SERIES 60 (LEVEL 68)

MULTICS PROGRAMMERS' MANUAL — SUBSYSTEM WRITERS' GUIDE

SUBJECT

Reference Guide for Advanced Multics Users Writing Their Own Subsystems

SPECIAL INSTRUCTIONS

This manual is one of six 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
Communicatons Input/Output	Order No. CC92
Peripheral Input/Output	Order No. AX49

This manual supersedes AK92, Rev. 1 dated September 1975, and its addenda (Addendum A dated July 1976, Addendum B dated February 1977, and Addendum C dated November 1977). Except in the areas where there have been extensive revisions, such as an entirely new command or subroutine, marginal change indicators have been included in this edition.

SOFTWARE SUPPORTED

Multics Software Release 7.0

ORDER NUMBER

AK92, Rev. 2

March 1979

Honeywell

PREFACE

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

Document	Referred To In Text As
Reference Guide (Order No. AG91)	MPM Reference Guide
Commands and Active Functions (Order No. AG92)	MPM Commands
Communications Input/Output (Order No. CC92)	MPM Communications I/O
Subroutines (Order No. AG93)	MPM Subroutines
Subsystem Writers' Guide (Order No. AK92)	MPM Subsystem Writers' Guide
Peripheral Input/Output (Order No. AX49)	MPM Peripheral I/O

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

The MPM Commands is organized into four sections. Section 1 contains a * list of the Multics command repertoire, arranged functionally. Section 2 describes the active functions. Section 3 contains descriptions of standard Multics commands, including the calling sequence and usage of each command. Section 4 describes the requests used to gain access to the system.

The MPM Peripheral 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.

The MPM Communications I/O manual contains information about the Multics communications system. Included are sections on the commands, subroutines, and I/O modules used to manipulate communications I/O. Special purpose communications I/O, such as binary synchronous communication, is also included.

File No.: 1L13

The MPM Communications I/O manual contains information about the Multics communications system. Included are sections on the commands, subroutines, and I/O modules used to manipulate communications I/O. Special purpose communications I/O, such as binary synchronous communication, is also included.

The MPM Subroutines is organized into three sections. Section 1 contains a list of the subroutine repertoire, arranged functionally. Section 2 contains descriptions of the standard Multics subroutines, including the declare statement, the calling sequence, and usage of each. Section 3 contains the descriptions of the I/O modules.

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

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

- A subsystem that precisely imitates the command environment of some system other than Multics.

- A subsystem intended to enforce restrictions on the services available to a set of users (e.g., an APL-only subsystem for use in an academic class).

- A subsystem that protects some kind of information in a way not easily expressible with ordinary access control lists (e.g., a proprietary linear programming system, or an administrative data base system that permits access only to program-defined, aggregated information such as averages and correlations).

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

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

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

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

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

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

- Each manual contains an index that lists items by name and page number.

Changes and Additions to MPM Subsystem Writers' Guide, AK92, Rev. 2, Addendum D

The following subroutine and entry point descriptions are new to this manual and do not contain change bars.

get_external_variable_
hcs_$get_uid_seg
msf_manager_$msf_get_ptr
read_password_$switch
set_ext_variable_
set_ext_variable_$locate
sus_signal_handler_
sus_signal_handler_$reconnect_ec_enable
sus_signal_handler_$reconnect_ec_disable

The signal command is new to this manual and does not contain change bars.

The display_component_name and list_external_external_variables commands were inadvertently omitted from the previous addendum. They are included in this addendum, and do not contain change bars.

The mode_string_ subroutine has been moved to the MPM Subroutines manual.

The following subroutine and entry point descriptions are obsolete and have been deleted.

convert_ipc_code_
resource_control_$assign
resource_control_$get_status
resource_control_$set_status
resource_control_$unassign

Throughout this manual change bars indicate technical additions and changes, and asterisks indicate deletions.

CONTENTS

CONTENTS (cont)

CONTENTS (cont)

CONTENTS (cont)

CONTENTS (cont)

ILLUSTRATIONS

SECTION 1

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 the MPM Reference Guide and "Standard Stack and Linkage Area Formats" in Section 2 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, defined by the system include file entry_sequence_info.incl.pl1:

```
dcl 1 parm_desc_ptrs          aligned,
      2 n_args                fixed bin(18) unaligned unsigned,
      2 descriptor_relp       (num_descs refer(parm_desc_ptrs.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         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. See "Parameter Descriptors" in Section 2.

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

the last parameter is not to be returned by this entry

7. code_sequence

is any sequence of machine instructions satisfying Multics standard calling conventions. See "Subroutine Calling Sequences" in Section 2.

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

Gate Segment Entry Point Transfer Vector

For protection purposes, control must not be passed to a gate procedure at other than its defined entry points. To enforce this restriction, the first _n_ words of a gate segment with n entry points must be an entry point transfer vector. That is, the kth word ($0 \leq k \leq n-1$) 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 _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 words, 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.

This page intentionally left blank.

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 4 in MPM Reference Guide.

Character strings in the definition section are stored in ALM "acc" format. This format is described by the following PL/I declaration, defined by the system include file acc.incl.pl1:

```
dcl 1 acc                  based aligned,
      2 num_chars          fixed bin(9) unsigned unaligned,
      2 string             char(0 refer(acc.num_chars)) unaligned;
```

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.

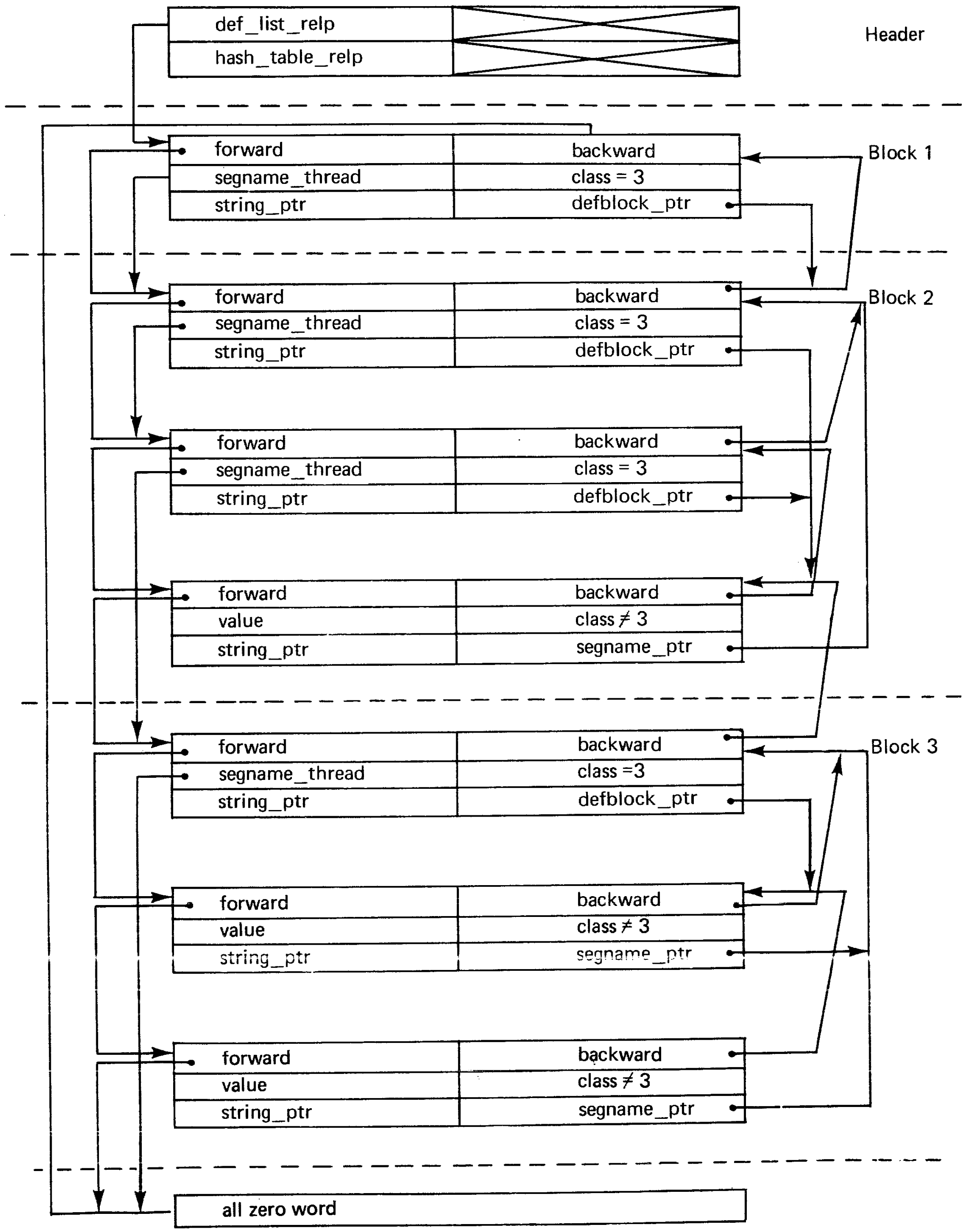

Figure 1-1. Sample Definition List

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 unused            bit(18) unaligned,
      2 hash_table_relp   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. unused
   is reserved for future use and must be "0"b.

3. 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.

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

A definition that is not class-3 has the following format, defined by the system include file definition.incl.pl1:

```
dcl 1 definition                               aligned,
      2 forward                                bit(18) unaligned,
      2 backward                               bit(18) unaligned,
      2 value                                  bit(18) unaligned,
      2 flags                                  unaligned,
        3 new                                  bit(1) unaligned,
        3 ignore                               bit(1) unaligned,
        3 entry                                bit(1) unaligned,
        3 retain                               bit(1) unaligned,
        3 argcount                             bit(1) unaligned,
        3 descriptors                          bit(1) unaligned,
        3 unused                               bit(9) unaligned,
      2 class                                  bit(3) unaligned,
      2 symbol                                 bit(18) unaligned,
      2 segname                                bit(18) unaligned,
      2 n_args                                 bit(18) unaligned,
      2 descriptor_relp(0 refer(n_args))       bit(18) unaligned;
```

where:

1. forward
   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
   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
   "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
   "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

   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
   is an offset (relative to the base of the definition section) to an aligned acc string representing the definition's symbolic name.

7. segname
   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. This item is not defined in the system include file.

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. This item is not defined in the system include file.

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            bit(18) unaligned,
      2 backward           bit(18) unaligned,
      2 segname_thread     bit(18) unaligned,
      2 flags              bit(15) unaligned,
      2 class              bit(3) unaligned,
      2 symbol             bit(18) unaligned,
      2 first_relp         bit(18) unaligned;
```

where:

1. forward
   is the same as above.

2. backward
   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
   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):

- forward points to an all zero word;
- the current entry's class is not 3, and forward points to a class-3 definition;
- the current definition is class 3, and both forward 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 format, defined in the system include file linkdcl.incl.pl1:

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

where:

1. type_ptr
   is an offset (relative to the base of the definition section) to the link's type pair.

2. exp
   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 defines the external symbol pointed to by a link and has the following format, defined in the system include file linkdcl.incl.pl1:

```
dcl 1 type_pair           aligned,
      2 type              bit(18) unaligned,
      2 trap_ptr          bit(18) unaligned,
      2 seg_ptr           bit(18) unaligned,
      2 ext_ptr           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

(obsolete) same as type 4 except that the external item is created if it is not found.

2. trap_ptr

is an offset (relative to the base of the definition section) to either an initialization structure (if type equals 5 and seg_ptr equals 5, or if type equals 6) or to a trap word.

3. seg_ptr

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 the MPM Reference Guide.)

4. ext_ptr

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 containing the entry point name of an external reference. If type equals 5 and seg_ptr equals 5, the acc string contains the name of the external variable. (See the MPM Reference Guide for a discussion of entry point names.)

Trap Word

The trap word is a structure that specifies a trap procedure to be called before the link associated with the trap word 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. The trap word has the following format, defined in the system include file linkdcl.incl.pl1:

```
dcl 1 trap_word      aligned,
      2 call_ptr     bit(18) unaligned,
      2 arg_ptr      bit(18) unaligned;
```

where:

1. call_ptr
   is an offset (relative to the base of the linkage section) to a link defining the entry point of the trap procedure.

2. arg_ptr
   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.

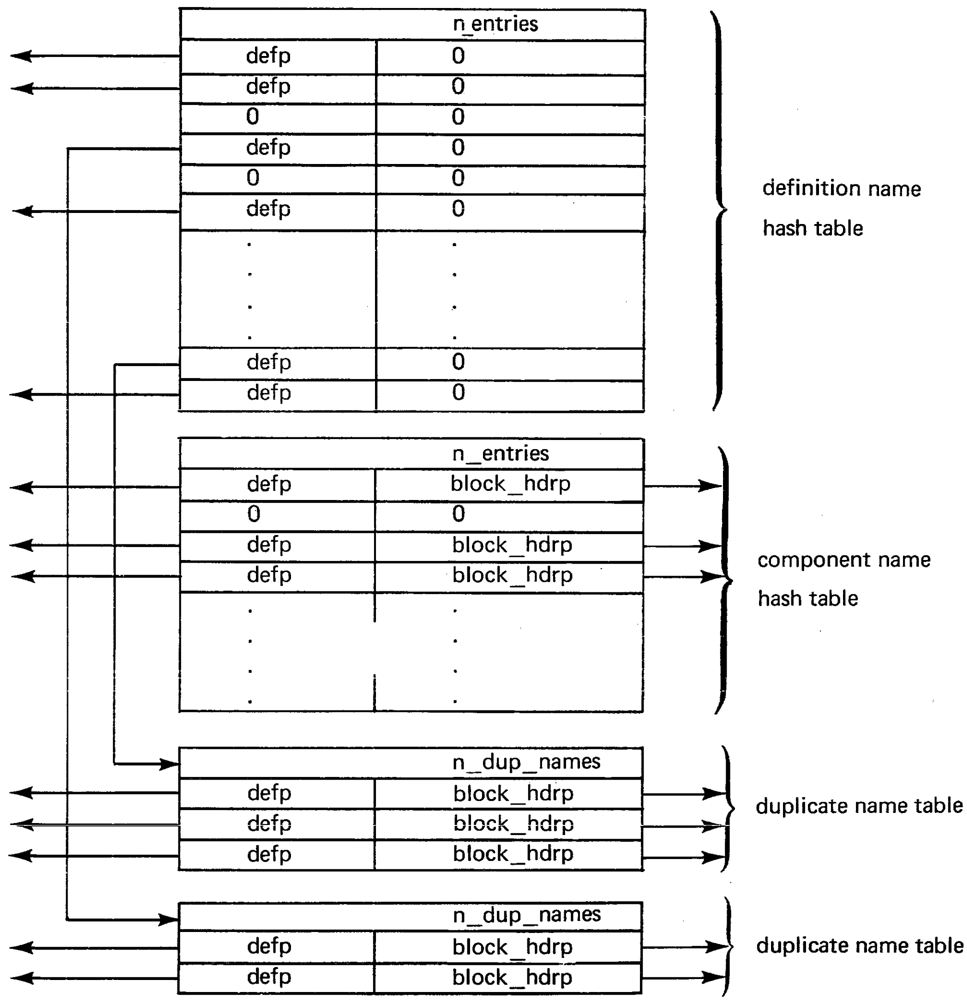

Figure 1-2. Definition Hash Table

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.

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 are filled in by the linker (when the header is copied) with a pointer to the beginning of the object segment's definition section. For more information see the MPM Reference Guide and "Standard Stack and Linkage Area Formats" in Section 2 of this manual.

Linkage Section Header

The header of the linkage section (in an object segment) has the following format, defined in the system include file object_link_dcls.incl.pl1:

```
dcl 1 virgin_linkage_header        aligned based,
      2 pad                        bit(30) unal,
      2 defs_in_link               bit(6) unal,
      2 def_offset                 fixed bin(18) uns unal,
      2 first_ref_relp             fixed bin(18) uns unal,
      2 filled_in_later            bit(144),
      2 link_begin                 fixed bin(18) uns unal,
      2 linkage_section_lng        fixed bin(18) uns unal,
      2 segno_pad                  fixed bin(18) uns unal,
      2 static_length              fixed bin(18) uns unal;
```

where:

1. pad
   is reserved for future use and must be 0.

2. defs_in_link
   Indicates whether or not there are definitions in the linkage section. If there are definitions in the linkage section, the value contained here is "010000"b.

3. def_offset
   is an offset (relative to the base of the object segment) to the base of the definition section.

4. first_ref_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, there is no first-reference trap.

5. filled_in_later
is initialized by the linker when the header is copied. As a result of initialization by the linker, the first word becomes a pointer to the object segment's symbol section. It is used by the linker to snap links relative to the symbol section. The second word becomes a pointer to the original linkage section within the object segment. It is used by the link unsnapping mechanism. The last two words remain unused.

6. link_begin
is an offset (relative to the base of the linkage section) to the first link (the base of the link array).

7. linkage_section_lng
is the entire length in words of the entire linkage section.

8. segno_pad
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. References to external entities are made by indirect references through a link, which has been copied from the pure linkage section of an object segment to the combined linkage section in the process directory. A link initially contains a fault tag 2 modification instead of an ITS modification. When the indirect reference is attempted, the fault occurs and is intercepted by the dynamic linking mechanism. Additional information in the link is used to locate the item referenced and, if successful, the link is replaced by an ITS pointer to the item. 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. A link has the following format, defined in the system include file object_link_dcls.incl.pl1:

```
dcl 1 object_link          aligned based,
      2 header_relp        fixed bin(17) unal,
      2 ringno             fixed bin(3) uns unal,
      2 mbz                bit(3) unal,
      2 run_depth          fixed bin(5) unal,
      2 tag                bit(6) unal,
      2 expression_relp    fixed bin(18) uns unal,
      2 mbz2               bit(12) unal,
      2 modifier           bit(6) unal;
```

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. ringno
   is the ring number of the ITS pointer.

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

4. run_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.

5. 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 the MPM Reference Guide.

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

7. mbz2
   is reserved for future use and must be "0"b.

8. modifier
   is a hardware address modifier. When the link is snapped, this becomes the modifier of the ITS pair.

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.

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 the MPM Reference Guide. The first reference trap has the following format, defined in the system include file object_link_dcls.incl.pl1:

```
dcl 1 fr_traps              aligned based,
      2 decl_vers           fixed bin,
      2 n_traps             fixed bin,
      2 trap_array          (n_fr_traps refer(fr_traps.n_traps)) aligned,
        3 call_relp         fixed bin(18) uns unal,
        3 info_relp         fixed bin(18) uns unal,
```

where:

1. decl_vers
   is the version number of the structure.

2. n_traps
   specifies the number of traps.

3. trap_array
   is an array of information about each first-reference procedure.

4. 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.

5. 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.

This page intentionally left blank.

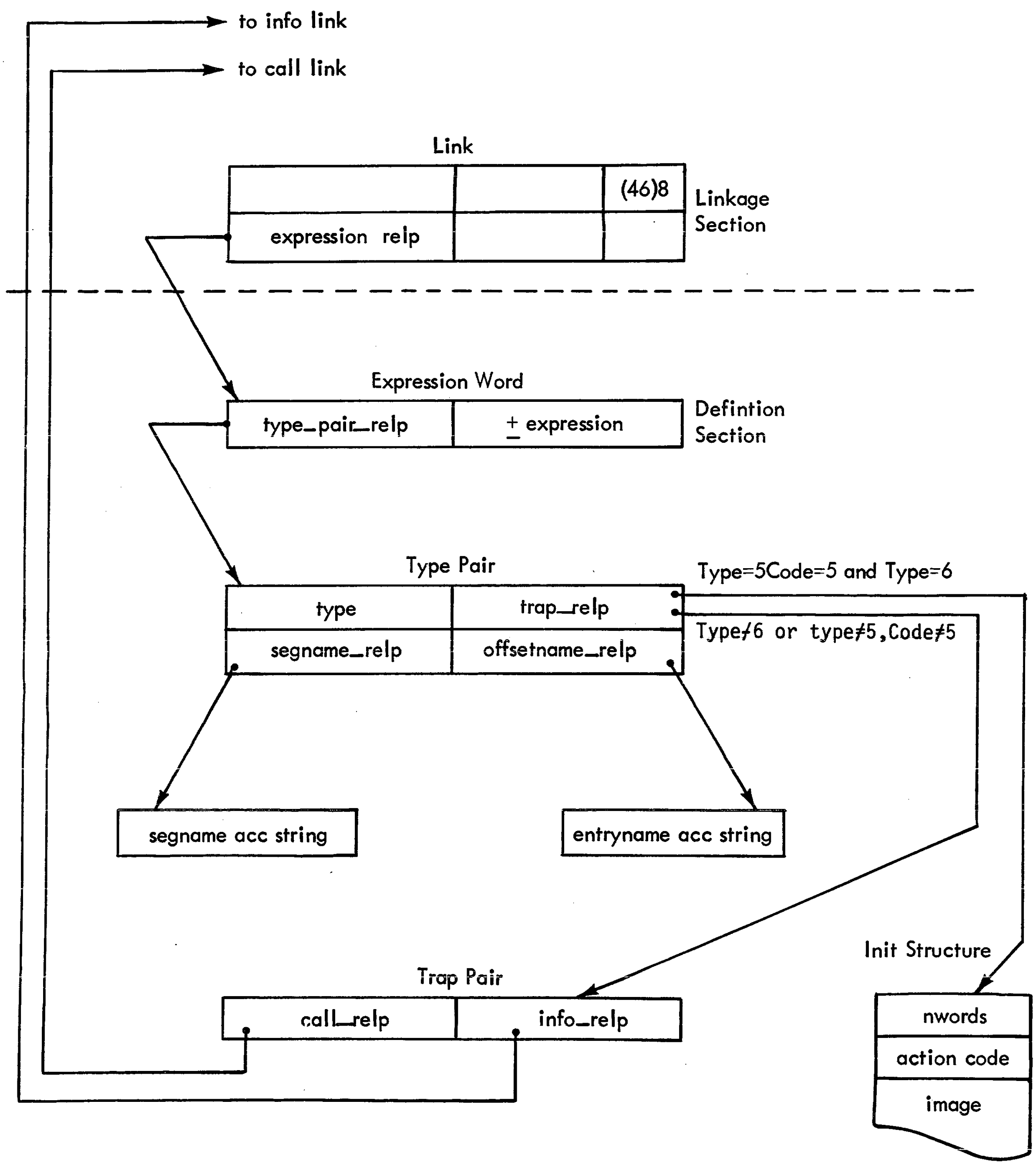

Figure 1-3. Structure of a Link

STRUCTURE OF THE SYMBOL SECTION

The symbol section consists of one or more symbol headers threaded together to form a single list. A symbol header 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 header, describing the circumstances under which the object segment was created. A symbol section can contain more than one symbol header. An example of multiple symbol headers is the case of a bound segment where in addition to the symbol header describing the segment's creation by the binder, there is also a symbol header for each of the component object segments.

Each symbol header can point to a free-format area. The free-format area can contain any information whatsoever, and the object segment will execute properly. However, the Multics debugging utilities (e.g., probe) place stringent requirements on the format of the free area, and these are followed by the translators for PL/I, FORTRAN, and COBOL. See Appendix B for additional information on the contents of the free-format area used by those three languages.

Symbol Block Header

All symbol blocks have a standard fixed-format block, although not all items in the block have meaning for all symbol blocks. The description of a particular symbol block lists items that have meaning for that symbol block. The block has the following format, defined by the system include file std_symbol_block.incl.pl1:

```
dcl 1 std_symbol_block          based aligned,
      2 decl_version             fixed bin initial(1),
      2 identifier               char(8) aligned,
      2 gen_number               fixed bin,
      2 gen_created              fixed bin(71),
      2 object_created           fixed bin(71),
      2 generator                char(8),
      2 gen_version              unaligned,
        3 offset                 bit(18),
        3 size                   bit(18),
      2 userid                   unaligned,
        3 offset                 bit(18),
        3 size                   bit(18),
      2 comment                  unaligned,
        3 offset                 bit(18),
        3 size                   bit(18),
      2 text_boundary            bit(18) unaligned,
      2 stat_boundary            bit(18) unaligned,
      2 source_map               bit(18) unaligned,
      2 area_pointer             bit(18) unaligned,
      2 backpointer              bit(18) unaligned,
      2 block_size               bit(18) unaligned,
      2 next_block               bit(18) unaligned,
      2 rel_text                 bit(18) unaligned,
      2 rel_def                  bit(18) unaligned,
      2 rel_link                 bit(18) unaligned,
      2 rel_symbol               bit(18) unaligned,
      2 mini_truncate            bit(18) unaligned,
      2 maxi_truncate            bit(18) unaligned;
```

where:

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

3. gen_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_created
   is a calendar clock reading specifying the date and time when this generator was created.

5. object_created
   is a calendar clock reading specifying the date and time when this symbol block was generated.

6. generator
   is the name of the program that generated this symbol block.

7. offset
   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_number.

8. size
   is the length of the aligned string describing the version of the generator.

9. userid
   is the name of the user for whom this symbol block was created.

10. offset
    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.

11. size
    is the length of the aligned string containing the access identification of the user for whom the symbol block was created.

12. comment
    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.

13. offset
    is an offset (relative to the base of the symbol block) to the comment. A value of "0"b indicates no comment.

14. size
    is the length of the aligned string containing generator-dependent symbolic information.

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, defined in the system include file source_map.incl.pl1:

```
dcl 1 source_map                              aligned based,
     2 version                                fixed bin initial(1),
     2 number                                 fixed bin,
     2 map (n refer (source_map.number))      aligned,
       3 pathname                             unaligned,
         4 offset                             bit(18),
         4 size                               bit(18),
```

THIS PAGE INTENTIONALLY LEFT BLANK

15. 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.

16. 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.

17. source_map
is an offset (relative to the base of the symbol block) to the source map (see "Source Map" below).

18. area_pointer
is an offset (relative to the base of the symbol block) to the free-format area of the symbol block. The contents of this area depend upon the symbol block. If the symbol block was created by a translator, this area may contain a runtime symbol table and/or a statement map. If the symbol block was created by the binder, this area contains the bind map.

19. backpointer
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.

20. block_size
is the size of the symbol block (including the block) in words.

21. next_block
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.

22. rel_text
is an offset (relative to the base of the symbol block) to text section relocation information (see "Relocation Information" below).

23. rel_def
is an offset (relative to the base of the symbol block) to definition section relocation information.

24. rel_link
is an offset (relative to the base of the symbol block) to linkage section relocation information.

25. rel_symbol
is an offset (relative to the base of the symbol block) to symbol section relocation information.

26. mini_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.

27. maxi_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.

```
          3 uid                              bit(36) aligned,
          3 dtm                              fixed bin(71);
```

where:

1. version
   is the version number of the structure.

2. number
   is the number of entries in the map array; that is, the number of source segments used to generate this object segment.

3. pathname
   an aligned string containing the absolute pathname of this source segment.

4. offset
   is an offset (relative to the base of the symbol block) to the pathname.

5. size
   is the length of the pathname.

6. uid
   is the unique identifier of this source segment at the time the object segment was generated.

7. 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., text_relocation_relp) 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(relinfo.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. Translators indicate the relocation code in the assembly-like listing of an object segment by a character. The second column below indicates the character used by standard translators. The third column indicates the character used by the ALM assembler.

"0"b	-	a	a	-	absolute
"10000"b	-	t	0	-	text
"10001"b	-	1	1	-	negative text
"10010"b	-	2	2	-	link 18
"10011"b	-	3	3	-	negative link 18
"10100"b	-	l	4	-	link 15
"10101"b	-	d	5	-	definition
"10110"b	-	s	6	-	symbol
"10111"b	-	7	7	-	negative symbol
"11000"b	-	8	8	-	internal storage 18
"11001"b	-	i	9	-	internal storage 15
"11010"b	-	r	L	-	self relative
"11011"b	-				unused
"11100"b	-				unused
"11101"b	-				unused
"11110"b	-				expanded absolute
"11111"b	-	e	*		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.

THIS PAGE INTENTIONALLY LEFT BLANK

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, defined in the system include file, object_map.incl.pl1:

```
dcl 1 object_map             aligned,
    2 decl_vers              fixed bin init(2),
    2 identifier             char(8) aligned,
    2 text_offset            bit(18) unaligned,
    2 text_length            bit(18) unaligned,
    2 definition_offset      bit(18) unaligned,
    2 definition_length      bit(18) unaligned,
    2 linkage_offset         bit(18) unaligned,
    2 linkage_length         bit(18) unaligned,
    2 static_offset          bit(18) unaligned,
    2 static_length          bit(18) unaligned,
    2 symbol_offset          bit(18) unaligned,
    2 symbol_length          bit(18) unaligned,
    2 break_map_offset        bit(18) unaligned,
    2 break_map_length        bit(18) unaligned,
    2 entry_bound            bit(18) unaligned,
    2 text_link_offset       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,
```

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

<u>Static Section</u>

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

<u>Symbol Section</u>

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 init. static for component 1 init. static for component 2 . . . init. 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 1-4. 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 std_symbol_header          based aligned,
      2 decl_version             fixed bin initial(1),
      2 identifier               char(8) aligned,
      2 gen_number               fixed bin,
      2 gen_created              fixed bin(71),
      2 object_created           fixed bin(71),
      2 generator                char(8),
      2 gen_version              unaligned,
        3 offset                 bit(18),
        3 size                   bit(18),
      2 userid                   unaligned,
        3 offset                 bit(18),
        3 size                   bit(18),
      2 comment                  unaligned,
        3 offset                 bit(18),
        3 size                   bit(18),
      2 text_boundary            bit(18) unaligned,
      2 stat_boundary            bit(18) unaligned,
      2 source_map               bit(18) unaligned,
      2 area_pointer             bit(18) unaligned,
      2 backpointer              bit(18) unaligned,
      2 block_size               bit(18) unaligned,
      2 next_block               bit(18) unaligned,
      2 rel_text                 bit(18) unaligned,
      2 rel_def                  bit(18) unaligned,
      2 rel_link                 bit(18) unaligned,
```

```
          2 rel_symbol                  bit(18) unaligned,
          2 mini_truncate               bit(18) unaligned,
          2 maxi_truncate               bit(18) unaligned;
```

where:

2. identifier
   is the string "bind_map".

6. generator
   is the string "binder".

13. comment
   is always "0"b.

19. area_pointer
   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, defined in the system include file bind_map.incl.pl1:

```
dcl 1 bindmap based                          aligned,
      2 dcl_version                          fixed bin,
      2 n_components                         fixed bin,
      2 component(0 refer(bindmap.n_components)) aligned,
        3 name
          4 name_ptr                         bit(18) unaligned,
          4 name_lng                         bit(18) unaligned,
        3 comp_name                          char(8) aligned,
        3 text_start                         bit(18) unaligned,
        3 text_lng                           bit(18) unaligned,
        3 stat_start                         bit(18) unaligned,
        3 stat_lng                           bit(18) unaligned,
        3 symb_start                         bit(18) unaligned,
        3 symb_lng                           bit(18) unaligned,
        3 defblock_ptr                       bit(18) unaligned,
        3 n_blocks                           bit(18) unaligned,
      2 bf_name                              aligned,
        3 bf_name_ptr                        bit(18) unaligned,
        3 bf_name_lng                        bit(18) unaligned,
      2 bf_date_up                           char(24),
      2 bf_date_mod                          char(24);
```

where:

1. dcl_version
   is a constant designating the format of this structure; this constant is modified whenever the structure is, allowing system tools to easily differentiate bind map formats. This structure is version one (1).

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
is 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_ptr
is the offset (relative to the base of the binder's symbol block).

6. name_lng
is the length (in characters) of the component's name.

7. comp_name
is the name of the translator that created this component object segment.

8. text_start
is the offset (relative to the base of the bound segment) of the component's text section.

9. text_lng
is the length (in words) of the component's text section.

10. stat_start
is the offset (relative to the base of the static section) of the component's internal static.

11. stat_lng
is the length of the component's internal static.

12. symb_start
is an offset (relative to the base of the symbol section) to the component's symbol section.

13. symb_lng
is the length of the component's symbol section.

14. defblock_ptr
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).

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

16. bf_name_ptr
is the offset (relative to the base of the binder's symbol block) of the symbolic name of the bindfile.

17. bf_name_lng
is the length (in characters) of the bindfile name.

18. bf_date_up
is the date, in symbolic form, that the bindfile was updated in the archive (of object segments) used as input by the binder.

19. bf_date_mod
is the date, in symbolic form, that the bindfile was last modified before being put into the binder's object archive.

SECTION 2

STANDARD EXECUTION ENVIRONMENT

STANDARD STACK AND LINK AREA FORMATS

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

Multics Stack

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

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

Stack Header

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

<table>
<tr><td>+0</td><td colspan="3">Reserved</td><td>Odd Lot Pointer</td><td>Combined Static Pointer</td></tr>
<tr><td>+8</td><td>Combined Linkage Pointer</td><td>Max Lot Size</td><td>Run Unit Depth</td><td>System Storage Pointer</td><td>User Storage Pointer</td></tr>
<tr><td>+16</td><td>Null Pointer</td><td colspan="2">Stack Begin Pointer</td><td>Stack End Pointer</td><td>Lot Pointer</td></tr>
<tr><td>+24</td><td>Signal Pointer</td><td colspan="2">BAR Mode Stack Pointer</td><td>PL/I Operators Pointer</td><td>Call Operator Pointer</td></tr>
<tr><td>+32</td><td>Push Operator Pointer</td><td colspan="2">Return Operator Pointer</td><td>Short Return Operator Ptr</td><td>Entry Operator Pointer</td></tr>
<tr><td>+40</td><td>Translator Operator Pointer</td><td colspan="2">Internal Static Offset Table Pointer</td><td>System Condition Table Pointer</td><td>Unwinding Procedure Pointer</td></tr>
<tr><td>+48</td><td>*system Link Info Pointer</td><td colspan="2">Reference Name Table Pointer</td><td>Event Channel Table Pointer</td><td>Assign Linkage Pointer</td></tr>
<tr><td>+56</td><td colspan="5">Reserved</td></tr>
<tr><td>+64</td><td colspan="5"></td></tr>
</table>

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 standard use in ALM). The following descriptions are based on that diagram and on the following PL/I declaration.

stack_frame +0	Pointer Register Storage					
+16	Previous Stack Frame Pointer	Next Stack Frame Pointer	Return Pointer		Entry Pointer	
+24	Operator Linkage Pointer	Argument Pointer	Internal Static Pointer	**	On Unit Relative Pointer	Operator Return Offset
+32	Register Storage					
+40	Temporaries					

** 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 1.

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 1.

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:

- Argument pointer
- Linkage pointer (and internal static pointer)
- Previous stack frame pointer
- 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.

call:

```
     OBJECT CODE       spri      pr6|0
                       sreg      pr6|32
                       epp0      arglist
                       epp2      entrypoint
                       tsp4      pr7|stack_header.call_op,*
     OPERATORS         spri4     pr6|stack_frame.return_ptr
                       sti       pr6|stack_frame.return_ptr+1
                       epp4      pr6|stack_frame.lp_ptr,*
                       call6     pr2|0
     OBJECT CODE       lpri      pr6|0
                       lreg      pr6|32
```

short_call:

```
     OBJECT CODE       epp2      entrypoint
                       tsp4      pr7|stack_header.call_op,*
     OPERATORS         (as above)
     OBJECT CODE       epp4      pr6|stack_frame.lp_ptr,*
```

return:

```
     OBJECT CODE       tra       pr7|stack_header.return_op,*
     OPERATORS         spri6     pr7|stack_header.stack_end_ptr
                       epp6      pr6|stack_frame.prev_sp,*
                       epbp7     pr6|0
                       epp0      pr6|stack_frame.operator_ptr,*
                       ldi       pr6|stack_frame.return_ptr+1
                       rtcd      pr6|stack_frame.return_ptr
```

short_return:

```
     OBJECT CODE       tra       pr7|stack_header.short_return_op,*
     OPERATORS         epbp7     pr6|0
                       epp0      pr6|stack_frame.operator_ptr,*
                       ldi       pr6|stack_frame.return_ptr+1
                       rtcd      pr6|stack_frame.return_ptr
```

entry:

```
     OBJECT CODE       tsp2      pr7|stack_header.entry_op,*
     OPERATORS         epp2      pr2|-1
                       epp4      pr7|stack_header.stack_end_ptr,*
                       spri2     pr4|stack_frame.entry_ptr
                       epaq      pr2|0
                       lprp5     pr7|stack_header.isot_ptr,*au
                       sprp5     pr4|stack_frame.static_ptr
                       lprp4     pr7|stack_header.lot_ptr,*au
                       tra       pr2|1
     OBJECT CODE       tra       executable_code
```

push:

```
OBJECT CODE     eax7     stack_frame_size
                tsp2     pr7|stack_header.push_op,*
OPERATORS       spri2    pr7|stack_header.stack_end_ptr,*
                epp2     pr7|stack_header.stack_end_ptr,*
                spri6    pr2|stack_frame.prev_sp
                spri0    pr2|stack_frame.arg_ptr
                spri4    pr2|stack_frame.lp_ptr
                epp6     pr2|0
                epp2     pr6|0,7
                spri2    pr7|stack_header.stack_end_ptr
                spri2    pr6|stack_frame.next_sp
                eax7     pr1
                stx7     pr6|stack_frame.translator_id
                tra      pr6|0,*
```

Register Usage Conventions

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

- 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).

- 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.

- 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 must begin on an even word boundary. It's format is given by the following PL/I declaration (arg_list.incl.pl1).

```
dcl 1 arg_list              aligned based,
      2 arg_count           fixed bin(17) unsigned unal,
      2 pad1                bit(1) unal,
      2 call_type           fixed bin(18) unsigned unal,
      2 desc_count          fixed bin(17) unsigned unal,
      2 pad2                bit(19) unal,
      2 arg_ptrs            (arg_list_arg_count) ptr,
      2 desc_ptrs           (arg_list_arg_count) ptr;
```

```
dcl 1 arg_list_with_envptr      aligned based,
      2 arg_count               fixed bin(17) unsigned unal,
      2 pad1                    bit(1) unal,
      2 call_type               fixed bin(18) unsigned unal,
      2 desc_count              fixed bin(17) unsigned unal,
      2 pad2                    bit(19) unal,
      2 arg_ptrs                (arg_list_arg_count) ptr,
      2 envptr                  ptr,
      2 desc_ptrs               (arg_list_arg_count) ptr;
```

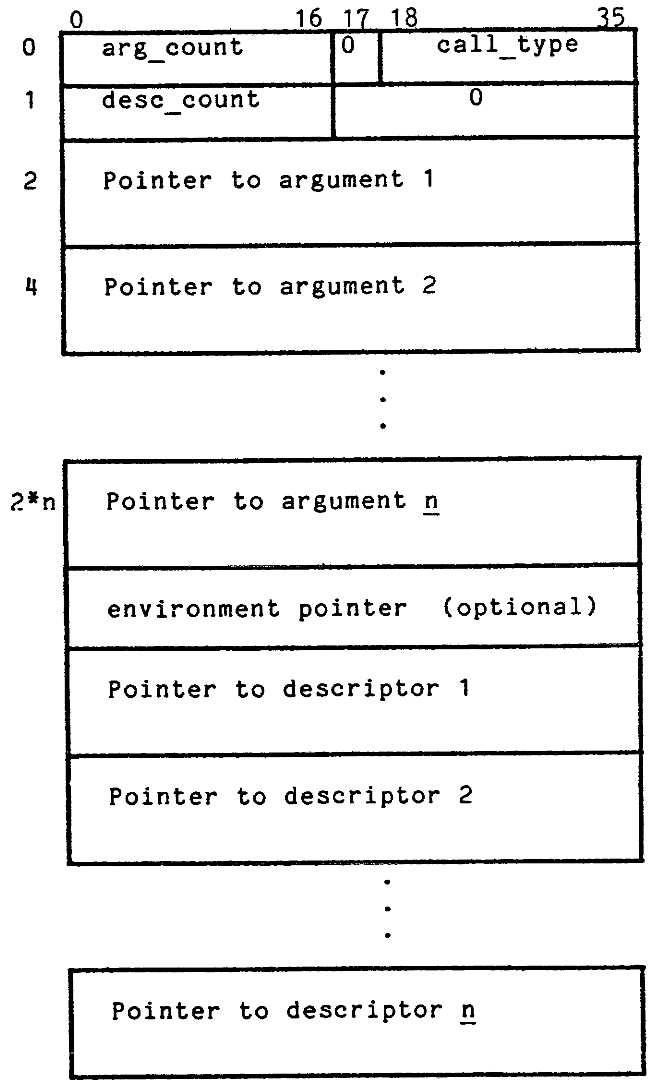

Figure 2-3. Standard Argument List

where:

1. arg_count
   is the number of arguments passed.

2. pad1
   is reserved and must be "0"b.

3. call_type
   is a code that describes the type of call being made. It can have one of the following values:

   0
      for quick internal calls.

   4
      for inter-segment calls.

   8
      for calls where an environment pointer is passed.

   The include file declares constants with these values:

```
dcl (
    Quick_call_type              init(0),
    Interseg_call_type           init(4),
    Envptr_supplied_call_type    init(8),
    ) fixed bin(18) unsigned unal int static
                                 options (constant);
```

4. desc_count
   is the number of argument descriptors being passed. If non-zero, it must be the same as arg_count.

5. pad2
   is reserved and must be "0"b.

6. arg_ptrs
   is an array of pointers to the arguments.

7. envptr
   is the environment pointer for the procedure being called. It is present only if call_type is 8.

8. desc_ptrs
   is an array of pointers to the argument descriptors, if present.

NOTES: 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.

The pointer envptr is present when a call is made to a non-quick internal procedure or when a call is made through an entry variable, regardless of whether the procedure being called is an external or internal procedure. When the called procedure is an internal procedure, envptr points to a stack frame of the activation of the block that contains the called procedure, and is used to set up the display pointer for the stack frame that the non-quick procedure will create. If the call is made through an entry variable, envptr is copied from the environment ptr of the entry variable. (See the MPM Reference Guide for the format of an entry variable.) If the call is to an internal entry constant, envptr is calculated by the PL/I operators. If a call is made through an entry variable to an external procedure, the environment pointer of the entry variable will be null, thus envptr is also null.

The include file also contains symbol names for the values that call_type takes on. They are: Quick_call_type, Interseg_call_type, and Envptr_supplied_call_type.

In the include file, the extent of the arrays, arg_ptrs, and desc_ptrs is determined by the variable arg_list_arg_count (which is not declared in the include file). In references to an already allocated argument list, the programmer should first set arg_list_arg_count to the value of arg_count in the appropriate structure (arg_list or arg_list_with_envptr).

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 one of the two following PL/I declarations, given in arg_descriptor.incl.pl1.

```
dcl 1 arg_descriptor        based aligned,
      2 flag                bit(1) unal,
      2 type                fixed bin(6) unsigned unal,
      2 packed              bit(1) unal,
      2 number_dims         fixed bin(4) unsigned unal,
      2 size                fixed bin(24) unsigned unal;

dcl 1 fixed_arg_descriptor  based aligned,
      2 flag                bit(1) unal,
      2 type                fixed bin(6) unsigned unal,
      2 packed              bit(1) unal,
      2 number_dims         fixed bin(4) unsigned unal,
      2 scale               fixed bin(11) unal,
      2 precision           fixed bin(12) unsigned unal;
```

The first four elements have the same meaning for all data 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 standard descriptor types (see Appendix D of the MPM Reference Guide). Named constants for the descriptor types are declared in the std_descriptor_types.incl.pl1 include 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.

If the data is fixed-point, then:

5. scale
is a 2's complement, signed value.

6. precision
is the number of bits used to represent the data (if binary) or the number of digits (if decimal).

For all other data:

5. size

   is the size (in bits, characters, or words) of string or area data, or the number of structure elements for structure data. In an argument descriptor for Algol68 array descriptor data, the size field is the number of dimensions of the array represented by the array decriptor datum. It is equal to the number_dims field of the second datum of the Algol68 array descriptor datum. In an argument descriptor for Algol68 union data, the size field is the number of words in the Algol68 union datum.

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 separation 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.

Parameter Descriptors

The parameter descriptors associated with an entry point have the same format as argument descriptors. The value 16777215 (77777777 octal) in the size field of an area, bit, or character parameter indicates an asterisk size. The value -34359738368 (400000000000 octal) in the lower bound, upper bound, or multiplier fields indicates asterisk array bounds.

SECTION 3

CLOSED 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, Order No. AK51.) If a user has the v_process_overseer attribute, she may specify a different initial procedure when she 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.

If the user has the v_process_overseer attribute, she may leave a program named "process_overseer_" in her homedir. Note that if the PMF specifies a reference-name other than "process_overseer_", the user must put whatever it specifies in her homedir. If the PMF provides an absolute pathname for the initial procedure, the user can not replace it in this manner.

Process Initialization

A process is created for a user when she logs in, or in response to either a new_proc command (described in the MPM Commands) or process termination signal. What follows is a brief description of the birth of a process. *

Unless otherwise noted, all of the modules described are in PL/I. It is helpful to follow along this discussion with a listing of the modules; the comments often provide useful amplification. To do so, use the library_fetch command. For example:

```
lf initialize_process_.pl1
```

Several items of information must be passed to all processes 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 because its write bracket is zero. The user_info_ subroutine (described in the MPM Subroutines) is used to extract information from the PIT.

A process begins, for all intents and purposes, with a call to the ring zero routine init_proc. This description will only mention those actions of init_proc which are of significance to visible features of the user environment.

The first action of init_proc is to initialize the known segment table (KST) by calling initialize_kst. Then init_proc initializes the PIT, and checks for the v_process_overseer attribute. If v_process_overseer is on, init_proc sets the working directory to the user's home directory. Until this point the user has no working directory, so that users without v_process_overseer do not get their home directory into the search rules until later on in their process. This prevents users without v_process_overseer from replacing their initial procedure, signaller, or unwinder.

Now init_proc calls makestack to create the stack in the user's initial ring. First, makestack creates a segment named stack_N in the process directory, where N is the number of the user's initial ring. It fills in the null pointer, begin pointer, and end pointer of the stack and calls the linker (via link_man$get_initial_linkage), to get the initial linkage for the ring.

The internal procedure initialize_rnt is then called by makestack in order to make a reference name table (RNT) for the ring in question. initialize_rnt calls define_area_ to get an area for the RNT, and puts a pointer to the RNT into the appropriate place in the stack header. Then initialize_rnt initializes the search rules to the default rules and returns.

At this point makestack adds the name of the stack it is creating to the RNT and calls the linker to snap links to signal_, unwinder_, the alm operators, and pl1_operators_. Thus users with v_process_overseer, whose working directories were set by init_proc before makestack was called, pick up any versions of these programs that are resident in their home directories. It then sets up the static condition handlers for no_write_permission, not_in_write_bracket, isot_fault, and lot_fault, fills in the thread pointers for the first stack frame and returns.

Now, init_proc is ready to find the initial procedure. For the purposes of this discussion, the initial procedure is the first procedure called in the user's initial ring. The term "process overseer" will refer to the program specified by the initproc keyword of the PMF or the argument to the -process_overseer control argument of the login access request. If the string ",direct" is appended to the pathname specified by either the initproc keyword or the -process_overseer control argument, then the specified pathname is both the process overseer and the initial procedure and init_proc parses the pathname and initiates it explicitly. This is because link_snap$make_ptr (the ring 0 entry that snaps links) will not take absolute or relative pathnames. Therefore init_proc parses the supplied pathname as either an absolute pathname or a relative pathname relative to the user's home directory. Note that this is independent of the state of v_process_overseer -- if the project administrator specified a ,direct overseer with a relative pathname, it will reference off of the home directory. This primarily provides a typing convenience to users with v_process_overseer specifying a ,direct overseer at login. If the name does not end with ,direct, the standard initial procedure, initialize_process_, is used.

At this point init_proc either has a pointer and a reference name for a ,direct overseer, or it has a reference name to the standard initial procedure initialize_process_.

Finally, init_proc calls call_outer_ring_ to call out to the user's initial ring. Note that a user without v_process_overseer is still lacking a working directory. It is the responsibility of any user-supplied ,direct initial procedure to set the working directory.

The user's process now begins execution in the initial ring in the program initialize_process_.

The initialize_process_ procedure first initiates the PIT. If the user lacks v_process_overseer it finds the appropriate process overseer. Then it sets the working directory, and finds the process overseer if it was not previously found. It sets up static condition handlers for cput, alrm, trm_, wkp_ and sus_.

Before calling the process overseer, initialize_process_ 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 by calling iox_$init_standard_iocbs. 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 Communications I/O.) For absentee processes it is abs_io_, and for daemons it is mr_.

The initialize_process_ procedure then calls the process overseer specified in the PIT. This is either the procedure specified in the "initproc" keyword of the PMF, or the -po argument to login. It is called with the following arguments:

```
declare process_overseer_ entry (ptr, bit (1) aligned, char (*) varying);

call process_overseer_ (pit_ptr, call_listen_, initial_cl);
```

where:

1. pit_ptr (Input)
   is a pointer to the PIT. It should be ignored.

2. call_listen_ (Output)
   if set to "1"b, initialize_process_ will call listen_ with the value of initial_cl as the first command line, thus starting the command environment. If it is set to "0"b , the process will be terminated, on the assumption that the process overseer already ran the entire process.

3. initial_cl (Output)
   Is the first command line to be executed, normally an exec_com of the start_up ec. It may be up to 256 characters long.

The system process overseers terminate processing by setting the call_listen flag in their calling sequence, setting the initial_cl argument to the initial command line, and returning to initialize_process_.

A user-supplied 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 initial command line used by the system process overseer is:

exec_com start_up_path>start_up.ec start_type proc_type

where:

1. start_up_path
   is the location of the user's start_up.ec. The system process overseers search for the start_up.ec in the following directories, in this order: >udd>Project>person, >udd>Project, and >system_control_1.

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

3. 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. 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 the process overseer returns to initialize_process_ with the call_listen flag set, initialize_process_ establishes an any_other handler of default_error_handler_$wall by executing the statement:

on any_other call wall_entry_variable;

An entry variable is used because initialize_process_ calls hcs_$make_entry with a null referencing pointer, so that users with v_process_overseer can put private versions of default_error_handler_ in their homedirs.

The default_error_handler_$wall procedure is invoked on all signals not intercepted by any subsequently established condition handler. In general, the default_error_handler_$wall 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.

If the process overseer does not use the call_listen_ flag, it must establish its own any_other handler, and call the listener if cleared.

Some Notes on Writing a Process Overseer

The best source of information on the writing of process overseers is the source of the standard one: process_overseer_.pl1. There are, however, several important considerations not obvious from the source.

The first is that process_overseer_ makes use of the pointer to the PIT that it gets as an argument. This means that if the PIT format changes, at best process_overseer_ must be recompiled. At worst, it may have to be recoded. If a user process overseer uses the PIT instead of calling user_info_, then it will likely stop working if the format of the PIT changes. For this reason, we strongly recommend that user-written process overseers do not directly reference the PIT. They should call user_info_, instead.

Both of the installed process overseers look for start_up exec_coms. The process_overseer_ and project_start_up_ procedures try to find start_up.ec in the home directory, the project directory, and >sc1 before giving up. Privately written process overseers should do so as well, unless they are putting the user in an environment for which this is obviously inappropriate.

Direct Process Overseers

The ,direct overseers are called as the first procedure in the user ring. In addition to setting up all I/O attachments for user_i/o, and static condition handlers for alrm, cput, trm_, wkp_ and sus_, ,direct overseers are responsible for setting the working directory for users without v_process_overseer. This is done to make protection somewhat easier, as the ,direct overseer can find anything it is interested in before setting the working directory.

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 4

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, and provides a list of implementation rules. For descriptions of the iox_ entry points, refer to the MPM Subroutines, and to the iox_$init_standard_iocbs entry point description in this manual.

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). The audit_ system I/O module (described in the MPM Subroutines) is an example of this sort of I/O module.

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. Most nonsystem I/O modules that perform true I/O work in this way, because a nonsystem I/O module (or some module that it calls) in turn calls 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 and the iox_$init_standard_iocbs entry point in this manual. This structure is given in iocb.incl.pl1.

```
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));
```

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 (attach_descrip.length));
```

The value of attach_descrip.string is the attach description. See "Multics Input/Output System" in Section 5 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 (open_descrip.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. The iox_modes.incl.pl1 include file gives standard names and named constants for the opening modes.

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. The iox_ subroutine provides four error entries that set the error code argument for the I/O module entry to an appropriate error_table_ value. The entries and the corresponding error codes are:

iox_$err_not_attached	(error_table_$not_attached)
iox_$err_not_closed	(error_table_$not_closed)
iox_$err_no_operation	(error_table_$no_operation)
iox_$err_not_open	(error_table_$not_open)

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 5 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).

THIS PAGE INTENTIONALLY LEFT BLANK

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 and the iox_$init_standard_iocbs entry point described in this manual.

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). Many I/O modules mask ips signals when the iocb is being modified. To do this:

   a. Get ready to change the iocb by copying all pointers or entries that the new iocb will contain into automatic variables. This will snap links to lessen the probability of a linkage error while interrupts are masked.

   b. Establish an any_other handler to call terminate_process_ with error_table_$unable_to_do_io or some other appropriate status code.

c. Execute the call:

```
call hcs_$set_ips_mask (0, mask);
```

The routine hcs_$set_ips_mask is used to disable one or more ips interrupts. (See the description of hcs_$set_ips_mask in this manual.)

d. Change the iocb.

e. 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.

f. Execute the call:

```
hcs_$reset_ips_mask (mask, mask);
```

This routine is used to enable one or more ips interrupts. (See the description of hcs_$reset_ips_mask in this manual.)

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

THIS PAGE INTENTIONALLY LEFT BLANK

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 (Input)
   points to the control block of the I/O switch to be attached.

2. option_array (Input)
   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.

3. com_err_switch (Input)
   indicates whether the attach routine should call the com_err_ subroutine (described in the MPM Subroutines) when an error is detected.
   "1"b yes
   "0"b no

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

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_$undefined_order_request. The io_call command, seeing this code, will call iox_$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.

Outer Modules

The iox_ I/O module with which user_i/o is attached at process initialization is called the outer module. In order to support reconnection of terminals, I/O modules used as outer modules must respect certain conventions. For an example of the appropriate techniques, examine the source of tty_.

All outer modules must support the -login_channel attach control argument, to mean that the switch will be connected to the device specified by user_info_$terminal_data.

When the user is disconnected, the special condition sus_ is signalled in the process. The program sus_signal_handler_ catches the condition, and blocks awaiting notification from the Answering Service that a new terminal is available. This may happen at any time, even when the process is compute-bound. When sus_signal_handler_ receives the notification, it searches the attach table for all switches with the control argument -login_channel in their attach description. Each one is closed, detached, attached, and opened.

The result of this is that an outer module may be interrupted in the middle of an operation, have its switch detached and closed, and be left to continue execution. Outer modules must be designed to avoid failure under these circumstances. An outer module may mask the sus_ IPS signal for the duration of all operations affecting the attachment data structures, but there is only a limited amount of CPU time available after the signal. If sus_signal_handler_ does not make the proper response to the Answering Service within this time, the process is terminated.

The alternative strategy is to detect asynchronous detachments. This can be done using a half lock in the attach data. As any operation is started, the half lock has one added to its value. When an operation is completed, one is subtracted. If the detach or close entrypoints are called and find a nonzero half lock, they may not free any storage that may be referenced by interrupted operations. Instead, they set flags in the attach data indicating that an asynchronous close or detach has taken place. When any of the other entrypoints detect these bits, they assume that a new attachment has been made, and call iox_ on the new attachment to complete their operation. Then they return.

For example, if tty_'s put_chars operation gets an error indicating that the process no longer has permission to use the terminal, it checks for the asynchronous bits. If they are not present, it blocks to await the arrival of the sus_ signal. If they are, it calls iox_$put_chars on its actual_iocb, and returns the results it returns.

SECTION 5

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, Logical Volumes
Storage System, Mailbox Manipulation
Storage System, Segment Manipulation

Detailed descriptions of these commands, arranged alphabetically rather than functionally, are given in Section 6 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 1 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
signal	signals Multics conditions

Input/Output System Control

dial_manager_call	provides command interface to answering service's dial facility

Language Translators, Compilers, Assemblers, and Interpreters

alm	invokes ALM assembler
alm_abs	invokes ALM assembler in absentee job

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
set_dir_ring_brackets	changes ring brackets of a directory

Storage System, Logical Volumes

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

*

Storage System, Mailbox Manipulation

*

mbx_create	creates mailbox
mbx_delete_acl	deletes entries from mailbox ACL
mbx_list_acl	lists ACL of mailbox
mbx_set_acl	adds and changes entries on mailbox ACL

archive_sort	sorts components of archive segment
copy_switch_off	turns off the copy switch of a specified segment
copy_switch_on	turns on the copy switch of a specified segment
reorder_archive	orders components of archive segment
set_max_length	specifies maximum length of nondirectory segment

SUBROUTINE REPERTOIRE

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

- Argument List Manipulation Routines
- 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
- Resource Control Package (RCP)
- Run Units
- 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 7 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 Section 1 of the MPM Subroutines for the functional grouping of the subroutines described in that manual.

Argument List Manipulation Routines

decode_descriptor_	extracts information from argument descriptors

get_entry_arg_descs_	returns information about the calling sequence of an entry point
get_entry_point_dcl_	returns attributes needed to construct a PL/I declare statement

Clock and Timer Procedures

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

Command Environment Utility Procedures

change_default_wdir_	changes the user's current default working directory
check_star_name_	verifies formation of entrynames according to star name rules
cv_userid_	converts a character string containing an abbreviated User_id into one containing all three components
dl_handler_	issues queries for situations involving deletion
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_arg_descs_	returns information about the calling sequence of an entry point
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
help_	locates info segs
nd_handler_	resolves name duplication
read_password_	reads user's password from the terminal
requote_string_	doubles all quotes within a character string and returns the result enclosed in quotes
terminate_process_	terminates the process in which it is called

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_frame_	returns a pointer to the most recent condition frame
find_condition_info_	returns information about condition when signal occurs
hcs_$get_exponent_control	returns flag settings that control handling of overflow and underflow conditions
hcs_$set_exponent_control	changes flag settings that control handling of overflow and underflow conditions

prepare_mc_restart_	checks machine conditions for restartability, and permits modifications to them for user changes to process execution, before condition handler returns
sct_manager_	manipulates the System Condition Table; can set a static handler, get a pointer to one, and call one
signal_	signals occurrence of given condition
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
char_to_numeric_	converts user-supplied string to a numeric type
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
valid_decimal_	checks decimal data for validity

Error Handling Procedures

active_fnc_err_	prints formatted error message and signals active_function_error condition
active_fnc_err$_suppress_name	prints formatted error message and signals active_function_error condition but suppresses name of calling function
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

Miscellaneous Procedures

add_epilogue_handler_	adds to execute_epilogue_'s list of programs
* execute_epilogue_	cleans up language I/O buffers in conjunction with run units
get_privileges_	returns process' access privileges
hash_index_	computes the value of a hash function
hcs_$get_process_usage	retrieves system resource usage information
mode_string_	manipulates mode strings; can parse, analyze, and create them
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

create_ips_mask_	returns a bit string that can be used to disable specified ips interrupts
get_lock_id_	returns a 36-bit unique identifier to be used in setting locks
hcs_$get_ips_mask	returns the value of the current ips mask
hcs_$reset_ips_mask	replaces the entire ips mask with a specified ips mask
hcs_$set_automatic_ips_mask	replaces the entire automatic ips mask with a specified ips mask
hcs_$set_ips_mask	replaces the entire ips mask with a specified ips mask
hcs_$wakeup	sends interprocess communication wakeup to blocked process over specified event channel
ipc_	user interface to Multics interprocess communication facility

Resource Control Package (RCP)

cv_rcp_attributes_	manipulates RCP resource attribute specifications and descriptions
interpret_resource_desc_	displays selected contents of RCP resource description
resource_control_	provides interface to Multics resource control facility
resource_info_	returns selected information about RCP resource types defined on the system

Run Units

run_	sets up special environment for executing programs
run_$environment_info	returns information about run environment

Storage System, Access Control and Rings of Protection

aim_check_	determines relationship between two access attributes
convert_aim_attributes_	converts representation of process'/segment's access authorization/class into character string of defined form
copy_acl_	copies the ACL from one segment, MSF, or directory to another.
cross_ring_	allows an outer ring to attach to a preexisting switch in an inner ring and perform I/O operations
cross_ring_io_$allow_cross	allows use of an I/O switch via cross_ring_ attachments from an outer ring
cv_dir_mode_	converts a character string containing access modes for directories into a bit string used by the ACL entries
cv_mode_	converts a character string containing access modes for segments into a bit string used by the ACL entries
get_privileges_	returns process' access privileges
get_ring_	returns number of current protection ring
hcs_$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_$del_dir_tree	deletes subdirectory's contents
hcs_$force_write	writes pages from memory to disk
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_$star_	returns storage system type and all names that match entryname according to star name rules
mdc_	provides entrypoints for master directory manipulation
shcs_$set_force_write_limit	fixes limit on number of pages to be written to disk

Storage System, Utility Procedures

area_info_	returns information about an area
define_area_	initializes a region of storage as an area
get_default_wdir_	returns pathname of user's current default working directory
get_definition_	returns pointer to specified definition within an object segment
get_entry_name_	returns associated name of externally defined location or entry point in segment
get_equal_name_	constructs target name by substituting from entryname into equal name
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 6

COMMANDS

COMMAND DESCRIPTION FORMAT

This section contains descriptions of 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

D error in macro definition or macro expansion; more detailed diagnostic for specific error given in listing

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

THIS PAGE INTENTIONALLY LEFT BLANK

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" below. 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,...,term*n*

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.

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.

 alm

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 literal pool is allocated in the text section. 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. The binary point in a literal with a binary scale factor is positioned to the right of the bit indicated by a decimal integer following the "b".

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.

If a variable-field literal, octal literal, or fixed point literal (decimal literals with a "b" binary scale factor) is used with du or dl modification, then the lower 18 bits of the literal are placed in the address field of the instruction. If any other type of literal is used with du or dl modification, then the upper 18 bits of the literal are placed in the address field of the instruction.

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.

THIS PAGE INTENTIONALLY LEFT BLANK

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 <u>Multics Processor Manual</u>.

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.
xN	Descriptor address is offset by the value in index register N (N can be 0 - 7, as above).

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,...,numbern
assembles the decimal integers number1, number2, through numbern 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).

entrybound

places the current value of the location counter in the object_map entrybound field. If more than one such operation is encountered, the last one is effective. See the gate_macros.incl.alm include file for an example of this operation's use. Note that setting the entry bound of the object segment's directory entry is still necessary. See hcs_$set_entry_bound for a description of that operation.

equu 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.

THIS PAGE INTENTIONALLY LEFT BLANK

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. The "translator" search list, which has the synonym "trans," is used to locate the segment (see the search facility commands in MPM Commands).

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.

perprocess_static
: turns on the object segment's perprocess static switch. See the description of the run command in the MPM Commands for an explanation of perprocess static.

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

rem
: (see the null pseudo-operation)

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

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

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

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

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

THIS PAGE INTENTIONALLY LEFT BLANK

short_return
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

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 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

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. &R*m*,*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 can contain other selection groups to a nesting depth of 10. 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. &s*n*

    selects the *n*th 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 can be placed anywhere in source segments and include files, as well as in macro code, but the conditional pseudo-operations can not be nested.

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

                    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:

```
                    tempd     argl(16)
```

appears at some point in the program.

```
                    macro     gcall
     &R2&(          dcl_&i
                    spri0     argl+2*&x
                    spri1     argl+2*&K-2+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*3-2+2*1
                    dcl_progname
                    spri0     argl+2*3-2
                    spri1     argl+2*3-2+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*3-2+2*1
                    epp0      lp|xlink,*

                    epp1      =v1/1,6/char,17/0,12/32
                    spri0     argl+2*2
                    spri1     argl+2*3-2+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.

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
      is the priority queue of the request. The default queue is defined by the system administrator. See "Notes" for a description of the interaction with the dprinting of listing files.

   -hold
      specifies that alm_abs should not dprint or delete the listing segment.

   -limit N, -li N
      places a limit on the CPU time used by the absentee process. The parameter N must be a positive decimal integer specifying the limit in seconds. The default limit is defined by the site for each queue. An upper limit is defined by the site for each queue on each shift. Jobs with limits exceeding the upper limit for the current shift are deferred to a shift with a higher limit.

   -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.

If the -queue control argument is not specified, the request is submitted into the default absentee priority queue defined by the site and, if requested (via -list), the listing files are dprinted in the default queue of the request type specified on the command line (via dp_args). (If no request type is specified, the "printer" request type is used.)

If requested (via -list) when the -queue control argument is specified, the listing files are dprinted in the same queue as is used for the absentee request. If the request type specified for dprinting (via dp_args) does not have that queue, the highest-numbered (i.e., the lowest priority) queue available for the request type is used and a warning is issued.

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 virtual_ptr {-control_args}

where:

1. virtual_ptr
   is a virtual pointer to the area to be looked at. The syntax of virtual pointers is described in the cv_ptr_ subroutine description.

2. control_args
   can be chosen from the following:

   -trace
      displays a trace of all free and used blocks in the area. *

   -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.

The copy_names command description, formerly on this page, has been moved to the MPM Commands and Active Functions manual, Order No. AG92.

Name: copy_switch_off, csf

This command turns off the copy switch of specified segments.

Usage

copy_switch_off paths

where paths are the pathnames of segments.

Notes

The current state of a segment's copy switch can be determined by issuing the command:

status path -copy_switch

This command replaces the resetcopysw command.

Name: copy_switch_on, csn

This command turns on the copy switch of specified segments.

Usage

```
copy_switch_on paths
```

where paths are the pathnames of segments.

Note

This command replaces the setcopysw command.

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. The minimum area if forty octal words.

   -id STR
   specifies a string to be used in constructing the names of the components of extensible areas.

Name: delete_external_variables, dev

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.

The delete_volume_quota command, formerly on this page, has been moved to the Multics Administrators' Manual Project, Order No. AK51. *

Name: dial_manager_call

The dial_manager_call command provides a command interface to the answering service's dial facility. All functions which are available through the dial_manager_ subroutine interface are available through this command. See the description of dial_manager_ for a more complete description of these functions.

Usage

```
dial_manager_call request {STR1 {STR2} {STR3}}
```

where:

1. request
   maps into a call to an identically named entry in dial_manager_. Each request requires the use of a particular STR which is listed in the request description. A request must be one of the following:

   allow_dials STR, ad STR
      requests that the answering service establish a dial line to allow terminals to dial to the calling process. STR must be a dial_qualifier as described below.

   dial_out STR1 STR2 {STR3}, do STR1 STR2 {STR3}
      requests 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. STR1 must be a channel_name and STR2 must be a dial_out_destination as described below. STR3, which can be omitted, is a reservation_string as described below.

   privileged_attach STR, pa STR
      allows a privileged process to attach any terminal that is in the channel master file, and is not already in use. See the description of dial_manager_$privileged_attach for information on what access is required. The effect is as if that terminal had dialed to the requesting process. STR must be a channel_name as described below.

   registered_server STR, rs STR
      requests that the answering service allow terminals to dial the calling process using only the dial qualifier. STR must be a dial_qualifier as described below.

   release_channel STR, rc STR
      requests the answering service to release the channel specified by channel_name. This channel must be dialed to the caller at the time of the request. If the channel was dialed, the channel is returned to the answering service and another access request may be issued. If the channel is a slave channel, the channel is hung up. STR must be a channel_name as described below.

   release_channel_no_hangup STR, rcnh STR
      is the same as release_channel except that this request does not hang up slave channels. STR must be a channel_name as described below.

release_dial_id STR, rdi STR
: informs the answering service that the user process wishes to prevent further dial connections, but that existing connections should be kept. Any connections kept can be released later with the release_channel request. STR must be a dial_qualifier as described below.

shutoff_dials STR, sd STR
: informs the answering service that the user process wishes to prevent further dial connections, and that existing connections should be terminated. STR must be a dial_qualifier as described below.

start_report, start
: turns on the reporting feature. See "Notes" below. STR is not used with this request.

stop_report, stop
: turns off the reporting feature. See "Notes" below. STR is not used with this request.

terminate_dial_out STR, tdo STR
: requests that the answering service hang up an auto call line and unassign it from the requesting process. STR must be a channel_name as described below.

2. STR
: depends on the request. STR is selected from the following list. (For details on the interpretation of the following qualifiers, see the description of the dial_manager_subroutine in this manual.)

channel_name
: is the name of a tty_channel.

dial_qualifier
: is the name for which the user is to be a dial server.

dial_out_destination
: is the destination (e.g., phone number) of up to 32 characters.

reservation_string
: is a dial_manager_ reservation string of up to 256 characters.

Notes

The dial_manager_call command establishes an event call channel for communication with the answering service. This event channel and its handler (which is an entry point in dial_manager_call) remain active after the command terminates. Any events which happen subsequent to the command termination, such as channel hang-ups, dial-ups, and dial requests will be decoded using convert_dial_message_ and reported on the user_output I/O switch when they happen. This reporting feature may be turned on and off by using the start_report and stop_report requests. The default is on.

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, or an octal segment number. A pathname that looks like an octal segment number can be specified by -name nnn.

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
```

If bound_zilch_ were known with segment number 532, the following command would generate the same output:

```
dcn 532 17523 64251
```

Name: list_external_variables, lev

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. Cannot be used with -all.

2. control_arg
   is -all (or -a) to list all temporary segments, including free ones. If the command is issued with no arguments (the default invocation), it lists only those temporary segments currently assigned to programs (i.e., free temporary segments are not listed).

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
```

The mbx_add_name command, formerly described on page 6-40, is obsolete and has been deleted. Use instead the add_name command described in the MPM Commands manual.

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 pathi 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 the 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:

adrosw	user who created the mailbox
as	*.SysDaemon.*
aow	*.*.*

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.

The mbx_delete command, formerly described on page 6-42, is obsolete and has been deleted. Use instead the delete command described in the MPM Commands manual.

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.

The mbx_delete_name command, formerly described on page 6-45, is obsolete and has been deleted. Use instead the delete_name command described in the MPM Commands manual.

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.

The mbx_rename, mbx_safety_switch_off, and mbx_safety_switch_on commands, formerly described on pages 6-47 through 6-49, are obsolete and have been deleted. Use instead the rename, switch_off, and switch_on commands described in the MPM Commands manual.

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 adrosw (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 adrosw for the user who created it, asw for *.SysDaemon.*, and aow 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
wakeup	w	can send a wakeup indicating that a message was added to the mailbox

Example

The command line:

mbsa Green adrosw Klein.. null Jones.Multics a *.*.*

manipulates the ACL of Green.mbx so that all previously existing entries with a first component of Klein have adrosw 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.

The mbx_set_max_length command, formerly described on page 6-52, is obsolete and has been deleted. Use instead the set_max_length command described later in this section.

The move_names command description has been moved to the MPM Commands manual.

The perprocess_static_sw_off command is obsolete and has been deleted. Use instead the switch_off command described in the MPM Commands manual.

The perprocess_static_sw_on command is obsolete and has been deleted. Use instead the switch_on command described in the MPM Commands manual.

Name: print_bind_map, pbm

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.

   -page_offset, -pgofs
      causes the page number of the first word of the text section of each component to be printed as an octal number, which is the format used by the cumulative_page_trace command. If the component crosses at least one page boundary, a "+" character follows the page number.

Name: print_link_info, pli

The print_link_info command prints selected items of information for the specified object segments. The archive component (::) convention is accepted.

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 "Note" 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, ra

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, rev

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_dir_ring_brackets, sdrb

The set_dir_ring_brackets command allows a user to modify the ring brackets of a specified directory.

Usage

```
set_dir_ring_brackets path {rb1 {rb2}}
```

where:

1. path
   is the relative or absolute pathname of the directory whose ring brackets are to be modified.

2. ring_numbers
   are the numbers that represent the directory ring brackets (rb1, rb2). The ring brackets must be in the allowable range v through 7 (where v depends upon the user's current validation level) and must have the ordering:

   rb1 ≤ rb2

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

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

   rb2
   is the number to be used for the second ring bracket of the directory.

Note

The user's process must have a validation level less than or equal to rb1. See the MPM Reference Guide for a discussion of ring brackets and validation levels.

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 6 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
   can be one of the following:

   -system
   to specify the area used for linkage sections

   -create
   to create (and initialize) a system_free segment in the user's process directory.

These control arguments 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_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
   may be one of the following:

   -system
      to specify the area used for linkage sections.

   -create
      to create (and initialize) a system_free segment in the user's process directory.

These control arguments 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.

Name: signal

The signal command signals Multics conditions, allowing the user to specify some information to be associated with the condition. The result of a condition signal depends on the user or system program handling the condition signal.

The descriptions that follow assume that the signal is handled by the default unclaimed signal handler, default_error_handler_$wall. Any messages described are sent over the error_output switch.

Usage

signal CONDITION_NAME {-control_args}

where:

1. CONDITION_NAME
   is the name of the condition to signal. It can not contain embedded white space, because condition names are only significant to the first space character. It can not be longer than 256 characters.

2. control_args
   can be chosen from the following:

   -info_string INFO_MESSAGE
   associates the string INFO_MESSAGE with this signal. If an error message is printed, this string is also printed. It must be enclosed in quotes if it contains whitespace or special characters. The string can not be longer than 256 characters.

   -code ET_CODE_NAME
   associates the error table code name ET_CODE_NAME with this signal. It must be a virtual pointer to an error_table acceptabe to cv_ptr_. If the segment name portion of the virtual pointer is omitted, error_table_ is assumed. The text message defined for this error table_ code_ is printed if an error message is printed. Thus an ET_CODE_NAME of noentry will be interpreted as error_table$noentry, not as a pointer to noentry|0.

   -cant_restart
   sets the cant_restart flag for this signal. The default handler establishes a new listener level after printing a message, and refuses to accept the "start" command. See "Notes" for a description of the default action.

   -default_restart
   sets the default_restart flag for this signal. The default handler prints a messages and restarts execution.

   -quiet_restart
   sets the quiet_restart flag for this signal. The default handler restarts execution without printing a message.

-support_signal
sets the support_signal flag for this signal. This indicates that the error is being signalled on behalf of another procedure, and should only be used when a user handler is present on the stack that expects it.

Notes

This command should not be used with any of the system conditions defined in the MPM Reference Guide, or with PL/I language conditions. These conditions require other associated information that cannot be specified with this command. As a result, the use of this command with these conditions may produce unpredictable results.

The on command can be used to handle signals produced with this command.

The default handler handles all condition signals that are otherwise unhandled by user or system programs on the stack. If neither of -quiet_restart, -cant_restart, or -default_restart are given, the default handler prints the error message described below, and establishes a new listener level. If the user types "start" at this point, execution continues. In particular, if the command is executed in an exec_com, and the user types start, execution continues with the next command in the exec_com.

The default message printed for a condition signalled is of the form:

```
Error: CONDITION_NAME condition by signal$signal|octalnumber
ERROR_TABLE_MESSAGE
INFO_STRING_MESSAGE
```

If -info_string is not given, the INFO_STRING_MESSAGE line is omitted. If -code is not given, the ERROR_TABLE_MESSAGE line is omitted.

SECTION 7

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 6 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 argi arguments described above. If the control_string and argi arguments are not given, user_message is omitted.

Entry: active_fnc_err_$suppress_name

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

Usage

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

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

where all arguments are the same as above.

Name: add_epilogue_handler_

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

Usage

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

call add_epilogue_handler_ (ev, code);
```

where:

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

2. code
   is a standard status code.

Note

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

Name: 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 areap            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. areap
    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_$ae_table

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

Usage

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

ISOMORPHIC ASCII/EBCDIC CONVERSION TABLE

ASCII		EBCDIC	
GRAPHIC	OCTAL	HEXADECIMAL	GRAPHIC
NUL	000	00	NUL
SOH	001	01	SOH
STX	002	02	STX
ETX	003	03	ETX
EOT	004	37	EOT
ENQ	005	2D	ENQ
ACK	006	2E	ACK
BEL	007	2F	BEL
BS	010	16	BS
HT	011	05	HT
LF	012	25	NL
VT	013	0B	VT
FF	014	0C	NP
CR	015	0D	CR
SO	016	0E	SO
SI	017	0F	SI
DLE	020	10	DLE
DC1	021	11	DC1
DC2	022	12	DC2
DC3	023	13	TM
DC4	024	3C	DC4
NAK	025	3D	NAK
SYN	026	32	SYN
ETB	027	26	ETB
CAN	030	18	CAN
EM	031	19	EM
SUB	032	3F	SUB
ESC	033	27	ESC
FS	034	1C	IFS
GS	035	1D	IGS
RS	036	1E	IRS
US	037	1F	IUS
space	040	40	space
!	041	5A	!
"	042	7F	"
#	043	7B	#
$	044	5B	$
%	045	6C	%
&	046	50	&
'	047	7D	'
(	050	4D	(
)	051	5D	)
*	052	5C	*
+	053	4E	+
,	054	6B	,
-	055	60	-
.	056	4B	.
/	057	61	/
0	060	F0	0
1	061	F1	1
2	062	F2	2
3	063	F3	3
4	064	F4	4

GRAPHIC	OCTAL	HEXADECIMAL	GRAPHIC
5	065	F5	5
6	066	F6	6
7	067	F7	7
8	070	F8	8
9	071	F9	9
:	072	7A	:
;	073	5E	;
<	074	4C	<
=	075	7E	=
>	076	6E	>
?	077	6F	?
@	100	7C	@
A	101	C1	A
B	102	C2	B
C	103	C3	C
D	104	C4	D
E	105	C5	E
F	106	C6	F
G	107	C7	G
H	110	C8	H
I	111	C9	I
J	112	D1	J
K	113	D2	K
L	114	D3	L
M	115	D4	M
N	116	D5	N
O	117	D6	O
P	120	D7	P
Q	121	D8	Q
R	122	D9	R
S	123	E2	S
T	124	E3	T
U	125	E4	U
V	126	E5	V
W	127	E6	W
X	130	E7	X
Y	131	E8	Y
Z	132	E9	Z
[	133	AD	[(see "Notes")
\	134	E0	\
]	135	BD	] (see "Notes")
^	136	5F	logical NOT
	137	6D	
`	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

GRAPHIC	OCTAL	HEXADECIMAL	GRAPHIC
n	156	95	n
o	157	96	o
p	160	97	p
q	161	98	q
r	162	99	r
s	163	A2	s
t	164	A3	t
u	165	A4	u
v	166	A5	v
w	167	A6	w
x	170	A7	x
y	171	A8	y
z	172	A9	z
{	173	C0	{
¦	174	4F	solid bar
}	175	D0	}
~	176	A1	~
DEL	177	07	DEL

Notes

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

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

This mapping differs from the ASCII to EBCDIC mapping discussed in "Punched Card Codes" in Section 5 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. This subroutine handles the following data types: 1-12, 19-22, 33, 34, 41-46. Any other type will produce an error. This subroutine uses rounding in the conversion when the target is floating point or when the source is floating and the target is character, and uses truncation in all other cases.

Usage

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

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

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 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 unaligned halfwords. The left halfword is a fixed bin(17) representing the signed scale and the right halfword is a fixed bin(18) unsigned integer representing the precision. The include file encoded_precision.incl.pl1 declares this as:

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

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.

Entry: assign_$computational_

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

Usage

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

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

where:

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

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

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

Notes

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

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

where:

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

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

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

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

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

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

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

<u>Entry</u>: assign_round_

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

<u>Entry</u>: assign_truncate_

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

Name: change_default_wdir_

The change_default_wdir_ subroutine changes the user's current default working directory to the directory specified. See the description of the change_wdir and change_default_wdir commands in the MPM Commands for a discussion of the default working directory.

Usage

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

call change_default_wdir_ (path, code);
```

where:

1. path (Input)
   is the pathname of the directory that is to become the default working directory.

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

Name: char_to_numeric_

The char_to_numeric_ subroutine converts a user-supplied string to a numeric type, or signals the conversion condition if it cannot be converted. The attributes of the numeric data created are returned.

Usage

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

call char_to_numeric_ (target_ptr, enc_type, enc_prec, source_ptr,
     source_len);
```

where:

1. target_ptr (Input)
   points to a buffer where the numeric data may be written. No check is made that the buffer is large enough to hold the data.

2. enc_type (Output)
   is the encoded type of the data created. Its value is 2*M+P, where M is a standard Multics type code, and P is 1 if the data is packed, or 0 if it is not. (P should always be 0.) The value of Multics type codes are defined in the MPM Reference Guide.

3. enc_prec (Output)
   is the encoded precision of the data created. The format of an encoded precision is given by encoded_precision.incl.pl1. See the description of the assign_ subroutine.

4. source_ptr (Input)
   points to the character string to convert to numeric.

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

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 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 a pathname as its input and validates the final entryname in that pathname.

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. These are described in "Notes" below. (For more information on conditions, see 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. The area size should be at least 300 words. If null, the message is printed on the error_output I/O switch.

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 under "List of System Conditions and Default Handlers" in the MPM Reference Guide.

Notes

The following conditions cause a return with no message:

command_error
command_question
finish
stringsize

*

Name: continue_to_signal_

The continue_to_signal_ subroutine enables an on unit that cannot completely handle a condition to tell the signalling program, upon its return, to search the stack for other on units for the condition. The search continues with the stack frame immediately preceding the frame for the block containing the on unit. However, if a separate on unit for the any_other condition is established in the same block activation as the caller of the continue_to_signal_ subroutine, that on unit is invoked before the stack is searched further.

Usage

```
declare continue_to_signal_ entry (fixed bin(35));

call continue_to_signal_ (code);
```

where code (Output) is a standard status code and is nonzero if continue_to_signal_ was called when no condition was signalled.

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 event channel into status information more easily used by the user.

Entry: convert_dial_message_$return_io_module

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

Usage

```
declare convert_dial_message_$return_io_module entry (fixed bin(71),
     char(*), char(*), fixed bin, 1 aligned, 2 bit(1) unal, 2 bit(1) unal,
     2 bit(1) unal, 2 bit(33) unal, fixed bin(35));

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

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.

Notes

The message may be either a control message or an informative message. Informative messages have flags.control off ("0"b), n_dialed is set to -1, channel is set to the name of the channel involved, io_module is set to the name of an I/O module, and either flags.dialed_up or flags.hung_up is on, indicating that the named channel has either just dialed up or just hung up. The io_module name is provided as a convenience; the caller is not required to use the name returned by this subroutine.

Control messages have flags.control on ("1"b), and n_dialed is set to the number of dialed terminals or -1. The code is either 0 (request accepted) or one of the following values:

error_table_$action_not_performed
: the requested action was not performed; typically, this indicates an attempt to manipulate a channel that the requesting process can not control.

error_table_$ai_out_range
: access to the requested channel is prohibited by AIM.

error_table_$bad_name
: the channel_name does not conform to required syntax.

error_table_$badcall
: the dial message was -1. The dial_manager_ subroutine will set dial_manager_arg.dial_message to -1 when an error occurs and there is no answering service dial_message to return.

error_table_$bigarg
: the dial_out_distination is too long.

error_table_$dial_active
: the process is already serving a dial qualifier.

error_table_$dial_id_busy
: the dial_qualifier is already being used by another process.

error_table_$insufficient_access
: the running process does not have the access permission required to perform the requested operation.

error_table_$invalid_resource_state
: the channel is not configured to allow the requested operation.

error_table_$name_not_found
: the dial_qualifier is not registered.

error_table_$no_connection
: it was not possible to complete the connection, e.g., dial-out failure.

error_table_$no_dialok
: the requesting process does not have the dialok attribute.

error_table_$order_error
: an error occurred while processing an order on this channel.

error_table_$request_not_recognized
indicates a software error.

error_table_$resource_not_free
the requested channel is already in use.

error_table_$resource_unavailable
no channel could be found that satisfied required characteristics.

error_table_$resource_unknown
the channel specified does not exist.

error_table_$unable_to_check_access
typically indicates that the process does not have required access, but may indicate an administrative error.

error_table_$unimplemented_version
the version of the dial_manager_arg structure supplied is not supported by dial_manager_. This error code may also indicate an internal software error.

THIS PAGE INTENTIONALLY LEFT BLANK

The convert_ipc_code_ subroutine is obsolete and has been deleted.

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 7 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: copy_acl_

The copy_acl_ subroutine copies the access control list (ACL) from one file, segment, multisegment file, or directory to another, replacing the current ACL if necessary.

Usage

```
declare copy_acl_ entry(char(*), char(*), char(*), char(*), bit(1) aligned,
     fixed bin(35));

call copy_acl_ (source_dir, source_ent, target_dir, target_ent,
     target_error_sw, code);
```

where:

1. source_dir (Input)
   is the pathname of the directory containing the source file or source directory whose ACL is to be copied.
2. source_ent (Input)
   is the entryname of the source file or source directory.
3. target_dir (Input)
   is the pathname of the directory containing the target file or target directory whose ACL is replaced.
4. target_ent (Input)
   is the entryname of the target file or target directory.
5. target_error_sw (Output)
   is "0"b if the status code reflects an error in listing the ACL of the source file or directory, and is "1"b if the code reflects an error in replacing the ACL of the target file or directory.
6. code (Output)
   is a standard status code.

Notes

An attempt to copy the ACL from a source file to a target directory, or from a source directory to a target file causes an error. Source and target must both be a file, or both a directory.

Links are chased in the processing of the source and target pathnames.

Name: create_ips_mask_

The create_ips_mask_ subroutine returns a bit string that can be used to disable specified ips interrupts (also known as ips signals).

Usage

```
declare create_ips_mask_ entry (ptr, fixed bin, bit(36) aligned);

call create_ips_mask_ (array_ptr, lng, mask);
```

where:

1. array_ptr (Input)
   is a pointer to an array of ips (interprocess signal) names that are char(32) aligned.

2. lng (Input)
   is the number of elements in the above array.

3. mask (Output)
   is a mask that disables all of the ips signals named in the array pointed to by array_ptr. (See "Notes" below.)

Notes

If any of the names are not valid ips signal names, the condition create_ips_mask_err is signalled.

If the first name in the array is -all, then a mask is returned that masks all interrupts.

Currently the allowed ips names are:

quit
cput
alrm
neti
sus_
trm_
wkp_

The returned mask contains a "0"b in the bit position corresponding to each ips name in the array, and a "1"b in all other bit positions. The bit positions are ordered as in the above list. It should be noted that it is necessary to complement this mask (using a statement of the form "mask = ^mask") in cases where the requirement is for a mask with "1" bits corresponding to specified interrupts. An ips mask is used as an argument to the following entry points: hcs_$reset_ips_mask, hcs_$set_automatic_ips_mask, and hcs_$set_ips_mask.

Name: cross_ring_

The cross_ring_ I/O module allows an outer ring to attach a switch to a preexisting switch in an inner ring, and to perform I/O operations by forwarding I/O from the attachment in the outer ring through a gate to an inner ring. The cross_ring_ I/O module is not called directly by users; rather the module is accessed through the I/O system.

Attach 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_.)

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_dir_mode_

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

Usage

```
declare cv_dir_mode_ entry (char(*), bit(*), fixed bin(35));

call cv_dir_mode_ (char_modes, bit_modes, code);
```

where:

1. char_modes (Input)
   are the character string access modes.

2. bit_modes (Output)
   are the bit string access modes.

3. code (Output)
   is a standard status code. It may be:
   error_table_$bad_acl_mode
   if char_modes contains an invalid directory access mode character

Notes

If char_modes is "null" or "n", bit_modes is set to "0"b. The mode characters in char_modes may occur in any order. Spaces are ignored. The following table indicates what bit in bit_modes is turned on when the access mode character is found.

Access Mode	Bit in bit_modes
s	1
m	2
a	3

This page intentionally left blank.

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.
dir>entry$entry_pt	entry at word identified by entry point entry_pt in segment identified by pathname dir>entry.
<dir>entry$entry_pt	entry at word identified by entry point entry_pt in segment identified by pathname <dir>entry.
<entry$entry_pt	entry at word identified by entry point entry_pt in segment identified by pathname <entry.
path	same as path¦[entry path]. If path contains no ">" or "<" characters, it is interpreted as a ref_name.
ref_name$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.

*

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_mode_

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

Usage

```
declare cv_mode_ entry(char(*), bit(*), fixed bin(35));

call cv_mode_ (char_modes, bit_modes, code);
```

where:

1. char_modes (Input)
   are the character string access modes.

2. bit_modes (Output)
   are the bit string access modes.

3. code (Output)
   is a standard status code. It may be:
   error_table_$bad_acl_mode
   if char_mode contains an invalid segment access mode character

Notes

If char_modes is "null" or "n", bit_modes is set to "0"b. The mode characters in char_modes may occur in any order. Spaces are ignored. The following table indicates what bit in bit_modes is turned on when the access mode character is found.

Access Mode	Bit in bit_modes
r	1
e	2
w	3

This page intentionally left blank.

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.
dir>entry$entry_pt	points to word identified by entry point entry_pt in segment identified by pathname dir>entry.
<dir>entry$entry_pt	points to word identified by entry point entry_pt in segment identified by pathname <dir>entry.
<entry$entry_pt	points to word identified by entry point entry_pt in segment identified by pathname <entry.
ref_name$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: cv_rcp_attributes_

The cv_rcp_attributes_ subroutine contains several entry points that are useful in manipulating RCP resource attribute specifications and descriptions.

RCP resource attribute descriptions are printable strings that describe the attributes of resources (devices and volumes). For a description of the syntax of attribute descriptions see the Multics Administrators' Manual Project, Order No. AK51.

RCP resource attribute specifications are encoded representations of attribute descriptions. They may be either absolute, relative, or multiple. An absolute attribute specification represents a complete and consistent state of all the attributes of a resource. A relative attribute description represents a desired modification to the state of all the attributes of a resource, and must be applied to an absolute attribute specification to produce the desired change in that absolute specification. A multiple attribute specification does not represent a consistent state of all the attributes of a resource at any given time, but is useful for representing the union of all such consistent states, i.e., potential attributes.

Entry: cv_rcp_attributes_$to_string

This entry point takes an RCP resource attribute specification and produces a printable RCP attribute description.

Usage

```
declare cv_rcp_attributes_$to_string entry (char (*), bit (72)
     dimension (2), char (*) varying, fixed bin (35));

call cv_rcp_attributes_$to_string (type, attributes, string, code);
```

where:

1. type (Input)
   specifies the type of resource from which attributes was obtained e.g., tape, disk_drive (see "Notes" below).

2. attributes (Input)
   is an RCP attribute specification (sees "Notes" below).

3. string (Output)
   is a printable RCP attribute description.

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

Notes

A list of defined resource types may be obtained via the list_resource_types command. *

Entry: cv_rcp_attributes_$from_string

This entry point accepts a printable RCP attribute description and produces an RCP attribute specification.

Usage

```
declare cv_rcp_attributes_$from_string entry (char (*), bit (72)
     dimension (2), char (*)varying, fixed bin (35));

call cv_rcp_attributes_$from_string (type, attributes, string, code);
```

where:

1. type (Input)
   specifies the type of resource to which attributes applies.

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

3. string (Input)
   is the same as above.

4. code (Output)
   is the same as above.

Entry: cv_rcp_attributes_$modify

This entry point applies a printable RCP resource attribute description (representing a relative attribute specification) to a given resource specification and returns a new attribute specification as the result. The resulting attribute specification consists of the original attribute specification, modified by the attributes specified in the printable description.

Usage

```
declare cv_rcp_attributes_$modify entry (char (*), bit (72) dimension (2),
     char (*)varying, bit (72) dimension (2), fixed bin (35));

call cv_rcp_attributes_$modify (type, attributes, string, new_attributes,
     code);
```

where:

1. type (Input)
   specifies the type of resource to which attributes and string apply.

2. attributes (Input)
   is an absolute RCP attribute specification.

3. string (Input)
   is a printable RCP attribute description that is to modify attributes.

4. new_attributes (Output)
   is the new absolute RCP attribute specification.

5. code (Output)
   is the same as above.

Entry: cv_rcp_attributes_$from_string_rel

This entry point generates a relative attribute specification that can later be applied to attribute specifications of specific resources via the cv_rcp_attributes_$modify_rel entry point.

Usage

```
declare cv_rcp_attributes_$from_string_rel entry (char (*),
     char (*)varying, bit (72) dimension (4), fixed bin (35));

call cv_rcp_attributes_$from_string_rel (type, string, rel_attributes,
     code);
```

where:

1. type (Input)
   specifies the type of resource to which string applies.

2. string (Input)
   is a printable RCP attribute description.

3. rel_attributes (Output)
   is the relative RCP attribute specification.

4. code (Output)
   is the same as above.

Entry: cv_rcp_attributes_$modify_rel

This entry point applies a relative attribute specification produced by the cv_rcp_attributes_$from_string_rel entry point to an absolute attribute specification of a specific resource.

Usage

```
declare cv_rcp_attributes_$modify_rel entry (bit (72) dimension (2),
     bit (72) dimension (4), bit (72) dimension (2));

call cv_rcp_attributes_$modify_rel (attributes, rel_attributes,
     new_attributes);
```

where:

1. attributes (Input)
   is an absolute attribute specification.

2. rel_attributes (Input)
   is a relative attribute specification to be applied to attributes.

3. new_attributes (Output)
   is the resulting absolute attribute specification.

Notes

The caller must ensure that attributes and rel_attributes refer to the same resource type, i.e., were generated by previous calls to cv_rcp_attributes_ where the type arguments were identical.

Entry: cv_rcp_attributes_$reduce_implications

This entry point accepts an attribute specification for a volume and returns the necessary minimal attribute specification that a device must possess to be able to accept the volume.

Usage

```
declare cv_rcp_attributes_$reduce_implications entry (char (*), bit (72)
     dimension(2), char (*), bit (72) dimension (4), fixed bin (35));
```

```
call cv_rcp_attributes_$reduce_implications (vol_type, vol_attributes,
     dev_type, dev_attributes, code);
```

where:

1. vol_type (Input)
   specifies the type of volume from which vol_attributes was obtained.

2. vol_attributes (Input)
   is an absolute attribute specification for the volume type specified.

3. dev_type (Output)
   is the resource type of the device that accepts the given volume type.

4. dev_attributes (Output)
   is a minimal relative attribute specification for a device capable of accepting a volume with the given attributes.

5. code (Output)
   is the same as above.

Entry: cv_rcp_attributes_$protected_change

This function entry point accepts an absolute attribute specification for a resource and a relative attribute specification which is to modify it. It returns a value expressing whether or not this modification would affect protected attributes of the resource. No modification is actually attempted by this entry.

Usage

```
declare cv_rcp_attributes_$protected_change entry (bit (72) dimension(2),
     bit (72) dimension(4)) returns (bit (1) aligned);

protected_change = cv_rcp_attributes_$protected_change (attributes,
     rel_attributes);
```

where:

1. attributes (Input)
   is an RCP attribute specification.

2. rel_attributes (Input)
   is a relative attribute specification to be applied to attributes.

3. protected_change (Output)
   is "1"b if this operation would modify protected attributes of the resource; otherwise, it is "0"b.

Entry: cv_rcp_attributes_$test_valid

This entry point is used to determine whether a given attribute specification is absolute, relative, multiple, or invalid.

Usage

```
declare cv_rcp_attributes_$test_valid entry (char(*), bit 72 dimension (2),
     fixed bin, fixed bin (35));

call cv_rcp_attributes_$test_valid (type, attributes, validity, code);
```

where:

1. type (Input)
   specifies the type of resource to which attributes applies.

2. attributes (Input)
   is an RCP attribute specification.

3. validity (Output)
   shows whether the attribute specification is absolute, relative, or multiple.
   0 is an absolute attribute specification
   1 is a relative attribute specification
   2 is a multiple attribute specification

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

Name: cv_userid_

The cv_userid_ subroutine converts a character string containing an abbreviated User_id into one containing all three components, i.e. Person_id.Project_id.tag.

Usage

```
declare cv_userid_ entry (char(*)) returns (char(32));

user_id = cv_userid_ (string);
```

where:

1. string (Input)
   is the abbreviated User_id.

2. user_id (Output)
   is a User_id containing all three components.

Notes

The Person_id, Project_id and tag components are truncated to 20, 9 and 1 characters, respectively. An asterisk ("*") is supplied for missing components.

Examples

```
Abbreviated User_id     Full User_id
--------------------    ------------
Smith.Project.a         Smith.Project.a
Smith.Project           Smith.Project.*
Smith                   Smith.*.*
.Project                *.Project.*
```

Name: 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 2 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
   <u>n</u> descriptor is an array of <u>n</u> 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 define_area_ subroutine gives the user more control over an area than is defined in the PL/I language. The PL/I empty built-in function cannot empty a define_area_ area; the release_area_ subroutine must be used instead. PL/I offset values and PL/I area assignment cannot be used with extensible areas. In PL/I, an area variable is always initialized. Consequently, if a based area is overlayed upon arbitrary storage instead of being allocated with a PL/I allocate statement, then the define_area_ subroutine must be used to turn the contents of the based area into a PL/I area value.

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

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

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 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. The minimum size is thirty-two (decimal) words. The maximum size is the maximum number of words in a segment.

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. areap
    is a pointer to the region to be initialized as an area. If this pointer is null, a temporary segment is acquired for the area and areap is set as a returned value. If areap is initially nonnull, it must point to a 0 mod 2 address.

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. This subroutine uses a structure, dial_manager_arg, to receive arguments from its caller. This structure is described below, under "Notes". For more information, see the description of the dial command in the MPM Commands.

The dial_manager_ subroutine uses an event channel to communicate with the answering service. This event channel is specified by dial_manager_arg.dial_channel. The channel must be created by the caller. The answering service sends notices of dial connections and hangups over this channel. The dial_manager_ subroutine goes blocked on the event-wait channel awaiting a response to the request from the answering service. When the user program receives wakeups over this channel, it should call the convert_dial_message_ subroutine to decode the event message.

The dial_manager_$allow_dials and dial_manager_$registered_server entry points establish a dial line. The dial_id specified in dial_manager_arg.dial_qualifier is used as the first argument to the dial command when connecting a terminal to a process. The dial_id may be an alphanumeric string from 1 to 12 characters long. The dial_id "system" and "s" are reserved for the Initializer process. A process can have only one dial line active at a time.

Entry: dial_manager_$allow_dials

This entry point requests that the answering service establish a dial line to allow terminals to dial to the calling process. The caller must set dial_manager_arg.dial_qualifier to the dial_id for the dial line. The caller must also set dial_manager_arg.dial_channel to an event-wait channel in the caller's process. After the dial_manager_$allow_dials entry point has been called, the event channel may be changed to an event-call channel. To connect a terminal to the process, the User_id of the process must be specified as the second argument of the dial command. If the process has already established another dial line, the request is rejected and code is set to error_table_$dial_active.

Usage

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

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

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 establish a dial line to allow terminals to dial to the calling process using only the dial qualifier. The calling process must have rw access to the access control segment dial.<dial qualifier>.acs in >sc1>rcp if this request is to be honored. If the process has already established a dial line, the request is rejected and code is set to error_table_$dial_active.

Usage

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

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

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 destination and, if the channel is successfully dialed, that the channel be assigned to the requesting process. The caller must set dial_manager_arg.dial_out_destination to the telephone number to be dialed. The caller must also set dial_manager_arg.dial_channel to an event-wait channel in his process. The answering service sends notice of dial completions and hangups over this channel. After the dial_manager_$dial_out entry point has been called the event channel may be changed to an event-call channel. The user programs receiving the wakeup should call the convert_dial_message_ subroutine to decode the event message. The caller may set dial_manager_arg.channel_name to the name of a specific channel to be used. It is also possible to set dial_manager_arg.channel_name to a starname, in which case the answering service chooses a channel that has a matching name and has all the attributes specified in dial_manager_arg.reservation_string. The name of the chosen channel is not returned by dial_manager_; it must be obtained via a call to convert_dial_message_.

Usage

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

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

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 caller must set dial_manager_arg.dial_channel to an event wait channel in the caller's process. The caller also must set dial_manager_arg.channel_name to the name of the channel to be released. The user must make dial_manager_arg.dial_channel an event-wait channel before using this call. If the channel was dialed, the channel is returned to the answering service and another access request may be issued. If the channel is a slave channel, the channel is hung up.

Usage

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

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

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

Entry: dial_manager_$release_channel_no_hangup

This entry point performs the same function as the dial_manager_$release_channel entry point except that slave channels are not hung up.

Entry: dial_manager_$release_no_listen

This entry point requests the answering service to release the channel specified in channel_name, which must have been attached by means of the dial_manager_$tandd_attach entry point. The channel is left in a hung-up state and is not available for use until an explicit "attach" operator command is issued for the channel. This entry point has the same requirements as the dial_manager_$release_channel entry point.

Usage

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

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

where the arguments are the same as for the dial_manager_$release_channel 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_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_$release_dial_id

This entry point functions as does dial_manager_$shutoff_dials, except that dialed terminals are not hung up. The user can later release dialed terminals by a call to dial_manager_$shutoff_dials or by calls to dial_manager_$release_channel.

Usage

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

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

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

Entry: dial_manager_$privileged_attach

This entry point allows a privileged process to attach a "slave" channel. The effect is as if that terminal had dialed to the requesting process. The caller must set all variables required by the dial_manager_$allow_dials entry point and then must set dial_manager_arg.channel_name to the name of the channel that is to be attached; dial_manager_arg.dial_qualifier is not used and should be set to the null string. This must be the same name as specified by the channel master file. The slave service type must be specified for this channel in the channel master file. The calling process must have rw access to the access control segment <channel_name>.acs in >sc1>rcp if this request is to be honored.

Usage

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

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

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

Entry: dial_manager_$tandd_attach

This entry point allows a process with appropriate access to attach any communications channel that is in the channel master file and not already in use, for the purpose of performing online testing of the channel. The requesting process acquires the channel in a hung-up, nonlistening state. The channel can be released using either the dial_manager_$release_channel or the dial_manager_$release_no_listen entry point. In the latter case, the channel will be unavailable to users until the operator enters an attach command for the channel. The caller must set all the variables required by the dial_manager_$privileged_attach entry point; dial_manager_arg.dial_qualifier is not used and should be set to the null string.

Usage

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

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

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

Access Required

The caller must have at least rw access to both >sc1>rcp>tandd.acs and >sc1>rcp>CHAN_NAME.acs, where CHAN_NAME is the name of the channel to be attached.

Entry: dial_manager_$terminate_dial_out

This entry point is used to request that the answering service hang up an
* auto call line and unassign it from the requesting process. The caller must set dial_manager_arg.channel_name to the name of the channel being used; channel_name cannot be null. The caller also must set dial_manager_arg.dial_channel to an event-wait channel.

Usage

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

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

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 is declared in dial_manager_arg.incl.pl1. It has the following declaration:

```
dcl 1 dial_manager_arg        aligned,
      2 version               fixed bin initial (2),
      2 dial_qualifier        char (22),
      2 dial_channel          fixed bin (71),
      2 channel_name          char (32),
      2 dial_out_destination  char (32),
      2 reservation_string    char (256),
      2 dial_message          fixed bin (71);
```

where:

1. version
   indicates the version of the structure that is being used. This is set by the caller and must be 2.

2. dial_qualifier
   is the dial qualifier for calls to the dial_manager_$allow_dials, dial_manager_$registered_server, dial_manager_$shutoff_dials, and dial_manager_$release_dial_id entry points. This field should be set to blanks if it is not used.

3. dial_channel
   is an interprocess communication channel used to receive messages from the answering service. The channel must always be an event-wait channel at the time a call to any dial_manager_ entry is made.

4. channel_name
   is used for calls to the dial_manager_$terminate_dial_out and dial_manager_$release_channel entry points to indicate which channel should be disconnected. In calls to the dial_manager_$privileged_attach entry point, it indicates which channel should be attached. In calls to the dial_manager_$dial_out entry point, it indicates which auto_call channel should be used for a dial-out attempt. For this entry, the following convention is observed: the caller can fully specify a channel name or can use the star convention to specify a group of channels from which the answering service is to pick one. A channel_value of "" (null string) is equivalent to "**"; other examples of acceptable values are: "a.h*.*" and "a.*.*.co". (Consult the MPM Reference Guide for a description of the star convention.) This field should be set to blanks if it is not used.

5. dial_out_destination
   is used for calls to the dial_manager_$dial_out entry point. Interpretation of this value is determined by the multiplexer that controls the channel being dialed out. The standard FNP multiplexer interprets this value as a telephone number and ignores all characters except decimal digits and the exclamation point (!). It recognizes "!" as a dial-tone-wait character and will suspend dialing until the autocall unit receives a dial tone. Any number of "!" characters can exist in a dial_out_destination, and the standard FNP multiplexer will pause at each. This field should be set to blanks if it is not used.

6. reservation_string
   is used to specify the desired characteristics of a channel in calls to the dial_manager_$dial_out entry. The reservation string (which can be null), consists of reservation attributes separated by commas. The channel used by a dial-out operation must have the characteristics specified in the reservation string. Reservation attributes consist of a keyword and optional argument. Attributes allowed are:

   ```
   baud_rate=BAUD_RATE
   line_type=LINE_TYPE
   ```

   The attribute name, such as "baud_rate", must appear literally in the string. BAUD_RATE is a decimal representation of the desired channel line speed and must appear in a baud_rate attribute. LINE_TYPE is a valid line type, chosen from line_types.incl.pl1 and must appear in a line_type attribute. Examples: "baud_rate=300, line_type=ASCII", "line_type=BSC". This field should be set to blanks if it is not used or no particular channel attributes are required.

7. dial_message (Output)
   is a copy of the dial_message received from the answering service. The dial_manager_ subroutine makes an answering service request based upon the arguments supplied by its caller; it then waits for a reply from the answering service. This reply is converted using convert_dial_message_, and some of the results of the conversion are immediately available to dial_manager_ callers as output arguments. To obtain other portions of the dial_message absorbed by dial_manager_, the user must call convert_dial_message_ specifying the value of this field. This field is set to -1 if an error occurs in the dial_manager_ or answering service request; convert_dial_message_ rejects attempts to convert such a message with the return code error_table$badcall.

The second argument in all calls (code) is an error status indicator. It can assume any value documented in the convert_dial_message_ description (earlier in this manual), or one of the following:

error_table_$bad_conversion
: a reservation_string value (BAUD_RATE) was not a proper decimal value.

error_table_$invalid_line_type
: the value of LINE_TYPE is not acceptable.

error_table_$bad_arg
: reservation_string contains an unrecognized attribute.

Name: dl_handler_

This subroutine has three entry points that issue queries for each of three situations involving deletion. These situations are:

1. Deletion of an entry whose safety switch or copy switch is on.

2. Deletion via a starname that matches all entries, e.g. "**".

3. Deletion of a directory (delete_dir always queries).

This subroutine returns a status code depending on the user's answer. If the user answers "yes", all three entry points turn off the safety and copy switches, and in the case of a directory, set sma to the user before returning.

Entry: dl_handler_

This entry point, called when an entry has its safety switch or copy switch on, issues a query of the form:

<caller>: <path> is protected. Do you want to delete it?

If the user answers yes, dl_handler_ turns off both switches and returns a zero status code.

Usage

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

call dl_handler_ (caller, dn, en, code);
```

where:

1. caller (Input)
   is the name of the calling program, used to print the query.

2. dn (Input)
   is the directory name.

3. en (Input)
   is the entry name.

4. code (Output)
   is a standard status code. It may be:
   0
       if the user has answered "yes", switches have been turned off and the entry can now be deleted
   error_table_$action_not_performed
       if the user answered "no"
   other codes
       mean that the switches could not be turned off

The two other entry points have the same calling sequence as dl_handler_.

Entry: dl_handler_$dblstar

This entry point issues the query:

Do you want to '<caller> <en>' in <dn>?

where caller, the name of the calling program, is assumed to be a suitable verb. This entry point is called, for example, by the delete and unlink commands, which also pass a double starname as en:

Do you want to 'delete **' in <dir_path>?
Do you want to 'unlink **' in <dir_path>?

Entry: dl_handler_$dirdelete

This entry point assumes it is given a directory pathname, and issues the query:

<caller>: Do you want to delete the directory dn>en?

This entry point is called, for example, by the delete_dir command.

Name: dprint_

This subroutine contains several entry points used to submit requests to the I/O daemon for printing or punching of segments and multisegment files.

Entry: dprint_

This entry point adds a request to print or punch a segment or multisegment file to the specified queue.

Usage

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

call dprint_ (dir_name, entryname, dprint_arg_ptr, code);
```

where:

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

2. entryname (Input)
   is the entry name of the segment, multisegment file, or link to the segment or multisegment file to be printed or punched.

3. dprint_arg_ptr (Input)
   is a pointer to the dprint_arg structure (described in "Notes" below) that defines the options for this request. If this pointer is null, the default settings are used for all options.

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

Notes

The dprint_ subroutine uses the following structure, defined in the system include file dprint_arg.incl.pl1, to determine the details of the request. If no structure is supplied, default values are used.

```
dcl 1 dprint_arg                  based aligned,
      2 version                   fixed bin,
      2 copies                    fixed bin,
      2 delete                    fixed bin,
      2 queue                     fixed bin,
      2 pt_pch                    fixed bin,
      2 notify                    fixed bin,
      2 heading                   char(64),
      2 output_module             fixed bin,
      2 dest                      char(12),
```

```
2 carriage_control,
  3 nep                      bit(1) unaligned,
  3 single                   bit(1) unaligned,
  3 non_edited               bit(1) unaligned,
  3 truncate                 bit(1) unaligned,
  3 center_top_label         bit(1) unaligned,
  3 center_bottom_label      bit(1) unaligned,
  3 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),
2 bit_count                  fixed bin(35),
2 form_name                  char(24),
2 destination                char(24),
2 chan_stop_path             char(168),
2 request_type               char(24)unaligned;
```

where:

1. version
   is the version number of the structure. This is set by the caller and must be the value of the named constant dprint_arg_version_6 also defined in the include file.

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 the default queue for the default print/punch request type and is site-defined).

5. pt_pch
   indicates whether the request is for printing, punching, or plotting.
   1 print request (default)
   2 punch request
   3 plot 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
   5 indicates plotting

9. dest
   is not used. See destination below.

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
   is not used. See request_type below.

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.)

25. bit_count
    is the segment bit count.

26. form_name
    is the name of special forms needed.

27. destination
    is the string to be used to indicate where the output should be delivered. If it is null, the requestor's Project_id is used. The default is the null string.

28. chan_stop_path
    is the path of user channel stops.

29. request_type
    is the request type name to be used to queue the request. If printing is requested, the request type must be of the generic type "printer"; if punching is requested, the request type must be of generic type "punch." (The default request type for printing is "printer"; the default for punching is "punch.")

Entry: dprint_$check_daemon_access

This entry point checks the I/O daemon's access to a given segment or multisegment file. It returns whether the daemon responsible for a given request type has "r" access to the file and "s" access to the containing directory and whether the I/O daemon coordinator can delete the file if requested.

Usage

```
declare dprint_$check_daemon_access entry (char(*), char(*), char(*),
     bit(1) aligned, bit(1) aligned, bit(1) aligned, char(*),
     fixed bin(35));

call dprint_$check_daemon_access (dirname, entryname, request_type,
     delete_permission, read_permission, status_permission, driver_userid,
     code);
```

where:

1. dirname (Input)
   is the absolute pathname of the containing directory.

2. entryname (Input)
   is the entry name of the segment, or multisegment file, or a link to the segment or multisegment file for which the daemon's access is to be checked.

3. request_type (Input)
   is the name of the request type in which a request to print or punch the file will be placed. The access of the driver process for this request type will be returned.

4. delete_permission (Output)
   indicates whether the I/O coordinator has sufficient access to delete the file if requested. The coordinator requires "m" access to the containing directory to delete the file.

5. read_permission (Output)
   indicates whether the driver process of the given request type has "r" access to the given segment or multisegment file.

6. status_permission (Output)
   indicates whether the driver process of the given request type has "s" access to the directory containing the segment or multisegment file.

7. driver_userid (Output)
   is the name of the process that processes requests for the specified type. This value is in the form "Person_id.Project_id.*".

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

Notes

The user must have "s" access to the directory containing the segment or multisegment file to determine whether the driver has read access to the file.

The user must have "s" access to the directory containing the directory containing the segment or multisegment file in order to determine whether the I/O coordinator can delete the file and whether the driver process has "s" access to the containing directory.

Entry: dprint_$queue_contents

This entry point returns the number of requests in a specific I/O daemon queue.

Usage

```
declare dprint_$queue_contents entry (char(*), fixed bin, fixed bin,
     fixed bin(35));

call dprint_$queue_contents (request_type, queue, n_requests, code);
```

where:

1. request_type (Input)
   is the name of the request type whose queue is to be checked.

2. queue (Input/Output)
   is the number of the queue to be examined. If -1 is specified, the default queue of the given request type is checked and the number of the default queue is returned in this parameter.

3. n_requests (Output)
   is the number of requests in the specified queue.

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

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: execute_epilogue_

The execute_epilogue_ subroutine is called during process or run unit termination to call the routines in the list of epilogue handlers. The logout and new_proc commands are the prime callers of execute_epilogue_. It is also called when the run unit terminates to allow programs executing in the run unit to clean up. The add_epilogue_handler_ subroutine is used to add a program to the list that execute_epilogue_ calls.

Usage

```
declare execute_epilogue_ entry (bit (1) aligned);

call execute_epilogue_ (run_only);
```

where run_only (Input) is set to "1"b if epilogue handlers are to be invoked only for the run unit and not for the entire process.

*

Name: find_condition_frame_

This subroutine returns a pointer to the most recent condition frame, or the most recent one before a specified frame.

Usage

```
dcl find_condition_frame_ entry (ptr) returns (ptr);

stack_ptr = find_condition_frame_ (start_ptr);
```

where:

1. start_ptr (Input)
   is a pointer to a stack frame. The most recent condition frame before this stack frame is returned. The start_ptr argument can be obtained by another call to find_condition_frame_. If start_ptr is null, the most recent condition frame is returned.

2. stack_ptr (Output)
   is a pointer to the desired condition frame.

Note

The condition history can be traced by repeated calls to find_condition_frame_, starting with a null start_ptr argument and repeatedly passing the output stack_ptr as input.

Name: find_condition_info_

This subroutine, given a pointer to a stack frame being used when a signal occurred, returns information relevant to that condition.

Usage

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

call find_condition_info_ (stack_ptr, condition_info_ptr, code);
```

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 find_condition_frame_; if null, the most recent condition frame is used.

2. condition_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 that condition_info_ptr points to is declared in the include file condition_info.incl.pl1. It is declared as:

```
dcl 1 condition_info     aligned based (condition_info_ptr),
      2 mc_ptr           ptr,
      2 version          fixed bin,
      2 condition_name   char(32) varying,
      2 info_ptr         ptr,
      2 wc_ptr           ptr,
      2 loc_ptr          ptr,
      2 flags            unaligned,
        3 crawlout       bit(1),
        3 pad1           bit(35),
      2 pad2             bit(36),
      2 user_loc_ptr     ptr,
      2 pad3             (4) bit(36);
```

where:

1. mc_ptr
   if not null, points to the machine conditions. Machine conditions are described in the MPM Reference Guide.

2. version
   is the version number of this structure. It should be set to condition_info_version_1. This variable is declared in condition_info.incl.pl1.

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 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. pad1
   is currently unused and should be set to "0"b.

9. pad2
   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. pad3
    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_arg_descs_

This subroutine returns information about the calling sequence of a procedure entry point.

Entry: get_entry_arg_descs_

This entry point, given a pointer to the entry sequence or segdef of a procedure entry point, returns a list of argument descriptors describing the parameters of the entry point.

Usage

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

call get_entry_arg_descs_ (entry_ptr, nargs, desc_ptrs, code);
```

where:

1. entry_ptr (Input)
   points to the entry sequence or segdef of the procedure entry point whose parameter descriptors are to be described.

2. nargs (Output)
   is the number of parameters declared in the procedure entry point.

3. desc_ptrs (Output)
   is an array of pointers to the argument descriptors describing the declared parameters of the entry point. If dimension (desc_ptrs, 1) is less than nargs, the pointers identify the first dimension (desc_ptrs, 1) parameter descriptors.

4. code (Output)
   is a standard status code. It may be:
   error_table_$nodescr
   the entry point did not have parameter descriptors.

Notes

For some version 0 object segments, a code of zero is returned, nargs is set, but the descriptor pointers in desc_ptrs are null.

Entry: get_entry_arg_descs_$info

This entry point, given a pointer to the entry sequence or segdef of a procedure entry point, returns a list of argument descriptors describing the parameters of the entry point, plus a set of entry sequence flags which further describe the entry point.

Usage

```
declare get_entry_arg_descs_$info entry (ptr, fixed bin, (*) ptr, ptr,
     fixed bin(35));

call get_entry_arg_descs_$info (entry_ptr, nargs, desc_ptrs,
     entry_desc_info_ptr, code);
```

where:

1. - 3.
are as described above.

4. entry_desc_info_ptr (Input)
points to the structure described under "Notes" below.

5. code (Output)
is as described above.

Notes

The entry_desc_info_ptr argument of get_entry_arg_descs_$info points to the structure shown below. This structure is declared in entry_desc_info.incl.pl1.

```
dcl 1 entry_desc_info            aligned based(entry_desc_info_ptr),
      2 version                  fixed bin,
      2 flags,
       (3 basic_indicator,
        3 revision_1,
        3 has_descriptors,
        3 variable,
        3 function)              bit(1) unaligned,
        3 pad                    bit(13) unaligned,
    entry_desc_info_version_1    fixed bin int static
                                 options(constant) init(1),
    entry_desc_info_ptr          ptr;
```

where:

1. version
   is the version number of this structure. The current version number is 1. The named constant, entry_desc_info_version_1 should be used to set this version number.

2. flags
   are the flags which further describe the procedure entry point.

3. basic_indicator
   is on if the entry point is in a program written in the BASIC language.

4. revision_1
   is on if the entry sequence has version 1 descriptor data.

5. has_descriptors
   is on if the entry sequence has argument descriptors describing its parameters.

6. variable
   is on if the entry point accepts an undefined number of arguments, and has been declared with the options(variable) attribute. This flag will usually be off for entry points in command and active function procedures, even though these procedures accept a variable number of arguments. Command and active function procedures usually do not declare their entry points with explicit parameters or with the options(variable) attribute.

7. function
   is on if the procedure entry point is a function which returns a value. The final parameter argument descriptor describes this return value.

8. entry_desc_info_version_1
   is a named constant which the caller should use to set the version number in the structure above.

9. entry_desc_info_ptr
   points to the structure above.

Entry: get_entry_arg_descs_$text_only

This entry point, given a pointer to the entry sequence of a procedure entry point, returns a list of argument descriptors describing the parameters of the entry point. It differs from the get_entry_arg_descs_ entry point, in that it assumes that it is given a pointer to an entry sequence in the text section of the procedure, rather than checking to see if it was given a pointer to a segdef.

Usage

```
declare get_entry_arg_descs_$text_only entry (ptr, fixed bin, (*) ptr,
     fixed bin(35));

call get_entry_arg_descs_$text_only (entry_ptr, nargs, desc_ptrs, code);
```

where the arguments are the same as for the get_entry_arg_descs_ entry point above. If entry_ptr does not point to an entry point in the text section, then error_table_$nodescr is returned as the value of code.

Entry: get_entry_arg_descs_$text_only_info

This entry point, given a pointer to the entry sequence of a procedure entry point, returns a list of argument descriptors describing the parameters of the entry point, plus a set of entry sequence flags which further describe the entry point. It differs from the get_entry_arg_descs_$info entry point, in that it assumes that it is given a pointer to an entry sequence in the text section of the procedure, rather than checking to see if it was given a pointer to a segdef.

Usage

```
declare get_entry_arg_descs_$text_only_info entry (ptr, fixed bin, (*) ptr,
     ptr, fixed bin(35));

call get_entry_arg_descs_$text_only_info (entry_ptr, nargs, desc_ptrs,
     entry_desc_info_ptr, code);
```

where the arguments are the same as for the get_entry_arg_descs_$info entry point above.

Name: get_entry_name_

The get_entry_name_ subroutine, given a pointer to an externally defined location or entry point in a segment, returns the associated name.

Usage

```
declare get_entry_name_ entry (ptr, char(*), fixed bin(18), char(8) aligned,
     fixed bin(35));

call get_entry_name_ (entry_ptr, symbolname, segno, lang, code);
```

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_entry_point_dcl_

The get_entry_point_dcl_ subroutine returns attributes needed to construct a PL/I declare statement for external procedure entry points and for error_table_ codes and other system-wide external data. The program obtains the attributes from data files declaring all unusual procedure entry points (e.g., ALM segments), from system-wide data values sys_info$max_seg_size), and from the argument descriptors describing the entry point's parameters that are included with the entry point itself.

Entry: get_entry_point_dcl_

This entry point returns the declaration for an external value, either from one of the data files, or by using the parameter argument descriptors associated with the procedure entry point. It makes a special case of error_table_ values by always returning 'fixed bin(35) ext static' for them. For example, given the name iox_$put_chars, it might return:

```
entry (ptr, ptr, fixed bin(21), fixed bin(35))
```

Note that neither the name of the external value nor any trailing semicolon (;) is returned as part of the declaration.

Usage

```
dcl get_entry_point_dcl_ entry (char(*), fixed bin, fixed bin,
     char(*) varying, char(32) varying, fixed bin(35));

call get_entry_point_dcl_ (name, dcl_style, line_length, dcl, type, code);
```

where:

1. name (Input)
   is the name of the external entry point or data item whose declaration must be obtained.

2. dcl_style (Input)
   is the style of indentation to be performed for the name. See "Notes" below for a list of allowed values.

3. line_length (Input)
   is the maximum length to which lines in return value are allowed to grow when indentation is performed.

4. dcl (Output)
   is the declaration that was obtained.

5. type (Output)
   is the type of declaration. In the current implementation, this is always a null string.

6. code (Output)
   is a standard status code describing any failure to obtain the declaration.

Notes

Three styles of declaration indentation are supported by the dcl_style argument described above. Style 0 (dcl_style = 0) involves no indentation. The declaration is returned as a single line.

Style 1 (dcl_style = 1) indents the declaration in the format similar to the indent command. Long declarations are broken into several lines. For example, a declare statement for hcs_$initiate_count would appear as:

```
dcl  hcs_$initiate_count entry (char(*), char(*), char(*), fixed bin(24),
          fixed bin(2), ptr, fixed bin(35));
```

when the string "dcl hcs_$initiate_count" is concatenated with the value returned by get_entry_point_dcl_, and a semicolon (;) is appended to this value.

Style 2 (dcl_style = 2) indents the declaration in an alternate format that makes the name of the entry point stand out from its declaration. It assumes that the name of the entry point begins in column 11 (indented one horizontal tab stop from left margin), and the declaration begins in column 41. In style 2, the declare statement for hcs_$initiate_count would appear as:

```
     dcl   hcs_$initiate_count          entry (char(*), (char(*), (char(*),
                                             fixed bin(24), fixed bin(2), ptr,
                                             fixed bin(35));
```

Most command and active function entry points do not declare arguments in their procedure statements since they accept a variable number of arguments. Neither do they use the options(variable) attribute in their procedure statements. Therefore, when get_entry_point_dcl_ encounters a procedure entry point with no declared arguments and without options(variable), it assumes the options(variable) attribute required for commands and active functions and returns:

```
          entry options(variable)
```

It distinguishes between such assumed options(variable) entries and those that explicitly use the options(variable) attribute in their procedure statement by returning "entry" for the assumed case and "entry()" for the explicit case. Thus, for the display_entry_point_dcl command, which explicitly uses options(variable) in its procedure statement, get_entry_point_dcl_ returns:

```
          entry() options(variable)
```

Search List

The get_entry_point_dcl_ subroutine uses the "declare" search list, which has the synonym "dcl", to find data files describing unusual procedure entry points. For more information about search lists, see the descriptions of the search facility commands and, in particular, the add_search_paths command description (in the MPM Commands). Type:

```
print_search_paths declare
```

to see what the current declare search list is. The default search list identifies the data file:

```
>sss>pl1.dcl
```

User-Provided Data Files

Users may provide data files that redeclare standard system entry points (e.g., redeclaring a subroutine as a function), or that declare their own entry points or external data items. The add_search_paths command can be used to place user-provided data files in the "declare" search list. For example:

```
add_search_paths declare [hd]>my_pl1.dcl -first
```

Declarations have the general form of:

```
virtual_entry declaration
```

For example:

```
ioa_ entry options(variable)
```

Note that the word "dcl" is not included in the data item, nor does the declaration end with a semicolon (;). External data values are declared in a similar fashion. For example:

```
iox_$user_output ptr external static
```

THIS PAGE INTENTIONALLY LEFT BLANK

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 3 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.

Entry: get_equal_name_$component

This entry point accepts an archive and component name and two equal names as input and constructs a target archive and component name by substituting components or characters from the archive and component names into the equal names, according to the Multics archive component pathname equal convention. Refer to "Archive Component Pathnames and Equal Names" in the MPM Reference Guide for a description of the convention.

Usage

```
declare get_equal_name_$component entry (char(*), char(*), char(*),
     char(*), char(32), char(32), fixed bin(35));

call get_equal_name_$component (entryname, component, equal_entryname,
     equal_component, target_entryname, target_component, code);
```

where:

1. entryname (Input)
   is the archive name from which the target archive name is constructed, or is the entryname from which the target component name is constructed if the source pathname is not an archive component pathname.

2. component (Input)
   is the component name from which the target component name is constructed or is a null string if the source pathname is not an archive component pathname.

3. equal_entryname (Input)
   is the equal name from which the target archive name is constructed or is the equal name from which the target entryname is constructed if the target pathname is not an archive component pathname.

4. equal_component (Input)
   is the equal name from which the target component name is constructed or is a null string if the target pathname is not an archive component pathname.

5. target_entryname (Output)
   is the target archive name that is constructed or is the target entryname that is constructed if the target pathname is not an archive component pathname.

6. target_component (Output)
   is the target component name that is constructed or is a null string if the target pathname is not an archive component pathname.

7. code (Output)
is a standard status code. It can be one of the following:
error_table_$bad_equal_name
either equal_entryname or equal_component has a bad format.
error_table_$badequal
there is no letter or component in the archive or component name that corresponds to a percent character (%) or an equal sign (=) in the appropriate equal name.
error_table_$longeql
the target archive or component name to be constructed is longer than 32 characters.
error_table_$no_archive_for_equal
the target pathname has an equal name in the archive name position but the source pathname is not an archive component pathname.

Notes

If the error_table_$badequal status code is returned, the name returned in the appropriate output argument is constructed using null character strings to represent the letters or component names missing from the source name.

If the error_table_$longeql status code is returned, the first 32 characters of the constructed name are returned in the appropriate output argument.

The two pairs of input arguments to this subroutine are expected to be the output arguments from two calls to expand_pathname_$component, one call for the source pathname and one for the pathname containing the equal names.

The output arguments of this subroutine should be used in a call to the initiate_file_$component subroutine documented in MPM Subroutines. For example:

```
call expand_pathname_$component (arg1, source_dir, source_ename,
          source_comp, code);
     if code ^= 0 then ...

call expand_pathname_$component (arg2, target_dir, equal_entry,
          equal_component, code);
     if code ^= 0 then ...

call get_equal_name_$component (source_ename, source_comp, equal_entry,
          equal_component, target_ename, target_comp, code);
     if code ^= 0 then ...

call initiate_file_$component (source_dir, source_ename, source_comp,
          R_ACCESS, source_ptr, source_bit_count, code);
     if code ^= 0 then ...

call initiate_file_$component (target_dir, target_ename, target_comp,
          R_ACCESS, target_ptr, target_bit_count, code);
     if code ^= 0 then ...
```

Name: get_external_variable_

The get_external_variable_ subroutine obtains the location and size of an external variable.

Usage

```
declare get_external_variable_ entry (char(*), ptr, fixed bin(19), ptr,
     fixed bin(35));

call get_external_variable_ (vname, vptr, vsize, vdesc_ptr, code);
```

where:

1. vname (Input)
   is the name of the external variable.

2. vptr (Output)
   is a pointer to the current allocation of the external variable.

3. vsize (Output)
   is the size (in words) of the external variable.

4. vdesc_ptr (Output)
   is a pointer to a standard argument descriptor array describing the external variable. If the external variable does not have descriptor information associated with it, a null pointer is returned.

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

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        bit(1),
      2 dir        bit(1),
      2 seg        bit(1),
      2 soos       bit(1),
      2 ring1      bit(1),
      2 rcp        bit(1),
      2 mbz        bit(30);
```

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. rcp

   indicates whether the AIM restrictions for accessing resources through RCP resource management are bypassed for the calling process.
   "1"b yes
   "0"b no

7. 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 6 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. 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: hash_index_

The hash_index_ function returns the value of a hash function of a character string.

Usage

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

hash_value = hash_index_ (string_ptr, string_len, mbz, table_size);
```

where:

1. string_ptr (Input)
   is a pointer to the character string to be hashed. This character string must be aligned.

2. string_len (Input)
   is the length of the character string.

3. mbz (Input)
   is reserved and must be zero.

4. table_size (Input)
   is the number of entries in the hash table.

Notes

The value returned is between zero and table_size-1, inclusive.

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 status, modify, and append modes. The remaining bits must be 0's. For example, status permission is expressed as "100"b. The access_mode_values.incl.pl1 include file defines mnemonics for these bit strings:

```
dcl    (S_ACCESS                    init ("100"b),
        M_ACCESS                    init ("010"b),
        A_ACCESS                    init ("001"b),
        SA_ACCESS                   init ("101"b),
        SM_ACCESS                   init ("110"b)
        SMA_ACCESS                  init ("111"b))
        bit(3) internal static options (constant);
```

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 read, execute, and write modes. The remaining bits must be 0's. For example, rw access is expressed as "101"b. The access_mode_values.incl.pl1 include file defines mnemonics for these values:

```
dcl   (N_ACCESS                   init ("000"b),
       R_ACCESS                   init ("100"b),
       E_ACCESS                   init ("010"b),
       W_ACCESS                   init ("001"b),
       RE_ACCESS                  init ("110"b)
       REW_ACCESS                 init ("111"b),
       RW_ACCESS                  init ("101"b))
       bit (3) internal static options (constant);
```

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 6 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_$force_write

The hcs_$force_write entry point causes the supervisor to force modified pages out of main memory.

Usage

```
declare hcs_$force_write entry (ptr, bit(36), fixed bin(35);

call hcs_$force_write (segp, flags, code);
```

where:

1. segp (Input)
   is a pointer to the segment whose modified pages are to be written.

2. flags (Input)
   specify a set of options. Currently, only one option is defined. The following structure (also defined in the system include file force_write_flags.incl.pl1) defines the options:

```
declare 1 force_write_options     based (addr (flags)) unaligned,
          2 mbz1                  bit(1),
          2 serial_write          bit(1),
          2 mbz2                  bit(34);
```

   serial_write:
   "0"b queue write requests for all modified pages in parallel, up to the maximum permitted by the supervisor's force-write limit (see shcs_$set_force_write_limit).
   "1"b queue write requests for all modified pages serially; one at a time.

   mbz1
   mbz2
   these fields must be zero.

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

Notes

Use of this entry point may introduce substantial real time delay into execution, since the caller must wait for the movement of the disk; other usage of the system, meanwhile, may cause further delay.

This entry point protects data against an unrecoverable main memory crash. On systems with bulk store paging devices, this subroutine may flush pages to the bulk store, which is recoverable in case of main memory crashes, rather than to the disk.

This entry point returns the following non-zero status codes. If the segment is an inner ring segment, error_table_$bad_ring_brackets is returned. If the user does not have write access to the segment, error_table_$moderr is returned. If the segment is not known, not active, or a hardcore segment, then error_table_$invalidsegno is returned. Because the user has no control over whether or not the segment is active, error_table_$invalidsegno should not be treated as an error.

THIS PAGE INTENTIONALLY LEFT BLANK

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 6 of the MPM Reference Guide.

Name: hcs_$get_exponent_control

This entry point returns the current settings of the flags that control the system's handling of exponent overflow and underflow conditions. For more information on exponent control see the description of hcs_$set_exponent_control.

Usage

```
declare hcs_$get_exponent_control entry (bit(1) aligned, bit(1) aligned,
     float bin(63));

call hcs_$get_exponent_control (restart_underflow, restart_overflow,
     overflow_value);
```

where:

1. restart_underflow (Output)
   is "1"b if underflows are currently being automatically restarted, and "0"b otherwise.

2. restart_overflow (Output)
   is "1"b if overflows are currently being automatically restarted, and "0"b otherwise.

3. overflow_value (Output)
   is the value used for the result of the computation in the case of overflow.

Name: hcs_$get_ips_mask

The hcs_$get_ips_mask entry point returns the value of the current ips mask.

Usage

```
declare hcs_$get_ips_mask entry (bit(36) aligned);

call hcs_$get_ips_mask (old_mask);
```

where:

1. old_mask (Output)
   is the current value of the ips mask.

Notes

A "1"b in any position in the mask means that the corresponding ips interrupt is enabled.

The thirty-sixth (rightmost) bit of old_mask does not correspond to an interrupt, but is used as a control bit, giving a positive indication that a particular masking or unmasking operation has taken place. No ips interrupts can occur in the time interval between the requested mask modification and the returning of the old_mask, with the control bit set appropriately.

Entry points used at the beginning of a critical section of code, to disable some or all ips interrupts, return a value of "1"b for the control bit, while those that are used at the end of a critical section of code, to re-enable those interrupts, return a value of "0"b for the control bit. Thus, a condition handler can interpret a value of "1"b in the control bit as meaning that execution was in a critical section of code, and the ips mask has been modified. See "Notes" in the description of the hcs_$set_automatic_ips_mask entry point for information about the state of the ips mask immediately after an ips interrupt occurs.

The control bit in the mask returned by this entry point is always "0"b.

Name: hcs_$get_link_target

The hcs_$get_link_target entry point returns the pathname of the ultimate target of a link if the ultimate target exists, or what that pathname would be if the target did exist.

Usage

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

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

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.

Notes

This entry chases the link to its ultimate target. The ultimate target of a link must be a directory or segment, which may or may not exist. If the immediate target of a link is another link, the chasing of links continues toward the ultimate target directory or segment until it is encountered or found to be nonexistent.

If the ultimate target of the link exists, the user must either have status permission on the directory containing the target or nonnull access to the target itself in order to determine its pathname. If appropriate access exists, the code is zero, and link_dir_name and link_entryname are set. If not, an error code is returned, and the link_dir_name and link_entryname are returned as blank.

If the ultimate target does not exist, the pathname of the last link encountered while chasing links will be returned if the user has status permission on the directoyr containing that final link. In this case, the returned code is error_table_$noentry, and the link_dir_name and link_entryname are set.

In all other cases, an error code is returned to indicate the lack of access, and link_dir_name and link_entryname are returned as blanks.

THIS PAGE INTENTIONALLY LEFT BLANK

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_process_usage

The hcs_$get_process_usage entry point returns information on system resource usage by the requesting process.

Usage

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

call hcs_$get_process_usage (process_usage_pointer, code);
```

where:

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

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

Notes

The following structure, declared in process_usage.incl.pl1, is pointed to by process_usage_pointer:

```
declare 1 process_usage       based (process_usage_pointer),
          2 number_wanted     fixed bin,
          2 number_can_return fixed bin,
          2 cpu_time          fixed bin (71),
          2 paging_measure    fixed bin (71),
          2 page_faults       fixed bin (34),
          2 pd_faults         fixed bin (34),
          2 virtual_cpu_time  fixed bin (71),
          2 segment_faults    fixed bin (34),
          2 bounds_faults     fixed bin (34),
          2 vtoc_reads        fixed bin (34),
          2 vtoc_writes       fixed bin (34);
```

where:

1. number_wanted
   specifies how much information is to be returned in the structure. It must be set prior to the call to hcs_$get_process_usage, and its interpretation is given below. It is the only input parameter in the structure; all other items are output from hcs_$get_process_usage or are ignored, depending on the value of number_wanted.

2. number_can_return
   is the number of system resource values which can be returned. It corresponds to the number of level 2 items in the structure following number_can_return. This is returned for all values of number_wanted.

3. cpu_time
   is the cumulative central processor time for the process. It includes all time spent executing instructions outside of ring 0, all time spent executing instructions in ring 0 as the result of explicit calls to ring 0, and all overhead time while executing instructions in the address space of this process (e.g., processing page faults for this process and interrupts where this process was interrupted). This is returned if number_wanted is 1 or greater.

4. paging_measure
   is the cumulative memory usage for the process in billable memory units. This is returned if number_wanted is 2 or greater.

5. page_faults
   is the cumulative number of page faults by the process. This number represents the number of times a page was referenced which was not in main memory. This is returned if number_wanted is 3 or greater.

6. pd_faults
   is the cumulative number of paging device faults by the process. This number will be nonzero only if a paging device configured at the site. The number represents the number of page faults where the page faulted was not on the paging device. This is returned if number_wanted is 4 or greater.

7. virtual_cpu_time
   is the cumulative virtual time for the process. This includes all time spent executing instructions outside of ring 0 and all time spent executing instructions in ring 0 as the result of explicit calls to ring 0. It does not include overhead time, such as the time spent processing page faults, segment faults, or interrupts. This is returned if number_wanted is 5 or greater.

8. segment_faults
   is the cumulative number of segment faults by the process. This represents the number of times a segment was referenced whose page table was not in main memory. This is returned if number_wanted is 6 or greater.

9. bounds_faults
   is the cumulative number of bounds faults by the process. This represents the number of times an address within a segment was referenced that was beyond the segment bound. This occurs most commonly when a segment expands to the point where it requires a larger page table. This is returned if number_wanted is 7 or greater.

10. vtoc_reads
    is the number of read I/Os done by the process to Volume Table of Contents Entries (VTOCEs). This is returned if number_wanted is 8 or greater.

11. vtoc_writes
    is the number of write I/Os done by the process to VTOCEs. This is returned if number_wanted is 9 or greater.

In the above description, cumulative activity by the requesting process is defined to mean all activity since login or since the most recent new_proc.

Name: hcs_$get_ring_brackets

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.

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_uid_seg

The hcs_$get_uid_seg entry point, when given a pointer to a segment, returns the unique identifier associated with the segment.

Usage

```
declare hcs_$get_uid_seg entry (ptr, bit (36) aligned, fixed bin (35));

call hcs_$get_uid_seg (seg_ptr, unique_id, code);
```

where:

1. seg_ptr (Input)
   is a pointer to the segment whose unique identifier is to be determined.

2. unique_id (Output)
   is the unique identifier associated with the segment.

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

This page intentionally left blank.

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 6 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, or -1. If the ring value is -1, a default value of the validation level of the calling process is used. This default should be used in all cases except those in which a different ring's access is explicitly required.

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 access_modes_values.incl.pl1 include file defines mnemonics for these values:

```
dcl (N_ACCESS_BIN      init (00000b),
     R_ACCESS_BIN      init (01000b),
     E_ACCESS_BIN      init (00100b),
     W_ACCESS_BIN      init (00010b),
     RW_ACCESS_BIN     init (01010b),
     RE_ACCESS_BIN     init (01100b),
     REW_ACCESS_BIN    init (01110b),

     S_ACCESS_BIN      init (01000b),
     M_ACCESS_BIN      init (00010b),
     A_ACCESS_BIN      init (00001b),
     SA_ACCESS_BIN     init (01001b),
     SM_ACCESS_BIN     init (01010b),
     SMA_ACCESS_BIN    init (01011b))
     fixed bin (5) internal static options (constant);
```

The user must have status permission on the containing directory, unless the access name supplied is that of the calling process or null.

Name: hcs_$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.

Name: hcs_$reset_ips_mask

The hcs_$reset_ips_mask entry point replaces the entire ips mask with a specified mask, and returns the previous value of the mask with a control bit of "0"b. It can be used at the end of a critical section of code to restore the mask to its former value. See "Notes" in the description of the hcs_$get_ips_mask entry point for a discussion of the control bit.

Usage

```
declare hcs_$reset_ips_mask entry (bit(36) aligned, bit(36) aligned);

call hcs_$reset_ips_mask (mask, old_mask);
```

where:

1. mask (Input)
   is the new ips mask, to replace the current one. A "1" bit in a mask position enables the corresponding ips interrupt.

2. old_mask (Output)
   is the former value of the ips mask, with a control bit of "0"b.

Notes

This entry point can be used at the end of a critical section of code to undo the mask changes made by the hcs_$set_ips_mask entry point. The old_mask returned by the latter entry point should be used as the value of the new_mask set by this entry point.

Name: hcs_$set_automatic_ips_mask

The hcs_$set_automatic_ips_mask entry point replaces the entire automatic ips mask with a supplied value, and returns the previous value of the automatic ips mask with a control bit of "1"b.

Usage

```
declare hcs_$set_automatic_ips_mask entry (bit(36) aligned, bit(36)
     aligned);

call hcs_$set_automatic_ips_mask (mask, old_mask);
```

where:

1. mask (Input)
   is the new value to replace the automatic ips mask.

2. old_mask (Output)
   is the former value of the automatic ips mask, with a control bit of "1"b.

Notes

The create_ips_mask_ subroutine (described in this manual) can be used to create a mask, given a set of ips names.

The automatic ips mask controls the state of the ips mask at the time that an ips signal handler is called. The interpretation of the bits in the automatic ips mask is quite different from that of the bits in the ips mask. When an ips interrupt occurs, if the bit corresponding to that interrupt is on in the automatic ips mask, then automatic ips masking takes place -- i.e., all ips interrupts are temporarily masked off, as described below. If the bit is off, then the ips mask is not changed.

If automatic ips masking is to take place for a given ips interrupt, then the current value of the ips mask is saved in the machine conditions, with its control bit on, and the ips mask is set to all zero bits, thus disabling all ips interrupts. This happens before the handler for the interrupt is called. When an ips interrupt handler returns, if the control bit in the saved ips mask is on, then the current ips mask is replaced by the saved one. It follows from this that the handler for an ips interrupt for which automatic ips masking is in effect can not make a permanent change to the ips mask unless it also modifies the machine conditions, turning off the control bit in the saved ips mask.

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 6 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_exponent_control

This entry point changes the current settings of the flags that control the system's handling of exponent overflow and underflow conditions. For more information on exponent control see "Notes".

Usage

```
declare hcs_$set_exponent_control entry (bit(1) aligned, bit(1) aligned,
     float bin(63), fixed bin (35));

call hcs_$set_exponent_control (restart_underflow, restart_overflow,
     overflow_value, code);
```

where:

1. restart_underflow (Input)
   is "1"b if underflows should be automatically restarted, and "0"b otherwise.

2. restart_overflow (Input)
   is "1"b if overflows should be automatically restarted, and "0"b otherwise.

3. overflow_value (Input)
   is the value used for the result of the computation in the case of overflow.

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

Notes

When either of the two flags are set to zero, the corresponding error condition causes the appropriate fault condition to be signalled. If a flag is set to one, then the computation resulting in the error is automatically restarted. In the case of underflow its result is set to zero. In the case of positive overflow, its value is set to the value specified in overflow_value. In the case of negative overflow, the negative of overflow_value is used. The default value is the largest representable positive number, available as Default_exponent_control_overflow_value in the include file exponent_control.incl.pl1.

This subroutine affects only the system's handling of exponent overflow and underflow when the overflow condition or the underflow condition is raised. In certain cases, the error condition is raised instead; this subroutine does not affect the system's handling of such cases.

In programs not written in PL/I, the exponent_control_ subroutine, described in MPM Subroutines, should be used in place of hcs_$set_exponent_control.

Name: hcs_$set_ips_mask

The hcs_$set_ips_mask entry point replaces the entire ips mask with a supplied value, and returns the previous value of the mask with a control bit of "1"b. It can be used at the beginning of a critical section of code, to disable one or more ips interrupts, and turn on the control bit to indicate that some interrupts are disabled. See "Notes" in the description of the hcs_$get_ips_mask entry point for a discussion of the control bit.

Usage

```
declare hcs_$set_ips_mask entry (bit(36) aligned, bit(36) aligned);

call hcs_$set_ips_mask (mask, old_mask);
```

where:

1. mask (Input)
   is the new value to replace the ips mask. A "1" bit in each mask position enables the corresponding ips interrupt.

2. old_mask (Output)
   is the former value of the ips mask, with a control bit of "1"b.

Notes

The create_ips_mask_ subroutine (described in this manual) can be used to create a mask, given a set of ips names.

The hcs_$reset_ips_mask entry point (described in this manual) can be used at the end of a critical section of code to undo the mask changes made by this entry point, by setting the mask to the old_mask value returned by this entry point.

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 6 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 2 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 2 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 3 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, star_select_sw, area_ptr,
     star_entry_count, star_entry_ptr, star_names_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. star_select_sw (Input)
   indicates what information is to be returned. It can be:

   star_LINKS_ONLY (=1)
   information is returned about link entries only

   star_BRANCHES_ONLY (=2)
   information is returned about segment and directory entries only

   star_ALL_ENTRIES (=3)
   information is returned about segment, directory, and link entries.

4. area_ptr (Input)
   is a pointer to the area in which information is to be returned. If the pointer is null, star_entry_count is set to the total number of selected entries. See "Notes" below.

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

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

7. star_names_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, star_entry_count is set to the total number of entries in the directory that match star_name. The setting of star_select_sw determines whether star_entry_count is the total number of link entries, the total number of segment and directory entries, or the total number of all entries.

If area_ptr is not null, the entry information structure and the name array are allocated in the user-supplied area.

This data structure is declared in star_structures.incl.pl1. The entry information structure is as follows:

```
declare 1 star_entries (star_entry_count)   aligned based (star_entry_ptr),
          2 type                         fixed binary (2) unsigned unaligned,
          2 nnames                       fixed binary (16) unsigned unaligned,
          2 nindex                       fixed binary (18) unsigned unaligned;
```

where:

1. type
   specifies the storage system type of entry (the following named constants are declared in star_structures.incl.pl1):
   star_LINK (0)
   star_SEGMENT (1)
   star_DIRECTORY (2)

2. nnames
   specifies the number of names for this entry that match star_name.

3. nindex
   specifies the offset in star_names of 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 and declared in star_structures.incl.pl1, is as follows:

```
declare star_names (sum (star_entries (*).nnames)) char(32)
     based (star_names_ptr);
```

The user must provide an area large enough for the hcs_$star_ entry point to store the requested information.

Status Codes

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

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

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

Using the include file

A program using star_structures.incl.pl1 should declare addr, binary, and sum to be builtin. The arguments star_entry_count, star_entry_ptr, and star_names_ptr are declared in the include file along with named constants for the value of star_select_sw and the storage system type. One of the named constants for star_select_sw can be passed as an argument to hcs_$star_ along with star_entry_count, star_entry_ptr and star_names_ptr.

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, star_select_sw,
     area_ptr, star_branch_count, star_link_count, star_list_branch__ptr,
     star_list_names_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. star_select_sw (Input)
   indicates what information is to be returned. It can be:

   star_LINKS_ONLY (=1)
   information is returned about link entries only

   star_BRANCHES_ONLY (=2)
   information is returned about segment and directory entries only

   star_ALL_ENTRIES (=3)
   information is returned about segment, directory, and link entries

   star_LINKS_ONLY_WITH_LINK_PATHS (=5)
   information is returned about link entries only, including the pathname associated with each link entry

   star_ALL_ENTRIES_WITH_LINK_PATHS (=7)
   information is returned about segment, directory, and link entries, including the pathname associated with each link entry

4. area_ptr (Input)
   is a pointer to the area in which information is to be returned. If the pointer is null, star_branch_count and star_link_count are set to the total number of selected entries. See "Notes" below.

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

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

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

8. star_list_names_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

The names star_LINKS_ONLY through STAR_ALL_ENTRIES_WITH_LINK_PATHS are declared in star_structures.incl.pl1.

Even if area_ptr is null, star_branch_count and star_link_count may be set. If information on segments and directories is requested, star_branch_count is set to the total number of segments and directories that match star_name. If information on links is requested, star_link_count is the total number of links that match star_name.

If area_ptr is not null, an array of entry information structures and the names array, as described in the hcs_$star_ entry point above, are allocated in the user-supplied area. Each element in the structure array may be either of the structures described below (the star_links structure for links or the star_list_branch structure for segments and directories). The correct structure is indicated by the type item, the first item in both structures.

If the system is unable to access the VTOC entry for a branch, values of zero are returned for records used, date_time_contents_modified, and date_time_used, and no error code is returned. Callers of this entry point should interpret zeros for all three of these values as an error indication, rather than as valid data.

The first three items in each structure are identical to the ones in the structure returned by the hcs_$star_ entry point.

The following structure, declared in star_structures.incl.pl1, is used if the entry is a segment or a directory:

```
declare 1 star_list_branch          (star_branch_count + star_link_count)
                                    aligned based (star_list_branch_ptr),
          2 type                    fixed binary(2) unsigned unaligned,
          2 nnames                  fixed binary(16) unsigned unaligned,
          2 nindex                  fixed binary(18) unsigned unaligned,
          2 dtcm                    bit(36) unaligned,
          2 dtu                     bit(36) unaligned,
          2 mode                    bit(5) unaligned,
          2 raw_mode                bit(5) unaligned,
          2 master_dir              bit(1) unaligned,
          2 pad                     bit(7) unaligned,
          2 records                 fixed bin(18) unsigned unaligned;
```

where:

1. type

   specifies the storage system type of entry:

   star_LINK (=0)
      link

   star_SEGMENT (=1)
      segment

   star_DIRECTORY (=2)
      directory

2. nnames
   specifies the number of names for this entry that match star_name.

3. nindex
   specifies the offset in star_list_names of 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. pad
   is unused space in the structure.

10. records
   is the number of 1024-word records of secondary storage that have been assigned to the segment or directory.

The following structure, declared in star_structures.incl.pl1, is used if the entry is a link:

```
declare 1 star_links            (star_branch_count + star_link_count)
                                aligned based (star_list_branch_ptr),
          2 type                fixed binary(2) unsigned unaligned,
          2 nnames              fixed binary(16) unsigned unaligned,
          2 nindex              fixed binary(18) unsigned unaligned,
          2 dtem                bit(36) unaligned,
          2 dtd                 bit(36) unaligned,
          2 pathname_len        fixed binary(18) unsigned unaligned,
          2 pathname_index      fixed binary(18) unsigned unaligned;
```

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 star_list_names of the link pathname.

If the pathname associated with each link was requested, the pathname is placed in the names array and occupies 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.

The following structure is the array of names. It is declared in star_structures.incl.pl1.

```
declare star_list_names (sum (star_links (*).nnames) +
     binary (star_select_sw >= star_LINKS_ONLY_WITH_LINK_PATHS,
     1) * 6 * star_link_count) char (32) based (star_list_names_ptr);
```

The following based variable is used to get the pathname associated with link star_linkx in the star_links array. It is declared in star_structures.incl.pl1.

```
declare star_link_pathname char (star_links (star_linkx).pathname_len)
     based (addr (star_list_names (star_links
     (star_linkx).pathname_index)));
```

Using the Include File

A program using star_structures.incl.pl1 should declare addr, binary and sum to be builtin. The star_branch_count, star_entry_ptr, star_link_count, star_linkx, star_list_names_ptr and star_select_sw variables are declared in the include file along with named constants for the value of star_select_sw and the storage system type.

To use the structures in the include file, first assign to star_select_sw the proper named constant and then pass star_select_sw as an argument to hcs_$star_list_ along with star_branch_count, star_link_count, star_list_branch_ptr, and star_list_names_ptr.

To get the link pathname associated with a link, assign to star_linkx the index of the link in star_links. Star_link_pathname will then be link_pathname.

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, star_select_sw, area_ptr,
     star_branch_count, star_link_count, star_list_branch_ptr,
     star_list_names_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.

Use the following structure if the entry is a segment or a directory. The star_dir_list_branch structure is the same as the star_list_branch structure except for the dtem and bit-count fields. This structure is declared in star_structures.incl.pl1.

```
declare 1 star_dir_list_branch        (star_branch_count + star_link_count)
                                      aligned based (star_list_branch_ptr),
          2 type                      fixed binary(2) unsigned unaligned,
          2 nnames                    fixed binary (16) unsigned unaligned,
          2 nindex                    fixed binary (18) unsigned unaligned,
          2 dtem                      bit(36) unaligned,
          2 pad                       bit(36) unaligned,
          2 mode                      bit(5) unaligned,
          2 raw_mode                  bit(5) unaligned,
          2 master_dir                bit(1) unaligned,
          2 bit_count                 fixed binary(24) unaligned;
```

where:

1. type
   specifies the storage system type of entry:

   star_LINK (=0)
      link

   star_SEGMENT (=1)
      segment

   star_DIRECTORY (=2)
      directory

2. nnames
   specifies the number of names for this entry that match star_name.

3. nindex
   specifies the offset in star_list_names of 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. pad
   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 star_links structure described for hcs_$star_list is used if the entry is a link.

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) aligned, 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.

*

Name: help_

The help_ subroutine performs the basic work of the help command (described in the MPM Commands). The help_ subroutine is called to print selected information from one or more info segments. The caller may select: what information is to be printed; what search list is to be used to find the info segments; what suffix the info segments must have. Thus, the help_ provides an interface for implementing a subsystem help command.

Several entry points in the help_ subroutine are described below. help_$init must be called before calling the help_ or help_$check_info_segs entry points. The help_ or help_$check_info_segs entry points may then be called one or more times. When the caller no longer needs the help_args structure, help_$term must be called to release the temporary segment containing the help_args structure.

Entry: help_$init

This entry point obtains a pointer to the help_args structure (see "Notes" below). This structure is used to pass information from the caller to the help_ entry point (described below). The structure is a based structure containing several arrays with adjustable extents. The help_$init entry point creates the structure in a temporary segment so that these arrays can be grown incrementally by the caller as information is added to the structure.

The help_ subroutine selects and prints info segments based upon the information given in the help_args structure. It also uses space in the temporary segment following the help_args structure for a work area. For this reason, space for help_args must be obtained by calling the help_$init entry point.

The help_$init entry point obtains the paths defined in a search list named by the caller. It stores these paths in the help_args structure for use by the help_ subroutine. Several other help_args elements are set, as described under "Notes" below.

Usage

```
declare help_$init entry (char(*), char(*), char(*), fixed bin, ptr,
     fixed bin(35));

call help_$init (caller, search_list_name, search_list_ref_dir,
     required_version, Phelp_args, code);
```

where:

1. caller (Input)
   is the name of the calling program, on whose behalf the temporary segment containing the help_args structure is obtained.

2. search_list_name (Input)
   is the name of the search list to be used in searching for info segments. A null string may be given if no search list is to be used.

3. search_list_ref_dir (Input)
   is the pathname of the directory to be used when expanding the referencing_dir search rule in the search list. If a null string is given, the referencing_dir search rule is omitted from the search list.

4. required_version (Input)
   is the version number of the help_args structure which the caller is prepared to accept. This argument should be set to the value of the Vhelp_args_1 constant, described under "Notes" below.

5. Phelp_args (Output)
   is a pointer to the help_args structure, described under "Notes" below.

6. code (Output)
   is a standard status code reporting any failure in obtain expanding the search list.

Entry: help_

This entry point searches for info segments, selects information blocks (infos), and prints the information. The caller provides information in the help_args structure (obtained in the call to help_$init) to select the infos to be printed and the type of information to be printed.

The help_ subroutine may ask the user questions about how much information should be printed. These questions and the responses the user may give are in the description of the help command in the MPM Commands. Questions are asked using the command_query_ subroutine, described elsewhere in this manual.

Usage

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

call help_ (caller, Phelp_args, suffix, progress, code);
```

where:

1. caller (Input)
   is as above.

2. Phelp_args (Input)
   is as above.

3. suffix (Input)
is the suffix which must appear in the entrynames of info segments to be processed by this invocation of help_. This suffix is also assumed when omitted from the (final or only) entryname of values given for help_args.path.value in the help_args structure (see "Notes" below). If a null string is given, then no suffix is required in info segment entrynames, and none is assumed in values of help_args.path.value.

4. progress (Output)
is a special status code that indicates which stage of processing help_ was performing when an error occurs. The following values may be returned:

1
the Phelp_args argument points to an unimplemented version of the help_args structure.

2
help_args.Npaths is not positive, indicating that no info_names were given. help_ is unable to select info segments for printing, and reports the error.

3
an error is encountered while evaluating one or more of the help_args.path.value values. help_args.path.code indicates the particular error encountered in each value.

4
no fatal errors are encountered. Some infos matching help_args.path were found. Any nonfatal errors encountered while finding the infos are diagnosed to the user. A list of infos to be compared with the -section and -search criteria is created.

5
infos matching the -section and -search criteria are printed. A nonzero code argument is returned only when no infos match the -section and -search criteria. help_ does not report such an error to the user. The caller is responsible for doing this.

5. code (Output)
is a standard status code. When progress is 1, the code may have the following value:

error_table_$unimplemented_version
help_ does not support the version of the help_args structure pointed to by the Phelp_args pointer argument.

When progress is 2, the code may have the following value:

error_table_$noarg
help_args.Npaths was not positive.

When progress is 3, the code may have any value returned by expand_pathname_$add_suffix or check_star_name_$entry, or it may have the following value:

error_table_$inconsistent
a star name was given when help_args.Sctl.ep = "1"b, or when a value of help_args.path.value contains a subroutine entry point name.

When progress is 4, the code may have the following value:

error_table_$nomatch
No info segments match any of the help_args.path elements. For each help_args.path.value element, help_ prints an error message when no matching info segments are found.

When progress is 5, the code may have the following value:

error_table_$nomatch
None of the infos selected by help_args.path contain sections whose titles match the selection criteria given in help_args.scn, or paragraphs that match the selection criteria given in help_args.srh. help_ does not report this error to the user. The caller of help_ must do this.

Notes

The Phelp_args argument points to the following structure, which is declared in help_args_.incl.pl1:

```
dcl  1 help_args aligned based (Phelp_args),
       2 version                    fixed bin,
       2 Sctl,
        (3 he_only,
         3 he_pn,
         3 he_info_name,
         3 he_counts,
         3 title,
         3 scn,
         3 srh,
         3 bf,
         3 ca,
         3 ep,
         3 all)                     bit(1) unal,
         3 pad1                     bit(25) unal,
       2 Nsearch_dirs               fixed bin,
       2 Npaths                     fixed bin,
       2 Ncas                       fixed bin,
       2 Nscns                      fixed bin,
       2 Nsrhs                      fixed bin,
       2 min_Lpgh                   fixed bin,
       2 max_Lpgh                   fixed bin,
       2 Lspace_between_infos       fixed bin,
       2 min_date_time              fixed bin(71),
       2 pad2 (10)                  fixed bin,
       2 search_dirs (0 refer (help_args.Nsearch_dirs))
                                    char (168) unal,
       2 path (0 refer (help_args.Npaths)),
         3 value                    char(425) varying,
         3 info_name                char(32) unal,
         3 dir (1)                  char(168) unal,
         3 ent                      char(32) unal,
         3 ep                       char(32) varying,
         3 code                     fixed bin(35),
         3 S,
          (4 pn_ctl_arg,
           4 info_name_not_starname,
           4 less_greater,
           4 starname_ent,
           4 starname_info_name,
           4 separate_info_name)    bit(1) unal,
           4 pad3                   bit(30) unal,
       2 ca (0 refer (help_args.Ncas))
                                    char(32) varying,
       2 scn (0 refer (help_args.Nscns))
                                    char(80) varying,
       2 srh (0 refer (help_args.Nsrhs))
                                    char(80) varying,
     Phelp_args                     ptr,
     Vhelp_args_1                   fixed bin int static
                                         options(constant) init(1);
```

where:

1. version
   is the version number of this structure (currently 1). The variable Vhelp_args_1 (see 45 below) should be used when checking this version number.

2. Sctl
   are flags controlling the operations which help_ performs on the info segments. help_$init sets all of these flags to "0"b.

3. Sctl.he_only
   help_ prints only a heading line identifying matching info segments. The heading line includes the info heading, plus heading fields selected by Sctl.he_pn, Sctl.he_info_name and Sctl.he_counts. No other information is printed. This flag is mutually exclusive with all other Sctl flags except those named above, Sctl.scn and Sctl.srh.

4. Sctl.he_pn
   help_ includes the info pathname in all heading lines. help_ prints other information along with the heading line, as requested by the other Sctl flags. If no other flags are set, help_ prints the heading line followed by the first paragraph of information.

5. Sctl.he_info_name
   help_ includes the info_name in all heading lines. This info_name is included only when help_args.path identifies an info segment containing more than one information block (info). (See 28 below for more information about info_names.) help_ prints other information along with the heading line, as requested by other Sctl flags. If no other flags are set, help_ prints the heading line followed by the first paragraph of information.

6. Sctl.he_counts
   help_ includes info line counts and subroutine info entry point counts in all heading lines. help_ prints other information along with the heading line, as requested by other Sctl flags. If no other flags are set, help_ prints the heading line followed by the first paragraph of information.

7. Sctl.title
   help_ prints all section titles (including section line counts), then asks if the user wants to see the first paragraph. Normally, help_ just begins printing the first paragraph.

8. Sctl.scn
   help_ searches section titles for one containing all of the substrings given in help_args.scn (see 42 below). If a matching title is found, help_ begins printing information requested by other Sctl flags. If no other flags are set, help_ prints the first paragraph of the matching section. If no matching title is found, help_ skips the info without comment.

9. Sctl.srh

help_ searches all paragraphs for one containing all of the substrings given in help_args.srh (see 43 below). If a matching paragraph is found, help_ begins printing information requested by other Sctl flags. If no other flags are set, help_ prints the matching paragraph. If no matching paragraph is found, help_ skips the info without comment. If Sctl.scn is also "1"b, then only paragraphs from the matching section to the end of the info are searched.

10. Sctl.bf

help_ prints only a brief summary of an info describing a command, active function, or subroutine. This flag is mutually exclusive with all other Sctl flags except Sctl.he_pn, Sctl.he_info_name, Sctl.he_counts, Sctl.ca, Sctl.scn and Sctl.srh.

11. Sctl.ca

for an info describing a command, active function, or subroutine, help_ prints only the descriptions of one or more arguments or control arguments identified by the substrings in help_args.ca (see 41 below). This flag is mutually exclusive with all other Sctl flags except Sctl.he_pn, Sctl.he_info_name, Sctl.he_counts, Sctl.bf, Sctl.scn and Sctl.srh.

12. Sctl.ep

help_ prints information describing the main entry point of a subroutine, rather than information describing the general characteristics of all subroutine entry points.

13. Sctl.all

help_ prints all of the info without asking the user any questions.

14. Sctl.pad1

is reserved for future use. help_$init sets this field to ""b.

15. Nsearch_dirs

is the number of directories help_ searches for info segments. The directory pathname are given in help_args.search_dirs (see 25 below). This number is set by help_$init to the number of paths in the search list named in the call to help_$init, but the caller may change it before calling help_.

16. Npaths

is the number of info names help_ searches for. The names are given in help_args.path (see 26 below). The caller must set this number before calling help_. help_$init initializes it to zero.

17. Ncas

is the number of substrings help_ uses in searching for argument or control argument descriptions when help_args.Sctl.ca is given. The substrings are given in help_args.ca (see 41 below). help_$init initializes this number to zero.

18. Nscns

is the number of substrings help_ uses in searching for a matching section title when help_args.Sctl.scn is given. The substrings are given in help_args.scn (see 42 below). help_$init initializes this number to zero.

19. Nsrhs
is the number of substrings help_ uses in searching for a matching paragraph when help_args.Sctl.srh is given. The substrings are given in help_args.srh (see 43 below). help_$init initializes this number to zero.

20. min_Lpgh
is the length (in lines) of the shortest paragraph that help_ will consider as a distinct unit. Paragraphs shorter than this may be printed with their preceding paragraph, rather than asking the user if he wants to see the short paragraph. help_$init initializes this number to 4.

21. max_Lpgh
is the maximum number of lines of information that help_ allows in grouper paragraphs before asking the user whether he wants to see more. help_ will never group short paragraphs with their preceding paragraph if the total number of lines to be printed (including 2 blank lines between paragraphs) would exceed this number. help_$init initializes this number to 15.

22. Lspace_between_infos
is the number of blank lines which help_ prints between the last paragraph of one info and the heading line (or first paragraph) of the next. help_$init initializes this number to 2.

23. min_date_time
is a Multics clock value. Only infos modified on or after the time given in this clock value are selected. Info modification time is based upon the date_time_entry_modified of the segment containing the info. When an info segment contains more than one info, any date given in the info heading is used as the modification date for that info. help_$init initializes this number to -1, indicating that all infos are eligible for selection.

24. pad2
is reserved for future use. This field should not be set or referenced.

25. search_dirs
is an array of absolute pathnames specifying directories that help_ will look in for named infos. help_ searches for an info unless help_args.path.value (see 27 below) contains less-than (<) or greater-than (>) characters, or unless help_args.path.S.pn_ctl_arg = "1"b (see 34 below). help_$init sets this array to the pathnames given for the search list named by its search_list_name argument. The caller can change this list before calling help_. Note that the search_dirs are absolute pathnames which are expanded from the rules in a search list. If the working directory may have changed between calls to help_, then the search list rules must be reevaluated before each call to help_. This can be accomplished by calling help_$init before each call to help_, and help_$term after each call.

26. path
is an array of minor structures that identify the infos to be printed.

27. path.value
is a value used to select one or more info segments. A relative or absolute pathname may be given, or just an entryname. The (final or only) entryname may be a starname. A subroutine entry point name may follow the entryname. For example

ioa_$rsnnl

or

my_info_dir>extend_subr$init

A starname may not be given with a subroutine entry point name or when Sctl.ep = "1"b (see 12 above). A proper suffix (as defined by the suffix argument to the help_ entry point) is assumed if not given. If path.value contains a less-than (<) or greater-than (>) character, it is assumed to be the pathname of an info to be printed. Otherwise, path.value is assumed to be the entryname of an info which is searched for in directories named in the search_dirs array (see 25 above). Note that path.value has a maximum length of 425 characters to accommodate a maximum size pathname (168 characters), a maximum size entry point name (256 characters), plus a dollar sign ($) separator.

28. path.info_name
selects an info within the info segments found by path.value. Normally, the caller of help_ sets the info_name to a null string, causing help_ to use the (final or only) entryname from path.value (without its suffix) as the info_name. help_ then searches for an info segment having the info_name (with an appropriate suffix) as one of its segment names. help_ looks inside the segment to see if it is divided into different information blocks (infos). Lines of the form

:Info: info_name1: ...info_nameN: date info_heading

divide the segment into infos. For each info segment containing multiple infos, help_ searches for infos having an info_namei matching the info_name and prints only those infos.

When the caller of help_ gives a nonnull value for path.info_name, then the info_name need not be a name on the info segment itself. This is sometimes useful for subsystems which want to store all of their infos in a single info segment (to reduce storage costs, simplify maintenance of the infos or facilitate printing all of the information), but which do not want to add all of the info_names to the segment. This avoids the need for many names on the segment, and also prevents the system help command from accessing the infos whose names do not appear on the info segment. The star convention may be used in the path.info_name. Note that the info_namei given in a :Info: line of an info segment correspond to names on the info segment when a null path.info_name is given. However, when a nonnull path.info_name is given, the info_namei need not be unique within the info segment. help_ selects all infos having a matching info_namei in the order in which they appear in the info segment, even when path.info_name is not a star name. If path.info_name is set to a nonnull value, the pathS.info_name_not_starname must also be set (see 35 below).

29. path.dir
is the directory part of a pathname given as the value of path.value. help_ sets this value, and the caller of help_ need not

set this value. The variable is a one-dimensional array so that it can be used interchangeably with the search_dirs array (see 25 above) in searching for info segments.

30. path.ent
    is the entryname part of a pathname given as the value of path.value. help_ sets this value, and the caller of help_ need not set this value.

31. path.ep
    is the entry point name part of a name given in path.value. help_ sets this value, and the caller of help_ need not set this value.

32. path.code
    is a standard status code associated with processing the value given in path.value. When help_ returns to its caller with a progress argument value of 3 and a nonzero status code argument, the caller of help_ should: examine each path.code; for nonzero values, report an error in path.value. path.code may have any of the values listed above for the code argument returned by help_ when the progress argument is 3.

33. path.S
    are flags controlling the interpretation of path.value (see 27 above).

34. path.S.pn_ctl_arg
    is "1"b if path.value is to be interpreted as a relative or absolute pathname, rather than as an entryname which should be searched for using the search_dirs (see 27 above). If the flag is "0"b, then help_ interprets path.value as a pathname only if it contains a less-than (<) or greater-than (<) character. The caller of help_ must set this flag to the appropriate value.

35. path.S.info_name_not_starname
    is "1"b if path.info_name is not a star name, even though it may contain * or ? characters. A value of "0"b causes path.info_name to be treated as a star name if it contains * or ? characters. If the caller sets path.info_name to a nonnull value (see 28 above), then this switch must be set.

36. path.S.less_greater
    is a flag that help_ uses to record that path.value contains less-than (<) or greater-than (>) characters, or that path.S.pn_ctl_arg was set. The caller of help_ need not set this flag.

37. path.S.starname_ent
    is a flag that help_ uses to record the fact that the (final or only) entryname in path.value is a star name. The caller of help_ need not set this value.

38. path.S.starname_info_name
    is a flag that help_ uses to record that path.info_name is a star name. The caller of help_ need not set this flag.

39. path.S.separate_info_name
    is a flag that help_ uses to record that path.info_name was supplied by the caller of help_, rather than being extracted from path.value by help_. The caller of help_ need not set this flag.

40. path.S.pad3
is a reserved field. The caller of help_ must set this field to zeros.

41. ca
is the array of substrings help_ uses in searching for argument or control argument descriptions when help_args.Sctl.ca is given. If any of these strings appears in the argument name line of an argument or control argument description, then help_ prints the entire description.

42. scn
is the array of substrings help_ uses in searching for a matching section title when help_args.Sctl.scn is given. All of these substrings must appear (in any order) in a matching section title. Comparisons are made after all substrings are translated to lowercase, so the letter case of the substrings does not matter.

43. srh
is the array of substrings help_ uses in searching for a matching paragraph when help_args.Sctl.srh is given. All of the substrings must appear (in any order) in a matching paragraph. Comparisons are made after all substrings are translated to lowercase, so the letter case of substrings does not matter.

44. Phelp_args
is a pointer to the help_args structure. help_$init returns a value for this pointer argument. help_, help_$check_info_segs and help_$term require the pointer as an input argument.

45. Vhelp_args_1
is a named constant which the caller of help_$init should use for the required_version argument. This constant can also be used to check the value of help_args.version.

The structure above is somewhat complex, due to the many options provided by the help_ subroutine. Callers of help_ or help_$check_info_segs can use the following steps to set structure elements:

1. Set the Sctl flags to the required values. Set min_Lpgh, max_Lpgh, Lspace_between_infos, and min_date_time values if you wish to change the defaults supplied by help_$init.

2. If any of the search_dirs are to be set (or changed from the pathnames given in the search list named in the call to help_$init), then set Nsearch_dirs to the correct value, and set the search_dir array elements to the desired values.

3. Set Npaths to the number of info pathname/info_name input values. Set the elements of help_args.path for each of these input values. If the values are arguments in a subsystem help request, they can be placed in the help_args.path structure as each argument is processed. In this case, add 1 to Npaths as each argument is processed, then set help_args.path(Npaths) to the appropriate input values.

4. Provide substrings used in searching for argument or control argument descriptions, if any. Set Ncas to the appropriate value, then store the substrings in the ca array.

5. Provide substrings used in searching for section titles, if any. Set Nscns to the appropriate value, then store the substrings in the scn array.

6. Provide substrings used in searching for matching paragraphs, if any. Set Nsrhs to the appropriate value, then store the substrings in the srh array.

Note that when substrings for argument and control argument matching, section title matching, or paragraph matching are not provided, Ncas, Nscns, or Nsrhs above need not be set. help_$init initializes these values to zero.

Entry: help_$check_info_segs

This entry point searches for info segments modified since a given date. It returns a sorted list of info segments matching the selection criteria. The list is sorted by directory name, and within a directory by entryname. In addition, the help_$check_info_segs entry point flags entrynames found in more than one directory. All but the first such duplicate segment are marked with a cross reference flag and are sorted after all unique info segments. The caller provides the selection criteria in the help_args structure, obtained by calling help_$init. In particular, help_args.min_date_time specifies the info segment modification threshold (see 23 in the "Notes" above).

Usage

```
declare help_$check_info_segs entry (char(*), ptr, char(*), fixed bin,
     fixed bin(35), ptr);

call help_$check_info_segs (caller, Phelp_args, suffix, progress, code,
     PPDinfo_seg);
```

where:

1. caller (Input)
   is as described above for the help_ entry point.

2. Phelps_args (Input)
   is as described above for the help_ entry point.

3. suffix (Input)
   is as described above for the help_ entry point.

4. progress (Output)
   is as described above for the help_ entry point.

5. code (Output)
   is as described above for the help_ entry point.

6. PPDinfo_seg (Output)
   points to the PDinfo_seg structure, described under "Notes" below. This structure contains a sorted list of pointers to descriptors for the selected info segments.

Notes

The PPDinfo_seg argument points to the PDinfo_seg structure that follows. This structure is declared in help_cis_args_.incl.pl1. All structure values are set by help_$check_info_segs.

```
     dcl  1 PDinfo_seg                     aligned based(PPDinfo_seg),
            2 version                      fixed bin,
            2 N                            fixed bin(24),
            2 P (0 refer (PDinfo_seg.N))
                                           ptr unal,
          PPDinfo_seg                      ptr,
          VPDinfo_seg_1                    fixed bin int static
                                                options(constant) init(1);
```

Each pointer PDinfo_seg.P points to the following info segment descriptor structure, which is also declared in help_cis_args_.incl.pl1.

```
     dcl  1 Dinfo_seg                      aligned based,
            2 Scross_ref                   bit(36) aligned,
            2 dir                          char(168) unal,
            2 ent                          char(32) unal,
            2 info_name                    char(32) unal,
            2 ep                           char(32) var,
            2 uid                          bit(36),
            2 I                            fixed bin(21),
            2 L                            fixed bin,
            2 date                         fixed bin(71),
           (2 segment_type                 bit(2),
            2 mode                         bit(3),
            2 pad1                         bit(31)) unal,
            2 code                         fixed bin(35);
```

where:

1. version
   is the version number of the PDinfo_seg and Dinfo_seg structures (currently 1). The variable VPDinfo_seg_1 (see 5 below) should be used when checking this version number.

2. N
   is the number of info segments found.

3. P
   is the array of pointers to the Dinfo_seg structures which describe the info segments found by the selection criteria.

4. PPDinfo_seg
   is a pointer to the PDinfo_seg structure.

5. VPDinfo_seg_1
   is a named constant which the caller of help_$check_info_segs should use when testing the value of PDinfo_seg.version.

6. Dinfo_seg
   is the structure which describes each info segment found by the selection criteria.

7. cross_ref
   is an info segment crossreference flag. If the flag equals "1"b, then several info segments were found having the same entryname but residing in different directories, and the info segment identified by this structure was not the first such duplicate.

8. dir
   is the directory part of the pathname of the info segment.

9. ent
   is the final entryname part of the pathname of the info segment.

10. info_name
    is reserved for use by help_, and is always a null character string.

11. ep
    is the subroutine entry point name given in the selection criteria for the info segment.

12. uid
    is reserved for use by help_, and is always 0.

13. I
    is reserved for use by help_, and is always 0.

14. L
    is the length (in characters) of the info segment.

15. date
    is the date_time_entry_modified of the info segment.

16. segment_type
    is the type of storage system entry identified by Dinfo_seg.dir and Dinfo_seg.ent. It may have one of the following values:
    "00"b link
    "01"b segment

17. mode
    is the user's access mode to the info segment. The three bits correspond to read, execute and write access mode. For example, rw access is expressed as "101"b.

18. pad1
    is reserved for future use.

19. code
    is a standard status code encountered while processing this info segment. It may have any of the following values:

    error_table_$noentry
    Dinfo_seg.dir and Dinfo_seg.ent identify a link whose target does not exist.

    error_table_$zero_length_seg
    the info segment is empty.

    error_table_$bad_syntax
    the info segment has a bit count which is not evenly divisible by 9. Therefore, the info segment does not contain a whole number of characters.

Entry: help_$term

This entry point releases the temporary segment in which the help_args structure (and the PDinfo_seg and Dinfo_seg structures of help_$check_info_segs) are created. This entry point should be called before calling help_$init again.

Usage

```
declare help_$term entry (char(*), ptr, fixed bin(35));

call help_$term (caller, Phelp_args, code);
```

where the arguments are as described above for the help_ entry point.

Name: interpret_resource_desc_

The interpret_resource_desc_ subroutine provides a facility for displaying the contents of an RCP resource description, in a format similar to that used by the resource_status command.

Usage

```
declare interpret_resource_desc_ entry (pointer, fixed bin, char (*),
     bit (36) aligned, bit (1) aligned, char (*) varying, fixed bin (35));

call interpret_resource_desc_ (resource_desc_ptr, nth, callername,
     string (rst_control), return_noprint, return_string, code);
```

where:

1. resource_desc_ptr (Input)
   is a pointer to the structure containing the RCP resource description to be displayed. (See the resource_control_ subroutine.)

2. nth (Input)
   specifies which element of the resource description is to be displayed (the index to the array resource_descriptions.item). If nth is zero, all elements will be displayed.

3. callername (Input)
   is the name of the command invoking interpret_resource_desc_. It is used in printing any necessary error messages.

4. rst_control (Input)
   is declared in the include file rst_control.incl.pl1. (See "Display Control" below.)

5. return_noprint (Input)
   specifies, if "0"b, that information about the resource description is to be written to the user_output I/O switch. If "1"b, the information is returned in return_string, nth must not be zero, and the elements of the structure rst_control must be set so that exactly one item of information is requested.

6. return_string (Output)
   contains, if return_noprint is "1"b, a printable representation of the information requested. Otherwise, its contents are undefined.

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

Display Control

The rst_control structure (declared in the include file rst_control.incl.pl1) is defined as follows:

```
dcl 1 rst_control              aligned,
      2 default                bit (1) unaligned,
      2 name                   bit (1) unaligned,
      2 uid                    bit (1) unaligned,
      2 potential_attributes   bit (1) unaligned,
      2 attributes             bit (1) unaligned,
      2 desired_attributes     bit (1) unaligned,
      2 potential_aim_range    bit (1) unaligned,
      2 aim_range              bit (1) unaligned,
      2 owner                  bit (1) unaligned,
      2 acs_path               bit (1) unaligned,
      2 location               bit (1) unaligned,
      2 comment                bit (1) unaligned,
      2 charge_type            bit (1) unaligned,
      2 mode                   bit (1) unaligned,
      2 usage_lock             bit (1) unaligned,
      2 release_lock           bit (1) unaligned,
      2 awaiting_clear         bit (1) unaligned,
      2 user_alloc             bit (1) unaligned,
      2 given_flags            bit (1) unaligned,
      2 mbz                    bit (16) unaligned,
      2 any_given_item         bit (1) unaligned;
```

where:

1. default
   if "1"b, signifies that certain items of information are to be displayed only if they are not in the most common state. This bit hould not be used by non-system commands.

2. name
   is "1"b if item.name is to be displayed.

3. uid
   is "1"b if item.uid is to be displayed.

4. potential_attributes
   is "1"b if item.potential_attributes is to be displayed.

5. attributes
   is "1"b if item.attributes is to be displayed.

6. desired_attributes
   is "1"b if item.desired_attributes is to be displayed.

7. potential_aim_range
   is "1"b if item.potential_aim_range is to be displayed.

8. aim_range
   is "1"b if item.aim_range is to be displayed.

9. owner
   is "1"b if item.owner is to be displayed.

10. acs_path
    is "1"b if item.acs_path is to be displayed.

11. location
    is "1"b if item.location is to be displayed.

12. comment
    is "1"b if item.comment is to be displayed.

13. charge_type
    is "1"b if item.charge_type is to be displayed.

14. mode
    is "1"b if item.mode is to be displayed.

15. usage_lock
    is "1"b if item.usage_lock is to be displayed.

16. release_lock
    is "1"b if item.release_lock is to be displayed.

17. awaiting_clear
    is "1"b if item.awaiting_clear is to be displayed.

18. user_alloc
    is "1"b if item.user_alloc is to be displayed.

19. given_flags
    is "1"b if the state of all the flags in the structure item.given is to be displayed.

20. mbz
    is unused and must be "0"b.

21. any_given_item
    is "1"b to display any field in the item structure for which the corresponding bit in the item.given structure is "1"b.

Name: 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 as 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 as 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.

Entry: iod_info_$queue_data

This entry point examines the I/O daemon tables and returns the default queue and maximum number of queues for a given request type.

Usage

```
declare iod_info_$queue_data entry (char(*), fixed bin, fixed bin, fixed
     bin(35);

call iod_info_$queue_data entry (request_type, default_q, max_queues,
     code);
```

where:

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

2. default_q (Output)
   is the number of the default queue for the request type.

3. max_queues (Output)
   is the number of queues for the request type.

4. 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_$rqt_list

This entry point examines the I/O daemon tables and returns a list of request types of a given generic type.

Usage

```
declare iod_info_$rqt_list entry (char(32), (*) char(32), fixed bin, fixed
     bin(35));

call iod_info_$rqt_list entry (gen_type, q_list, n_queues, code);
```

where:

1. gen_type (Input)
   is the generic type of request types to be listed. If the string is blank, then all request types are listed.

2. q_list (Output)
   is an array that is filled in with the request type names to be returned. If the h-bound of this array is less than the number of names to be returned, the code error_table_$too_many_names will be returned, with the partial list.

3. n_queues (Output)
   is the number of entries returned in the q_list array.

4. code (Output)
   is a standard status code. If there are no matching entries, the code error_table_$no_entry is returned.

Name: iox_$init_standard_iocbs

The iox_$init_standard_iocbs entry point attaches the standard switches for a user process. These are currently user_input, user_output, and error_output, and they are attached with an attach description of:

syn_ user_i/o

The variables iox_$user_input, iox_$user_output, and iox_$error_output are set to the iocb pointers for these switches.

Usage

```
declare iox_$init_standard_iocbs entry ();

call iox_$init_standard_iocbs;
```

Notes

Should the standard attachments change, this program will change to establish whatever they are. It should therefore be used in any direct process overseer that wishes to establish standard attachments.

THIS PAGE INTENTIONALLY LEFT BLANK

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 standard status code.

Entry: ipc_$delete_ev_chn

This entry point destroys an event channel previously created by the process.

Usage

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

call ipc_$delete_ev_chn (channel_id, code);
```

where:

1. channel_id (Input)
   is the same as described above for ipc_$create_ev_chn.

2. code (Output)
   is the same as described above for ipc_$create_ev_chn.

Entry: ipc_$decl_event_call_chn

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

Usage

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

call ipc_$decl_event_call_chn (channel_id, call_chn_procedure, data_ptr,
     priority, code);
```

where:

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

2. call_chn_procedure (Input)
   is the procedure entry point invoked when an event occurs on the specified channel.

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 standard status code.

Entry: ipc_$decl_ev_wait_chn

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

Usage

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

call ipc_$decl_ev_wait_chn (channel_id, code);
```

where:

1. channel_id (Input)
   is the same as described above for ipc_$create_ev_chn.

2. code (Output)
   is 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:

1. channel_id (Input)
   is the same as described above for ipc_$create_ev_chn.

2. code (Output)
   is 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:

1. channel_id (Input)
   is the same as described above for ipc_$create_ev_chn.

2. code (Output)
   is 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:

1. channel_id (Input)
   is the same as described above for ipc_$create_ev_chn.

2. code (Output)
   is 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 standard status code.

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 standard status code.

Entry: ipc_$mask_ev_calls

This entry point causes the ipc_$block entry point (see below) to completely ignore event-calls occurring in the user's ring at the time of this call. This call causes a mask counter to be incremented. Event calls are masked if this counter is greater than zero.

Usage

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

call ipc_$mask_ev_calls (code);
```

where code (Output) is a standard status code.

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 standard status code. 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 (event_wait_list_ptr, event_wait_info_ptr, code);
```

where:

1. event_wait_list_ptr (Input)
   is a pointer to a structure that specifies the channels on which events are being awaited. This structure is declared in event_wait_list.incl.pl1.

```
dcl 1 event_wait_list      based aligned (event_wait_list_ptr),
      2 n_channels         fixed bin,
      2 pad                bit(36),
      2 channel_id   (event_wait_list_n_channels refer
                     (event_wait_list.n_channels)) fixed bin(71);
```

   where:

   n_channels
      is the number of channels. This item must be allocated on an even-word boundary.

   pad
      must be zero.

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

   Frequently ipc_$block is called with only one channel in the wait list. In this case, the following structure may be used. It is declared in event_wait_channel.incl.pl1.

```
dcl 1 event_wait_channel   aligned,
        2 n_channels       fixed bin initial (1),
        2 pad              bit(36),
        2 channel_id       (1) fixed bin(71);
```

2. event_wait_info_ptr (Input)
   is a pointer to a structure into which the ipc_$block entry point can put information about the event that caused it to return (i.e., that awakened the process). This structure is declared in event_wait_info.incl.pl1.

```
dcl 1 event_wait_info      based aligned (event_wait_info_ptr),
      2 channel_id         fixed bin(71),
      2 message            fixed bin(71),
      2 sender             bit(36),
      2 origin,
        3 dev_signal       bit(18) unaligned,
        3 ring             fixed bin(17) unaligned,
      2 channel_index      fixed bin;
```

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 event_wait_list structure above.

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

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 standard status code.

Invoking an Event-Call Procedure

When a process is awakened on an event-call channel, control is immediately passed to the procedure specified by the ipc_$decl_event_call_channel entry point. The procedure is called with one argument, a pointer to the following structure. This structure is declared in event_call_info.incl.pl1.

```
dcl 1 event_call_info   based aligned  (event_call_info_ptr),
      2 channel_id      fixed bin(71),
      2 message         fixed bin(71),
      2 sender          bit(36),
      2 origin,
        3 dev_signal    bit(18) unaligned,
        3 ring          fixed bin(17) unaligned,
      2 data_ptr        ptr;
```

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.

Notes

A user should be familiar with interprocess communication in Multics and the pitfalls of writing programs that can run asynchronously within a process. For example, if a program does run asynchronously within a process and it does input or output with the tty_ I/O module, then the program should issue the start control order of tty_ before it returns. This is necessary because a wakeup from tty_ may be intercepted by the asynchronous program.

If a program establishes an event-call channel, and the procedure associated with the event-call channel uses static storage, then the event-call procedure should have the perprocess_static attribute. This is not necessary if the procedure is part of a limited subsystem in which run units cannot be used. See the description of the run command in MPM Commands for more information on run units and perprocess_static.

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 3 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: mdc_

The mdc_ subroutine (actually a ring 1 gate) provides a series of entry points for manipulation of master directories.

Entry: mdc_$create_dir

This entry point is used to create a new master directory. Its arguments are roughly analogous to the hcs_$append_branchx entry point.

Usage

```
declare mdc_$create_dir entry (char(*), char(*), char(*), fixed bin(5),
     (3) fixed bin(3), char(*), fixed bin, fixed bin(35));

call mdc_$create_dir (dir_name entryname, volume, mode, rings, user_id,
     quota, code);
```

where:

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

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

3. volume (Input)
   is the name of the logical volume that is to contain segments created in the new directory.

4. mode (Input)
   is the user's access mode.

5. rings (Input)
   are the ring brackets of the directory.

6. user_id (Input)
   is an access control name.

7. quota (Input)
   is the quota to be placed on the new directory.

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

Entry: mdc_$create_dirx

This entry point is an extension of the mdc_$create_dir entry point, which is similiar to hcs_$create_branch_ entry point.

Usage

```
declare mdc_$create_dirx entry (char(*), char(*), char(*), ptr,
     fixed bin(35));

call mdc_$create_dirx (dir_name, entryname, volume, info_ptr, code);
```

where:

1. dir_name
   is as above.

2. entryname
   is as above.

3. volume
   is as above.

4. info_ptr (Input)
   is a pointer to a status structure as described under the hcs_$create_branch_ entry point.

Entry: mdc_$delete_dir

This entry point is used to delete a master directory.

Usage

```
declare mdc_$delete_dir entry (char(*), char(*), fixed bin(35));

call mdc_$delete_dir (dir_name, entryname, code);
```

where:

1. dir_name
   is as above.

2. entryname
   is as above.

3. code
   is as above.

Entry: mdc_$set_mdir_quota

This entry point is used to set the quota on a master directory.

Usage

```
declare mdc_$set_mdir_quota entry (char(*), char(*), bit(1) aligned,
     fixed bin, fixed bin(35));

call mdc_$set_mdir_quota (dir_name, entryname, sw, quota, code);
```

where:

1. dir_name
   is as above.

2. entryname
   is as above.

3. sw (Input)
   is a switch indicating the kind of quota change.
   "0"b sets the directory quota to the quota parameter.

   "1"b algebraically adds the quota parameter to the current directory quota.

4. quota
   is as above.

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

Entry: mdc_$set_volume_quota

This entry point is used to set the volume quota for a quota account on a logical volume.

Usage

```
declare mdc_$set_volume_quota entry (char(*), char(*), bit(1) aligned,
     fixed bin, fixed bin(35));

call mdc_$set_volume_quota (volume, account, sw, quota, code);
```

where:

1. volume
   is as above.

2. account (Input)
   is the name of the quota account in the form Person_id.Project_id.tag. The quota account name may contain stars.

3. sw
   is as above.

2. account (Input)
   is the name of the quota account in the form Person_id.Project_id.tag. The quota account name may contain stars.

3. sw
   is as above.

4. quota
   is as above.

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

Entry: mdc_$set_mdir_owner

This entry point is used to set the owner name of a master directory.

Usage

```
declare mdc_$set_mdir_owner entry (char(*), char(*), char(*),
     fixed bin(35));

call mdc_$set_mdir_owner (dir_name, entryname, owner, code);
```

where:

1. dir_name
   is as above.

2. entryname
   is as above.

3. owner (Input)
   is the new owner name of the master directory, in the form person_id.project_id.tag.

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

Entry: mdc_$set_mdir_account

This entry point is used to set the quota account of a master directory.

Usage

```
declare mdc_$set_mdir_account entry (char(*), char(*), char(*),
     fixed bin(35));

call mdc_$set_mdir_account (dir_name, entryname, account, code);
```

where:

1. dir_name
   is as above.

2. entryname
   is as above.

3. account
   is the name of the new quota account. The directory quota is returned to the old account and redrawn from this new account.

4. code
   is as above.

Entry: mhcs_$get_seg_usage

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

Usage

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

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

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.

The mode_string_ subroutine has been moved to the MPM Subroutines manual.

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.

Note

If the file does not exist, fcb_ptr is nonnull and the code error_table_$noentry is returned. If the file cannot be opened, fcb_ptr is null and the value of code returned indicates the reason for failure.

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_$msf_get_ptr

The msf_manager_$msf_get_ptr entry point returns a pointer to a specified component in the multisegment file. The component can be created if it does not exist. If the file is a single segment file, and the requested component is not component 0, the single segment is converted to a multisegment file. This change does not affect a previously returned pointer to component 0. If the file does not exist, it is created as a "mulit-segment file" with a single component. This entry point never creates a single segment file. (See also the msf_manager_$get_ptr entrypoint.)

Usage

```
declare msf_manager_$msf_get_ptr entry (ptr, fixed bin, bit(1), ptr, fixed
     bin(24), fixed binZ(35));

     call msf_manager_$msf_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: nd_handler_

This subroutine attempts to resolve the name duplication caused when a program tries to create a segment, multisegment file, or link in a directory that already contains an entry by the same name. If the existing entry has additional names, nd_handler_ tries to delete the name needed for the new entry and, if successful, prints a warning message. If the existing entry has only one name, nd_handler_ queries the user whether or not to delete it. A zero status code in either case means that nd_handler_ has succeeded, and the calling program can retry creating the new entry.

Entry: nd_handler_

Usage

```
dcl nd_handler_ entry (char(*), char(*), char(*), fixed bin(35));
call nd_handler_ (caller, dn, en, code);
```

where:

1. caller (Input)
   is the name of the calling program, used in printed messages.

2. dn (Input)
   is the pathname of the directory involved.

3. en (Input)
   is the name of the entry that the calling program wants to create.

4. code (Output)
   is a standard status code. It may be:
   0
       if the old entryname has been removed
   error_table_$action_not_performed
       if the user answered "no" to a query
   other codes
       if the old entryname could not be removed for some other reason such as lack of access. An error message is then printed by nd_handler_.

Notes

This subroutine is usually called after another subroutine call has returned error_table_$namedup. If nd_handler_ returns a zero status code, the other subroutine is called a second time. A warning message of the following kind is printed if the existing entry has multiple names:

```
caller:  Name duplication.  Old name foo removed
           from >udd>m>Smith>oldseg.
```

If the existing entry has only one name, wording of the query depends on the existing entry's type:

```
caller:  Do you want to delete the old segment <path>?
caller:  Do you want to delete the old multisegment file <path>?
caller:  Do you want to unlink the old link <path>?
                  (Target <path2> exists.)
             or:  (Target <path2> does not exist.)
             or:  (Cannot get info for target <path2>.)
             or:  (No target pathname.)
```

The following entry points have the same calling sequence.

Entry: nd_handler_$force

This entry point deletes the existing entry if it has only one name, rather than issue a query.

Entry: nd_handler_$del

This entry point queries whether or not to delete the existing entry, regardless of whether or not it has additional names.

Entry: nd_handler_$del_force

This entry point deletes the old entry (no query), regardless of whether it has additional names.

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 based,
      2 version_number          fixed bin,
      2 textp                   ptr,
      2 defp                    ptr,
      2 linkp                   ptr,
      2 statp                   ptr,
      2 symbp                   ptr,
      2 bmapp                   ptr,
      2 tlng                    fixed bin(18),
      2 dlng                    fixed bin(18),
      2 llng                    fixed bin(18),
      2 ilng                    fixed bin(18),
      2 slng                    fixed bin(18),
      2 blng                    fixed bin(18),
      2 format,
        3 old_format            bit(1) unaligned,
        3 bound                 bit(1) unaligned,
        3 relocatable           bit(1) unaligned,
        3 procedure             bit(1) unaligned,
```

```
      3 standard              bit(1) unaligned,
      3 gate                  bit(1) unaligned,
      3 separate_static       bit(1) unaligned,
      3 links_in_text         bit(1) unaligned,
      3 perprocess_static     bit(1) unaligned,
      3 pad                   bit(27) unaligned,
    2 entry_bound             fixed bin,
    2 textlinkp               ptr,

/*This is the limit of the $brief info structure.*/

    2 compiler                char(8) aligned,
    2 compile_time            fixed bin(71),
    2 userid                  char(32) aligned,
    2 cvers                   aligned,
      3 offset                bit(18) unaligned,
      3 length                bit(18) unaligned,
    2 comment                 aligned,
      3 offset                bit(18) unaligned,
      3 length                bit(18) unaligned,
    2 source_map              fixed bin,

/*This is the limit of the $display info structure.*/

    2 rel_text                ptr,
    2 rel_def                 ptr,
    2 rel_link                ptr,
    2 rel_static              ptr,
    2 rel_symbol              ptr,
    2 text_boundary           fixed bin,
    2 static_boundary         fixed bin,
    2 default_truncate        fixed bin,
    2 optional_truncate       fixed bin;

/*This is the limit of the $long info structure.*/
```

where:

1. version_number
   is the version number of the structure (currently this number is 2). This value is input.

2. textp
   is a pointer to the base of the text section.

3. defp
   is a pointer to the base of the definition section.

4. linkp
   is a pointer to the base of the linkage section.

5. statp
   is a pointer to the base of the static section.

6. symbp
   is a pointer to the base of the symbol section.

7. bmapp
   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. textlinkp
    is a pointer to the first text-embedded link if links_in_text is equal to "1"b.

This is the limit of the info structure for the object_info_$brief entry point.

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. userid
    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 7 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 7 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 6 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_password_

The read_password_ subroutine reads a single line from the users' terminal (actually from the user_input I/O switch). It attempts to hide the input line by turning the printing mechanism off before reading and turning it back on afterwards. If the printing mechanism cannot be turned off, then a mask consisting of several layers of printing designed to "black out" the page is printed. One of the layers of printing is pseudo-randomly generated so that it will be different each time the subroutine is called, thus making it difficult to analyze the layers of overprinting. The mask is 12 characters long.

Usage

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

call read_password_ (prompt, password);
```

where:

1. prompt (Input)
   is a message to be printed before the password is read. It can be any length. A newline character is always printed after the prompting message.

2. password (Output)
   is the password that the user typed. It can be up to 120 characters long.

Note

The password is processed as follows: Tab characters are translated to blanks. Leading blanks are removed. Characters after any embedded blanks are removed. If the resulting password is all blank, a single asterisk ("*") is returned, otherwise the password is returned.

Entry: read_password_$switch

This entry is similar to read_password_, but it allows the caller to specify the I/O switches to be used to print the prompt and read the password.

Usage

```
declare read_password_$switch entry (ptr, ptr, char(*), char(*),
     fixed bin(35));

call read_password_$switch (output_switch, input_switch, prompt, password,
     code);
```

where:

1. output_switch (Input)
   is a pointer to the I/O switch on which the prompt, and if necessary the password mask, is printed.

2. input_switch (Input)
   is a pointer to the I/O switch from which the password is read.

3. prompt (Input)
   is a message to be printed before the password is read. It can be any length. A newline character is always printed after the prompting message.

4. password (Output)
   is the password that the user typed. It can be up to 120 characters long.

5. code (Output)
   is a standard system status code which is non-zero only if a password could not be read.

Note

The password is processed as follows: Tab characters are translated to blanks. Leading blanks are removed. Characters after any embedded blanks are removed. If the resulting password is all blank, a single asterisk ("*") is returned; otherwise the password is returned.

This page intentionally left blank.

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 6 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: requote_string_

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

Usage

```
declare requote_string_ entry (char(*)) returns(char(*));
requoted_string = requote_string_ (string);
```

where:

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

2. requoted_string (Output)
   is the string with all quotes doubled and enclosed in quotes.

Examples

```
"""a""" = requote_string_ ("a")

"""a""""b""" = requote_string_ ("a""b")
```

Name: resource_control_

The resource_control_ subroutine provides an interface to the Multics resource control facility. Entry points in this subroutine allow programs to reserve or cancel I/O devices and volumes.

Note

Not all sites enable the resource_control_ subroutine. Consult your system administrator to find out if your site has this capability.

Entry: resource_control_$reserve

This entry point reserves a resource or group of resources for use by a process.

Usage

```
declare resource_control_$reserve entry (pointer, pointer, bit (1) aligned,
     bit (72) aligned, fixed bin (35));

call resource_control_$reserve (descriptions_ptr, reservation_desc_ptr,
     authorization, system, code);
```

where:

1. descriptions_ptr (Input)
   is a pointer to the structure containing a description of the resources to be reserved (see "Resource Description" below).

2. reservation_desc_ptr (Input)
   is a pointer to the structure containing reservation information for the resources to be reserved (see "Reservation Description" below).

3. authorization (Input)
   checks the user's authorization to use the devices or volumes and is only valid if system = "1"b.

4. system (Input)
   specifies, if "1"b, that the calling process wishes to perform a privileged reservation (see "Notes" below).

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

Reservation Description

The reservation_desc_ptr argument points to the following structure (declared in the include file resource_control_desc.incl.pl1):

```
dcl 1 reservation_description aligned based,
     2 version_no               fixed bin,
     2 reserved_for             char (32),
     2 reserved_by              char (32),
     2 reservation_id           fixed bin (71),
     2 group_starting_time      fixed bin (71),
     2 asap_duration            fixed bin (71),
     2 flags                    aligned,
    (3 auto_expire              bit(1),
      3 asap                    bit (1),
      3 rel                     bit (1),
      3 sec                     bit (1)) unaligned,
     2 n_items                  fixed bin,
     2 reservation_group (Resource_count refer
       (reservation_description.n_items)),
      3 starting_time           fixed bin (71),
      3 duration                fixed bin (71);
```

where:

1. version_no (Input)
   is the current version number of this structure. It should be set to "resource_control_version_1".

2. reserved_for (Input)
   specifies the User_id of the process for whom this reservation is made. The use of an asterisk (*) for a component name is permitted. If this element is blanks, the User_id of the current process is used.

3. reserved_by (Input)
   is the User_id of the process which is charged for this reservation (see "Notes" below). This element is ignored for an unprivileged reservation and the current User_id is used.

4. reservation_id (Input or Output)
   is an identifier for this reservation group. It is currently returned as an absolute clock time.

5. n_items (Input)
   is the number of items being reserved.

The rest of the items in this structure are currently ignored and should be set to zero.

Notes

If system = "1"b, reservation_description.reserved_by is used to specify the User_id of the process to be charged for this reservation.

The reservation_description structure is strongly dependent on the resource_descriptions structure. That is, for each resource described in resource_descriptions there must be a corresponding entry of the same index in reservation_description.

Access Restrictions

Execute access to the rcp_sys_ gate is necessary to perform a privileged reservation.

Entry: resource_control_$cancel

This entry point cancels the reservation of a resource or group of resources.

Usage

```
declare resource_control_$cancel_id_string entry (char(*), char(*),
     bit(1) aligned, fixed bin (35));

call resource_control_$cancel_id_string (reservation_id, group_id, system,
     code);
```

where:

1. reservation_id (Input)
   is the character string representation of the reservation identifier to be cancelled.

2. group_id (Input)
   is the group id of the user to whom the reservation belongs. This is only valid if system = "1"b.

3. system (Input)
   specifies, if "1"b, that a privileged cancellation is to be performed (see "Notes" below).

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

Notes

If system = "1"b, then the reservation group is forcibly cancelled whether or not it belongs to the current process. *

Access Restrictions

Execute access to the rcp_sys_ gate is necessary to perform a privileged cancellation.

This page intentionally left blank.

This page intentionally left blank.

This page intentionally left blank.

This page intentionally left blank.

This page intentionally left blank.

Resource Description

The descriptions_ptr argument points to the following structure (this structure is declared in the include file resource_control_desc.incl.pl1):

```
dcl 1 resource_descriptions based (resource_desc_ptr) aligned,
      2 version_no fixed bin,
      2 n_items fixed bin,
      2 item (Resource_count refer (resource_descriptions.n_items)) aligned,
        3 type char (32),
        3 name char (32),
        3 uid bit (36),
        3 potential_attributes bit (72),
        3 attributes (2) bit (72),
        3 desired_attributes (4) bit (72),
        3 potential_aim_range (2) bit (72),
        3 aim_range (2) bit (72),
        3 owner char (32),
        3 acs_path char (168),
        3 location char (168),
        3 comment char (168),
        3 charge_type char (32),
        3 rew bit (3) unaligned,
        3 (usage_lock,
           release_lock,
           awaiting_clear,
           user_alloc) bit (1) unaligned,
        3 pad2 bit (29) unaligned,
        3 given aligned,
          4 (name,
             uid,
             potential_attributes,
             desired_attributes,
             potential_aim_range,
             aim_range,
             owner,
             acs_path,
             location,
             comment,
             charge_type,
             usage_lock,
             release_lock,
             user_alloc) bit (1),
          4 pad1 bit (22)) unaligned,
        3 state bit (36) aligned,
        3 status_code fixed bin (35);
```

where:

1. version_no (Input)
   is the current version number of the structure. It should be set to "resource_control_version_1".

2. n_items (Input)
   specifies the number of resources described by this structure. A consistent combination of the following elements must be supplied for each resource described.

3. type (Input)
specifies the type of resource desired (e.g., tape, disk_drive). It must be supplied (see "Notes" below).

4. name (Input or Output)
is a specific resource name. If flags.name_given = "1"b, the named resource is chosen. If flags.name_given = "0"b, a resource is chosen depending on criteria specified by other elements of the structure, and the name of the resource chosen is returned in this element (see "Notes" below).

5. uid (Input or Output)
is the unique identifier of a specific resource. If flags.uid_given = "1"b, the specified resource is chosen. If flags.uid_given = "0"b, a resource is chosen depending on criteria specified by other elements of the structure, and the unique identifier of the resource chosen is returned in this element.

6. potential_attributes (Output)
specifies the potential attributes of the resource chosen.

7. attributes (Input or Output)
contains, if flags.attr_given = "1"b, the specification of attributes which the resource chosen must possess. If flags.attr_given = "0"b, the resource to be chosen need not possess any particular attributes. The attributes of the resource chosen are returned in these elements (see "Notes" below).

8. desired_attributes (Input)
specifies the desired attributes of the resource chosen.

9. potential_aim_bounds (Output)
are a pair of AIM access classes, specifying the minimum and maximum process authorization that can be permitted to acquire this resource.

10. aim_bounds (Input or Output)
are a pair of AIM access classes, specifying the minimum and maximum process authorization that can be permitted to both read and write this resource. If flags.aim_bounds_given = "1"b, this element is input. Otherwise, it is output.

11. owner (Input or Output)
is the owner of the resource. If flags.owner = "1"b, this element is input. Otherwise, this element is output (see "Notes" and "Access Restrictions" below).

12. acs_path (Input)
is the pathname of the access control segment (ACS) for this resource (see "Access Restrictions" below).

13. location (Output)
contains a character string description of the location of this resource.

14. comment (Input)
    contains a character-string comment which is associated with this resource.

15. charge_type (Input)
    is the accounting identifier for this resource.

16. rew (Output)
    is the effective access of the user to this resource.

17. usage_lock (Input)
    if "1"b, specifies that this resource cannot be used by any user, regardless of the state of the resource.

18. release_lock (Input)
    if "1"b, specifies that the owner of the resource is not allowed to release the resource. Unless system = "1"b, this element is ignored (see "Notes" below).

19. awaiting_clear (Output)
    specifies that the resource is awaiting manual clear.

20. user_alloc (Input)
    if "1"b, specifies that the user has not allocated the resource to any use.

21. pad2 (Input)
    is unused and must be zero.

22. name (Input)
    is "1"b if item.name has been supplied by the caller.

23. uid (Input)
    is "1"b if item.uid has been supplied by the caller.

24. potential_attr (Input)
    is "1"b, if item.potential_attributes has been supplied by the caller.

25. desired_attr (Input)
    is "1"b if item.desired_attributes has been supplied by the caller.

26. potential_aim_bounds (Input)
    is "1"b if item.potential_aim_bounds has been supplied by the caller.

27. aim_bounds (Input)
    is "1"b if item.aim_bounds has been supplied by the caller.

28. owner (Input)
    is "1"b if item.owner has been supplied by the caller.

29. acs_path (Input)
    is "1"b if item.acs_path has been supplied by the caller.

30. location (Input)
    is "1"b if item.location has been supplied by the caller.

31. comment (Input)
    is "1"b if item.comment has been supplied by the caller.

32. charge_type (Input)
    is "1"b if item.charge_type_given has been supplied by the caller.

33. usage_lock (Input)
    is "1"b if item.usage_lock has been supplied by the caller.

34. release_lock (Input)
    is "1"b if item.release_lock has been supplied by the caller.

35. user_alloc (Input)
    is "1"b if item.user_alloc_given has been supplied by the caller.

36. pad1 (Input)
    is unused and must be zero.

37. state (Output)
    is for the use of resource_control_ and should not be used by the user.

38. status_code (Output)
    is a standard status code. If the subroutine argument code is nonzero, one or more items in the structure have a nonzero status_code specifying in more detail why the attempt to manipulate the described resource was refused.

Notes

A list of defined resource types may be obtained via the list_resource_types command.

Suitable values for the attributes element may be constructed using the cv_rcp_attributes_$from_string subroutine.

Access Restrictions

The user must have at least sm permission to the directory in which the ACS is specified to reside.

Unless otherwise stated, the user must have re access to the rcp_sys_ gate to specify system = "1"b in the calling sequence for any entry point of the resource_control_ subroutine.

Name: resource_info_

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

Entry: resource_info_$get_type

This entry point, given the name of a resource type, indicates whether the resource type named is a device or a volume.

Usage

```
declare resource_info_$get_type entry (char (*), bit (1), fixed bin (35));

call resource_info_$get_type (name, is_volume, code);
```

where:

1. name (Input)
   is the name of a defined resource type (see "Notes" below).

2. is_volume (Output)
   is "1"b if the resource type given specifies a class of volumes. If "0"b, the resource type given specifies a class of devices.

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

Notes

A list of defined resource types may be obtained via the list_resource_types command (see Section 4).

Entry: resource_info_$limits

This entry point returns information about quantity and time limits for a given resource type.

Usage

```
declare resource_info_$limits entry (char (*), fixed bin, fixed bin,
     fixed bin, fixed bin (35));

call resource_info_$limits (name, max_quantity, default_time, max_time,
     code);
```

where:

1. name (Input)
   is the name of a defined resource type.

2. max_quantity (Output)
   is the maximum number of this type of resource that a process may assign at one time.

3. default_time (Output)
   is the default reservation time, in minutes, for this type of resource.

4. max_time (Output)
   is the maximum allowed reservation time, in minutes, for this type of resource.

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

Notes

The information returned by this entry point is from the RTDT. These are not the limits currently enforced by RCP (see "Device Limits" in Section 1 of the *Multics Resource Control Users' Guide* (CT38)).

Entry: resource_info_$mates

This entry provides information about the resource type or types with which the given resource type may be mounted.

Usage

```
declare resource_info_$mates entry (char (*), fixed bin, char (*)
     dimension (*), fixed bin (35));

call resource_info_$mates (name, n_mates, mates, code);
```

where:

1. name (Input)
   is the name of a defined resource type.

2. n_mates (Output)
   is the number of mates returned.

3. mates (Output)
contains the name or names of the resource type(s) that may be mounted with this resource (see "Notes" below).

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

Notes

If the number of elements in mates is too small to hold all the mates for the given resource type, code is set to error_table_$smallarg and mates is set to the null string. However, n_mates still contains the number of mates associated with the given resource type.

Entry: resource_info_$defaults

This entry point fills a resource_descriptions structure with the default registration parameters defined in the RTDT.

Usage

```
dcl resource_info_$defaults entry (char(*), char(*), pointer,
     fixed bin(35));

call resource_info_$defaults (name, subtype, item_ptr, code);
```

where:

1. name (Input)
is the name of a defined resource type.

2. subtype (Input)
is the name of a subtype of the resource type, defined in the RTDT. If subtype is the null string, the master defaults for the resource type are used.

3. item_ptr
points to a structure declared like resource_descriptions.item (see the resource_control_ subroutine).

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

Entry: resource_info_$lock_on_release

This entry point returns a value specifying whether resources of a given type are to be locked for manual clearing at release time.

Usage

```
dcl resource_info_$lock_on_release entry (char(*), bit(1) aligned,
     fixed bin(35));

call resource_info_$lock_on_release (name, lock_sw, code);
```

where:

1. name (Input)
   is the name of a defined resource type.

2. lock_sw (Output)
   specifies whether the resource is locked at release time.
   "1"b lock the resource
   "0"b do not lock the resource

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

Entry: resource_info_$canonicalize_name

This entry point applies the proper canonicalization to a resource name of a given resource type. See "Canonicalization Routines" in the Multics Administrators' Manual - Resource Control (Order No. CC74).

Usage

```
declare resource_info_$canonicalize_name entry (char(*), char(*), char(*),
     fixed bin(35));

call resource_info_$canonicalize_name (resource_type, resource_name,
     canonicalized_name, code);
```

where:

1. resource_type (Input)
   is the name of a defined resource type.

2. resource_name (Input)
   is the string to be canonicalized.

3. canonicalized_name (Output)
   is the canonicalized representation of resource_name.

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

Name: run_

The run_ subroutine manages the environment for a run unit and invokes the main program of a run unit. See the documentation of the run command in the MPM Commands for an explanation of run units.

Entry: run_

This entry sets up the run unit environment, invokes the main program, and restores the environment when the run ends.

Usage

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

call run_ (main_entry, arglist_ptr, run_cs_ptr, code);
```

where:

1. main_entry (Input)
   is the entry point to be called as the main program of the run unit.

2. arglist_ptr (Input)
   points to the argument list for the main program.

3. run_cs_ptr (Input)
   points to the following structure which is declared in run_control_structure.incl.pl1:

```
dcl 1 run_control_structure    aligned based(run_cs_ptr),
      2 version                fixed bin,
      2 flags                  aligned,
        3 ec                   bit(1) unaligned,
        3 pad                  bit(35) unaligned,
      2 reference_name_switch  fixed bin,
      2 time_limit             fixed bin(35);
```

   where:

   1. version
      is the version number of the structure. It should be set to run_control_structure_version_1.

   2. ec
      is "1"b if the main program is exec_com (main_entry must still be set), otherwise ec must be "0"b.

3. pad
   must be "0"b.

4. reference_name_switch
   is set to one of the named constants NEW_REFERENCE_NAMES, COPY_REFERENCE_NAMES or OLD_REFERENCE_NAMES delcared in run_control_structure.incl.pl1.

5. time_limit
   is the interval in cpu seconds after which the program is to be interrupted.

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

Entry: run_$environment_info

This entry enables the symbolic debugging tools to obtain the saved stack header information used by a given stack frame.

Usage

```
declare run_$environment_info entry (ptr, ptr, fixed bin(35));

call run_$environment_info (stack_frame_ptr, info_ptr, code);
```

where:

1. stack_frame_ptr (Input)
   points to an active stack frame on the current stack.

2. info_ptr (Input)
   points to the following structure, declared in env_ptrs.incl.pl1:

```
dcl 1 env_ptrs              aligned based,
      2 version             fixed bin,
      2 pad                 fixed bin(35),
      2 lot_ptr             ptr,
      2 isot_ptr            ptr,
      2 clr_ptr             ptr,
      2 combined_stat_ptr   ptr,
      2 user_free_ptr       ptr,
      2 sys_link_info_ptr   ptr,
      2 rnt_ptr             ptr,
      2 sct_ptr             ptr;
```

   where:

   1. version
      is the version number of this structure; it must be 1.

   2. pad
      is unused.

3. lot_ptr
   points to the linkage offset table (LOT).

4. isot_ptr
   points to the internal static offset table (ISOT).

5. clr_ptr
   points to the area where linkage sections are allocated.

6. combined_stat_ptr
   points to the area where separate static sections are allocated.

7. user_free_ptr
   points to the area where user storage is allocated.

8. sys_link_info_ptr
   points to the control structure for external static variables.

9. rnt_ptr
   points to the reference name table.

10. sct_ptr
    points to the static handler array.

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

Name: sct_manager_

The sct_manager_ subroutine manipulates the System Condition Table (SCT), which is used to provide static handlers for certain conditions. It has entries to set a handler, get a pointer to a handler, and call a handler if one exists.

Entry: sct_manager_$set

This entry point sets the handler for the given index to the one given in the call.

Usage

```
declare sct_manager_$set entry (fixed bin, ptr, fixed bin (35));

call sct_manager_$set (fcode, hptr, code);
```

where:

1. fcode (Input)
   is a fixed binary index into the SCT table. Appropriate values can be selected from static_handlers.incl.pl1, which gives symbolic names for all indices currently defined.

2. hptr (Input)
   is a pointer to the static handler, if it exists.

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

Entry: sct_manager_$get

This entry point returns a pointer to the handler for the given index, or null if it does not exist.

Usage

```
declare sct_manager_$get entry (fixed bin, ptr, fixed bin (35));

call sct_manager_$get (fcode, hptr, code);
```

where:

1. fcode (Input)
   is a fixed binary index into the SCT table. Appropriate values can be selected from static_handlers.incl.pl1, which gives symbolic names for all indices currently defined.

2. hptr (Output)
   is a pointer to the static handler, if it exists.

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

Entry: sct_manager_$call_handler

This entry point calls a handler if it exists. If none exists, the "continue" bit is set on to pass this information to the caller.

Usage

```
declare sct_manager_$call_handler entry (ptr, char(*), ptr, ptr, bit (1)
     aligned);

call sct_manager_$call_handler (mcptr, cname, null(), null(), continue);
```

where:

1. mcptr (Input)
   is a pointer to the machine conditions for the condition to be handled. The fault code within the scu data determines the handler to use.

2. cname (Input)
   is the name of the condition being signalled. It is passed to the condition handler, if there is one.

5. continue (Output)
   is set to "1"b if there is no handler, otherwise it is set by the handler.

The third and fourth arguments are ignored; they must be null. They are declared for compatibility with the standard condition handler mechanism.

Notes

The System Condition Table is a based array of 127 packed pointers, pointed to by the sct_pointer in the stack_header of the stack for the ring in which sct_manager_ is executing. The pointers point to the entry to call, and a null value is used for the environment portion of the entry. A static handler has the same calling sequence as any other condition handler. SCT indices are assigned by hardcore systems programmers. Since sct_manager_$call_handler uses machine conditions to locate the handler, conditions without machine conditions (e.g., software conditions such as PL/I support) cannot have static handlers. Ring 0, rather than the user, ensures that there is a proper fault code in the conditions.

Name: set_ext_variable_

The set_ext_variable_ subroutine allows the caller to look up an external variable by name. If the name is not found, the variable is added to the list of external variables.

Usage

```
dcl set_ext_variable_ entry (char(*), ptr, ptr, bit(1) aligned, ptr,
     fixed bin(35));

call set_ext_variable_ (ext_name, init_info_ptr, sb_ptr, found_sw,
     node_ptr, code);
```

where:

1. ext_name (Input)
   is the name of the external variable.

2. init_info_ptr (Input)
   is a pointer to the initialization info (see "Notes" below).

3. sb_ptr (Input)
   is a pointer to the base of the stack of the caller.

4. found_sw (Output)
   is set to indicate whether the variable was found or not.

5. node_ptr (Output)
   is a pointer to the external variable node. (see "Notes" below)

6. code (Output)
   is an error code.

Notes

When a new external variable is allocated (not found), it must be initialized. The following structure, described in system_link_init_info.incl.pl1, is pointed to by init_info_ptr:

```
dcl 1 init_info            aligned based,
    2 size                 fixed bin(19),
    2 type                 fixed bin,
    2 init_template
      (init_size refer
      (init_info.size))    fixed bin(35);
```

where:

1. size
   is the initialization template size, in words.

2. type
   is the type of initialization to be performed.
   0 no init
   3 init from template
   4 init area to empty ()

3. init_template
   is the initialization template to be used when type = 3. Great care should be taken when referencing with the node_ptr. The node structure should never be modified. Modifications to the node will have unpredictable results.

Notes

A pointer to the following structure is returned by the locate entry to set_ext_variable_ (found in system_link_names.incl.pl1):

```
dcl 1 variable_node       based aligned,
      2 forward_thread     ptr unal,
      2 vbl_size           fixed bin(23) unal,
      2 init_type          fixed bin(11) unal,
      2 time_allocated     fixed bin(71),
      2 vbl_ptr            ptr,
      2 init_ptr           ptr,
      2 name_size          fixed bin,
      2 name_char          (nchars refer (variable_node.name_size));
```

where:

1. forward_thread
   is used by the linker to thread this variable to the next.

2. vbl_size
   is the size, in words, of this variable.

3. init_type
   is the type of initialization that is performed:
   0 none
   3 initialize from template
   4 initialize to an empty area

4. time_allocated
   is the clock reading at the time this variable was allocated.

5. vbl_ptr
   is a pointer to the variable's storage.

6. init_ptr
   is a pointer to the initialization template.

7. name_size
   is the number of characters in the variable name.

8. name
   is the name of the variable.

Entry: set_ext_variable_$locate

This entry point locates the specified external variable and returns a pointer to the structure describing the variable.

Usage

```
dcl set_ext_variable_$locate entry (char(*), ptr, ptr, fixed bin(35));

call set_ext_variable_$locate (ext_name, sb_ptr, node_ptr, code);
```

where:

1. ext_name (Input)
   is the name of the external variable.

2. sb_ptr (Input)
   is a pointer to the base of the stack of the caller.

3. node_pointer (Output)
   is a pointer to the variable_node describing the specified variable. This structure is defined in the system_link_names.incl.pl1 include file. (see "Notes" above)

4. code (Output)
   is an error code.

Name: shcs_$set_force_write_limit

The shcs_$set_force_write_limit entry point sets the write limit of the calling process. This limit specifies the maximum number of pages that may be queued for I/O at the same time by calls to hcs_$force_write. The default for this limit is 1.

Usage

```
declare shcs_$set_force_write_limit entry (fixed bin, fixed bin (35));

call shcs_$set_force_write_limit (npages, code);
```

where:

1. npages (Input)
   is the maximum number of pages that will be allowed to be queued for I/O at the same time.

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

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 7 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 pointer) to information relating to the condition being raised. The structure of the information is dependent upon the condition being signalled; however, conditions raised with the same name should provide the information in the same structure. All structures must begin with a standard header. The format for the header as well as the structures provided with system conditions are described in "List of System Conditions and Default Handlers" in Section 7 of the MPM Reference Guide. This argument is intended for use in signalling conditions other than hardware faults. This argument is optional.

4. wc_ptr
   is a pointer (declared as an aligned pointer) to the machine conditions at the time a lower ring was entered to process a fault. This argument is used only by the system and only in the case where a condition that occurred in a lower ring is being signalled in the outer ring and when the lower ring has been entered to process a fault occurring in the outer ring. This argument is optional.

Notes

If the signal_ subroutine returns to its caller, indicating that the handler has returned to it, the calling procedure should retry the operation that caused the condition to be signalled.

The PL/I signal statement differs from the signal_ subroutine in that the above parameters cannot be provided in the signal statement. Also, for PL/I-defined conditions, a call to the signal_ subroutine is not equivalent to a PL/I signal statement since information about these conditions is kept internally.

Name: 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 describing the reason for calling the sub_err_ subroutine. (It is normally declared fixed bin(35); but it can be any computational data type. If not fixed bin(35), it will be converted to fixed bin(35)).

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 options associated with the error. The flags argument should be declared as a nonvarying bit string. The following values, located in the include file sub_err_flags.incl.pl1, are permitted:

```
ACTION_CAN_RESTART       init (""b),
ACTION_CANT_RESTART      init ("1"b),
ACTION_DEFAULT_RESTART   init ("01"b),
ACTION_QUIET_RESTART     init ("001"b)
ACTION_SUPPORT_SIGNAL    init ("0001"b)) bit (36) aligned
                         internal static options (constant);
```

   Each bit corresponds to one of the action flags in the standard condition_info_header structure, declared in condition_info_header.incl.pl1. If multiple bits are on in the supplied string, all the specified flags are set. See the MPM Reference Guide for definitions of the flags.

4. info_ptr (Input)
is a pointer (declared as an aligned pointer) to optional information specific to the situation. This argument is used as input to initialize info.retval (see "Info Structure," below). 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. This argument is used as input to initialize info.retval (see "Info Structure," below). The standard system environment sets this value to zero. Other environments may set the retval argument to other values, which may be used to select recovery strategies. The retval argument should be declared fixed bin(35).

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.

Note

There is an obsolete calling sequence to this subroutine, in which the flags argument is a character string instead of a bit string. In that calling sequence, the legal values are "s" for ACTION_CAN_RESTART, "h" for ACTION_CANT_RESTART, "q" for ACTION_QUIET_RESTART, and "c" for ACTION_DEFAULT_RESTART.

Operation

The sub_err_ subroutine proceeds as follows: the structure described below is filled in from the arguments to the sub_err_ subroutine and the signal_ subroutine is called to raise the sub_error_ condition.

When the standard system environment receives a sub_error_ signal, it prints a message of the form:

```
name error by sub_name|location
Status code message. Message from ctl_string.
```

The standard environment then sets retval to zero and returns, if the value ACTION_DEFAULT_RESTART is specified; otherwise it calls the listener. If the start command is invoked, the standard environment returns to sub_err_, which returns to the subroutine caller of the sub_err_ subroutine unless ACTION_CANT_RESTART is specified. If the value ACTION_CANT_RESTART is specified, the sub_err_ subroutine signals the illegal_return condition.

Handler Operation

All handlers for the any_other condition must either pass the sub_error_ condition on to another handler, or else must handle the condition correctly. Correct handling consists of printing the error message and of respecting the cant_restart, default_restart, and quiet_restart flags, unless the environment deliberately countermands these actions (for example, for debugging purposes).

If an application program wishes to call a subsystem that reports errors by the sub_err_ subroutine and wishes to replace the standard system action for some classes of sub_err_ subroutine calls, the application should establish a handler for the sub_error_ condition by a PL/I on statement. When the handler is activated as a result of a call to the sub_err_ subroutine by some dynamic descendant, the handler should call the find_condition_info_ subroutine to obtain the sub_error_info_ptr that points to the structure described in "Info Structure" below.

Info Structure

The structure pointed to by sub_error_info_ptr is declared as follows in the sub_error_info.incl.pl1 include file:

```
dcl 1 sub_error_info          aligned based,
      2 header                aligned like condition_info_header,
      2 retval                fixed bin(35),
      2 name                  char(32),
      2 info_ptr              ptr;
```

where:

1. header
   is a standard header required at the beginning of each information structure provided to an on unit. See "Information Header Format" in the MPM Reference Guide for further details.

2. retval
   is the return value. The standard environment sets this value to zero.

3. name
   is the name of the module encountering the condition.

4. info_ptr
   is a pointer to additional information associated with the condition.

The handler should check sub_err_info.name and sub_err_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: sus_signal_handler_

The sus_signal_handler_ subroutine is for use as the static condition handler for the sus_ condition. The standard process overseers establish this handler by calling sct_manager_$set. For interactive processes, the sus_ condition typically occurs when the process is disconnected from its login terminal channel. For absentee processes, the sus_ condition occurs when the operators suspend the job.

When the user reconnects to the process, sus_signal_handler_ may attempt to execute an exec_com, according to whether reconnect_ec_enable or reconnect_ec_disable was last called before disconnection.

Entry: sus_signal_handler_$reconnect_ec_enable

This entry point enables searching for the segment reconnect.ec when the user reconnects to a disconnected process. As a result, sus_signal_handler_ looks first in the user's home directory, then in his project directory (>user_dir_dir>Project_name), and finally in >system_control_dir. When the reconnect.ec segment is found, the command "exec_com >Directory_name>reconnect" is executed.

Usage

```
declare sus_signal_handler_$reconnect_ec_enable entry;

call sus_signal_handler_$reconnect_ec_enable ();
```

Notes

The use of reconnect.ec is enabled automatically by the standard process overseer process_overseer_.

Invocation of the reconnect.ec is not automatically enabled by the project_start_up_ process overseer. Thus, when using project_start_up_, the project administrator may enable the invocation of reconnect.ec at any point in the project_start_up.ec by using the reconnect_ec_enable command (See MPM Commands).

The current command processor is used to execute the reconnect.ec command. If the user is using the abbrev command processor, any applicable abbreviation will be expanded.

Entry: sus_signal_handler_$reconnect_ec_disable

This entry point reverses the effect of the sus_signal_handler_$reconnect_ec_enable entry. After reconnection to a disconnected process, there is no attempt made to find or invoke the exec_com "reconnect.ec".

Usage

```
declare sus_signal_handler_$reconnect_ec_disable entry;

call sus_signal_handler_$reconnect_ec_disable ();
```

Name: 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 how_many_users command (described in the MPM Commands) when the -long control argument is specified.

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 when 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 when the system will return, if these data are available.

Usage

```
declare system_info_$next_shutdown entry (fixed bin(71), char(*),
     fixed bin(71));

call system_info_$next_shutdown (td, rsn, tn);
```

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 (dev_ptr) 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 for the device.

Entry: system_info_$resource_price

This entry point returns the price of a specified resource.

Usage

```
declare system_info_$resource_price entry (char(*), float bin,
     fixed bin (35));

call system_info_$resource_price entry (name, price, code);
```

where:

1. name (Input)
   is the name of the resource.

2. price (Output)
   is the price of the resource in dollars per unit.

3. code (Output)
   is a standard status code. It will be error_table_$noentry if the resource is not in the price list.

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_$rs_name

This entry point returns the rate structure name corresponding to a rate structure number.

Usage

```
declare system_info_$rs_name entry (fixed bin(17), char(*), fixed bin(35));

call system_info_$rs_name (rs_number, rs_name, code);
```

where:

1. rs_number (Input)
   is the number of a rate structure.

2. rs_name (Output)
   is the name corresponding to rs_number. (The name can be up to 32 characters long.)

3. code (Output)
   is zero if no error occurred, or error_table_$no_entry if rs_number is not the number of a defined rate structure.

Entry: system_info_$rs_number

This entry point returns the rate structure number corresponding to a rate structure name.

Usage

```
declare system_info_$rs_number entry (char(*), fixed bin(17),
     fixed bin(35));

call system_info_$rs_number (rs_name, rs_number, code);
```

where:

1. rs_name (Input)
   is the name of a rate structure.

2. rs_number (Output)
   is the number corresponding to rs_name.

3. code (Output)
   is zero if no error occurred, or error_table_$no_entry if rs_name is not the name of a rate structure.

Entry: system_info_$max_rs_number

This entry point returns the largest valid rate structure number.

Usage

```
declare system_info_$max_rs_number entry (fixed bin(17);

call system_info_$max_rs_number (rs_number);
```

where:

1. rs_number (Output)
   is the largest valid rate structure number. If it is zero, there are no rate structures defined, other than the default one in installation_parms.

Entry: system_info_$default_absentee_queue

This entry point returns the number of the default absentee queue used for submission of absentee jobs by the enter_abs request, pl1_abs, fortran_abs, etc., commands.

Usage

```
declare system_info_$default_absentee_queue entry (fixed bin);

call system_info_$default_absentee_queue (default_q);
```

where:

1. default_q (Output)
   is the default absentee queue.

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: terminate_process_

This procedure causes the process in which it is called to be terminated. The arguments determine the exact nature of the termination.

Usage

```
declare terminate_process_ entry (char(*), ptr);

call terminate_process_ (action, info_ptr);
```

where:

1. action (Input)
   specifies one of four general actions to be taken upon process termination. The permissible values are logout, new_proc, fatal_error, or init_error (see "Notes").

2. info_ptr (Input)
   points to more specific information about the action to be taken at termination. The structure pointed to by info_ptr depends upon action (see "Notes").

Notes

If action is logout then the user's process is logged out. The info_ptr points to:

```
dcl 1 logout_info    aligned,
      2 version      fixed bin,
      2 hold         bit(1) unaligned,
      2 brief        bit(1) unaligned,
      2 pad          bit(34) unaligned,
```

where:

1. version
   must be 0.

2. hold
   must be "1"b if the terminal associated with this process is not to be hung up, so that another user may log in.

3. brief
   must be "1"b if the logout message is to be suppressed.

4. pad
   must be "0"b.

If action is new_proc, then the user's current process is logged out and a new process is created. The info_ptr points to:

```
dcl 1 new_proc_info               aligned,
      2 version                   fixed bin,
      2 authorization_option      bit(1) unaligned,
      2 pad                       bit(35) unaligned,
      2 new_authorization         bit(72) aligned;
```

where:

1. version
   must be 1.

2. authorization_option
   must be 1 if new_authorization is to be used.

3. pad
   must be 0.

4. new_authorization
   is the authorization of the new process.

If action is fatal_error, then the user's current process is terminated due to an unrecoverable error. A fatal error message is printed on the terminal and a new process is created. The info_ptr points to:

```
dcl 1 fatal_error_info      aligned,
      2 version             fixed bin,
      2 status_code         fixed bin(35);
```

where:

1. version
   must be 0.

2. status_code
   is an error_table_code indicating the nature of the fatal error, the corresponding error message will be printed on the user's console.

If action is init_error, then the user's process is logged out and a message indicating that his process could not be initialized is printed. The info_ptr points to:

```
dcl 1 init_error_info      aligned,
      2 version            fixed bin,
      2 status_code        fixed bin(35);
```

where:

1. version
   must be 0.

2. status_code
   is a standard Multics code indicating the nature of the error.

See the MPM Commands for a description of the logout and new_proc commands.

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. For example, if a program does run asynchronously within a process and it does input or output with the tty_ I/O module, then the program should issue the "start" control order of tty_ before it returns. This is necessary because a wakeup from tty_ may be intercepted by the asynchronous program. Most pitfalls can be avoided by using only the timer_manager_$sleep entry point.

For most uses of the timer_manager_ subroutine, a cleanup condition handler, which resets all the timers that might be set by a software subsystem, should be set up. If the subsystem is aborted and released, any timers set up by the subsystem can be reset instead of going off at undesired times.

To be used, the timer_manager_ subroutine must be established as the condition handler for the alrm and cput conditions. This is done automatically by the standard Multics environment.

Generic Arguments

At least one of the following arguments is called in all of the timer_manager_ entry points. For convenience, these common arguments are described below rather than in each entry point description.

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 entry value must be valid when the routine is invoked, i.e., if the routine is an internal procedure, the proceudre that created the entry value must still be on the stack. 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.

Before the routine is called, a condition wall is established. The wall is established with the following statement:

```
on any_other system;
```

See the MPM Reference Guide and the Multics PL/1 Reference Manual (AM83) for more information. Any conditions signalled in the routine are handled by default_error_handler_ if the routine does not handle them. They are not handled by user condition handlers on the stack above the call to the 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. If the segment existed before the call to tssi_$get_segment, the ACL is reset to the way it was before the tssi_$get_segment entry point was called. If no ACL existed for the current user, the mode is set to "mode" for the current user. If the segment was created, and the "mode" parameter contains the "e" mode, all entries on the segment's ACL (as derived from the containing directory's Initial ACL) receive the "e" bit, as well as the other modes specified. The current user, if not specified on the Initial ACL, receives an ACL term of "mode" on the segment. Otherwise, the segment's Initial ACL is restored, and, if the current user does not have an ACL term, the segment receives an ACL term of "mode" for the user.

Usage

```
declare tssi_$finish_segment entry (ptr, fixed bin(24), bit(36) aligned,
     ptr, fixed bin(35));

call tssi_$finish_segment (seg_ptr, bc, mode, aclinfo_ptr, code);
```

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: 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: valid_decimal_

The valid_decimal_ function tests decimal data for validity.

Usage

```
declare valid_decimal_ entry (fixed bin, ptr, fixed bin) returns (bit(1));

b = valid_decimal_ (dtype, dptr, dprec);
```

where:

1. dtype (Input)
   is the data type descriptor of the decimal data. It must be one of the following: 9-12, 29:30, 35-36, 38-39, 41-46.

2. dptr (Input)
   is a pointer to the data to be tested for validity.

3. dprec (Input)
   is the precision of the data.

4. b (Output)
   is the value returned by valid_decimal_. It is "1"b if the data is valid, "0"b otherwise.

Notes

For decimal data to be valid, it must pass the following tests: (1) The precision must be > 0 and <= 59; (2) The data type descriptor must be one handled by valid_decimal_; (3) If the data is stored as nonoverpunched 9-bit characters, then if it has a sign, then the sign must be either "+" or "-". The digits must all be one of the ASCII characters "0123456789"; (4) If the data is stored as overpunched 9-bit characters, then the sign character must be either octal 173, 175, or in the range 101 to 122. The remaining digits must all be one of the ASCII characters "0123456789"; (5) If the data is stored as 4-bit characters, then if it has a sign, then sign must be in the range "1010"b to "1111"b. All digits must be in the range "0000"b to "1001"b.

Name: 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 6 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 8

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$clock_                       bit(3) aligned external static;
dcl 1 sys_info$ips_mask_data              aligned external static,
      2 count                             fixed binary,
      2 masks (sys_info$ips_mask_data.count),
        3 name                            char(32) aligned,
        3 mask                            bit(35)aligned;
dcl sys_info$page_size                    fixed binary(19) external static;
dcl sys_info$max_seg_size                 fixed binary(19) external static;
dcl sys_info$default_stack_length         fixed binary(19) external static;
dcl sys_info$default_max_length           fixed binary(19) external static;
dcl sys_info$access_class_ceiling         bit(72) aligned external static;
dcl sys_info$time_correction_constant fixed binary(71) external static;
dcl sys_info$time_delta                   fixed binary(35) external static;
dcl sys_info$maxlinks                     fixed binary external static;
dcl sys_info$time_of_bootload             fixed binary(71) external static;
dcl sys_info$time_zone                    char(3) aligned external static;
```

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. name
   is the name used to signal the IPS condition.

5. mask
   is the IPS mask for the corresponding name. The mask has one bit on, and the rest of the bits are off.

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. default_max_length
   is the default maximum length of segments in words.

10. access_class_ceiling
    is the maximum access class.

11. time_correction_constant
    is the correction from Greenwich mean time (GMT) in microseconds.

12. time_delta
    is the same as time_correction_constant, only in single precision.

13. maxlinks
    is the maximum depth to which the system chases a link without finding a branch.

14. time_of_bootload
    is the clock reading at the time of bootload.

15. 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.
AST Atlantic Standard Time, 4 hours before GMT, including Carribean Islands.
ADT Atlantic Daylight Time, applies daylight savings to AST zone, giving time 3 hours before GMT.
BST British Summer Time, applies daylight savings to GMT zone, giving time 1 hour after GMT.
FWT French Winter Time, 1 hour after GMT, including Western Europe.
FST French Summer Time, applies daylight savings to FWT zone, giving time 2 hours after GMT.
HFH Heure Francais D'Hiver, the French representation of French Winter Time (FWT), giving time 1 hour after GMT.
HFE Heure Francais D'Ete, the French representation of French Summer Time (FST), giving time 2 hours after GMT.
Z Universal Time, an alternate name for 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

Appendix A, "Approved Control Arguments", has been deleted since the * information is available in the Standards System Designers' Notebook, Order No. AN82.

APPENDIX B

SYMBOL TABLE ORGANIZATION

The information in this section is subject to change. Future Multics releases may use a different format of runtime symbol information.

The free-format area can contain any information whatsoever, and the object segment will execute properly. However, the Multics debugging utilities (e.g., probe) place stringent requirements on the format of the free area, and these are followed by the translators for PL/I, FORTRAN and COBOL.

The free-format area begins with a fixed-format header, called the pl1_symbol_block. Despite the name, this block is present even in FORTRAN and COBOL-produced object segments. The pl1_symbol_block gives the options used in compiling the segment, and the offsets of the statement map, the root block node, and the profile information.

The remainder of the free-format area consists of the statement map, the symbol tree, and the profile information, which are discussed below.

The PL/I Symbol Block

The PL/I symbol block has the following format (declared in pl1_symbol_block.incl.pl1):

```
declare 1 pl1_symbol_block          aligned,
          2 version                 fixed binary,
          2 identifier              char(8),
          2 flags,
            3 profile               bit(1) unaligned,
            3 table                 bit(1) unaligned,
            3 map                   bit(1) unaligned,
            3 flow                  bit(1) unaligned,
            3 io                    bit(1) unaligned,
            3 table_removed         bit(1) unaligned,
            3 long_profile          bit(1) unaligned,
            3 pad                   bit(29) unaligned,
          2 greatest_severity       fixed binary,
          2 root                    bit(18) unaligned,
          2 profile                 bit(18) unaligned,
          2 map,
            3 first                 bit(18) unaligned,
            3 last                  bit(18) unaligned,
          2 segname,
            3 offset                bit(18) unaligned,
            3 length                bit(18) unaligned;
```

where:

1. version
   is the version number of the structure. For this version the version number is 1.

2. identifier
   is the constant "pl1info".

3. profile
   is "1"b if the object program contains an execution profile table. This table is generated if the -profile control argument is specified when the source program is compiled.

4. table
   is "1"b if the object program contains a runtime symbol table. A runtime symbol table is generated if the -table control argument is specified when the source program is compiled or if the runtime table is required by PL/I put data or get data or FORTRAN namelist input/output statements in the source program (see "The PL/I Runtime Symbol Table" below).

5. map
   is "1"b if the object segment contains a statement map that gives the correspondence between source line numbers and locations in the object segment (see "The Statement Map" below). The statement map is present if the -brief_table, -profile, or -table control arguments are specified when the source program is compiled.

6. flow
   is "1"b if the object program contains additional instructions for monitoring program flow. This facility is not yet available.

7. io
   is "1"b if the object program contains a runtime symbol table that is required by PL/I put data or get data or FORTRAN namelist input/output statements in the source program. In this case the runtime symbol table cannot be removed.

8. table_removed
   is "1"b if the object segment originally contained a runtime symbol table that has subsequently been removed.

9. long_profile
   is "1"b if the object segment contains a long profile table.

10. greatest_severity
    contains the greatest severity level of all error messages issued during the compilation of the source program. A value of 0 means that no errors were found during compilation.

11. root
    is nonzero only if the object segment contains a runtime symbol table; in this case, root is a pointer (relative to the base of the symbol header block) to the root block of the runtime symbol table.

12. profile
    is nonzero if the object segment contains a profile table. If it is nonzero, it is the offset in the linkage section of the table.

13. first
    is nonzero only if the object segment contains a statement map; in this case, first is a pointer (relative to the base of the symbol header block) to the first entry in the statement map.

14. last

    is nonzero only if the object segment contains a statement map; in this case, last is a pointer (relative to the base of the symbol header block) to the last entry in the statement map.

15. offset

    is a pointer (relative to the base of the symbol header block) to an aligned character string that gives the name of the segment; this is the same as the name used for the class 3 definition of the object segment.

16. size

    is the length of the segment name string.

The PL/I Runtime Symbol Table

The PL/I runtime symbol table contains information needed to support source language debugging and PL/I data-directed or FORTRAN namelist input/output statements. Most of the information that the compiler has in its compile-time symbol table is placed, in a different format, in the runtime symbol table; this permits attributes of a variable such as data type, storage class, or location to be determined during execution of the program. If the runtime symbol table is present, it follows the PL/I symbol block.

There are two types of runtime symbol tables: partial tables and full tables.

A partial table is generated when the source program contains data-directed input/output statements; it contains information only about variables that are transmitted via PL/I data-directed or FORTRAN namelist input/output statements. A partial runtime symbol table cannot be removed.

A full symbol table is generated if the table control argument is specified when the source program is compiled; it contains information about all variables, labels, and entries referenced by the source program. A full symbol table can be removed from the object program (when binding) if the source program does not contain data-directed input/output statements that would require a partial table to be generated.

The existence of a runtime symbol table does not affect the executable code normally generated by the compiler. There are no instructions that must routinely be executed by the object program in order to support the runtime symbol table. In some cases (described later), the compiler generates additional code sequences solely because a runtime symbol table is being created, but these extra instructions are not executed unless particular fields of the runtime symbol table are actually referenced.

An internal static variable that has an initial value and is never set is normally treated just as if it were a constant. If all references to the value of the internal static variable can be made using DU or DL modifiers in the instructions making the reference, the variable is not assigned a location. If all references cannot be made via DU or DL modifiers, the variable is assigned one or more locations in the text section. When a runtime symbol table is being generated, internal static variables that are initialized and never set are always assigned locations in the text section. This does not affect references to these variables since DU or DL modifiers continue to be used wherever possible.

The runtime symbol table is a list structure that consists of interconnected runtime_token, runtime_block, and runtime_symbol nodes. Normally, when node A in the runtime symbol table contains a pointer to node B, the pointer is relative to the start of the node in which it occurs; such a pointer is called a self-relative pointer. The format of the nodes in the runtime symbol table are described in the sections that follow.

THE RUNTIME_TOKEN NODE

The runtime_token node holds the name of an identifier used elsewhere in the runtime symbol table. The runtime_token nodes for all identifiers in the runtime symbol table are threaded together on a list that is ordered alphabetically by size (all 1 character names before all 2 character names, etc.); there are no duplicate names on this list. This ordering is used to increase the speed with which the runtime symbol table can be searched. Each runtime_token node contains a pointer to the runtime_symbol node for the first variable having the name stored in the runtime_token node. The runtime_token node has the following format (and appears in runtime_symbol.incl.pl1):

```
dcl 1 runtime_token     based aligned,
      2 next            bit(18) unaligned,
      2 dcl             bit(18) unaligned,
      2 name,
        3 size          fixed bin(9) unsigned unaligned,
        3 string        char(0 refer(runtime_token.size)) unaligned;
```

where:

1. next
   is a self-relative pointer to the next token on the alphabetic by size list of tokens. This field is zero in the last runtime_token node on the list.

2. dcl
   is a self-relative pointer to the runtime_symbol node for the first identifier having the name stored in this runtime_token node. This field is zero if there are no identifiers declared with this name.

3. name
   is an ACC string that gives the name of the identifier represented by this node (see "The Structure of the Definition Section" for a description of ACC strings).

THE RUNTIME_BLOCK NODE

Each procedure or begin block in the source program has a corresponding runtime_block node in the runtime symbol table. The manner in which these nodes are connected reflects the block structure of the source program. Each runtime_block node contains a pointer to a list of runtime_symbol nodes that represent declarations defined immediately internal to the block (i.e. internal to the block but not internal to any other block contained in the block). These declarations correspond to the variables and label or entry constants used in the block. The runtime_block node has the following format (which appears in runtime_symbol.incl.pl1):

```
dcl 1 runtime_block       aligned,
      2 flag              bit(1) unaligned,
      2 quick             bit(1) unaligned,
      2 fortran           bit(1) unaligned,
      2 standard          bit(1) unaligned,
      2 owner_flag        bit(1) unaligned,
```

```
2 skip                bit(1) unaligned,
2 type                bit(6) unaligned,
2 number              bit(6) unaligned,
2 start               bit(18) unaligned,
2 name                bit(18) unaligned,
2 brother             bit(18) unaligned,
2 father              bit(18) unaligned,
2 son                 bit(18) unaligned,
2 map,
  3 first             bit(18) unaligned,
  3 last              bit(18) unaligned,
2 entry_info          bit(18) unaligned,
2 header              bit(18) unaligned,
2 chain(4)            bit(18) unaligned,
2 token(0:5)          bit(18) unaligned,
2 owner               bit(18) unaligned;
```

where:

1. flag
   is always "1"b and is used to tell this version of the structure from an earlier one.

2. quick
   is "1"b if the procedure or begin block that corresponds to this runtime_block node is a quick block that does not have a stack frame of its own. By definition, when a quick block is called, pr6 (the stack pointer) points at the stack frame shared by the quick block in which the quick block allocates its storage. This bit is always "0"b in the runtime_block that corresponds to an external procedure.

3. fortran
   is "1"b if this program was compiled by the FORTRAN compiler. This bit is used to tell the programs that access the runtime symbol table that array elements are stored in column-major order instead of row-major order. The object program contains other places that indicate the compiler that processed the program; this bit was added to increase the speed with which this information could be obtained.

4. standard
   is "1"b if this object segment is in standard Multics format. Here, too, information that is available elsewhere is repeated for the sake of convenience.

5. owner_flag
   is "1"b if this block has a valid owner field.

6. skip
   is reserved for future expansion.

7. type
   is zero if this runtime_block node corresponds to a begin block. A nonzero value indicates that the runtime_block node corresponds to a procedure block.

8. number
   is used to number begin blocks. All begin blocks in the source program are assigned a sequence number in the order in which they are encountered by the program that generates the runtime symbol table.

9. start

is a self-relative pointer to the runtime_symbol node for the first declaration in the block represented by the runtime_block node. This declaration list gives all level 0 (nonstructure) and level 1 (top level structure) symbols defined immediately internal to the block; the runtime_symbol nodes on this list are ordered alphabetically by size. The start field is zero if there are no declarations in the block.

10. name

is a self-relative pointer to the ACC string that gives the name of the block; this field is zero for a begin block. The block compiled for an on-unit is a procedure block whose name is derived from the name of the condition, e.g. "overflow.1". For historical reasons, the name component points at runtime_token.name instead of the beginning of runtime_token.

11. brother

is a self-relative pointer to the next runtime_block node at the same nesting level. This field is zero if there is no other block at the same nesting level.

12. father

is a self-relative pointer to the immediately containing runtime_block node of which this block is a son. If the current block is the root of the symbol tree, this pointer points to the symbol header block.

13. son

is a self-relative pointer to the first runtime_block node contained within the current block. This field is zero if the current block does not contain any other blocks.

14. first

is nonzero if the object program contains a statement map; in this case first is a self-relative pointer to the entry in the statement map that corresponds to the first executable statement in this block. If block B is contained in block A, the entries in the statement map for block B are also contained in the statement map entries for block A.

15. last

is a self-relative pointer to the word after the entry that corresponds to the last executable statement. Note that zero is a meaningful value.

16. entry_info

is nonzero only for a runtime_block that corresponds to a procedure without its own stack frame (quick = "1"b). It gives the location in the stack frame shared by the quick block of the entry information block used by the quick block. The format of an entry information block is described below.

17. header

is a self-relative pointer to the start of the symbol header block.

18. chain

is a vector of self-relative pointers that point at runtime_symbol nodes on the declaration list for this block. The chain(i) points at the runtime_symbol node for the first declaration whose name is longer than 2**i; chain(i) is zero if the longest name in the declaration list is shorter than 2**i.

19. token
is a vector of self-relative pointers that point at runtime_token nodes. The token(i) points at the runtime_token node for the first name longer than 2**i; token(i) is zero if the longest name in the token list is shorter than 2**i.

20. owner
is a self-relative pointer to the runtime_block node whose stack frame will be shared by this block. This field is valid only if owner_flag is set.

THE ENTRY INFO BLOCK

An entry info block consists of one, two, or three pointers, depending on the procedure. It has the following format (declared in quick_entry.incl.pl1):

```
dcl 1 quick_entry          aligned,
      2 return             ptr,
      2 argptr             ptr,
      2 descptr            ptr;
```

where:

1. return
points at the return location of the quick block.

2. argptr
if present, points at the argument list of the quick block.

3. descptr
if present, points at the descriptor list of the quick procedure.

THE RUNTIME_SYMBOL NODE

Each runtime_symbol node in the runtime symbol table corresponds to an identifier in the source program. The manner in which these nodes are connected reflects the structural relationship of variables in the source program. Level 0 (nonstructure) and level 1 (top level structure) variables have the runtime_symbol nodes that correspond to them threaded on a list of runtime_symbol nodes ordered alphabetically by size.

The format of the runtime_symbol node is (declared in runtime_symbol.incl.pl1):

```
dcl 1 runtime_symbol       aligned,
      2 flag               bit(1) unaligned,
      2 use_digit          bit(1) unaligned,
      2 array_units        bit(2) unaligned,
      2 units              bit(2) unaligned,
      2 type               bit(6) unaligned,
      2 level              bit(6) unaligned,
      2 ndims              bit(6) unaligned,
      2 bits               unaligned,
        3 aligned          bit(1),
        3 packed           bit(1),
        3 simple           bit(1),
        3 decimal          bit(1),
      2 scale              bit(8) unaligned,
      2 name               bit(18) unaligned,
      2 brother            bit(18) unaligned,
```

```
2 father            bit(18) unaligned,
2 son               bit(18) unaligned,
2 address           unaligned,
  3 location        bit(18),
  3 class           bit(4),
  3 next            bit(14),
2 size              fixed binary(35),
2 offset            fixed binary(35),
2 virtual_org       fixed binary(35),
2 bounds(1),
  3 lower           fixed binary(35),
  3 upper           fixed binary(35),
  3 multiplier      fixed binary(35);
```

In the discussion that follows, the term "current identifier" means the indentifier represented by the runtime_symbol node under consideration, and the term "current block" means the block in which the current identifier is declared:

1. flag

   is always "1"b and distinguishes this version of the structure from an earlier one.

2. use_digit

   contains the most significant bit of the three bit binary integers that identify the addressing units for arrays and offsets.

3. array_units

   contains the low order two bits of a three bit positive binary integer that gives the addressing units to be used when computing the address of a subscripted array element; this field is meaningful only when ndims is not zero. The high order bit is supplied by the use_digit bit. The possible values for this three bit number, and the corresponding factor by which an offset should be multiplied to convert to a bit offset are:

units	factor
0 word	36
1 bit	1
2 byte	9
3 half word	18
4 word	36
5 bit	1
6 byte	9
7 digit	4.5

4. units

   contains the low order two bits of a positive binary integer that gives the addressing units of the offset field in the runtime_symbol node. The high order bit is supplied by use_digit. The possible values and associated conversion factors are the same as for array_units.

5. type

   contains a positive binary integer that gives the data type of the current identifier. The numeric values used to encode the data type are the same as the values used in the Multics descriptor, supplemented with additional values. See Appendix D of the MPM Reference Guide.

   When the identifier is a pictured variable, the real data type is given by the picture information block, which can be found by using information in the size field of the runtime_symbol node.

6. level

contains a positive binary integer that gives the structure nesting level of the current identifier as determined by the compiler; nonstructure variables have level = 0.

7. ndims

contains a positive binary integer that gives the number of array dimensions of the current identifier; a value of zero means the current identifier is not an array. The ndims gives the total number of subscripts that must be provided to access an element of the array and is the sum of the number of dimensions with which the identifier was explicitly declared and the number of dimensions inherited from a containing structure.

8. aligned

is "1"b if the current identifier is aligned and is "0"b if the identifier is unaligned.

9. packed

is "1"b if the current identifier is any one of the following: an unaligned aggregate of packed data, unaligned arithmetic data, unaligned nonvarying string data, or unaligned pointer data.

10. simple

is "1"b if an abbreviated form of the runtime_symbol node is being used for the current identifier; in this case fields after size in the runtime_symbol node are <u>not</u> present and the current identifier is a scalar with zero offset. If simple is "0"b, <u>all</u> fields in the runtime_symbol node are present.

11. decimal

is reserved for future expansion.

12. scale

is the arithmetic scale factor of the current identifier. Although stored in a bit (8), it is logically a fixed bin (7). Be warned that COBOL and PL/I both define negative scale factors, and that PL/I bit to fixed conversion assumes unsigned, not signed.

13. name

is a self-relative pointer to the ACC string that gives the name of the current identifier. For historical reasons, the name component points at runtime_token.name instead of the beginning of runtime_token.

14. brother

is a self-relative pointer to the runtime_symbol node for the next identifier at the same structure level; levels 0 and 1 are considered to be the same level. Within a structure (level > 1), brother points to the runtime_symbol node for the identifier that immediately follows the current identifier in the structure; brother is zero if the current identifier is the last element in the structure that immediately contains it. Outside of a structure (level <= 1), brother points to the next element on the list of runtime_symbol nodes ordered alphabetically by size.

15. father

is a self-relative pointer to either a runtime_block node or a runtime_symbol node. If level <= 1, father points to the runtime_block node that represents the block in which the current identifier is declared. If level > 1, father points to the runtime_symbol node for the structure that immediately contains the current identifier as a son.

16. son

is a self-relative pointer to the first son of a structure (the runtime_symbol node for the first identifier in the structure with a level number one greater than the level of the current identifier). This field is zero if the current identifier is not a structure.

17. location

usually contains a positive integer L that is used in combination with class to determine the address of the current identifier. L is normally an offset with respect to the start of a given class of storage; its interpretation depends on the value of the class field in the runtime_symbol node.

18. class

contains a positive binary integer that gives the storage class of the current identifier; the possible classes are:

class	storage class
1	automatic; L is the offset at which the current identifier is defined in the stack frame associated with the current block.
2	automatic adjustable; the address of the current identifier is not known at the time the runtime symbol table is created. Location L in the stack frame associated with the current block contains a pointer to the storage for the current identifier.
3	based; location is a self-relative pointer to the runtime_symbol for the pointer used in the declaration of the current identifier or is zero if a pointer was not specified. The user must provide a pointer, either explicitly at run time or implicitly through the default pointer, in order to reference the current identifier.
4	internal static; L is the offset at which the current identifier is assigned storage in the linkage section associated with the current block.
5	external static; L is the offset in the linkage section of a link that points to the current identifier.
6	internal controlled; L is the offset of the control block of the current identifier in the linkage section of the current block.
7	external controlled; L is the offset in the linkage section of a link that points to the control block for the current identifier.
8	parameter; at L in the stack frame corresponding to the current block there is a pointer to the storage for the current identifier. This storage class is used when the current identifier appears in more than one position in procedure and/or entry statements in the block.
9	parameter; L gives the position of the current identifier in the argument list provided to the current block. This class is used when the current identifier appears in the same position in every procedure or entry statement in the current block.
10	not used
11	not used
12	text reference; the current identifier is defined at L in the text section of the object segment.

13 link reference; the current identifier is defined at L in the linkage section corresponding to the current block.

14 not used

15 not used

19. next
is a self-relative pointer to the runtime_symbol node of the next identifier having the same name as the current identifier.

20. size
is the arithmetic precision, string size, or area size of the identifier. If the identifier is a string or area, it may be an encoded value. If the current identifier is a picture variable, size contains the offset at which the picture information block can be found in the text section of the object segment. If the current identifier is an offset variable, size is a self-relative pointer to the runtime_symbol node for the area, if any, associated with the current identifier.

21. offset
is the encoded value of the offset of the start of the current identifier with respect to the address specified by location and class. The units of the offset value are given by the units field in the runtime_symbol node. This field is not present, and its value is assumed to be zero, if the simple bit is "1"b.

22. virtual_org
is the encoded value of the virtual origin of an array, in units given by array_units. Its value should be subtracted from the base address specified by location and class. This field is not present, and the current identifier is a scalar, if the simple bit is "1"b.

23. bounds
is an array that gives information about each dimension of an array identifier, from left to right. The upper bound for the bounds array that appears in the declaration is actually a dummy; the true upper bound for the bounds array is given by the ndims field. All the fields in the bounds array are not present, and the current identifier is a scalar, if the simple bit is "1"b. A bound structure is declared in runtime_bound in runtime_symbol.incl.pl1.

24. lower
is the encoded value of the lower bound of this dimension of the current identifier.

25. upper
is the encoded value of the upper bound of this dimension of the current identifier.

26. multiplier
is the encoded value of the multiplier of this dimension of the current identifier.

The address of an identifier is calculated in the following manner. The base address is determined by the class and location fields. If the identifier is "simple", this is all. Otherwise, the offset field (which may be encoded) is multiplied by the conversion factor given by use_digit and units to give a bit offset, which is added to the base address. If the identifier is not an array element, that is all; otherwise, the virtual origin is computed (an encoded value converted to bits by the factor given by use_digit and array_units) and subtracted from the address. The array offset is computed by taking the dot product of the subscripts supplied and the multipliers for the identifier. The array offset is converted to a bit offset using the array_units conversion factor, and added to the address previously computed. This gives the final address of the data.

Encoded Values

The runtime_symbol node contains information about the attributes of an identifier. In many cases, the value of attributes such as string length, array bounds, or address cannot be determined at the time the runtime symbol table is created. For example, given the declaration

```
dcl x char(n+m);
```

the length of the variable x can be different each time the block in which it is declared is entered; the location of x is not known because a variable with nonconstant size is allocated when the block is entered. If x were declared instead:

```
dcl x char(n+m) based;
```

the length of x could be different at each reference.

The problem of representing nonconstant attributes values is handled by encoding the values that can be nonconstant. A field in the runtime_symbol node that can have a nonconstant value is called an <u>encoded value</u>; it is declared fixed binary(35) in the node declaration, but actually has the following format (declared in runtime_symbol.incl.pl1):

```
dcl 1 encoded_value          aligned,
      2 flag                 bit(2) unaligned,
      2 code                 bit(4) unaligned,
      2 (n1,n2)              bit(6) unaligned,
      2 n3                   bit(18) unaligned;
```

If flag ^= "10"b, the encoded value is the constant given in the entire word. If flag = "10"b, the positive binary integer contained in the code field determines the value as follows:

<u>Code</u>	<u>Value</u>
0	Value is the contents of the word at location n3 in the stack frame of the block n1 static levels before the block in which the declaration occurs.
1	Value is the contents of the word at location n3 in the linkage section of the block in which the declaration occurs.
2	Value is the contents of the word with positive offset n1 from the word pointed at by the link at location n3 in the linkage section of the block in which the declaration occurs.
3	Value is n3 plus the contents of the bit offset field of the pointer used to access the variable, which must be based. This encoding was only used by the compiler before version 2 EIS.

4 Value is the contents of the word with positive offset n2 based on the pointer at location n3 in the stack frame n1 static levels before the block in which the declaration occurs.

5 Value is the contents of the word with positive offset n2 based on the pointer at location n3 in the linkage section of the block in which the declaration occurs.

6 Value is the contents of the word with positive offset n2 based on the pointer with positive offset n1 from the word pointed at by the link at location n3 in the linkage section of the block in which the declaration occurs.

7 Value is the contents of the word with positive offset n2 based on the pointer used to access the variable, which must be based. This encoding is used for refer extents.

8 Value is the value returned by the internal procedure at location n3 in the text section of the block in which the declaration occurs. This procedure is compiled as if it were declared in the block in which the declaration occurs. This encoding is used whenever one of the other more specific encodings cannot be used. The calling sequence of this procedure is

```
dcl f entry(ptr) returns(fixed binary(24));
value = f(refp);
```

where refp is the pointer that could be used to access a based variable. Note that this procedure is never called by the executable code in the object program, it is used only by the programs that reference the runtime symbol table.

9 Value is the contents of the word with positive offset n3 from the start of argument n2 of the procedure n1 static levels before the block in which the declaration occurs.

10 Value is the contents of the word with positive offset n3 from the word pointed at by the pointer that is argument n2 of the procedure n1 static levels above the block in which the declaration occurs.

11 Value is the contents of the size field of descriptor n2 of the procedure n1 static levels before the block in which the declaration occurs.

12 Value is the contents of the word with positive offset n3 from the start of descriptor n2 of the procedure n1 static levels before the block in which the declaration occurs.

13 Value is the size field at positive offset n2 from the start of the descriptor for a controlled variable. For all encodings having to do with controlled variables, if n1 = 0 the variable is internal, if n1 = 1 it is external. For an internal controlled variable a pointer to the descriptor (control_block.descriptor) is located at n3 in the static secion. For an external variable, a ptr to the descriptor ptr is at n3 in the linkage section.

14 Value is the contents of the word with positive offset n2 from the start of the descriptor for a controlled variable. The descriptor is located in the same manner used for type 13 encoding.

15 Value is the contents of the word with positive offset n2 from the start of a controlled variable. If n1 = 0 the controlled variable is internal and its control block is located at n3 in the linkage section of the block in which the declaration occurs. If n1 = 1 the controlled variable is external and location n3 in the linkage section of the block in which the declaration occurs contains a pointer to the control block. The data itself is found using the data pointer of the controlled variable control block.

Controlled Variable Control Block

The format of the control block for a controlled variable is given in ctl_block.incl.pl1:

```
declare 1 control_block  aligned,
          2 data         ptr,
          2 descriptor   ptr,
          2 previous     ptr;
```

where:

1. data
   points at the current generation of the controlled variable. It is null if the controlled variable does not have a current generation.

2. descriptor
   points at the descriptor for the current generation of the controlled variable.

3. previous
   points at the control block of the previous generation of the controlled variable. It is null or points to a null ptr if there is no previous generation.

Picture Information Block

A picture variable of any type is stored in edited form as a character string. Each picture variable has an "associated value" that gives the value of the picture variable in internal form, either as a character string or as a decimal number. When the current identifier is a picture variable, the size field in the runtime_symbol node specifies the location of the picture information block, whose format is (declared in picture_image.incl.pl1):

```
dcl 1 picture_info   based aligned,
      2 type         fixed binary(8) unaligned,
      2 prec         fixed binary(8) unaligned,
      2 scale        fixed binary(8) unaligned,
      2 piclength    fixed binary(8) unaligned,
      2 varlength    fixed binary(8) unaligned,
      2 scalefactor  fixed binary(8) unaligned,
      2 explength    fixed binary(8) unaligned,
      2 drift        char(1) unaligned,
      2 chars        char(0 refer(picture_info.piclength)) aligned;
```

where:

1. type

   is the true data type of the current identifier according to the following encoding:

type	data type	named constants in picture_image.incl.pl1
24	character string	picture_char_type
25	real fixed decimal	picture_realfix_type
26	complex fixed decimal	picture_complexfit_type
27	real float decimal	picture_realflo_type
28	complex float decimal	picture_complexflo_type

2. prec

   is the arithmetic precision or string length of the associated value. Note that the length of a character picture variable must be constant.

3. scale

   for arithmetic picture variables is the number of digits, if any, after the "v" in the picture constant minus scale factor (see below).

4. piclength

   is the length of the normalized picture constant string.

5. varlength

   is the length of the edited form of the picture variable in characters. Note that the length of a picture variable must be constant.

6. scalefactor

   is the picture scale factor.

7. explength

   is the length in characters of the exponent field of a floating point picture variable.

8. drift

   is the picture drifting character. It is blank if the picture constant does not specify a drifting field.

9. chars

   is the normalized picture constant.

SPECIAL RUNTIME SYMBOL DATA TYPE CODES

type	data type
24	label constant (used in symbol tables only)
25	internal entry constant (used in symbol tables only)
26	external entry constant (used in symbol tables only)
27	external procedure (used in symbol tables only)
63	picture (used in symbol tables only)

These types are used in runtime_symbol values only, and not in argument descriptors. The user is referred to std_descriptor_types.incl.pl1, which gives named constants for these codes. See Appendix D of the MPM Reference Guide for more information.

The Statement Map

The statement map contains information about each statement in the source program for which instructions were generated. The statement map is normally placed after the runtime symbol table, if the table is present. All the entries are contiguous. Each entry in the statement map has the following format (declared in statement_map.incl.pl1):

```
dcl 1 statement_map      aligned based,
      2 location         bit(18) unaligned,
      2 source_id        unaligned,
        3 file           bit(8),
        3 line           bit(14),
        3 statement      bit(5),
      2 source_info      unaligned,
        3 start          bit(18),
        3 length         bit(9);
```

where:

1. location
   is location in the object segment of the first instruction generated for the statement that corresponds to this entry in the statement map.

2. source_id
   describes the line on which the statement begins. The last entry in the statement map is a dummy that has string(source_id) = (27)"1"b.

3. file
   contains a positive binary integer that specifies the number of the source segment in which the current statement is contained (see "The Source Map").

4. line
   contains a positive binary integer that specifies the number of the line on which the current statement begins. The first line in a file is number 1.

5. statement
   contains a positive binary integer that specifies the position of the current statement on the line in which it begins. The first statement on a line is number 1.

6. source_info
   specifies the starting position and length of the string of characters that are the source for the current statement.

7. start
   contains a positive binary integer S that specifies the number of characters that precede the first character of the source of the current statement (see below).

8. length
   contains a positive binary integer L that gives the number of characters occupied by the current statement in the source file; a statement is assumed to be entirely contained in a single segment. If string is the contents of the source file that contains the current statement considered as a single string, the source string for the current statement is substr(string,S+1,L).

INDEX

D

E

S

T

U

V

W

HONEYWELL INFORMATION SYSTEMS

Technical Publications Remarks Form

CUT ALONG LINE

TITLE	SERIES 60 (LEVEL 68) MULTICS PROGRAMMERS' MANUAL - SUBSYSTEM WRITERS' GUIDE	ORDER NO.	AK92, REV. 2
		DATED	MARCH 1979

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 ______________________

____

PLEASE FOLD AND TAPE—
NOTE: U. S. Postal Service will not deliver stapled forms

CUT ALONG LINE

FOLD ALONG LINE

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 39531 WALTHAM, MA02154

POSTAGE WILL BE PAID BY ADDRESSEE

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTN: PUBLICATIONS, MS486

FOLD ALONG LINE

Honeywell

Honeywell

Honeywell Information Systems
In the U.S.A.: 200 Smith Street, MS 486, Waltham, Massachusetts 02154
In Canada: 2025 Sheppard Avenue East, Willowdale, Ontario M2J 1W5
In the U.K.: Great West Road, Brentford, Middlesex TW8 9DH
In Australia: 124 Walker Street, North Sydney, N.S.W. 2060
In Mexico: Avenida Nuevo Leon 250, Mexico 11, D.F.

27286, 7.5C480, Printed in U.S.A.

AK92, Rev. 2