Format

```
dcl 1 stack_header based       aligned,
    2 pad1(4)                  fixed bin,
    2 old_lot_ptr              ptr,
    2 combined_stat_ptr        ptr,
    2 clr_ptr                  ptr,
    2 max_lot_size             fixed bin(17) unaligned,
    2 run_unit_depth           fixed bin(17) unaligned,
    2 cur_lot_size             fixed bin(17) unaligned,
    2 pad2                     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 rnt_ptr                    ptr,
2 ect_ptr                    ptr,
2 assign_linkage_ptr         ptr,
2 pad3(8)                    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. run_unit_depth — is the current run unit level.

7. cur_lot_size — is the current number of words (entries) in the LOT and ISOT.

8. pad2 — 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. rnt_ptr — points to the reference name table (RNT).

29. ect_ptr — points to the event channel table (ECT).

30. assign_linkage_ptr — points to the area used by certain critical system programs whose operations must not be modified by run unit. This pointer initially points to the same area as stack_header.clr_ptr but is not changed by the run unit mechanism.

31. pad3 — 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.

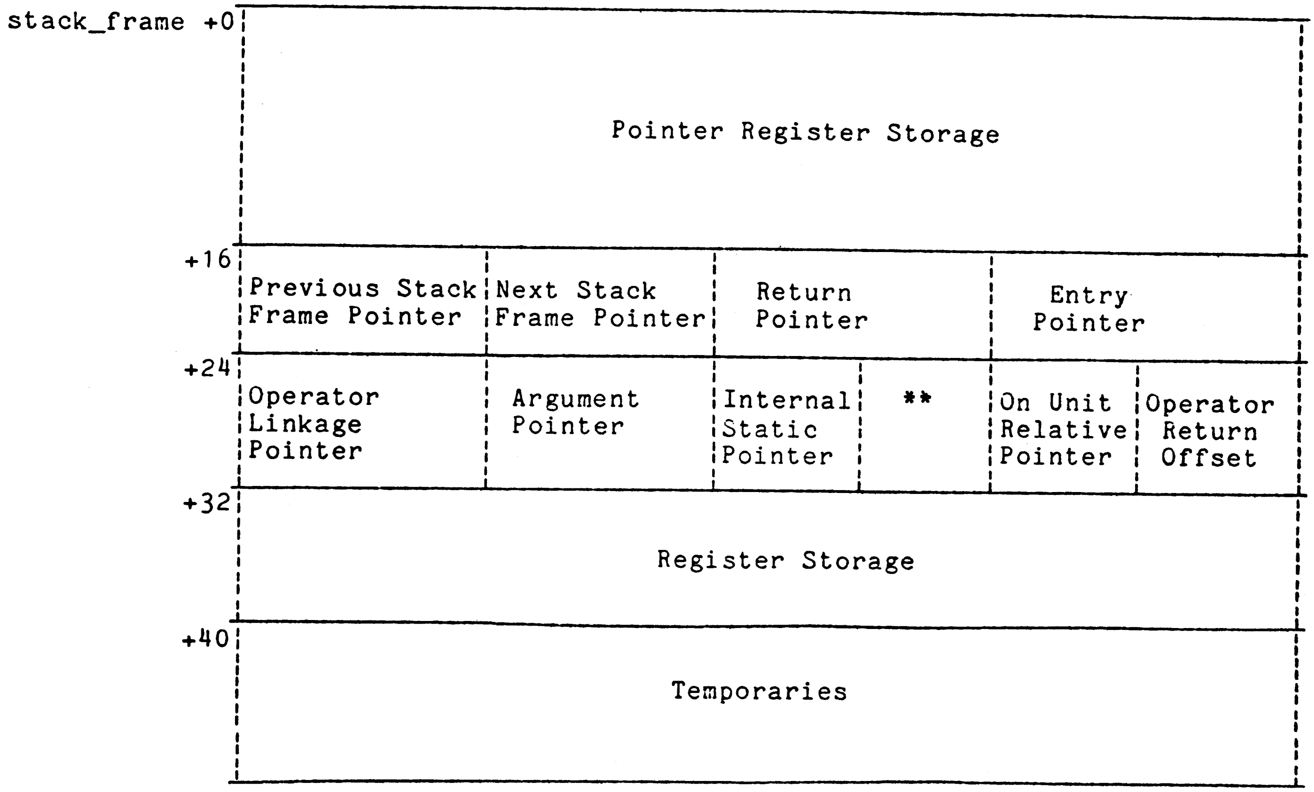
The PL/I and FORTRAN compilers use slightly different operators that perform equivalent and compatible functions. All supported translators, however, depend on the effects generated by these operators.

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.

If one of the slots in the linkage_ptr array contains all 0's except for "111"b in the high-order three bits (a lot fault), the segment number associated with the slot corresponds to a known segment that either does not have a linkage section or whose linkage section has not been combined (i.e., the segment has not been executed).

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:

          spri     6|0
          sreg     6|32
          epp0     arglist
          epp2     entrypoint
          tsp4     7|call_op,*
                                         spri4    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,*
                                         spri6    7|stack_end_ptr
                                         epp6     6|prev_sp,*
                                         epbp7    6|0
                                         epp0     6|op_ptr,*
                                         ldi      6|return_ptr+1
                                         rtcd     6|return_ptr

short_return:

          tra      7|short_return_op,*
                                         epbp7    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,*
                                         spri2    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.

Offset		
0	arg_count	code
1	desc_count	0
2	Pointer to argument 1	
4	Pointer to argument 2	
	. . .	
2*n	Pointer to argument n	
	Optional pointer to stack frame of containing block	
	Pointer to descriptor 1	
	Pointer to descriptor 2	
	. . .	
	Pointer to descriptor n	

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, absentee, or daemon.

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 returned to 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 OF 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, which does not have an entry variable since there can be only one attach entry point in an I/O module. 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);
```

can be thought of as:

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

Certain system routines make the latter call directly, without going through the appropriate iox_ subroutine; all other routines must call the iox_ subroutine, as the internal representation of the control block may change.

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 V 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 detached.

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 last switch in a chain of switches that have been connected to each other by the syn_ I/O module. (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 actual 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 any other control blocks that have been attached as synonyms to the actual I/O control block.

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 V 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 iocb_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 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 is not and 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.

5. Call iox_$propagate.

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. The following is a list of possible I/O operations:

    detach_iocb
    open
    close
    get_line
    get_chars
    put_chars
    modes
    position
    control
    read_record
    write_record
    rewrite_record
    delete_record
    seek_key
    read_key
    read_length

3. If either the modes operation or the control operation is enabled with the I/O switch attached but not enabled when the switch is 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.

6. Call iox_$propagate.

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 not enabled with the switch open and should be enabled with the switch closed, set iocb_ptr->iocb.actual_iocb_ptr->iocb.op, where op is modes or control.

If the operation is not 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.

6. Call iox_$propagate.

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.

4. Call iox_$propagate.

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. In this situation, the state of the I/O switch must not be changed.

Performing Control Operations From Command Level

Most of the operations supported by an I/O module may be used directly from command level by using the io_call command (see the MPM Commands). When a control operation requires an info structure see iox_$control, MPM Subroutines. A special interface the "io_call" order, is used to make these control operations from command level possible. All standard I/O modules that implement control operations requiring info structures should implement this interface, as described below.

When an io_call command of the form:

io_call control switch_name {optional_args} is issued, the io_call command performs an "io_call" control operation to the switch specified using the following info structure (found in io_call_info.incl.pl1):

```
dcl 1 io_call_info          aligned based (io_call_infop),
    2 version               fixed bin,
    2 caller_name           char(32),
    2 order_name            char(32),
    2 report                entry options (variable),
    2 error                 entry options (variable),
    2 af_returnp            ptr,
    2 af_returnl            fixed bin,
    2 fill (5)              bit(36),
    2 nargs                 fixed bin,
    2 max_arglen            fixed bin,
    2 args                  (0 refer  (io_call_info.nargs))
                               char (0 refer (io_call_info.max_arglen))
                               varying;
```

where:

1. version
   is the version number of this structure, currently 1.

2. caller_name
   is the name of the caller (normally io_call) to be used in any error message or output.

3. order_name
   is the order specified in the command line.

4. report
   is an entry like ioa_ to be called to report the results of the order.

5. error
   is an entry like com_err_ to be called to report any errors.

6. af_returnp
   is a pointer to the active function return string if the io_call command was invoked as an active function.

7. af_returnl
   is the maximum length of the active function return string.

8. nargs
   is the number of optional_args specified in the command line.

9. max_arglen
   is the length of the longest argument.

10. args
    is an array of the actual arguments from the command line.

The I/O module, upon receipt of an io_call order, should do the following:

1. If io_call_info.order_name specifies an order that requires an info structure with input values, the I/O module should use io_call_info.args to determine what data should be placed into the info structure. Once the structure is complete, the I/O module should call iox_$control, passing it io_call_info.order_name and a pointer to the info structure just created. Exactly how io_call_info.args is to be interpreted in order to build the info structure depends on the I/O module and what order is being performed. This should be documented along with the I/O module.

2. If io_call_info.order_name specifies an order that requires an info structure with output values, the I/O module should call iox_$control passing it io_call_info.order_name and a pointer to a structure of the appropriate kind. Then, using io_call_info.report, the I/O module should display the results of the control operation in some meaningful way. It is possible in this case that io_call_info.args could be used for control arguments to determine exactly what will be displayed. As in input type orders, the interpretation of these arguments is completely at the discretion of the I/O module.

3. If io_call_info.order_name specifies an order that does not require an info structure, or is an invalid order, then the I/O module should return error_table_$no_operation. The io_call command, seeing this code, will call iox_$control again, this time passing the original control order name, and a null info_ptr.

4. If the I/O module detects an error in handling an io_call order, it must do one of two things. First, it may return an error code, in which case io_call prints an error message. Secondly, it may call io_call_info.error (used like the com_err_ subroutine) to report the error directly. In this case, a zero error code should be returned to the caller. The latter choice is recommended, especially in cases where the I/O module can print a more informative error message.

I/O modules that do not support control operations that require info structures need not implement the io_call order at all. The io_call order can be rejected along with all other invalid orders in which case the order is performed with a null info_ptr by the io_call command as described in item 3 above.

Control operations can also be performed through the active function interface of the io_call command. In this case, the mechanism is basically the same with the following differences:

1. The order issued by the io_call command is io_call_af, not io_call.

2. Instead of printing a result, the I/O module should store its result in the varying string defined by io_call_info.af_returnp and io_call_info.af_returnl.

The io_call_af order should only be supported for orders that have meaning as an active function. As in the io_call order, the interpretation of io_call_info.args is completely up to the I/O module.

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. The following modifications to the I/O control block of the actual I/O switch can be made:

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 unrecoverable 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
Input/Output System Control
Language Translators, Compilers, Assemblers, and Interpreters
Object Segment Manipulation
Storage System, Access Control and Rings of Protection
Storage System, Directory Manipulation
Storage System, Logical Volumes
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

Input/Output System Control

set_ttt_path	changes pathname of terminal type table

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 and Rings of Protection

set_ring_brackets	changes ring brackets of segment

Storage System, Logical Volumes

delete_volume_quota	deletes a quota account for a logical volume and is used by volume executives

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
Formatted Output Facilities
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
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_dec_	converts an ASCII representation of a decimal integer to fixed binary(35)
cv_dec_check_	same as cv_dec_ except that a code is returned indicating the possibility of a conversion error
cv_entry_	converts a virtual entry to an entry value
cv_hex_	converts an ASCII representation of a hexadecimal integer to fixed binary(35)
cv_hex_check_	same as cv_hex_ except that a code is returned indicating the possibility of a conversion error
cv_oct_	converts an ASCII representation of an octal integer to fixed binary(35) of an octal integer.
cv_oct_check_	same as cv_oct_ except that a code is returned indicating the possibility of a conversion error
cv_ptr_	converts a virtual pointer to a pointer value
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

Formatted Output Facilities

dump_segment_	prints a dump formatted the same way as dump_segment command

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
pl1_io_	extracts information about PL/I
ttt_info_	extracts information from the terminal type table (TTT) files not available within the language

Miscellaneous Procedures

decode_descriptor_	extracts information from argument descriptors
get_privileges_	returns process' access privileges
system_info_	provides user with information on system parameters

Object Segment Manipulation

component_info_	returns information similar to object_info_ information about a component of a bound segment
object_info_	prints structural and identifying information extracted from object segment
tssi_	simplifies use of storage system by language translators

Process Synchronization

get_lock_id_	returns a 36-bit unique identifier to be used in setting locks
hcs_$wakeup	sends interprocess communication wakeup to blocked process over specified event channel
ipc_	user interface to Multics interprocess communication facility

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
cross_ring_	allows an outer ring to attach to a preexisting switch in an inner ring and perform I/O operations
cross_ring_io_$allow_cross	allows use of an I/O switch via cross_ring_ attachments from an outer ring
get_privileges_	returns process' access privileges
get_ring_	returns number of current protection ring
hcs_$add_dir_inacl_entries hcs_$add_inacl_entries	adds specified access modes to initial ACL for segments or directories
hcs_$delete_dir_inacl_entries hcs_$delete_inacl_entries	deletes specified entries from initial ACL for segments or directories
hcs_$get_dir_ring_brackets hcs_$get_ring_brackets	returns ring brackets for specified segment or subdirectory
hcs_$list_dir_inacl hcs_$list_inacl	returns all or part of initial ACL for segments or directories
hcs_$replace_dir_inacl hcs_$replace_inacl	replaces initial ACL with user-provided one for segments or directories
hcs_$set_dir_ring_brackets hcs_$set_ring_brackets	sets ring brackets for specified segment or directory
read_allowed_ read_write_allowed_ write_allowed_	determines if AIM allows specified operations on object given process' authorization and object's access class

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 hcs_$get_max_length_seg	returns maximum length of segment in words
hcs_$get_safety_sw hcs_$get_safety_sw_seg	returns safety switch value of directory or segment
hcs_$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 hcs_$set_entry_bound_seg	sets entry point bound of segment
hcs_$set_max_length hcs_$set_max_length_seg	sets maximum length of segment
hcs_$set_safety_sw hcs_$set_safety_sw_seg	sets safety switch of segment
hcs_$star_	returns storage system type and all names that match entryname according to star name rules

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
hcs_$get_link_target	returns the target pathname of a link
hcs_$get_user_effmode	returns a user's effective access mode to a branch
mhcs_$get_seg_usage	returns the number of page faults taken on a segment since its creation
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
suffixed_name_	aids in processing suffixed names
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 demonssrates 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 {... xxxn yyyn}).

5. To indicate that the same generic argument must be given in pairs, the arguments are given letters and numbers (e.g., pathA1 pathB1 {... pathAn pathBn}).

6. To indicate one of a group of the same arguments, an "i" is added to the argument name (e.g., pathi, 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).

   -arguments STR, -ag STR
      indicates that the assembled program may expect arguments. If present, it must be the last control argument to the alm command and must be followed by at least one argument. See "Macros in ALM" later in this description.

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 re access to the segment with ring brackets v,v,v where v is the validation level of the process that is active when the object segment is created.

If the user specifies the -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 rpdx.

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 opr*n*, where *n* specifies the pointer register to be used. ALM extends this form to allow:

```
opr pointer,operand
```

which assembles as opr*n*, 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.

alm

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.
        lxl0    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*
```

An alternative form of external expression must begin with a segment name followed by a dollar sign. This may be followed by an entryname, an arithmetic expression, or a modifier, all of which are optional. For example:

```
segname$
segname$entryname-1
segname$+3,5
```

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

A segment name of *system indicates a reference to the external variable (or common block) entryname, which is managed by the linker.

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.

This page intentionally left blank.

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.

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

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.

Keyword	Meaning
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. Additional pseudo-operations are provided by the macro facility. See "Macros in ALM" (following this list of pseudo-operations) for a further description of their syntax.

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 character other than white space. 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. If it is longer, it will be truncated to the defined length.

aci /string/,expression
: is similar to acc, but no length is stored. The first character position contains the first character in aci format.

ac4 /string/,expression
: is similar to aci, but only the rightmost four bits of each ASCII character are stored into the corresponding character position of a string of 4-bit characters. If the given string is shorter than the defined length, it is padded on the right with zeros.

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.

dup expression
duplicates all source statements following the statement containing the dup pseudo-operation up to (but not including) the statement containing the dupend pseudo-operation. The number of times that the statements are duplicated is equal to the value of the expression. This value must be positive and nonzero. Also, dup statements may not be nested.

dupend
terminates the range of a dup pseudo-operation.

eight
(see the even pseudo-operation)

end
terminates the source segment.

entry name1,name2,...,name*n*
generates entry sequences for labels name1, name2, through name*n* and makes the externally-defined symbols name1, name2, through name*n* 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,*
```

maclist keyword {save}
indicates how listing of statements generated by macro expansion is to be done. The following keywords are accepted:

off
suppresses the listing of macro-generated statements and object code

on
lists such statements and their associated object code

object
lists only the object code

restore
reverts the macro listing mode to a previously saved setting

The save argument, if present, saves the current macro listing in a pushdown stack. The default macro listing mode is on.

macro name
indicates the start of a macro definition. When a macro name is defined, it may then be used as a pseudo-operation to trigger the expansion of the macro. See "Macros in ALM" below for a complete description of the definition and expansion of macros in ALM.

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.

alm

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

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.

Macros in ALM

The ALM macro facility provides a means for defining and using sequences of text to be inserted at various points in an ALM program. Each such sequence of text, called a macro, is defined by the use of the macro pseudo-operation in ALM. A macro definition consists of all text following the line containing the macro pseudo-operation until the character string, &end. The sequence of text is named by the symbol appearing as the operand to the macro pseudo-operation.

At any point in a program subsequent to the definition of a macro, the macro name can be used as a pseudo-operation in ALM. Whenever it is so used, ALM inserts the text sequence defined as that macro.

The macro facility is purely text manipulative. It deals with macro definitions as a continuous stream of text characters interspersed with control sequences. Each control sequence begins with the & character. The control sequence, &end, terminates the macro definition. When a macro is invoked by using its name as a pseudo-operation, the macro definition is scanned from left to right. All text between control sequences is copied, and variable information is inserted in place of the control sequences. The resulting macro expansion is presented to ALM for assembly.

Macros may be given arguments by placing operands in fields corresponding to the operands of a pseudo-operation. These arguments can be substituted into the expanded copy of the macro as specified by various control sequences within the macro definition. Control sequences are also provided to facilitate iteration, conditional text selection, unique symbol generation, and other operations.

The macro facility also provides a set of special pseudo-operations that are distinct from the regular ALM pseudo-operations. These special pseudo-operations allow for the conditional assembly of source lines and the printing of messages to the user's terminal during assembly. The argument syntax of these pseudo-operations is the same as that of macros, not the expressions and symbols of the ALM assembler.

Contents of a Macro

The body of a macro (i.e., the text starting on the line following the macro pseudo-operation and ending just before the character string &end) can include any text and control sequences which, when expanded, yield valid ALM source code. The body of a macro can include invocations of other macros and even the definition of other macros.

Macro definitions are shown in the assembly listing with their internal line numbers to the left of the ALM source line number. (These internal numbers are used in diagnostics produced by the macro expander.) Macros may be redefined, the later definition replacing the earlier. Macros may also redefine all existing ALM operations and pseudo-operations.

An example macro is given below:

```
macro  move_a_to_b
lda    a
sta    b
&end
```

Invoking a Macro

A macro is invoked by specifying its name as a pseudo-operation. Arguments to the macro can appear in the variable field separated by commas. A comment may follow the argument list, separated from it by white space or a double quote. Arguments to macros that include spaces, tabs, newline characters, commas, or semicolons must be enclosed in matching parentheses. The parentheses are stripped from the argument during macro expansion. The use of parentheses allows macro invocations to extend over several lines. Argument lists may also be continued on successive source lines by following the last macro argument of a line with a comma. Leading white space preceding the continuation of the argument list on the next line is ignored.

Code and statements produced by the macro facility are placed in the assembly listing without source line numbers. Symbols used by a macro expansion appear in the cross-reference listing as though they were referenced on the line of the macro invocation. The listing of statements produced by macro expansion may be controlled through the use of the maclist pseudo-operation. See the description under "Pseudo-operations" above.

Restrictions

Macro expansions cannot generate the include pseudo-operation. Any macro definition that begins in an include file must end in that include file.

A macro must be defined before it is expanded. It can appear before its definition within another macro definition, but that other macro may not be expanded until the macro it invokes is defined.

Macros may be invoked in code produced by macro expansions. The depth of such recursion, however, must not exceed the current limit of 100.

Control Sequences

Character substitutions and conditional expansions at the time of macro expansion are effected by the control sequences detailed below. The use of any ampersand followed by any sequence not defined below is noted by ALM as an assembly error.

1. &0, & 1.&2, ...

   the character & followed immediately by any positive decimal integer (< 100) is replaced, upon expansion, with the corresponding argument passed to the macro (see "Notes" and "Examples" below).

   The special sequence &0 causes a reference to a unique label at the start of the macro expansion. The label is generated only if the &0 sequence is generated within a macro.

2. &u

   is expanded to be a unique character string of the form ...00000, ...00001, etc., that is different from any other such strings expanded with &u control.

3. &p

   is expanded to be the same string as the previous &u expansion.

4. &n

   is expanded to be the same string as the next &u expansion.

5. &U

   is expanded to be a unique character string of the form .._00000, .._00001; however, multiple occurrences of &U within the same macro yield the same string.

6. &(n

   indicates the beginning of an iteration sequence. The text following the &(n and up to but not including the next &) is expanded repeatedly (see "Iteration" below).

7. &i

   is expanded to the particular element of the iteration set for which the current iteration is being performed (see "Iteration" below).

8. &x

   is expanded into the decimal integer corresponding to the relative position of the particular element of the iteration set over which the current iteration is being performed.

9. &An

   is expanded to be the nth argument following the -ag or -arguments control argument to the alm command.

10. &K

    is expanded as a decimal number equal to the number of arguments in the current macro invocation.

11. &k

    is expanded as a decimal number equal to the number of elements in the current iteration set.

12. &ln

    is expanded as a decimal number equal to the length in characters of the nth argument in the current macro invocation.

13. && is expanded to a single & character. This facilitates macro definitions within macro expansions.

14. &Fn expands to a string constructed by concatenating all arguments to the macro invocation, from the nth onward, separated by commas. If n is not given, 1 is assumed.

15. &Fqn or &FQn is similar to &Fn, except that each argument is enclosed in parentheses as it is concatenated to the expanded string. This control sequence should be used when sublists of macro arguments are to be passed to other macros and there is a possibility that some of these arguments may contain white space, newline characters, etc.

16. &fn is similar to &Fn, except that the elements of the current iteration set are concatenated.

17. &fqn or &fQn is similar to &Fqn and &FQn, except that the elements of the current iteration set are enclosed in parentheses.

18. &Rm,n is used to cause iteration over the arguments in a macro invocation, as opposed to the iteration elements of a single macro argument. The use of &R affects the operation of the next &(control sequence. The m is a decimal number equal to the number of the first argument to be selected; n is a decimal number equal to the number of the last argument to be selected. If n is missing or zero, it is assumed to be equal to the number of arguments in the macro invocation. If m is missing or zero, it is assumed to be 1 (see "Notes" below).

19. &[marks the start of a selection group. The text following the &[and up to but not including the matching &] is expanded conditionally. The elements of a selection group are separated by the control sequence &;. Each element may contain other selection groups to a nesting depth of 15. When a macro is expanded, only one element of a selection group is used. This element is chosen by a control sequence preceding the &[control sequence.

20. &sn selects the nth element of the following selection group. All expanded text between the &s and &[control sequences is interpreted as the decimal number n. If n is zero or greater than the number of elements in the selection group, no element is selected.

21. &=c1,c2 all expanded text between the &= and the next &[control sequence is broken into two character strings. If no comma is found in the expanded text, c2 is taken to be a null string. If the two strings are equal, by character string comparison, the first element of the following selection group is used. Otherwise, the second element, if present, is used.

22. &^=c1,c2 the &^= control sequence is identical to the &= control sequence, except that the first element is selected if the strings are unequal, and the second, if present, is selected if they are equal.

23. &>n1,n2
&<n1,n2
&>=n1,n2
&<=n1,n2

these control sequences are similar to the &= and &^= control sequences, except that the expanded text between this control sequence and the next &[control sequence is interpreted as two decimal integers. If no comma is found, n2 is taken to be zero. An arithmetic comparison of the numbers is performed, as specified by the particular control sequence used. A result of true causes the first element of the following selection group to be used. A result of false causes the second element, if present, to be used.

24. &end

signifies the end of the macro definition. The statement containing the &end control sequence is not part of the macro body, and hence, is not included as part of the macro definition.

Notes

Decimal numbers produced by &K, &k, and &x are generated with no leading blanks or zeros. The number zero is generated as the single digit 0.

Numeric arguments to &n, &(n, &Fn, &fn, &Fqn, &fqn, and &An can be comprised of from zero to three digits. These numbers must appear as such in the unexpanded macro definition. If numeric text is to follow one of the above control sequences, all three digits of n must be supplied.

The numbers used by &Rm,n, as well as the strings and numbers used by the relational and selection control sequences can be of any length. They appear in the expanded text and need not necessarily be in the macro definition. These expanded strings and numbers are, of course, not placed in the final macro expansion being generated.

If a given macro argument is not specified in a particular invocation of that macro, a null character string is used for that argument during macro expansion.

Iteration

The macro facility provides the ability to map the expansion of a subset of a macro definition over a set of elements, expanding that part of the definition repeatedly, selectively substituting each element of the iteration set in turn. By means of this technique, lists may be processed.

An iteration set consists of elements separated by commas. It has the same syntax as the argument list of a macro invocation, including conventions on the use of parentheses for quoting and continuation via the trailing comma. Two types of iteration sets may be referenced in a macro expansion:

1. The argument list to a macro invocation itself may be used as an iteration set, in which case the arguments of the macro invocation are the elements. This type of iteration set is specified by means of the &R control sequence.

2. Any argument to a macro invocation may be used as an iteration set, if it, internally, has the same syntax as an argument list to a macro invocation. This type of iteration set is specified when &R is not used.

The text between the sequences &(and &) is expanded once for each element in the iteration set, in left to right order. If the second form of iteration set is used, the number of the argument to the macro invocation may appear (one to three digits, no digits are mapped into 1) immediately after the &(sequence. Any occurrence of the sequence &i between the sequences &(and &) is replaced by the current element of the iteration set. The sequence &x is replaced by the decimal number of the relative position of that element in the iteration set (not the argument number, in the first type of iteration set).

Iterations may not be nested. Any iteration that starts in an element of a selection group must end in that element of a selection group. No iteration may end in any element of a selection group unless it started in that element of that selection group.

Macro Facility Pseudo-Operations

The macro facility provides a set of pseudo-operations in addition to the macro pseudo-operation already described. These pseudo-operations are different from the other pseudo-operations provided by the assembler insofar as the syntax of their arguments, which is the syntax of macro invocation arguments, with all quoting and continuation conventions of them, and not the syntax of other pseudo-operation arguments to the assembler.

The use of these pseudo-operations, like all other ALM pseudo-operations, is not limited to code produced by macro expansion. They may be freely used in source segments and include files, as well as in macro code.

1. warn

   prints out its first argument on the user's terminal, preceded by the string "ALM assembly:" and followed by a newline character. This argument, without the prefix, is also placed in the program listing.

2. ife

   the character strings that are the first and second arguments to ife are compared. If they are the same character string, all assembler statements between the one containing the end of the argument list to ife, and the next one containing the string ifend in any context at all are assembled. No part of the line containing the string ifend is assembled. If the first and second arguments are not equal, none of these lines are assembled.

3. ine

   the same as ife, but assembly of the text up to ifend proceeds only if the first two arguments are not equal by character string comparison.

4. ifint

   the first argument to the ifint pseudo-operation is inspected to see if it is a valid decimal integer. If so, all assembler statements between the one containing the end of the argument list to ifint and the next one containing the string ifend in any context at all are assembled. No part of the line containing the ifend is assembled. If the first argument to ifint is not a valid integer, none of these lines are assembled.

5. inint

   the same as ifint, but assembly of the text up to ifend proceeds only if the first argument is not a valid decimal integer.

6. ifarg

   all of the arguments to the alm command following the -ag or -arguments control argument are inspected, and compared with the first argument to ifarg. If any of these command arguments compare equal, by character string comparison, to the first argument to ifarg, all assembler statements between the one containing the end of the argument list to ifarg and the first one containing the string ifend in any context at all are assembled. No part of the line containing the ifend is assembled. If the first argument to ifarg does not appear among the arguments following -ag or -arguments, none of these lines are assembled.

7. inarg

   the same as ifarg, but assembly of the text up to ifend proceeds only if the first argument to inarg is not found among the arguments to the alm command following -ag or -arguments.

In all of the conditional constructs above, the key string, ifend, must appear in the same source segment or macro expansion as the statement containing the conditional pseudo-operation. If the ifend key string appears in the ifend_exit string, and the entire construct appears in a macro expansion, and the predicate of the conditional construct is met (i.e., the statements are being assembled, not skipped), the assembler ceases to take input from that macro expansion, as though the last statement in that macro expansion had been assembled.

Examples

The following macro definitions show typical expansions:

```
          macro     load
          ld&1      &2
          &end
```

might be used as follows:

```
     load      x0,temp                  ldx0      temp
```

or:

```
     load      a,(sp|3,*)               lda       sp|3,*
```

The use of parentheses in the second example causes the comma to be ignored as a parameter delimiter. The macro definition:

```
          macro     test
          lda       &1
          tpl       &U
          sta       last_minus
&U        sta       2
          &end
```

might be used as follows:

```
     test      a,b                      lda       a
                                        tpl       .._00000
                                        sta       last_minus
                              .._00000: sta       b
```

The following example shows how iteration is used. The macro definition:

```
          macro     table
&R&(      vfd       18/&i,18/&0
&)
          & end
```

might be used as follows:

```
e1:       table     4,6,8,10            vfd       18/4,18/e1
                                        vfd       18/6,18/e1
                                        vfd       18/8,18/e1
                                        vfd       18/10,18/e1
```

The following example shows how conditional expansion can be used. The macro definition:

```
                    macro     meter
                    lda       &1
                    ife       &2,on
                    aos       meterword,al
                    ifend
                    &end
```

might be used as follows:

```
     meter          foo,on                        lda      foo
                                                  aos      meterword,al
```

The following macro shows how &x might be used. The macro definition:

```
                    macro     callm
     &(3            eppbp     &i
                    spribp    &2+&x*2
     &)
                    eaq       2*&x-2
                    lls       36
                    staq      &2
                    call      &1(&2)
                    &end
```

might be used as follows:

```
                    callm     sys,arg,(=1,(=14aError from ^d.))
```

yielding:

```
                    eppbp     =1
                    spribp    arg+1*2
                    eppbp     =14aError from ^d.
                    spribp    arg+2*2
                    eppbp     did
                    spribp    arg+3*2

                    eaq       2*4-2
                    lls       36
                    staq      arg
                    call      sys(arg)
```

The following macro definition shows how conditional expansion might be used:

```
                    macro     tab9
     &R&(&=&x,1&[             vfd       &;,&]o9/&i&)
                    &end
```

This macro might be invoked as follows:

```
                    tab9      16,42,13,36,67
```

expanding to:

```
                    vfd       o9/16,o9/42,o9/13,o9/36,o9/67
```

The following example shows how macros may be defined by macros, and used to powerful effect. These macros allow a call like a PL/I call to be generated, with descriptors.

The following macro is invoked to declare variables by specifying their address, data type, and precision:

```
                    macro     declare
                    macro     dcl_&1
                    epp0      &2
                    epp1      =v1/1,6/&3,17/0,12/&4
                    &&end
                    &end
```

This macro may be invoked as follows:

```
     declare        count,buffer+2,fixed,17
or:
     declare        progname,(lp|xlink,*),char,32
```

These macro invocations cause the following macro definitions to be produced:

```
                    macro     dcl_count
                    epp0      buffer+2
                    epp1      =v1/1,6/fixed,17/0,12/17
                    &end

                    macro     dcl_progname
                    epp0      lp|xlink,*
                    epp1      =v1/1,6/char,17/0,12/32
                    &end
```

Assume that at some point in the assembly the statements:

```
                    equ       char,21
                    equ       fixed,1
```

defining the PL/I descriptor types for these data types appear.

The following macro definition, when invoked, generates a full PL/I call with descriptors. Assume that the statement:

```
                    temp      argl,16
```

appears at some point in the program.

```
                    macro     gcall
     &R2&(          dcl_&i
                    spri0     argl+2*&x
                    spri1     argl+2*&K+2*&x
     &)
                    ldaq      =v18/2*&K-2,18/0,18/2*&K-2,18/4
                    staq      argl
                    call      &1(argl)
                    &end
```

When the following macro invocation is issued:

```
gcall program,count,progname
```

the following expansion is immediately produced:

```
dcl_count
spri0     argl+2*1
spri1     argl+2*2+2*1
dcl_progname
spri0     argl+2*2
spri1     argl+2*s+2*2

ldaq      =v18/2*3-2,18/0,18/2*3-2,18/4
staq      argl
call      program(argl)
```

This is further expanded when the dcl_count and dcl_progname macros are expanded to:

```
epp0      buffer 2
epp1      =v1/1,6/fixed,17/0,12/17
spri0     argl+2*1
spri1     argl+2*2+2*1
epp0      lp|xlink,*

epp1      =v1/1,6/char,17/0,12/32
spri0     argl+2*2
spri1     argl+2*2+2*2
ldaq      =v18/2*3-2,18/0,18/2*3-2,18/4
staq      argl

call      program(argl)
```

which is precisely the code required for a full PL/I call.

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

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.

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.

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.

Name: delete_volume_quota, dlvq

The delete_volume_quota command is used to delete a quota account for a logical volume. This command is to be used by volume executives.

Usage

delete_volume_quota logical_volume account

where:

1. logical_volume
   is the name of the logical volume from which quota is to be deleted.

2. account
   is the name of the quota account (in the form Person_id.Project_id.tag) to be deleted.

Notes

To use this command, the user must have execute access to the logical volume. It is not necessary that the volume be mounted.

The quota account cannot be deleted if there are still master directories whose quotas are charged against the account to be deleted. Such directories must either be deleted or transferred to another account (see the set_mdir_account command).

This page intentionally left blank.

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;
```

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
  !BBBCdffddfdffkl.temp.0247   work
  !BBBCddffdfffhhh.temp.0253   (free)
  !BBBCdgdgfhfgfsf.temp.0254   (free)
  !BBBCvdvfgvdgvvv.temp.0321   editor
```

To list the segments currently in use, type:

```
! list_temp_segments

      3 Segments

  !BBBCdfghgffkkkl.temp.0246   work
  !BBBCdffddfdffkl.temp.0247   work
  !BBBCvdvfgvdgvvv.temp.0321   editor
```

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".

Name: mbx_create, mbcr

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:

```
mbcr 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 path*i* 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, mbda

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. Instead, this command sets their extended access to null. The command line "mbda path *.*.*" has the same effect as the command line "mbsa path null *.*.*".

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:

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

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.

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

The user must have modify permission on the containing directory.

The entryname portion of pathi 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.

Name: mbx_rename, mbrn

The mbx_rename 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. pathi
   is the pathname of a mailbox. The entryname portion is the name to be replaced. The star convention is allowed.

2. namei
   is the new name to be placed on the mailbox. The equal convention is allowed.

Notes

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

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

Since two entries in a directory cannot have the same entryname, special action is taken by this command if namei already exists in the directory specified by pathi. If the mailbox having the entryname namei has an additional name, entryname namei is removed and the user is informed of this action by a message printed on his terminal. If the mailbox having the entryname namei 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. modei
   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_namei
   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 pathi.

   -entry, -et
   print only a listing of the pathi 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 pathi.

   -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.

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.

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.

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.

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_.

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_ttt_path

The set_ttt_path command changes the pathname of the terminal type table (TTT) associated with the user's process.

Usage

```
set_ttt_path {path} {-control_arg}
```

where:

1. path
   is the pathname of the TTT.

   If no path argument is given, the control argument must be given.

2. control_arg
   can be -reset (-rs) to reset the TTT pathname to its default value of >system_control_1>ttt.

Note

The use of path argument and the -reset control argument are mutually exclusive; only one may be given in any invocation of the set_ttt_path command.

This page intentionally left blank.

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_.

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

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

3. control_string (Input)
   is an ioa_ subroutine control string (char(*)). (The ioa_ subroutine is described in the MPM Subroutines.) This argument is optional. See "Note" below.

4. argi (Input)
   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.

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 arg*i* arguments described above. If the control_string and arg*i* arguments are not given, user_message is omitted.

Name: aim_check_

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

Entry: aim_check_$equal

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

Usage

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

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

where:

1. acc_atti (Input)
   are access attributes.

2. returned_bit (Output)
   is the result of the comparison.
   "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_atti (Input)
   are access attributes.

2. returned_bit (Output)
   is the result of the comparison.
   "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_atti (Input)
   are access attributes.

2. returned_bit (Output)
   is the result of the comparison.
   "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 (Input)
   points to the structure described in "Notes" below.

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

Notes

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

```
dcl 1 area_info         aligned based,
      2 version         fixed bin,
      2 control,
        3 extend        bit(1) unaligned,
        3 zero_on_alloc bit(1) unaligned,
        3 zero_on_free  bit(1) unaligned,
        3 dont_free     bit(1) unaligned,
        3 no_freeing    bit(1) unaligned,
        3 system        bit(1) unaligned,
        3 mbz           bit(30) unaligned,
      2 owner           char(32) unaligned,
      2 n_components    fixed bin,
      2 size            fixed bin(30),
      2 version_of_area fixed bin,
      2 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.

Name: ascii_to_ebcdic_

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

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

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

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;
```

ascii_to_ebcdic_

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
5	065	F5	5

ascii_to_ebcdic_

GRAPHIC	OCTAL	HEXADECIMAL	GRAPHIC
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	
`	140	79	`
a	141	81	a
b	142	82	b
c	143	83	c
d	144	84	d
e	145	85	e
f	146	86	f
g	147	87	g
h	150	88	h
i	151	89	i
j	152	91	j
k	153	92	k
l	154	93	l
m	155	94	m
n	156	95	n
o	157	96	o

GRAPHIC	OCTAL	HEXADECIMAL	GRAPHIC
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 (Input)
   points to the target of the assignment; it can contain a bit offset.

2. target_type (Input)
   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 Appendix D of the MPM Reference Guide) and P is 0 if the target is unpacked and 1 if the target is packed.

3. target_length (Input)
   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.

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

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

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

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 III of the MPM Reference Guide. 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 (Input)
   is the pathname whose final entryname is to be validated. Trailing spaces in the pathname character string are ignored.

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

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.

2. code
   is as described above.

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: component_info_

This subroutine returns information about a component of a bound segment similar to that returned by object_info_. The component may be specified either by name or by offset.

Entry: component_info_$name

This entry point specifies the component by name.

Usage

```
declare component_info_$name entry (ptr, char(32) aligned, ptr,
     fixed bin(35));

call component_info_$name (seg_ptr, comp_name, arg_ptr, code);
```

where:

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

2. comp_name (Input)
   is the name of the component.

3. arg_ptr (Input)
   is a pointer to a structure to be filled in (see "Notes" below).

4. code (Output)
   is a standard status code.

Entry: component_info_$offset

This entry point specifies the component by its offset.

Usage

```
declare component_info_$offset entry (ptr, fixed bin(18), ptr,
     fixed bin(35));

call component_info_$offset (seg_ptr, offset, arg_ptr, code);
```

where:

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

2. offset (Input)
   is an offset into the bound segment corresponding to the text, internal static or symbol section of some component.

3,4.
   are as above.

Notes

The structure to be filled in (a declaration of which is found in component_info.incl.pl1) is declared as follows:

```
dcl 1 ci                    aligned,
     2 dcl_version          fixed bin,
     2 name char(32)        aligned,
     2 text_start           ptr,
     2 stat_start           ptr,
     2 symb_start           ptr,
     2 defblock_ptr         ptr,
     2 text_lng             fixed bin,
     2 stat_lng             fixed bin,
     2 symb_lng             fixed bin,
     2 n_blocks             fixed bin,
     2 standard             bit(1) aligned,
     2 compiler             char(8) aligned,
     2 compile_time         fixed bin(71),
     2 user_id              char(32) aligned,
     2 cvers                aligned,
       3 offset             bit(18) unaligned,
       3 length             bit(18) unaligned,
     2 comment              aligned,
       3 offset             bit(18) unaligned,
       3 length             bit(18) unaligned,
     2 source_map           fixed bin;
```

where:

1. dcl_version
   is the version number of this structure. It is set by the caller and must be 1.

2. name
   is the name of the component, i.e., the name specified in a bindfile objectname statement; also, the name of the component as archived.

3. text_start
   is a pointer to the base of the component's text section.

4. stat_start
   is a pointer to the base of the component's internal static.

5. symb_start
   is a pointer to the base of the component's symbol section.

6. defblock_ptr
   is a pointer to the component's definition block.

7. text_lng
   is the length, in words, of the component's text section.

8. stat_lng
   is the length, in words, of the component's internal static.

9. symb_lng
   is the length, in words, of the component's symbol section.

10. n_blocks
    is the number of blocks in the component's symbol section.

11. standard
    is on if the component is in standard object format.

12. compiler
    is the name of the component's compiler.

13. compile_time
    is a clock reading of the date/time the component was compiled.

14. user_id
    is the standard Multics User_id of the component's creator.

15. cvers.offset
    is the offset of the printable version description of the component's compiler, in words, relative to symb_start.

16. cvers.length
    is the length, in characters, of the component's compiler version.

17. comment.offset
    is the offset of the component's compiler comment, in words, relative to symb_start.

18. comment.length
    is the length, in characters, of the component's comment.

19. source_map
    is the offset of the component's source map structure, in words, relative to symb_start.

Name: condition_interpreter_

The condition_interpreter_ subroutine can be used by subsystem condition handlers to obtain a formatted error message for all conditions except quit, alrm, and cput. Some conditions do not have messages and others cause special actions to be taken (such as finish). These are described in "Notes" below. For more information on conditions, see "Multics Condition Mechanism" in Section VII 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 (Input)
   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.

2. m_ptr (Output)
   points to the allocated message if area_ptr is not null; otherwise it is not set.

3. mlng (Output)
   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.

4. mode (Input)
   is the desired mode of the message to be printed or returned. It can have the following values:
   1 normal mode
   2 brief mode
   3 long mode

5. mc_ptr (Input)
   if not null, points to machine conditions describing the state of the processor at the time the condition was raised.

6. cond_name (Input)
   is the name of the condition being raised.

7. wc_ptr (Input)
   is usually null; but when mc_ptr points to machine conditions from ring 0, wc_ptr points to alternate machine conditions.

8. info_ptr (Input)
   if not null, points to the information structure described in the find_condition_info_ subroutine in this document.

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 swtiches to be closed, so that no input/output may be done upon return.

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 (Output) is a standard status code and is nonzero if continue_to_signal_ was called when no condition was signalled. is not found.

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. *

Usage

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

call convert_aim_attributes_ (aim_bits, aim_chars);
```

where:

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

2. aim_chars (Output)
   is the character string representation.

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 a dialed terminal or an auto call line. In addition to returning line status, this entry point also returns the device name and I/O module name for use in attaching the line through the iox_ subroutine. See the MPM Subroutines for further description of the iox_ subroutine.

Usage

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

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

where:

1. message (Input)
   is the event message to be decoded.

2. channel_name (Output)
   is the name of the channel that has dialed up or hung up.

3. io_module (Output)
   is the name of the iox_ I/O module to be used with the assigned device.

4. n_dialed (Output)
   is the number of terminals currently dialed to the process or -1.

5. flags (Output)
   is a bit string of the following structure:

   ```
   dcl 1 flags       aligned,
         2 dialed_up bit(1) unal,
         2 hung_up   bit(1) unal,
         2 control   bit(1) unal,
         2 pad       bit(33) unal;
   ```

   Only the first three bits have meaning, and only one can be on at a time. See "Notes" below for complete details.

6. code (Output)
   is a standard status code.

convert_dial_message_

Notes

The message may be either a control message or an informative message. Control messages have flags.control on ("1"b), n_dialed is set equal to the number of dialed terminals, and code is set either to 0 (request to become dial server accepted) or error_table_$action_not_performed (request to become dial server denied).

Informative messages have flags.control off, n_dialed is set to -1, channel is set to the name of the channel involved, io_module is set to the name of an I/O module, and either flags.dialed_up or flags.hung_up is on, indicating that the named channel has either just dialed up, or just hung up.

The io_module name is provided as a convenience; the caller is not required to use the name returned by this subroutine.

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. See "Status Codes" in Section VII of the MPM Reference Guide.

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 (Input)
   is a standard status code.

2. shortinfo (Output)
   is a short status message corresponding to code.

3. longinfo (Output)
   is a long status message corresponding to code; the message is padded on the right with blanks.

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.

Name: cross_ring_

The cross_ring_ I/O module allows an outer ring to attach a switch to a preexisting switch in an inner ring, and to perform I/O operations by forwarding I/O from the attachment in the outer ring through a gate to an inner ring. The cross_ring_ I/O module is nto called directly by users; rather the module is accessed through the I/O system.

Attach Descriptions

cross_ring_ switch_name N

where:

1. switch_name
   is a previously registered switch name in ring N.

2. N
   is a ring number from 0 to 7.

Opening

The inner ring switch may be open or not. If not open, it will be opened on an open call. All modes are supported.

Close Operation

The inner switch is closed only if it was opened by cross_ring_.

Other Operations

All operations are passed on to the inner ring I/O switch.

Notes

This I/O module allows a program in an outer ring, if permitted by the inner ring, to use I/O services that are available only from an inner ring via cross_ring_io_$allow_cross. By the use of the cross_ring_io_$allow_cross subroutine a subsystem writer is able to introduce into an outer ring environment many features from an inner ring, thereby tailoring it to fit the user's specific needs.

The switch in the inner ring must be attached by the inner ring before cross_ring_ can be attached in the outer ring.

Name: cross_ring_io_$allow_cross

The cross_ring_io_$allow_cross entry point must be called to allow use of an I/O switch via cross-ring attachments from an outer ring. The call must be made in the inner ring before the outer ring attempts to attach.

Usage

```
declare cross_ring_io_$allow_cross entry (char(*), fixed bin,
     fixed bin(35));

call cross_ring_io_$allow_cross (switch_name, ring, code);
```

where:

1. switch_name (Input)
   is the inner ring switch name.

2. ring (Input)
   is the highest validation level from which switch_name may be used.

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

Notes

This entry may be called more than once with the same switch_name argument. Subsequent calls are ignored.

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 (Input)
   is the binary integer to be converted.

2. string (Output)
   is the ASCII equivalent of n.

3. base (Input)
   is the base to use in converting the binary integer (e.g., base is 10 for decimal integers).

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 (Input)
   is the binary integer to be converted.

2. string (Output)
   is the ASCII equivalent of n.

Entry: cv_bin_$oct

This entry point converts the binary representation of an octal integer to a 12-character ASCII string.

Usage

```
declare cv_bin_$oct entry (fixed bin, char(12) aligned);

call cv_bin_$oct (n, string);
```

where:

1. n (Input)
   is the binary integer to be converted.

2. string (Output)
   is the ASCII equivalent of n.

Note

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

Name: cv_dec_

The cv_dec_ function accepts an ASCII representation of a decimal integer and returns the fixed binary(35) representation of that number. (See also cv_dec_check_.)

Usage

```
declare cv_dec_ entry (char(*)) returns (fixed bin(35));

a = cv_dec_ (string);
```

where:

1. string (Input)
   is the string to be converted.

2. a (Output)
   is the result of the conversion.

Note

If string is not a proper character representation of a decimal number, a will contain the converted value of the string up to, but not including, the incorrect character within the string.

Name: cv_dec_check_

This function differs from cv_dec_ only in that a code is returned indicating the possibility of a conversion error. (See also cv_dec_.)

free format integer conversion with sign.

Usage

```
declare cv_dec_check_ entry (char(*), fixed bin(35))
     returns (fixed bin(35));

a = cv_dec_check_ (string, code);
```

where:

1. string (Input)
   is the string to be converted.

2. code (Output)
   is a code that equals 0 if no error has occurred; otherwise, it is the index of the character of the input string that terminated the conversion. See "Note" below.

3. a (Output)
   is the result of the conversion.

NOTE

Code is not a standard status code and, therefore, cannot be passed to com_err_ and other subroutines that accept only standard status codes.

Name: cv_entry_

The cv_entry_ function converts a virtual entry to an entry value. A virtual entry is a character-string representation of an entry value. The types of virtual entries accepted are described under "Virtual Entries" below.

Usage

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

entry_value = cv_entry_ (ventry, referencing_ptr, code);
```

where:

1. ventry (Input)
   is the virtual entry to be converted. See "Virtual Entries" below for more information.

2. referencing_ptr (Input)
   is a pointer to a segment in the referencing directory. This directory is searched according to the referencing_dir search rule to find the entry. A null pointer may be given if the referencing_dir search rule is not to be used.

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

4. entry_value (Output)
   is the entry value that results from the conversion.

Virtual Entries

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

Virtual Entry	Interpretation
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$0.
ref_name	same as ref_name$ref_name but like "path" if it contains ">" or "<" characters.

Notes

Use of a pathname in a virtual entry causes the referenced segment to be initiated with a reference name equal to its final entryname. Name duplication errors occurring during the initiation are resolved by terminating the previously known name.

The referencing_ptr is used in a call to the hcs_$make_entry entry point. Refer to the description of this entry point in the MPM Subroutines for more information.

The cv_entry_ function returns an entry value that may be used in a call to cu_$generate_call. If an entry pointer is required, rather than an entry variable, make a call to cu_$decode_entry_value. (The cu_ subroutine is documented in the MPM Subroutines.) For pointers not used as entry pointers, use the cv_ptr_ function to convert a virtual pointer.

A virtual entry not containing the "$" or "¦" characters is interpreted as a pathname if it contains a ">" or "<" character, otherwise, it is a reference name.

Name: cv_hex_

The cv_hex_ function takes an ASCII representation of a hexadecimal integer and returns the fixed binary(35) representation of that number. The ASCII representation may contain either uppercase or lowercase characters. (See also cv_hex_check_.)

Usage

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

a = cv_hex_ (string);
```

where:

1. string (Input)
   is the string to be converted. It must be nonvarying.

2. a (Output)
   is the result of the conversion.

Name: cv_hex_check_

This function differs from the cv_hex_ function only in that a code is returned indicating the possibility of a conversion error. (See also cv_hex_.)

Usage

```
declare cv_hex_check_ entry (char(*), fixed bin(35)),
     returns (fixed bin(35));

a = cv_hex_check_ (string, code);
```

where:

1. string (Input)
   is the string to be converted. It must be nonvarying.

2. code (Output)
   is a code that equals 0 if no error occurred; otherwise, it is the index of the character that terminated the conversion. See "Note" below.

3. a (Output)
   is the result of the conversion.

Note

Code is not a standard status code and, therefore, cannot be passed to com_err_ and other subroutines that accept only standard status codes.

Name: cv_oct_

The cv_oct_ function takes an ASCII representation of an octal integer and returns the fixed binary(35) representation of that number. (See also cv_oct_check_.)

Usage

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

a = cv_oct_ (string);
```

where:

1. string (Input)
   is the string to be converted.

2. a (Output)
   is the result of the conversion.

Name: cv_oct_check_

This function differs from the cv_oct function only in that a code is returned indicating the possibility of a conversion error. (See also cv_oct_.)

Usage

```
declare cv_oct_check_ entry (char(*), fixed bin(35)) returns
     (fixed bin(35));

a = cv_oct_check_ (string, code);
```

where:

1. string (Input)
   is the string to be converted. It must be nonvarying.

2. code (Output)
   is a code that equals 0 if no error occurred; otherwise it is the index of the character that terminated the conversion. See "Note" below.

3. a (Output)
   is the result of the conversion.

Note

Code is not a standard status code and, therefore, cannot be passed to com_err_ and other subroutines that accept only standard status codes.

Name: cv_ptr_

The cv_ptr_ function converts a virtual pointer to a pointer value. A virtual pointer is a character-string representation of a pointer value. The types of virtual pointers accepted are described under "Virtual Pointers" below.

Usage

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

ptr_value = cv_ptr_ (vptr, code);
```

where:

1. vptr (Input)
   is the virtual pointer to be converted. See "Virtual Pointers" below for more information.

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

3. ptr_value (Output)
   is the pointer that results from the conversion.

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

Notes

Pointers returned by the cv_ptr_ function cannot be used as entry pointers. The cv_ptr_ function constructs the returned pointer to a segment in a way that avoids copying of the segment's linkage and internal static data into the combined linkage area. The cv_entry_ function is used to convert virtual entries to an entry value.

The segment pointed to by the returned ptr_value is initiated with a null reference name. The cv_ptr_$terminate entry point should be called to terminate this null reference name.

Virtual Pointers

The cv_ptr_ function converts virtual pointers that contain one or two components -- a segment identifier and an optional offset into the segment. Altogether, 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.

Virtual Pointer	Interpretation
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, BASIC, COBOL, 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 (Input)
   points either directly at the descriptor to be decoded or at the argument list in which the descriptor appears.

2. n (Input)
   controls which descriptor is decoded. If n is 0, ptr points at the descriptor to be decoded; otherwise, ptr points at the argument list header and the nth descriptor is decoded.

3. type (Output)
   is the data type specified by the descriptor. Type codes appearing in an old form of descriptor are mapped into the new codes.
   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 nth descriptor does not exist

4. packed (Output)
   describes how the data is stored.

   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 (Output)
   indicates either the number of dimensions of the descriptor array or whether the descriptor is an array or a scalar.

   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 (Output)
   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.

7. scale (Output)
   is the scale of an arithmetic value for a new format descriptor. This value is 0 for an old form of descriptor.

Name: define_area_

The define_area_ subroutine is used to initialize a region of storage as an area and to enable special area management features as well. The region being initialized may or may not consist of an entire segment or may not even be specified at all, in which case a segment is acquired (from the free pool of temporary segments) for the caller.

See the release_area_ subroutine for a description of how to free up segments acquired via this interface.

Usage

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

call define_area_ (info_ptr, code);
```

where:

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

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

Notes

The structure pointed to by info_ptr is the standard area_info structure used by the various area management routines and is described by the following PL/I declaration defined by the system include file, area_info.incl.pl1:

```
dcl 1 area_info         aligned based,
      2 version         fixed bin,
      2 control,
        3 extend        bit(1) unaligned,
        3 zero_on_alloc bit(1) unaligned,
        3 zero_on_free  bit(1) unaligned,
        3 dont_free     bit(1) unaligned,
        3 no_freeing    bit(1) unaligned,
        3 system        bit(1) unaligned,
        3 pad           bit(30) unaligned,
      2 owner           char(32) unaligned,
      2 n_components    fixed bin,
      2 size            fixed bin(30),
      2 version_of_area fixed bin,
      2 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 per-process, 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.)

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.)

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 12 characters in length. This string corresponds to the first argument of the dial command. It is used, together with the second argument of the dial command (Person_id.Project_id), to uniquely identify a dial server. It can also be a registered dial qualifier (see below). 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. The dial_manager_ subroutine goes blocked on the event-wait channel awaiting a response to the request from the answering service. After the dial_manager_$allow_dials entry point has been called, the event channel may be changed to an event-call channel. When the user program receives wakeups over this channel, it 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 (Input)
   is a pointer to the dial_manager_arg structure described in "Notes" below.

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

Entry: dial_manager_$registered_server

This entry point is used to request that the answering service allow terminals to dial to the calling process using only the dial qualifier. The calling process must have rw access to the access control segment dial.<dial qualifier>.acs in >sci>rcp if this request is to be honored.

Usage

```
declare dial_manager_$registered_server entry (ptr, fixed bin(35));

call dial_manager_$registered_server (request_ptr, code);
```

where the arguments are the same as for the dial_manager_$allow_dials entry point.

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. Note that the name of the chosen channel is not returned by dial_manager_; it must be obtained via a call to convert_dial_message_.

Usage

```
declare dial_manager_$dial_out entry (ptr, fixed bin(35));

call dial_manager_$dial_out (request_ptr, code);
```

where the arguments are the same as for the dial_manager_$allow_dials entry point.

Entry: dial_manager_$release_channel

This entry point is used to request the answering service to release the channel specified in channel name. This channel must be dialed to the caller at the time of this request. The same information should be passed to this entry point as to the dial manager_$allow_dials entry point. The caller must set dial_manager_arg.channel_name to the name of the channel to be released. The user must make dial_manager_arg.dial_channel an event-wait channel before using this call.

Usage

```
declare dial_manager_$release_channel entry (ptr, fixed bin(35));

call dial_manager_$release_channel (request_ptr, code);
```

where the arguments are the same as for the dial_manager_$allow_dials entry point.

Entry: dial_manager_$shutoff_dials

This entry point informs the answering service that the user process wishes to prevent further dial connections, and that existing connections should be terminated. The same information should be passed to this entry point as was passed to the dial_manager_$allow_dials or dial_manager_$registered_server entry point. The dial event_channel must be an event-wait channel.

Usage

```
declare dial_manager_$shutoff_dials (ptr, fixed bin(35));

call dial_manager_$shutoff_dials (request_ptr, code);
```

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 the arguments are the same as for the dial_manager_$allow_dials entry point.

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. This is set by the caller and must be 1.

2. dial_qualifier
   is either a telephone number or a dial qualifier. It is the telephone number to be called for calls to the dial_manager_$dial_out entry point. Notice that nonnumeric characters are ignored, so the user need not remove them from a telephone number string. It is the dial qualifier for calls to the dial_manager_$allow_dials entry point.

3. dial_channel
   is an interprocess communication channel used to receive messages from the answering service. Note that the channel is a per-process item, rather than a per-terminal item. It must be the same for all calls to the dial_manager_ subroutine in the same process.

4. channel_name
   is used for calls in the dial_manager_$terminate_dial_out entry point to indicate which channel should be disconnected. In calls to the dial_manager_$dial_out entry point, 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. In calls to the dial_manager_$release_channel entry point, it indicates the dialed channel to be released.

Name: dprint_

This subroutine adds a request to print or punch a segment 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 (Input)
   is the pathname of the containing directory.

2. entryname (Input)
   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.

3. arg_ptr (Input)
   is a pointer to the argument structure described in "Notes" below. If no argument structure is supplied, arg_ptr should be null.

4. code (Output)
   is a standard status code.

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. This is set by the caller and must be 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.
   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 zeros.

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: dump_segment_

This subroutine prints the dump of a segment formatted in the same way as the dump_segment command (MPM Commands) would print it. The output format is controlled by a bit string that allows most of the formatting control arguments available to dump_segment.

Usage

```
declare dump_segment_ entry (ptr, ptr, fixed bin, fixed bin(18),
     fixed bin(18), bit(*));

call dump_segment_ (iocb_ptr, first, block_size, offset, count, format);
```

where:

1. iocb_ptr (Input)
   is a pointer to the I/O control block that specifies where the dump is to be written.

2. first (Input)
   is a pointer to the first word of the data to be dumped.

3. block_size (Input)
   is the number of words in the block if blocked output is desired. If unblocked output is desired, this is zero.

4. offset (Input)
   is an arbitrary offset to be printed in addition to the address of the first word of data to be dumped if the offset option in the format string is specified. (It is reset to this initial value at the start of each block.)

5. count (Input)
   is the number of words to dump, starting with the word pointed to by first.

6. format (Input)
   is a format control bit string with the following definition: (See the dump_segment documentation, MPM Commands, for a full discussion of these arguments.)

bit	definition	default value
1	address column	on
2	offset column	off
3	short	off
4	bcd	off
5	ascii	off
6	long	off
7	ebcdic9	off
8	ebcdic8	off
9	4bit	off
10	hex8	off
11	hex9	off

Name: ebcdic_to_ascii_

The ebcdic_to_ascii_ subroutine performs isomorphic (one-to-one reversible) conversion from EBCDIC to ASCII. The input data is a string of valid EBCDIC characters. A valid EBCDIC character is defined as a 9-bit byte with a hexadecimal value in the range 00 ≤ hex_value ≤ FF (octal value in the range 000 ≤ oct_value ≤ 377).

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 (Input)
   is the string of EBCDIC characters to be converted.

2. ascii_out (Output)
   is the ASCII equivalent of the input string.

Entry: ebcdic_to_ascii_$ea_table

This entry point defines the 256-character translation table used to perform conversion from EBCDIC to ASCII. Of the 256 valid EBCDIC characters, only 128 have ASCII equivalents. These latter 128 characters are defined in the Isomorphic ASCII/EBCDIC Conversion Table (in the ascii_to_ebcdic_ subroutine description.) For defined characters, the mappings implemented by the ebcdic_to_ascii_ and ascii_to_ebcdic_ subroutines are isomorphic; i.e., each character has a unique mapping, and mappings are reversible. An undefined (but valid) EBCDIC character is mapped into the ASCII SUB (substitute) character, octal 032; the mapping of such a character is anisomorphic. The result of converting an invalid character is undefined.

Usage

```
declare ebcdic_to_ascii_$ea_table char(256) external static;
```

Note

* 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 (Input)
   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.

2. cond_info_ptr (Input)
   is a pointer to the structure (see "Notes" below) in which information is returned.

3. code (Output)
   is the standard status code. It is nonzero when the stack_ptr argument does not point to a condition frame or, if the stack_ptr argument is null, when no condition frame can be found.

Notes

The structure referred to in item 2 above is declared as follows:

```
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:

1. 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.

2. version
   is the version number of this structure (currently this number is 1).

3. condition_name
   is the condition name.

4. 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 VII of the MPM Reference Guide.

5. 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.

6. 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.

7. 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

8. mbz1
   is currently unused and should be set to "0"b.

9. mbz2
   is currently unused and should be set to "0"b.

10. 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.

11. mbz
   is currently unused and should be set to "0"b.

Name: get_default_wdir_

The get_default_wdir_ function returns the pathname of the user's current default working directory.

Usage

```
declare get_default_wdir_ entry returns (char(168) aligned);

default_wdir = get_default_wdir_ ();
```

where default_wdir (Output) is the pathname of the user's current default working directory.

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 (Input)
   is a pointer to the definition section of the object segment. This pointer can be obtained via the object_info_ subroutine.

2. segname (Input)
   is the name of the object segment.

3. entryname (Input)
   is the name of the desired entry point.

4. def_ptr (Output)
   is a pointer to the definition for the entry point.

5. code (Output)
   is a standard status code. If the entry point is found, code is 0.

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 (Input)
   is a pointer to a procedure entry point.

2. symbolname (Output)
   is the name corresponding to the location specified by entry_ptr. The maximum length is 256 characters.

3. segno (Output)
   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.

4. lang (Output)
   is the language in which the segment or component pointed to by entry_ptr was compiled.

5. code (Output)
   is a standard status code.

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 III of the MPM Reference Guide 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 (Input)
   is the entryname from which the target is to be constructed. Trailing blanks in the entryname character string are ignored.

2. equal_name (Input)
   is the equal name from which the target is to be constructed. Trailing blanks in the equal name character string are ignored.

3. target_name (Output)
   is the target name that is constructed.

4. code (Output)
   is a standard status code. It can be one of the following:
   error_table_$bad_equal_name
   the equal name has a bad format
   error_table_$badequal
   there is no letter or component in the entryname that corresponds to a percent character (%) or an equal sign (=) in the equal name
   error_table_$longeql
   the target name to be constructed is longer than 32 characters

Notes

If the error_table_$badequal status code is returned, then a target_name is returned in which null character strings are used to represent the missing letter or component of entryname.

If the error_table_$longeql status code is returned, then the first 32 characters of the target name to be constructed are returned as target_name.

The entryname argument that is passed to get_equal_name_ can also be used as the target_name argument, as long as the argument has a length of 32 characters.

Name: get_lock_id_

The get_lock_id_ subroutine returns the 36-bit unique lock identifier to be used by a process in setting locks. By using this lock identifier, a convention can be established so that a process wishing to lock a data base and finding it already locked can verify that the lock is set by an existing process.

Usage

```
declare get_lock_id_ entry (bit(36) aligned);

call get_lock_id_ (lock_id);
```

where lock_id (Output) is the unique identifier of this process used in locking. For a more detailed discussion of locking see the set_lock_ description in the MPM Subroutines.

Name: get_privileges_

The get_privileges_ function returns the access privileges of the process. (See "Access Control" in Section VI 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 (Output) is a bit string with a bit set ("1"b) for each access privilege the process has.

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 is "0"b.

Name: get_ring_

The get_ring_ function returns to the caller the number of the protection ring in which the caller is executing. For a discussion of rings see "Intraprocess Access Control" in Section VI of the MPM Reference Guide.

Usage

```
declare get_ring_ entry returns (fixed bin(3));

ring_no = get_ring_ ();
```

where ring_no (Output) is the number of the ring in which the caller is executing.

Name: get_system_free_area_

The get_system_free_area_ function returns a pointer to the system free area for the ring in which it was called (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 (Output) points to the system free area.

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 (Input)
   is the pathname of the containing directory.

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

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

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

5. ring (Input)
   is the ring number of the initial ACL.

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

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 (Input)
   is the pathname of the containing directory.

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

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

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

5. ring (Input)
   is the ring number of the initial ACL.

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

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 (Input)
   is the pathname of the containing directory.

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

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

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" in Section VI 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 (Input)
   is the pathname of the containing directory.

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

3. acl_ptr (Input)
   points to the user-filled delete_acl structure as described in the hcs_$delete_inacl_entries entry point.

4. acl_count (Input)
   is the number of initial ACL entries in the delete_acl structure.

5. ring (Input)
   is the ring number of the initial ACL.

6. code (Output)
   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 (Input)
   is the pathname of the containing directory.

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

3. acl_ptr (Input)
   points to the user-filled delete_acl structure. See "Notes" below.

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

5. ring (Input)
   is the ring number of the initial ACL.

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

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.

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 (Input)
   is the pathname of the containing directory.

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

3. chase (Input)
   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.
   0 return link author
   1 return segment, directory, or multisegment file author

4. author (Output)
   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.

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

Note

The user must have status permission on the containing directory.

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 (Input)
   is the pathname of the containing directory.

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

3. bc_author (Output)
   is the bit count author of the segment or directory in the form of Person_id.Project_id.tag with a maximum length of 32 characters. An error is not detected if the string, bc_author, is too short to hold the bit count author.

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

Note

The user must have status permission on the containing directory.

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 (Input)
   is the pathname of the containing directory.

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

3. drb (Output)
   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.

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

Notes

The user must have status permission on the containing directory.

Ring brackets are discussed in "Intraprocess Access Control" in Section VI of the MPM Reference Guide.

Name: hcs_$get_link_target

The hcs_$get_link_target entry point returns the target pathname of a link. Links are not chased.

Usage

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

call hcs_$get_link_target (dir_name, entryname, link_dir_name,
     link_entryname, code);
```

where:

1. dir_name (Input)
   is the directory name containing the link.

2. entryname (Input)
   is the entryname of the link for which target information is desired.

3. link_dir_name (Output)
   is the directory name of the link target with a maximum length of 168 characters.

4. link_entryname (Output)
   is the entryname of the link target with a maximum length of 32 characters.

5. code (Output)
   is a standard status code.

Note

The user must have status permission on the containing directory.

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 (Input)
   is the pathname of the containing directory.

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

3. max_length (Output)
   is the maximum length of the segment in words.

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

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 (Input)
   is a pointer to the segment whose maximum length is to be returned.

2. max_length (Output)
   is the maximum length of the segment in words.

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

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 (Input)
   is the pathname of the containing directory.

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

3. rb (Output)
   is a three-element array that contains the segment's ring brackets. Ring brackets and validation levels are discussed in "Intraprocess Access Control" in Section VI of the MPM Reference Guide.

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

Note

The user must have status permission on the containing directory.

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 (Input)
   is the pathname of the containing directory.

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

3. safety_sw (Output)
   is the value of the safety switch.
   "0"b the segment or directory can be deleted
   "1"b the segment or directory cannot be deleted

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

Note

The user must have status permission on the containing directory or nonnull access to the directory or segment.

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 (Input)
   is a pointer to the segment whose safety switch is to be examined.

2. safety_sw (Output)
   is the value of the segment safety switch.
   "0"b the segment can be deleted
   "1"b the segment cannot be deleted

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

Note

The user must have status permission on the directory containing the segment or must have nonnull access to the segment.

Name: hcs_$get_search_rules

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 (Input) is a pointer to a user-supplied search rules structure. See "Note" below.

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.)

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 (Input)
   is a pointer to the structure described in "Notes" below.

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

Notes

The structure pointed to by search_rules_ptr is declared as follows:

```
dcl 1 drules            based aligned,
      2 ntags           fixed bin,
      2 nrules          fixed bin,
      2 tags (10),
        3 name          char(32),
        3 flag          bit(36),
      2 rules (50),
        3 name          char(168),
        3 flag          bit(36);
```

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.

hcs_$get_system_search_rules

7. rules.name
   is the absolute pathname of the directory.

8. rules.flag
   is a bit field with bits on for every tag that selects this directory.

Name: hcs_$get_user_effmode

The hcs_$get_user_effmode entry point returns the effective access mode of a user to a branch, given the pathname of the branch, the name of the user, and the validation level (ring number) of the user. (For a description of this mode, see "Effective Access" in Section VI of the MPM Reference Guide.)

Usage

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

call hcs_$get_user_effmode (dir_name, entryname, user_id, ring, mode,
     code);
```

where:

1. dir_name (Input)
   is the directory name of the branch.

2. entryname (Input)
   is the entry name of the branch.

3. user_id (Input)
   is the access name of the user in the form Person_id.Project_id.tag. This is limited to 32 characters. If null, the access name of the calling process is used.

4. ring (Input)
   is the validation level that is to be used in computing effective access. It must be a value between 0 and 7 inclusive.

5. mode (Output)
   is the effective access mode of the user to the branch (see "Notes" below).

6. code (Output)
   is a standard status code.

Notes

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

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

The modes for directories are:

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

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

The user must have status permission on the containing directory.

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 (Input)
   is a pointer to a structure containing the new search rules. See "Notes" below.

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

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 that may be returned from this entry point are:

```
error_table_$bad_string   (not a pathname or keyword)
error_table_$notadir
error_table_$too_many_sr
```

Additional codes can be returned from other procedures that are called by hcs_$initiate_search_rules.

For the values of the site-defined keywords, the user may call the hcs_$get_system_search_rules entry point.

Name: hcs_$list_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 (Input)
   is the pathname of the containing directory.

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

3. area_ptr (Input)
   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).

4. area_ret_ptr (Output)
   points to the start of the allocated list of initial ACL entries.

5. acl_ptr (Input)
   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.

6. acl_count (Input or Output)
   is the number of entries in the ACL structure.
   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 (Input)
   is the ring number of the initial ACL.

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

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 (Input)
   is the pathname of the containing directory.

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

3. area_ptr (Input)
   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).

4. area_ret_ptr (Output)
   points to the start of the allocated list of initial ACL entries.

5. acl_ptr (Input)
   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.

6. acl_count (Input or Output)
   is the number of entries in the initial ACL structure.
   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 (Input)
   is the ring number of the initial ACL.

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

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.

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 (Input)
   is the pathname of the containing directory.

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

3. quota_change (Input)
   is the number of records of secondary storage quota to be moved between the superior directory and the inferior directory. (See "Notes" below.)

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

Notes

The entryname specified by the entryname argument must be a directory.

The user must have modify permission on both directories.

After the quota change, the remaining quota in each directory must be greater than the number of records used in that directory.

The quota_change argument can be either a positive or negative number. If it is positive, the quota is moved from dir_name to entryname. If it is negative, the move is from entryname to dir_name. If the change results in zero quota left on entryname, that directory is assumed to no longer contain a terminal quota and all of its used records are reflected up to the used records on dir_name. It is a restriction that no quota in any of the directories superior to entryname can be modified from a nonzero value to a zero value by this subroutine.

Name: hcs_$quota_read

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 (Input)
   is the pathname of the directory for which quota information is desired.

2. quota (Output)
   is the segment record quota in the directory.

3. trp (Output)
   is the time-record product (trp) charged to the directory. This double-precision number is in units of record-seconds.

4. tup (Output)
   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.

5. sons_lvid (Output)
   is the logical volume ID for segments contained in this directory.

6. tacc_sw (Output)
   is the terminal account switch. The setting of this switch determines how charges are made.
   1 records are charged against the quota in this directory
   0 records are charged against the quota in the first superior directory with a terminal account

7. used (Output)
   is the number of records used by segments in this directory and by segments in nonterminal inferior directories.

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

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 (Input)
   is the pathname of the containing directory.

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

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

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

5. no_sysdaemon_sw (Input)
   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.
   "0"b adds sma *.SysDaemon.* entry
   "1"b replaces the existing initial ACL with only the user-supplied dir_acl

6. ring (Input)
   is the ring number of the initial ACL.

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

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 (Input)
   is the pathname of the containing directory.

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

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

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

5. no_sysdaemon_sw (Input)
   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.
   "0"b adds rw *.SysDaemon.* entry
   "1"b replaces the existing initial ACL with only the user-supplied segment_acl

6. ring (Input)
   is the ring number of the initial ACL.

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

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.

hcs_$set_dir_ring_brackets

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 (Input)
   is the pathname of the containing directory.

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

3. drb (Input)
   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.

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

Notes

The user must have modify permission on the containing directory. Also, the validation level must be less than or equal to both the present value of the first ring bracket and the new value of the first ring bracket that the user wishes set.

Ring brackets and validation levels are discussed in "Intraprocess Access Control" in Section VI the MPM Reference Guide.

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 (Input)
   is the pathname of the containing directory.
2. entryname (Input)
   is the entryname of the segment.
3. entry_bound (Input)
   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.
4. code (Output)
   is a storage system status code. (See "Notes" below.)

Notes

A directory cannot have its entry point bound changed.

The user must have modify permission on the containing directory.

If an attempt is made to set the entry point bound of a segment greater than the system maximum of 16383, code is set to error_table_$argerr.

The hcs_$set_entry_bound_seg entry point can be used when a pointer to the segment is given, rather than a pathname.

Name: hcs_$set_entry_bound_seg

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 (Input)
   is a pointer to the segment whose entry point bound is to be changed.

2. entry_bound (Input)
   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.

3. code (Output)
   is a storage system status code. (See "Notes" below.)

Notes

A directory cannot have its entry point bound changed.

The user must have modify permission on the containing directory.

If an attempt is made to set the entry point bound of a segment to greater than the system maximum of 16383, code is set to error_table_$argerr.

The hcs_$set_entry_bound entry point can be used when a pathname of the segment is given, rather than a pointer.

Name: hcs_$set_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 (Input)
   is the pathname of the containing directory.

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

3. max_length (Input)
   is the new value in words for the maximum length of the segment.

4. code (Output)
   is a storage system status code. (See "Notes" below.)

Notes

A directory cannot have its maximum length changed.

The user must have modify permission on the containing directory.

The maximum length of a segment is accurate to units of 1024 words, and if max_length is not a multiple of 1024 words, it is set to the next multiple of 1024 words.

If an attempt is made to set the maximum length of a segment to greater than the system maximum, sys_info$max_seg_size, code is set to error_table_$argerr. The sys_info data base is described in Section VIII of this manual.

If an attempt is made to set the maximum length of a segment to less than its current length, code is set to error_table_$invalid_max_length.

The hcs_$set_max_length_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 (Input)
   is the pointer to the segment whose maximum length is to be changed.

2. max_length (Input)
   is the new value in words for the maximum length of the segment.

3. code (Output)
   is a storage system status code. (See "Notes" below.)

Notes

A directory cannot have its maximum length changed.

The user must have modify permission on the containing directory.

The maximum length of a segment is accurate to units of 1024 words, and if max_length is not a multiple of 1024 words, it is set to the next multiple of 1024 words.

If an attempt is made to set the maximum length of a segment to greater than the system maximum, sys_info$max_seg_size, code is set to error_table_$argerr. The sys_info data base is described in Section VIII of this manual.

If an attempt is made to set the maximum length of a segment to less than its current length, code is set to error_table_$invalid_max_length.

The hcs_$set_max_length entry point can be used when a pathname of the segment is given, rather than the pointer.

Name: hcs_$set_ring_brackets

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 (Input)
   is the pathname of the containing directory.

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

3. rb (Input)
   is a three-element array specifying the ring brackets of the segment; see "Notes" below.

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

Notes

Ring brackets must be ordered as follows:

rb1 <= rb2 <= rb3

The user must have modify permission on the containing directory. Also, the validation level must be less than or equal to both the present value of the first ring bracket and the new value of the first ring bracket that the user wishes set.

Ring brackets and validation levels are discussed in "Intraprocess Access Control" in Section VI 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 II 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 (Input)
   is the pathname of the containing directory.

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

3. safety_sw (Input)
   is the new value of the safety switch.
   "0"b if the segment can be deleted
   "1"b if the segment cannot be deleted

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

Notes

The user must have modify permission on the containing directory.

The hcs_$set_safety_sw_seg entry point can be used when the pointer to the segment is given, rather than a pathname.

Name: hcs_$set_safety_sw_seg

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 II 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 (Input)
   is the pointer to the segment.

2. safety_sw (Input)
   is the new value of the safety switch.
   "0"b if the segment can be deleted
   "1"b if the segment cannot be deleted

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

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 Reference Guide.) 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.

The main entry point returns the storage system type and all names that match the given entryname. (The hcs_$star_dir_list_ and hcs_$star_list_ entry points described below return more information about each entry. The hcs_$star_dir_list_ entry point returns only information kept in the directory branch, while the hcs_$star_list_ entry point returns information kept in the volume table of contents (VTOC). Accessing the VTOC is an additional expense, and it can be quite time consuming to access the VTOC entries for all branches in a large directory. Further, if the volume is not mounted, it is impossible to access the VTOC. Therefore, use of the hcs_$star_dir_list_ entry point is recommended for all applications in which information from the VTOC is not essential.

Status permission is required on the directory to be searched.

Usage

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

call hcs_$star_ (dir_name, star_name, select_sw, area_ptr, entry_count,
     entry_ptr, n_ptr, code);
```

where:

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

2. star_name (Input)
   is the entryname that can contain asterisks or question marks.

3. select_sw (Input)
   indicates what information is to be returned. 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 (Input)
   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.

5. entry_count (Output)
   is a count of the number of entries that match the entryname.

6. entry_ptr (Output)
   is a pointer to the allocated structure in which information on each entry is returned.

7. n_ptr (Output)
   is a pointer to the allocated array of all the entrynames in this directory that match star_name. See "Notes" below.

8. code (Output)
   is a storage system status code. See "Status Codes" below.

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.

Status Codes

If no match with star_name was found in the directory, code will be returned as error_table_$nomatch.

If star_name contained illegal syntax with respect to the star convention, code will be returned as error_table_$badstar.

If the user did not provide enough space in the area to return all requested information, code will be returned as error_table_$notalloc. In this case, the total number of entries (for hcs_$star_) or the total number of branches and the total number of links (for hcs_$star_list_ and hcs_$star_dir_list_) will be returned, to provide an estimate of space required.

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_dir_list_ entry point.

Usage

```
declare hcs_$star_list_ entry (char(*), char(*), fixed bin(3), ptr,
     fixed bin, fixed bin, ptr, ptr, fixed bin(35));

call hcs_$star_list_ (dir_name, star_name, select_sw, area_ptr,
     branch_count, link_count, entry_ptr, n_ptr, code);
```

where:

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

2. star_name (Input)
   is the entryname that can contain asterisks or question marks.

3. select_sw (Input)
   indicates what information is to be returned. 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 (Input)
   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.

5. branch_count (Output)
   is a count of the number of segments and directories that match the entryname.

6. link_count (Output)
   is a count of the number of links that match the entryname.

7. entry_ptr (Output)
   is a pointer to the allocated structure in which information on each entry is returned.

8. n_ptr (Output)
   is a pointer to the allocated array in which selected entrynames and pathnames associated with link entries are stored.

9. code (Output)
   is a storage system status code. See "Status Codes" above in the description of hcs_$star_ entry point.

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;
```

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.

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 above.

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 the "Notes" section in the description of hcs_$get_user_effmode in this manual for a more detailed description of access modes.

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.

Name: hcs_$wakeup

The hcs_$wakeup entry point sends an interprocess communication wakeup signal to a specified process over a specified event channel. If that process has previously called the ipc_$block entry point, it is 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 (Input)
   is the process identifier of the target process.

2. channel_id (Input)
   is the identifier of the event channel over which the wakeup is to be sent.

3. message (Input)
   is the event message to be interpreted by the target process.

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

   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., process_id 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 (Input)
   is the name of a request type defined in the I/O daemon tables.

2. generic_type (Output)
   is the name of the generic type of the above request type.

3. code (Output)
   is a standard status code. If the specified request type is not found, the code error_table_$id_not_found is returned.

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 (Input)
   is the name of a request type defined in the I/O daemon tables.

2. access_name (Output)
   is the driver access name for the above request type.

3. code (Output)
   is a standard status code. If the specified request type is not found, the code error_table_$id_not_found is returned.

Name: ipc_

The Multics system supports an interprocess communication facility. The basic purpose of the facility is to provide control communication (by means of stop and go signals) between processes.

The ipc_ subroutine is the user's interface to the Multics interprocess communication facility. Briefly, that facility works as follows: a process establishes event channels in the current protection ring and waits for an event on one or more channels.

Event channels can be thought of as numbered slots in the interprocess communication facility tables. Each channel is either an event-wait or event-call channel. An event-wait channel receives events that are merely marked as having occurred and awakens the process if it is blocked waiting for an event on that channel. On an event-call channel, the occurrence of an event causes a specified procedure to be called if (or when) the process is blocked waiting for an event on any channel. Naturally, the specific event channel must be made known to the process that expected to notice the event. For an event to be noticed by an explicitly cooperating process, the event channel identifier value is typically placed in a known location of a shared segment. For an event to be noticed by a system module, a subroutine call is typically made to the appropriate system module. A process can go blocked waiting for an event to occur or can explicitly check to see if it has occurred. If an event occurs before the target process goes blocked, then it is immediately awakened when it does go blocked.

The user can operate on an event channel only if his ring of execution is the same as his ring when the event channel was created (for a discussion of rings see "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 (Output)
   is the identifier of the event channel.

2. code (Output)
   is a nonstandard status code; see "Status Code Values" later in this description.

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 channel_id (Input) and code (Output) are the same as described above for ipc_$create_ev_chn.

Entry: ipc_$decl_event_call_channel

This entry point changes an event-wait channel into an event-call channel.

Usage

```
declare ipc_$decl_event_call_channel entry (fixed bin(71), ptr, ptr,
     fixed bin, fixed bin(35));

call ipc_$decl_event_call_channel (channel_id, procedure_ptr, data_ptr,
     priority, code);
```

where:

1. channel_id (Input)
   is the identifier of the event channel.

2. procedure_ptr (Input)
   is a pointer to the procedure entry point invoked when an event occurs on the specified channel. The procedure entry point should not be an internal procedure.

3. data_ptr (Input)
   is a pointer to a region where data to be passed to and interpreted by that procedure entry point is placed.

4. priority (Input)
   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.

5. code (Output)
   is a nonstandard status code; see "Status Code Values" later in this description.

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 channel_id (Input) and code (Output) are the same as described above for ipc_$create_ev_chn.

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 channel_id (Input) and code (Output) are the same as described above for ipc_$create_ev_chn.

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 channel_id (Input) and code (Output) are the same as described above for ipc_$create_ev_chn.

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 channel_id (Input) and code (Output) are the same as described above for ipc_$create_ev_chn.

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 (Output) 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 and waiting on any of a list of channels. Only event channels in the current ring are affected. By default, event-call channels have priority.

Usage

```
declare ipc_$set_call_prior entry (fixed bin(35));

call ipc_$set_call_prior (code);
```

where code (Output) 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. This call causes a mask counter to be incremented. Event calls are masked if this counter is greater than zero.

Usage

```
declare ipc_$mask_ev_calls entry (fixed bin(35));

call ipc_$mask_ev_calls (code);
```

where code (Output) is a nonstandard status code; see "Status Code Values" later in this description.

Entry: ipc_$unmask_ev_calls

This entry point causes the event-call mask counter to be decremented. Event calls remain masked as long as the counter is greater than zero. To force event calls to become unmasked, call this entry point repeatedly, until a nonzero code is returned.

Usage

```
declare ipc_$unmask_ev_calls entry (fixed bin(35));

call ipc_$unmask_ev_calls (code);
```

where code (Output) is a nonstandard status code; see "Status Code Values" later in this description. A nonzero code is returned if event calls were not masked at the time of the call.

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 (Input)
   is a pointer to the base of a structure that specifies the channels on which events are being awaited.

```
dcl 1 wait_list             based aligned,
      2 nchan               fixed bin,
      2 pad                 fixed bin,
      2 channel_id (n refer (wait_list.nchan) fixed bin(71));
```

where:

nchan
is the number of channels.

pad
must be zero.

channel_id
: is an array of channel identifiers selecting the channels to wait on.

2. info_ptr (Input)
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).

```
dcl 1 event_info       based aligned,
      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 (Output)
is a nonstandard status code; see "Status Code Values" later in this description.

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 (Input)
   is the identifier of the event channel.

2. ev_occurred (Output)
   indicates whether an event occurred on the specified channel.
   0 no event occurred
   1 an event occurred

3. info_ptr (Input)
   is as above.

4. code (Output)
   is a nonstandard status code; see "Status Code Values" below.

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_$decl_event_call_channel entry point. The procedure is called with one argument, a pointer to the following structure:

```
dcl 1 event_info          based aligned,
      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 III of the MPM Reference Guide 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 (Input)
   is the entryname to be compared with the star name. Trailing spaces in the entryname are ignored.

2. star_name (Input)
   is the star name with which entryname is compared. Trailing spaces in the star name are ignored.

3. code (Output)
   is a standard status code. It can be:
   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: mhcs_$get_seg_usage

This entry point returns the number of page faults taken on a segment since its creation.

Usage

```
declare mhcs_$get_seg_usage entry (char(*), char(*), fixed bin(35),
     fixed bin(35));

call mhcs_$get_seg_usage (dir_name, entryname, use, code);
```

where:

1. dir_name (Input)
   is the directory containing the segment.

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

3. use (Output)
   is the page fault count.

4. code (Output)
   is a standard status code.

Notes

This entry point works for segments only and cannot be used to determine the page faults on a directory.

Entry: mhcs_$get_seg_usage_ptr

This entry point works the same as mhcs_$get_seg_usage except that it takes a pointer to the segment.

Usage

```
declare mhcs_$get_seg_usage_ptr entry (ptr, fixed bin(35), fixed bin(35));

call mhcs_$get_seg_usage_ptr (s_ptr, use, code)
```

where:

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

2. use (Output)
   is as above.

3. code (Output)
   is as above.

Name: msf_manager_

The msf_manager_ subroutine provides a centralized and consistent facility for handling multisegment files. Multisegment files are files that can require more than one segment for storage. Examples of multisegment files are listings, data used through I/O switches, and APL workspaces. The msf_manager_ subroutine makes multisegment files almost as easy to use as single segment files in many applications.

A multisegment file is composed of one or more components, each the size of a segment, identified by consecutive unsigned integers. Any word in a single segment file can be specified by a pathname and a word offset. Any word in a multisegment file can be specified by a pathname, component number, and word offset within the component. The msf_manager_ subroutine provides the means for creating, accessing, and deleting components, truncating the multisegment file, and controlling access.

In this implementation, a multisegment file with only component 0 is stored as a single segment file. 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 (Input)
   is the pathname of the containing directory.

2. entryname (Input)
   is the entryname of the multisegment file.

3. fcb_ptr (Output)
   is a pointer to the file control block.

4. code (Output)
   is a storage system status code. The code error_table_$dirseg is returned when an attempt is made to open a directory.

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 (Input)
   is a pointer to the file control block.

2. component (Input)
   is the number of the component desired.

3. create_sw (Input)
   is the create switch.
   "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 (Output)
   is a pointer to the specified component in the file, or null (if there is an error).

5. bc (Output)
   is the bit count of the component.

6. code (Output)
   is a storage system status code. 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_$adjust entry point optionally sets the bit count, truncates, and terminates the components of a multisegment file. The number of the last component and its bit count must be given. The bit counts of all components with numbers less than the given component are set to sys_info$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 (Input)
   is a pointer to the file control block.

2. component (Input)
   is the number of the last component.

3. bc (Input)
   is the bit count to be placed on the last component.

4. switch (Input)
   is a 3-bit count/truncate/terminate switch.

   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 (Output)
   is a storage system status code.

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.

Entry: msf_manager_$acl_list

This entry point returns the access control list (ACL) of a multisegment file.

Usage

```
declare msf_manager_$acl_list entry (ptr, ptr, ptr, ptr, fixed bin,
     fixed bin(35));

call msf_manager_$acl_list (fcb_ptr, area_ptr, area_ret_ptr, acl_ptr,
     acl_count, code);
```

where:

1. fcb_ptr (Input)
   is a pointer to the file control block.

2. area_ptr (Input)
   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).

3. area_ret_ptr (Output)
   points to the start of the allocated list of ACL entries.

4. acl_ptr (Input)
   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.

5. acl_count (Input/Output)
   is the number of entries in the segment_acl structure.
   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 (Output)
is a storage system status code.

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 mode of the file or has status_code set to error_table_$user_not_found and contains a mode of 0.

Entry: msf_manager_$acl_replace

This entry point replaces the ACL of a multisegment file.

Usage

```
declare msf_manager_$acl_replace entry (ptr, ptr, fixed bin, bit(1),
     fixed bin(35));

call msf_manager_$acl_replace (fcb_ptr, acl_ptr, acl_count, no_sysdaemon_sw
     code);
```

where:

1. fcb_ptr (Input)
   is a pointer to the file control block.

2. acl_ptr (Input)
   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.

3. acl_count (Input)
   is the number of entries in the segment_acl structure.

4. no_sysdaemon_sw (Input)
   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.
   "0"b adds rw *.SysDaemon.* entry
   "1"b replaces the existing ACL with only the user-supplied segment_acl

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

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 (Input)
   is a pointer to the file control block.

2. acl_ptr (Input)
   points to the user-supplied segment_acl structure (described in the msf_manager_$acl_list entry point above).

3. acl_count (Input)
   is the number of ACL entries in the segment_acl structure.

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

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 (Input)
   is a pointer to the file control block.

2. acl_ptr (Input)
   points to a user-supplied delete_acl structure. See "Notes" below.

3. acl_count (Input)
   is the number of ACL entries in the delete_acl structure.

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

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 structure described in "Information Structure" below.

Entry: object_info_$brief

This entry point returns only the structural information necessary to locate the object's major sections.

Usage

```
declare object_info_$brief entry (ptr, fixed bin(24), ptr, fixed bin(35));

call object_info_$brief (seg_ptr, bc, info_ptr, code);
```

where:

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

2. bc (Input)
   is the bit count of the object segment.

3. info_ptr (Input)
   is a pointer to the info structure in which the object information is returned. See "Information Structure" later in this description.

4. code (Output)
   is a standard status code.

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 all the arguments are the same as for the object_info_$brief entry point above.

Entry: object_info_$long

This entry point returns, in addition to the information supplied by the object_info_$display entry point, the data required by the Multics binder.

Usage

```
declare object_info_$long entry (ptr, fixed bin(24), ptr, fixed bin(35));

call object_info_$long (seg_ptr, bc, info_ptr, code);
```

where all the arguments are the same as in the object_info_$brief entry point above.

Information Structure

The information structure is as follows (as defined in the system include file object_info.incl.pl1):

```
dcl 1 object_info             aligned,
    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 perprocess_static    bit(1) unaligned,
        3 pad                  bit(27) 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,
        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. perprocess_static
    indicates whether the static section should be reinitialized for a run unit.
    "1"b static section is used as is
    "0"b static section is per run unit

23. pad
    is currently unused.

24. entry_bound
    is the entry bound if this is a gate procedure.

25. 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.

26. compiler
    is the name of the compiler that generated this object segment.

27. compile_time
    is the date and time this object was generated.

28. access_name
    is the access identifier (in the form Person_id.Project_id.tag) of the user in whose behalf this object was generated.

29. 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.

30. cvers.length
    is the length (in characters) of the compiler version string.

31. 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.

32. comment.length
    is the length (in characters) of the comment string.

33. 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.

34. rel_text
    is a pointer to the object's text section relocation information.

35. rel_def
    is a pointer to the object's definition section relocation information.

36. rel_link
    is a pointer to the object's linkage section relocation information.

37. rel_static
    is a pointer to the object's static section relocation information.

38. rel_symbol
    is a pointer to the object's symbol section relocation information.

39. 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.

40. static_boundary
    is analogous to text_boundary for internal static.

41. 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).

42. 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: pl1_io_

The pl1_io_ subroutine is a collection of utility functions for extracting information about PL/I files that is not available within the language itself.

Entry: pl1_io_$get_iocb_ptr

This function returns the I/O control block pointer for the Multics I/O System switch associated with an open PL/I file. This pointer may be used to perform control and modes operations upon the switch associated with that file.

Usage

```
declare pl1_io_$get_iocb_ptr entry (file) returns (ptr);

iocb_ptr = pl1_io_$get_iocb_ptr (file_variable);
```

where:

1. file_variable (Input)
   is a PL/I file value.

2. iocb_ptr (Output)
   is a pointer to the I/O control block for the file.

Notes

Performing explicit operations via the Multics I/O System upon switches in use by PL/I I/O is potentially dangerous unless care is taken that certain conventions are observed. No calls should be made that affect the data in the PL/I data set being accessed, the positioning of the data set, or the status or interpretation of any I/O operations that may be in progress. In general, this limits such calls to those which obtain status information.

Entry: pl1_io_$error_code

This function returns the last nonzero status code encountered by PL/I I/O while performing file operations. This is a standard Multics status code and describes the most recent error more specifically than the PL/I condition which is raised after an error.

Usage

```
declare pl1_io_$error_code entry (file) returns (fixed bin(35));

code = pl1_io_$error_code (file_variable);
```

where:

1. file_variable (Input)
   is a PL/I file value.

2. code (Output)
   is the last nonzero status code associated with the file.

Notes

The specific values returned by this function are subject to change. See "Handling Unusual Occurrences" in Section VII of the MPM Reference Guide.

Name: prepare_mc_restart_

The prepare_mc_restart_ subroutine to checks machine conditions for restartability, and makes modifications to the machine conditions (to accomplish user modifications to process execution) before a condition handler returns.

The prepare_mc_restart_ subroutine should be called by a condition handler, which was invoked as a result of a hardware-detected condition, if the handler wishes the process to:

1. retry the faulting instruction
2. skip the faulting instruction and continue
3. execute some other instruction instead of the faulting instruction and continue
4. resume execution at some other location in the same program

When a condition handler is invoked for a hardware-detected condition, it is passed a pointer to the machine-conditions data at the time of the fault. If the handler returns, the system attempts to restore these machine conditions and restart the process at the point of interruption encoded in the machine-conditions data. After certain conditions, however, the hardware is unable to restart the processor. In other cases, an attempt to restart always causes the same condition to occur again, because the system software has already exhausted all available recovery possibilities (e.g., disk read errors).

Entry: prepare_mc_restart_$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 (Input)
   is a pointer to the machine conditions.

2. code (Output)
   is a standard status code. If it is nonzero on return, the machine conditions cannot be restarted. See "Notes" below.

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 (Input)
   is as above.

2. new_ins (Input)
   is the desired substitute machine instruction.

3. code (Output)
   is as above.

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 (Input)
   is the same as in the prepare_mc_restart_$retry entry point above.

2. newp (Input)
   is used in replacing the instruction counter in the machine conditions.

3. code (Output)
   is the same as in the prepare_mc_restart_$retry entry point above.

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 VII of the MPM Reference Guide.

For all entry points in the prepare_mc_restart_ subroutine, the following codes can be returned:

error_table_$badarg	an invalid mc_ptr was provided
error_table_$no_restart	the machine conditions cannot be restarted
error_table_$bad_ptr	the restart location is not accessible
error_table_$useless_restart	the same error will occur again if restart is attempted

Name: read_allowed_

The read_allowed_ function determines whether a subject of specified authorization has access (with respect to the access isolation mechanism) to read an object of specified access class. For information on access classes, see "Nondiscretionary Access Control" in Section VI of the MPM Reference Guide.

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 (Input)
   is the authorization of the subject.

2. access_class (Input)
   is the access class of the object.

3. returned_bit (Output)
   indicates whether the subject is allowed to read the object.
   "1"b read is allowed
   "0"b read is not allowed

Name: read_write_allowed_

The read_write_allowed_ function 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. For information on access classes see "Nondiscretionary Access Control" in Section VI of the MPM Reference Guide.

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 (Input)
   is the authorization of the subject.

2. access_class (Input)
   is the access class of the object.

3. returned_bit (Output)
   indicates whether the subject is allowed to both read and write the object.
   "1"b read and write are allowed
   "0"b read and write are not allowed

Name: release_area_

The release_area_ subroutine cleans up an area after it is no longer needed. If the area is a segment acquired via the define_area_ subroutine, the segment is released to the free pool via the temporary segment manager. If the area was not acquired (only initialized) via the define_area_ subroutine then the area itself is reinitialized to the empty state. In certain cases when the area is defined by the system or when the area is extended in ring 0, the temporary segment manager is not used and the area segments are actually created and deleted. Segments acquired to extend the area are released to the free pool of temporary segments or deleted if they are not obtained from the temporary segment manager.

Usage

```
declare release_area_ entry (ptr);

call release_area_ (area_ptr);
```

where area_ptr (Input/Output) points to the area to be released.

Note

The release_area_ subroutine sets area_ptr to null after copying it to a local variable.

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 VII of the MPM Reference Guide.

Usage

```
declare signal_ entry options (variable);

call signal_ (name, mc_ptr, info_ptr, wc_ptr);
```

where:

1. name (Input)
   is the name (declared as a nonvarying character string) of the condition to be signalled.

2. mc_ptr (Input)
   is a pointer (declared as an aligned pointer) to the machine conditions at the time the condition was raised. This argument is used by system programs only in order to signal hardware faults. In user programs, this argument should be null if a third argument is supplied. This argument is optional.

3. info_ptr (Input)
   is a pointer (declared as an aligned character) 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 VII of the MPM Reference Guide. This argument is intended for use in signalling conditions other than hardware faults. This argument is also optional.

4. wc_ptr
   points to the machine conditions at the time a lower ring was entered to process a fault. This argument is used only by the system and only in the case where a condition that occurred in a lower ring is being signalled in the outer ring and when the lower ring has been entered to process a fault occurring in the outer ring. This argument is also optional.

Notes

If the signal_ subroutine returns to its caller, indicating that the handler has returned to it, the calling procedure should retry the operation that caused the condition to be signalled.

The PL/I signal statement differs from the signal_ subroutine in that the above parameters cannot be provided in the signal statement. Also, for PL/I-defined conditions, a call to the signal_ subroutine is not equivalent to a PL/I signal statement since information about these conditions is kept internally.

Name: 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 (Input)
   is a standard status code (declared fixed bin(35)) describing the reason for calling the sub_err_ subroutine.

2. name (Input)
   is the name (declared as a nonvarying character string) of the subsystem or module on whose behalf the sub_err_ subroutine is called.

3. flags (Input)
   describe how and whether restart may be attempted. The flags argument should be declared as a nonvarying character string. 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 (Input)
   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.

5. retval (Input/Output)
   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).

6. ctl_string (Input)
   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.

7. ioa_args (Input)
   are any arguments required for conversion by the ctl_string argument.

Operation

The sub_err_ subroutine proceeds as follows: the structure described below is filled in from the arguments to the sub_err_ subroutine and the signal_ subroutine is called to raise the sub_error_ condition.

When the standard system environment receives a sub_error_ signal, it prints a message of the form:

```
name error by sub_name|location
Status code message. Message from ctl_string.
```

The standard environment then sets retval to zero and returns, if the value 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 the structure described in "Info Structure" below.

Info Structure

The structure pointed to by software info_ptr is declared as follows:

```
dcl 1 info               aligned based (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: suffixed_name_

This subroutine handles storage system entrynames. It provides an entry point that creates a properly suffixed name from a user-supplied name that might or might not include a suffix, an entry point that changes the suffix on a user-supplied name that might or might not include the original suffix, and an entry point that finds a segment, a directory, or a multisegment file whose name matches a user-supplied name that might or might not include a suffix. It is intended to be used by commands that deal with segments with a standard suffix, but that do not require the user to supply the suffix in the command arguments.

Entry: suffixed_name_$find

This entry point attempts to find a directory entry whose name matches a user-supplied name that might or might not include a suffix. This directory entry can be a segment, directory, or a multisegment file.

Usage

```
declare suffixed_name_$find entry (char(*), char(*), char(*), char(32),
     fixed bin(2), fixed bin(5), fixed bin(35));

call suffixed_name_$find (directory, name, suffix, entry, type, mode,
     code);
```

where:

1. directory (Input)
   is the name of the directory in which the entry is to be found.

2. name (Input)
   is the name that has been supplied by the user, and that might or might not include a suffix.

3. suffix (Input)
   is the suffix that is supposed to be part of name. It should not contain a leading period.

4. entry (Output)
   is a version of name that includes a suffix. It is returned even if the directory entry, directory>entry, does not exist.

5. type (Output)
   is a switch indicating the type of directory entry that was found.

   0 no entry was found
   1 a segment was found
   2 a directory was found
   3 a multisegment file was found

6. mode (Output)
is the caller's access mode to the directory entry that was found. See the hcs_$append_branch entry point in the MPM Subroutines for a description of mode. The the caller's access mode to the multisegment file directory is returned for a multisegment file.

7. code (Output)
is a standard status code. It may be one of the following:
error_table_$noentry
    no directory entry that matches name was found
error_table_$no_info
    no directory entry that matches name was found, and furthermore, the caller does not have status permission to the directory
error_table_$incorrect_access
    a directory entry that matches name was found, but the caller has null access to this entry, and to the directory containing this entry
error_table_$entlong
    the properly suffixed name that was made is longer than name

---

Entry: suffixed_name_$make

This entry point makes a properly suffixed name out of a name supplied by the user that might or might not include a suffix.

Usage

```
declare suffixed_name_$make entry (char(*), char(*), char(32),
     fixed bin(35));

call suffixed_name_$make (name, suffix, proper_name, code);
```

where:

1. name (Input)
is as above.

2. suffix (Input)
is as above.

3. proper_name (Output)
is the suffixed version of name.

4. code (Output)
is a standard status code. It may be one of the following:
error_table_$entlong
    the properly suffixed name that was made is longer than proper_name; proper_name contains only a part of the properly suffixed name

Entry: suffixed_name_$new_suffix

This entry point creates a name with a new suffix by changing the (possibly existing) suffix on a user-supplied name to the new suffix. If there is no suffix on the user-supplied name, then the new suffix is merely appended to the user-supplied name.

Usage

```
declare suffixed_name_$new_suffix entry (char(*), char(*), char(*),
     char(32), fixed bin(35));

call suffixed_name_$new_suffix (name, suffix, new_suffix, new_name, code);
```

where:

1. name (Input)
   is as above.

2. suffix (Input)
   is the suffix that might or might not already be on name.

3. new_suffix (Input)
   is the new suffix.

4. new_name (Output)
   is the name that was created. If name ends with .suffix, then .new_suffix replaces .suffix in new_name. Otherwise, new_name is formed by appending .new_suffix to name.

5. code (Output)
   is a standard status code. It may be one of the following:
   error_table_$entlong
      meaning that the suffixed new name is longer than new_name and therefore new_name contains only part of the suffixed new name

Note

If error_table_$no_s_permission is encountered during the processing for suffixed_name_$find, it is ignored and is not returned in the status code.

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 that is 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 (Output) is the installation identifier.

Entry: system_info_$sysid

This entry point returns the eight-character system identifier that is typed in the header of the who command and at dial-up time.

Usage

```
declare system_info_$sysid entry (char(*));

call system_info_$sysid (sys);
```

where sys (Output) is the system identifier that identifies the current version of the system.

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 (Output)
   is the company or institution name (a maximum of 64 characters).

2. d (Output)
   is the department or division name (a maximum of 64 characters).

3. cc (Output)
   is the company name, double spaced (a maximum of 120 characters).

4. dd (Output)
   is the department name, double spaced (a maximum of 120 characters).

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 (Output)
   is the maximum number of users.

2. nn (Output)
   is the current number of users.

3. mu (Output)
   is the maximum number of load units (times 10).

4. nu (Output)
   is the current number of load units (times 10).

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 (Output) is the time the system came up.

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 (Output)
   is the time of the next scheduled shutdown. If none is scheduled, this is 0.

2. rsn (Output)
   is the reason for the next shutdown (a maximum of 32 characters). If it is not known, it is blank.

3. tn (Output)
   is the time the system will return. If it is not known, it is 0.

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 (Output)
   is the CPU-hour rate per shift.

2. log (Output)
   is the connect-hour rate per shift.

3. prc (Output)
   is the process-hour rate per shift.

4. cor (Output)
   is the page-second rate for main memory per shift.

5. dsk (Output)
   is the page-second rate for secondary storage.

6. reg (Output)
   is the registration fee per user per month.

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 (Output)
   is the number of devices with prices.

2. dev_ptr (Input)
   points to an array where device prices are stored.

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 is 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_$abs_chn

This entry point returns the event channel and process ID for the process that is running the absentee user manager.

Usage

```
declare system_info_$abs_chn entry (fixed bin(71), bit(36) aligned);

call system_info_$abs_chn (ec, p_id);
```

where:

1. ec (Output)
   is the event channel over which signals to absentee_user_manager_ should be sent.

2. p_id (Output)
   is the process ID of the absentee manager process (currently the initializer).

Entry: system_info_$next_shift_change

This entry point returns the number of the current shift, the time it started, the time it will end, and the number of the next shift.

Usage

```
declare system_info_$next_shift_change entry (fixed bin, fixed bin(71),
     fixed bin, fixed bin(71));

call system_info_$next_shift_change (now_shift, change_time, new_shift,
     start_time);
```

where:

1. now_shift (Output)
   is the current shift number.

2. change_time (Output)
   is the time the shift changes.

3. new_shift (Output)
   is the shift after change_time.

4. start_time (Output)
   is the time the current shift started.

Entry: system_info_$shift_table

This entry point returns the local shift definition table of the system.

Usage

```
declare system_info_$shift_table entry ((336) fixed bin);

call system_info_$shift_table (stt);
```

where stt (Output) is a table of shifts, indexed by half-hour within the week e.g., stt(1) gives the shift for 0000-0030 Mondays.

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 (Output)
   is the price per CPU hour for absentee queues 1 to 4.

2. realrate (Output)
   is the memory unit rate for absentee queues 1 to 4.

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 (Output) is the price per 1000 lines for each I/O daemon queue.

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 (Output)
   is the clock time of the last shutdown.

2. erfno (Output)
   is the ERF number of the last crash, or blank.

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 (Output) is the access ceiling.

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 (Output)
   is an array of the long level names.

2. short (Output)
   is an array of the short level names.

Entry: system_info_$category_names

This entry point returns the 32-character long names and the eight-character short names for the access categories.

Usage

```
declare system_info_$category_names entry (dim(18) char(32), dim(18) char(8));

call system_info_$category_names
     (long, short);
```

where the arguments are the same as for the system_info_$level_names entry point.

Entry: system_info_$ARPANET_host_number

This entry point returns the Advanced Research Projects Agency Network (ARPANET) address of the installation. If the installation is not attached to the ARPANET, the value -1 is returned.

Usage

```
declare system_info_$ARPANET_host_number entry (fixed bin(16));

call system_info_$ARPANET_host_number (host_num);
```

where host_num (Output) is the ARPANET host address.

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, alrm 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 (Input)
   is a pointer to a structure containing the machine conditions at the time of the process interrupt.

   name (Input)
   is the condition name: alrm for a real-time timer and cput for a CPU timer.

   (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. The call is made with all interrupts inhibited (i.e., all interprocess signal (IPS) are masked off). When the handler routine returns, interrupts are reenabled. If the handler routine does not return, interrupts are not reenabled and the user process may malfunction.

Usage

```
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 when the timer goes off. The call is made with all interrupts inhibited (i.e., all IPS are masked off). When the handler routine returns, interrupts are reenabled. If the handler routine does not return, interrupts are not reenabled and the user process may malfunction.

Usage

```
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.

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 (Input)
   is the pathname of the containing directory.

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

3. seg_ptr (Output)
   is a pointer to the segment, or is null if an error is encountered.

4. aclinfo_ptr (Output)
   is a pointer to ACL information (if any) needed by the tssi_$finish_segment entry point.

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

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.

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 (Input)
   is the pathname of the containing directory.

2. entryname (Input)
   is the entryname of the multisegment file.

3. seg_ptr (Output)
   is a pointer to component 0 of the file.

4. aclinfo_ptr (Output)
   is a pointer to ACL information (if any) needed by the tssi_$finish_file entry point.

5. fcb_ptr (Output)
   is a pointer to the file control block needed by the msf_manager_ subroutine.

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

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 (Input)
   is a pointer to the segment.

2. bc (Input)
   is the bit count of the segment.

3. mode (Input)
   is the access mode to be put on the segment.
   "110"b   re access
   "101"b   rw access

4. aclinfo_ptr (Input)
   is a pointer to the saved ACL information returned by the tssi_$get_segment entry point.

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

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 (Input)
   is a pointer to the file control block returned by the tssi_$get_file entry point.

2. component (Input)
   is the highest-numbered component in the file.

3. bc (Input)
   is the bit count of the highest-numbered component.

4. mode (Input)
   is the access mode to be put on the multisegment file.

5. aclinfo_ptr (Input)
   is a pointer to the saved ACL information returned by the tssi_$get_file entry point.

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

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).

Usage

```
declare tssi_$clean_up_segment entry (ptr);

call tssi_$clean_up_segment (aclinfo_ptr);
```

where aclinfo_ptr (Input) is a pointer to the saved ACL information returned by the tssi_$get_segment entry point.

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 (Input)
   is a pointer to the file control block returned by the tssi_$get_file entry point.

2. aclinfo_ptr (Input)
   is a pointer to the saved ACL information returned by the tssi_$get_segment entry point.

Name: ttt_info_

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

Entry: ttt_info_$terminal_data

This entry point returns a collection of information that describes a specified terminal type.

Usage

```
declare ttt_info_$terminal_data entry (char(*), fixed bin, fixed bin, ptr,
     fixed bin(35));

call ttt_info_$terminal_data (tt_name, line_type, baud, ttd_ptr, code);
```

where:

1. tt_name (Input)
   is the terminal type name.

2. line_type (Input)
   is a line type number against which the compatibility of the terminal type is verified. If nonpositive, the line type number is ignored. For further description, see the tty_ I/O module in the MPM Subroutines.

3. baud (Input)
   is a baud rate used to select the appropriate delay table.

4. ttd_ptr (Input)
   is a pointer to a structure in which information is returned. (See "Notes" below.)

5. code (Output)
   is a standard status code. If the terminal type is incompatible with the line type, a value of error_table_$incompatible_term_type is returned.

Notes

The ttd_ptr argument should point to the following structure:

```
dcl 1 terminal_type_data    aligned,
    2 version               fixed bin,
    2 old_type              fixed bin,
    2 name                  char(32) unaligned,
    2 tables,
      3 input_tr_ptr        ptr,
```

```
        3 output_tr_ptr       ptr,
        3 input_cv_ptr        ptr,
        3 output_cv_ptr       ptr,
        3 special_ptr         ptr,
        3 delay_ptr           ptr,
      2 editing_chars         unaligned,
        3 erase_char(1)       unaligned,
        3 kill char(1)        unaligned,
      2 mbz                   fixed bin(17) unaligned,
      2 flags,
        3 keybd_locking       bit(1) unaligned,
        3 mbz                 bit(35) unaligned;
```

where:

1. version (Input)
   is the version number of the above structure. It must be 1.

2. old_type (Output)
   is the old terminal type number that corresponds to the terminal type name. (The old terminal type number is provided only for compatibility with the obsolete tty_ order requests set_type and info.) A value of -1 indicates that no corresponding old type exists.

3. name (Output)
   is the terminal type name.

4. input_tr_ptr (Output)
   is a pointer to a structure containing the input translation table. This structure is identical to the info structure for the set_input_translation order of the tty_ I/O module described in the MPM Subroutines.

5. output_tr_ptr (Output)
   is a pointer to a structure containing the output translation table. This structure is identical to the info structure for the set_output_translation order of the tty_ I/O module described in the MPM Subroutines.

6. input_cv_ptr (Output)
   is a pointer to a structure containing the input conversion table. This structure is identical to the info structure for the set_input_conversion order of the tty_ i/O module described in the MPM Subroutines.

7. output_cv_ptr (Output)
   is a pointer to a structure containing the output conversion table. This structure is identical to the info structure for the set_output_conversion order of the tty_ I/O module described in the MPM Subroutines.

8. special_ptr (Output)
   is a pointer to a structure containing the special characters table. This structure is identical to the info structure for the set_special order of the tty_ I/O module described in the MPM Subroutines.

9. delay_ptr (Output)
   is a pointer to a structure containing the delay table. This structure is identical to the info structure for the set_delay order of the tty_ I/O module described in the MPM Subroutines.

10. erase (Output)
is the erase character.

11. kill (Output)
is the kill character.

12. keybd_locking (Output)
indicates whether the terminal type requires keyboard locking and unlocking.
"1"b yes
"0"b no

13. mbz (Input)
must be "0"b.

Entry: ttt_info_$modes

This entry point returns the default modes for a specified terminal type.

Usage

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

call ttt_info_$modes (tt_name, modes, code);
```

where:

1. tt_name (Input)
is the terminal type name.

2. modes (Output)
is the default modes string for the terminal type.

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

Entry: ttt_info_$preaccess_type

This entry point returns the terminal type name associated with a specified preaccess request.

Usage

```
declare ttt_info_$preaccess_type entry (char(*), char(*),

call ttt_info_$preaccess_type (request, tt_name, code));
```

where:

1. request (Input)
   is one of the following three preaccess requests: MAP, 963, or 029.

2. tt_name (Output)
   is the name of the associated terminal type.

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

Entry: ttt_info_$additional_info

This entry point returns additional information for a specified terminal type to be used by I/O modules other than tty_.

Usage

```
dcl ttt_info_$additional_info entry (char(*), char(512) varying,
     fixed bin(35));

call ttt_info_$additional_info (tt_name, add_info, code);
```

where:

1. tt_name (Input)
   is the terminal type name.

2. add_info (Output)
   is the additional information string. If no additional information is defined for the terminal type, a null string is returned.

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

Entry: ttt_info_$initial_string

This entry point returns a string that can be used to initialize terminals of a specified terminal type. The string must be transmitted to the terminal in raw output (rawo) mode. The initial string is most commonly used to set tabs on terminals that support tabs set by software.

Usage

```
declare ttt_info_$initial_string entry (char(*), char(512) varying,
     fixed bin(35));

call ttt_info_$initial_string (tt_name, istr_info, code);
```

where:

1. tt_name (Input)
   is the terminal type name.

2. istr_info (Output)
   is the initial string. If no initial string is defined for the terminal type, a null string is returned.

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

Entry: ttt_info_$dialup_flags

This entry point returns the values of two flags for a specified terminal type.

Usage

```
declare ttt_info_$dialup_flags entry (char(*), bit(1), bit(1),
     fixed bin(35));

call ttt_info_$dialup_flags (tt_name, ppm_flag, cpo_flag, code);
```

where:

1. tt_name (Input)
   is the terminal type name.

2. ppm_flag (Output)
   indicates whether a preaccess message should be printed when an unrecognizable login line is received from a terminal of the specified type:
   "1"b yes
   "0"b no

3. cpo_flag (Output)
   indicates whether "conditional printer off" is defined for the terminal type, i.e., if the answerback indicates whether a terminal is equipped with the printer off feature:
   "1"b yes
   "0"b no

4. code (Output)
   is a standard status code.

Entry: ttt_info_$decode_answerback

This entry point decodes a specified answerback string into a terminal type name and terminal identifier.

Usage

```
declare ttt_info_$decode_answerback entry (char(*), fixed bin, char(*),
     char(*), fixed bin(35));

call ttt_info_$decode_answerback entry (ansb, line_type, tt_name, id,
     code);
```

where:

1. ansb (Input)
   is the answerback string.

2. line_type (Input)
   is a line type number with which the decoded terminal type must be compatible. A nonpositive line type number is ignored. For further description, see the tty_ I/O module in the MPM Subroutines.

3. tt_name (Output)
   is the terminal type name decoded from the answerback. If no terminal type is indicated, a blank string is returned.

4. id (Output)
   is the terminal identifier decoded from the answerback. If no id is indicated, a blank string is returned.

5. code (Output)
   is a standard status code.

Entry: ttt_info_$encode_type

This entry point obtains a code number that corresponds to a specified terminal type name.

Usage

```
declare ttt_info_$encode_type entry (char(*), fixed bin, fixed bin(35));

call ttt_info_$encode_type (tt_name, type_code, code);
```

where:

1. tt_name (Input)
   is the terminal type name.

2. type_code (Output)
   is the corresponding terminal type code number.

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

Entry: ttt_info_$decode_type

This entry point obtains the terminal type name that corresponds to a specified terminal type code number.

Usage

```
declare ttt_info_$decode_type entry (fixed bin, char(*), fixed bin(35));

call ttt_info_$decode_type (type_code, tt_name, code);
```

where:

1. type_code (Input)
   is the terminal type code number.

2. tt_name (Output)
   is the corresponding terminal type name.

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

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 (Input) is a nonlocal label variable.

Name: write_allowed_

The write_allowed_ function determines whether a subject of specified authorization has access (with respect to the access isolation mechanism) to write an object of specified access class. For information on access classes, see "Nondiscretionary Access Control" in Section VI of the MPM Reference Guide.

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 (Input)
   is the authorization of the subject.

2. access_class (Input)
   is the access class of the object.

3. returned_bit (Output)
   indicates whether the subject is allowed to write the object.
   "1"b write is allowed
   "0"b write is not allowed

SECTION VIII

DATA BASE DESCRIPTIONS

This section contains descriptions of some Multics data bases presented in alphabetical order. Each description contains the name of the data base, discusses its purpose, and shows the correct usage.

Name

The "Name" heading shows the acceptable name by which the data base is referenced. The name is usually followed by a discussion of the purpose and function of the data base and the results that may be expected from referencing it.

Usage

This part of the data base description contains a declaration of the data base and its structure.

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 sys_info$ips_mask_data           count)),
        3 mask                                 bit(35) aligned,
        3 name                                 char(4) aligned,
      sys_info$page_size                       fixed bin(35),
      sys_info$max_seg_size                    fixed bin(35),
      sys_info$default_stack_length            fixed bin(35),
      sys_info$access_class_ceiling            bit(72),
      sys_info$time_correction_constant        fixed bin(71),
      sys_info$maxlinks                        fixed bin,
      sys_info$time_delta                      fixed bin(35),
      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.

3. count
   is the current number of valid IPS names.

4. mask
   is the IPS mask for the corresponding name. The mask has one bit on, and the rest of the bits are off.

5. name
   is the name used to signal the IPS condition.

6. page_size
   is the page size in words.

7. max_seg_size
   is the maximum segment size in words.

8. default_stack_length
   is the default stack maximum size in words.

9. access_class_ceiling
   is the maximum access class.

10. time_correction_constant
    is the correction from Greenwich mean time (GMT) in microseconds.

11. maxlinks
    is the maximum depth to which the system chases a link without finding a branch.

12. time_delta
    is the same as time_correction_ constant, only if single precision.

13. time_of_bootload
    is the clock reading at the time of bootload.

14. time_zone
    is the name of the time zone (e.g., EST).

Name: time_table_$zones

This data base is a table that defines the list of time zones accepted by the convert_date_to_binary_, decode_clock_value_, and encode_clock_value_ subroutines (all described in the MPM Subroutines). The table structure is defined the system include file, in time_zones_.incl.pl1. Time zones may be referenced using either uppercase or lowercase abbreviated zone names. The following is a list of abbreviations given in the system-supplied table. A site may modify this table to define other appropriate time zone abbreviations.

- GMT Greenwich mean time, zone east of the prime meridian (0 longitude), which runs through Greenwich, England, UK.
- EST Eastern Standard Time, 5 hours before GMT, including the eastern US.
- EDT Eastern Daylight Time, applies daylight savings to EST zone, giving time 4 hours before GMT.
- CST Central Standard Time, 6 hours before GMT, including the mid-western US.
- CDT Central Daylight Time, applies daylight savings to CST zone, giving time 5 hours before GMT.
- MST Mountain Standard Time, 7 hours before GMT, including the Rocky Mountain states of the US.
- MDT Mountain Daylight Time, applies daylight savings to MST zone, giving time 6 hours before GMT.
- PST Pacific Standard Time, 8 hours before GMT, including the west coastal states of the US.
- PDT Pacific Daylight Time, applies daylight savings to PST zone, giving time 7 hours before GMT.

Usage

```
dcl 1 time_zones       aligned based (addr (time_table_$zones)),
      2 version        fixed bin,
      2 number         fixed bin,
      2 values (0 refer (time_zones.number)),
        3 zone         char(3) aligned,
        3 pad          fixed bin,
        3 zone_offset  fixed bin(71);
```

where:

1. time_zones
   is the structure located in time_table_$zones.

2. version
   is the version number of this structure (currently version 1).

3. number
   is the number of time zones in the table.

4. zone
   is the abbreviated time zone character string in uppercase or lowercase.

5. pad
   must be set to zero.

6. zone_offset
   is the offset, in microseconds, which must be added to convert a time expressed in this time zone to a time expressed in the GMT zone.

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

-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	

A

(continued)

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

(continued)

T

U

V

W

CUT ALONG LINE

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

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Your comments will be promptly investigated by appropriate technical personnel and action will be taken as required. If you require a written reply, check here and furnish complete mailing address below. ☐

FROM: NAME ____________________ DATE ____________

TITLE ____________________

COMPANY ____________________

ADDRESS ____________________

____

CUT ALONG LINE

FOLD ALONG LINE

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

FOLD ALONG LINE

Honeywell