Honeywell

# MESSAGE SEGMENT FACILITY PROGRAM LOGIC MANUAL

SERIES 60 (LEVEL 68) MULTICS

RESTRICTED DISTRIBUTION

SUBJECT:

Internal and User Interfaces of the Message Segment Facility.

SPECIAL INSTRUCTIONS:

This Program Logic Manual (PLM) describes certain internal modules constituting the Multics System. It is intended as a reference for only those who are thoroughly familiar with the implementation details of the Multics operating system; interfaces described herein should not be used by application programmers or subsystem writers; such programmers and writers are concerned with the external interfaces only. The external interfaces are described in the Multics Programmers' Manual, Commands and Active Functions (Order No. AG92), Subroutines (Order No. AG93), and Subsystem Writers' Guide (Order No. AK92).

As Multics evolves, Honeywell will add, delete, and modify module descriptions in subsequent PLM updates. Honeywell does not ensure that the internal functions and internal module interfaces will remain compatible with previous versions.

This PLM is one of a set, which when complete, will supersede the System Programmers' Supplement to the Multics Programmers' Manual (Order No. AK96).



DATE:

February 1975

ORDER NUMBER:

AN69, Rev. 0

# PREFACE

Multics Program Logic Manuals (PLMs) are intended for use by Multics system maintenance personnel, development personnel, and others who are thoroughly familiar with Multics internal system operation. They are not intended for application programmers or subsystem writers.

The PLMs contain descriptions of modules that serve as internal interfaces and perform special system functions. These documents do not describe external interfaces, which are used by application and system programmers.

Since internal interfaces are added, deleted, and modified as design improvements are introduced, Honeywell does not ensure that the internal functions and internal module interfaces will remain compatible with previous versions. To help maintain accurate PLM documentation, Honeywell publishes a special status bulletin containing a list of the PLMs currently available and identifying updates to existing PLMs. This status bulletin is distributed automatically to all holders of the System Programmers' Supplement to the Multics Programmers' Manual (Order No. AK96) and to others on request. To get on the mailing list for this status bulletin, write to:

Large Systems Sales Support
Multics Project Office
Honeywell Information Systems Inc.
Post Office Box 6000 (MS A-85)
Phoenix, Arizona 85005



File No.: 2L13

AN69

CONTENTS

CONTENTS (cont)

Page

ILLUSTRATIONS

# SECTION I

# OVERVIEW

The message segment facility enables the user to create and delete repositories for messages. It can perform a set of operations on the repository and control access to it in a unique way called extended access.

The message segment facility is layered. It consists of a primitive facility on which software for specific applications can be built. The subsystem programmer can define a class of repository and a subset of the available operations.

The message repository used by the facility is an inner ring segment that will be referred to as a message segment.

# SECTION II

# DESIGN GOALS

The message segment facility is designed to:

1. Provide for the protected and ordered exchange of messages between processes and within processes.

2. Provide a partitioned facility to allow various higher level applications.

3. Be able to restore its message data to a consistent form so that it continues functioning if a message is damaged.

4. Make the time during which any primitive operation puts the message data in an inconsistent state as small as possible, so as to minimize the chance that it will be left in this state by crash or malfunction.

5. Allow the size of the message data storage unit to be changed easily so that the most efficient unit size can be determined and used.

6. Be executable in the user ring for debugging purposes.

## REQUIREMENTS

The requirements of the message segment facility are threefold. It must have one higher level subsystem to provide a queue facility for the I/O and Absentee Daemons. This queue facility must perform the operations listed below:

1. Enter a request so that its order is retained in the queue.

2. Obtain a request in some ordered fashion from the queue.

3. Delete a request from the queue.

4. Rewrite the request within the queue.

5. Obtain certain status concerning the queue.

This higher level queue facility must also provide the appropriate access control that allows users of the facility to enter, obtain, and cancel their own requests without being able to tamper with requests of other users.

There must be a second higher level subsystem to support the mail and send_message facilities. Within this subsystem the user must be able to:

1. Add, read, and delete mail and messages in an ordered fashion.

2. Send, obtain, and delete mail and messages without affecting mail and messages sent by other users.

3. Allow and defer messages.

The message segment facility must also provide the user with the ability to specify other types of message segments and to design high level subsystems to implement these types of message segments.

DESIGN

Use of extended access requires that the message repository and the procedures that access it are in an inner ring restricted to system use. Ring 1 has been reserved for this purpose. All message segments are bracketed 1,1,1.

The internal consistency of a message segment is protected as follows:

1. The sequence of operations that leave the message segment in an inconsistent state is short to minimize the probability of being interrupted by system failure.

2. Quits are inhibited during this sequence so that the user cannot interrupt it.

3. A switch is turned on while the sequence is executing. In the rare case when system failure or an unrecoverable error has left the message segment inconsistent, this switch will be on.

4. There is a salvager, called by any message operation when the switch is found to be on, that is capable of restoring the message segment to a usable state. This salvager cannot be responsible for the text portion of a message because the text does not follow any particular rules.

The choice of what message operations should be available is influenced by the need to make each one brief. The read operations in particular are affected by this consideration. If a message segment contains messages in threaded form and a user wishes to perform a read operation that involves chasing the threads (i.e., read the fifth message, or read all messages with a given property), then one call can take considerable time. To avoid this problem, only the following read operations exist:

1. Read the first message.

2. Read the last message.

3. Read the next message.

4. Read the previous message.

The caller thus has the ability, by combining calls, to find any desired message. The chasing of message threads is done in the user ring.

The implementation of these read operations poses a problem with respect to access control. Assume a user wishes to read a message from the message segment but has access to read only his own messages and no access to obtain status information such as the number of messages in the message segment. If a read operation returned nothing when it encountered a message that the user does not have access to read, the user could find out how many messages there are and which ones are his. To avoid this, the read first operation should, for example, return the first message that the user has access to.

A set of own read operations are provided for this purpose. These operations are implemented a level above the actual read operations in the administrative ring. They invoke the appropriate read operations repetitively until they read a message sent by the caller and then return that message. Although they are in the administrative ring, they do not access the repository directly and therefore can be interrupted.

To facilitate adding new types of message segments to the system, the primitives are partitioned. All code specific to one type of message segment is placed in a separate module. Each module performs the appropriate checks for an individual type of message segment and then passes the call on to the primitive message facility. The relationship can be diagrammed as follows. Solid arrows represent the flow of control between modules. Notice that control can pass from module I downward to either the own or nonown entries in the primitive facility.

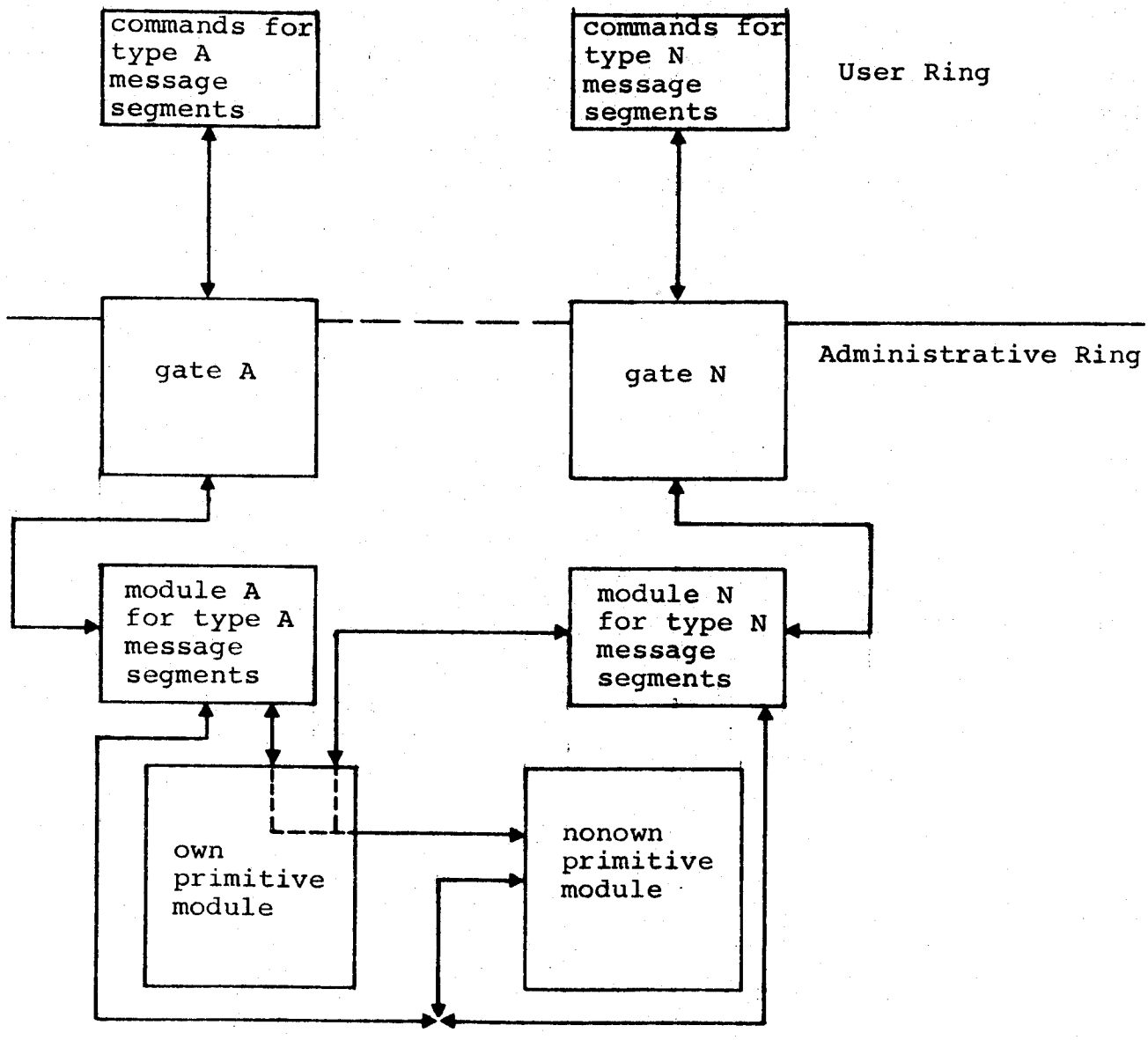


Figure 2-1. Sample Message Segment Facility Configuration

# SECTION III

# PROPERTIES AND STRUCTURE

## PROPERTIES

A message segment is accessible only in the administrative ring. It has a suffix name defined for some type of message segment. A subset of the extended access bits in its ACL entries is defined for that type of message segment. These bits are set by the administrative ring software and used by that software to determine access to the message segment. A message segment may not be a multisegment file.

## STRUCTURE

### Message Segment Structure

The first block of words in the message segment is the message segment header. It contains:

1. A lock (a word set by the standard Multics locking and unlocking operations).

2. An identification bit pattern checked by the message segment primitive to help ensure the identity of the message segment.

3. An offset to the beginning of the first message in the message segment.

4. An offset to the beginning of the last message in the message segment.

5. A count of the number of messages in the message segment.

6. A version number.

7. A block of words for a special header message.

8. A count of the number of bits in this header message if the header message is present.

9. Switches:

   a. A switch that is on whenever the message segment is being manipulated by the primitives and is in an inconsistent state.

   b. A switch that is on whenever the message segment has been restored by the message segment salvager.

   c. A switch that is on whenever a special message is present in the message segment header.

To maintain the ability to vary the size of the block in which messages are stored, the following data is also kept:

1. An allocation bit string where each bit represents a block of words in the message segment. A particular bit is turned on when the block corresponding to it is in use.

2. A number indicating the length of the allocation bit string.

3. A number indicating the size of a message block.

4. A count of the number of unused blocks in the message segment.

The remainder of the message segment consists of messages. Each message is threaded to the next and previous message with a forward and backward thread. A message consists of one or more message blocks. Each block of a message is threaded to the next block of that message with a forward thread.

## Message Block Structure

The first portion of every message block contains the following header information:

1. An offset to the next block in the message. If the current block in the message is the last block, this value is zero.

2. A first block switch that is on if the current block is the first block in the message.

3. A count of the number of message bits used in the block.

Following the header is all or part of a message. Present only in the first block of each message is a message block trailer, consisting of:

1. A unique bit pattern used to identify the beginning of the trailer when dumping message segments.

2. An offset to the first block of the next message in the message segment. If the current message is the last message in the segment, this value is zero.

3. An offset to the first block of the previous message in the message segment. If the current message is the first message in the segment, this value is zero.

4. The size, in bits, of the current message.

5. The time the message was sent.

6. The validation level of the sender of the message.

7. The Person_id.Project_id identification of the sender of the message.

The include files that describe the message segment header, the message header, and the message trailer are in Appendix A.

## SECTION IV

## DETAILS OF THE PRIMITIVE MESSAGE SEGMENT FACILITY

The primitive message segment facility consists of the following modules:

1. A module to create and delete message segments.

2. A module to open and close message segments. These operations are analogous to initiating and terminating a segment. When the message segment is opened, an index is returned. This index is used to refer to the message segment in subsequent operations.

3. A module to manipulate the extended access of a message segment. This module contains entries analogous to the current ACL primitive entries.

4. A module to lock and unlock a message segment.

5. A module to manipulate messages.

6. A module to manipulate own messages.

7. A module to salvage a message segment.

8. A data base containing constants relevant to the operation of the message segment primitives.

9. A module to convert a message segment from a previous format.

10. A metering data base.

Following is a description of each module. None of these modules can be called from the user ring. Each is

called by the administrative ring module that defines a particular type of message segment. References to ring number are implemented in such a way that if the user has appropriate access to certain system procedures, he can execute the primitives in any ring.

## CREATING AND DELETING MESSAGE SEGMENTS (msu_)

This procedure has two entries: one to create a message segment and one to delete a message segment. Both entries accept pathname and entryname arguments. The create entry accepts an extended access bit string argument representing full extended access privileges for the given type of message segment to be created.

Creating a message segment consists of:

1. Setting the validation level to the called ring.

2. Creating the message segment with rew real access and null extended access for *.*.*.

3. Adding the creator to the ACL with rew real access and full extended access.

This particular sequence is followed to ensure that no window to access the segment occurs during creation. The segment is created first with null access and then access is added.

Deleting a message segment consists of:

1. Setting the validation level to the called ring.

2. Verifying that the segment is a message segment. This is done by checking the name of the segment and its ring brackets when it is initiated.

3. Checking that the caller has appropriate extended access to delete the message segment, currently delete extended access (d, the second extended access bit, must be on). This access check serves also as a further identity check.

4. Setting up a cleanup handler to turn off guaranteed eligibility. This step is needed because the locking procedure called next turns on guaranteed eligibility. If a crawlout occurs in the locking procedure, a return

to the user ring with guaranteed eligibility on must be prevented.

5. Locking the message segment.

6. Deleting the message segment.

7. Turning off guaranteed eligibility.

8. Resetting the validation level to the calling ring.

## OPENING AND CLOSING MESSAGE SEGMENTS (mseg_index_)

It is a violation of access control conventions to initiate a message segment in an inner ring and pass a pointer to it across rings. Therefore a message segment is opened and closed rather than initiated and terminated. This procedure opens and closes message segments and maintains the correspondence between inner ring pointers to message segments and the indices used in outer ring references to message segments.

The procedure mseg_index_ contains an entry to open a message segment, an entry to close a message segment, and an entry to return a pointer to a message segment given its index. The last is used by the other primitive functions and is not available in the user ring.

## MANIPULATING EXTENDED ACCESS (mseg_access_)

This procedure is used to manipulate and to read extended access. It contains entries corresponding to the hardcore ACL primitive entries. Calls are mapped and passed to the hardcore entries.

The procedure mseg_access_ also ensures that at least two entries are maintained on the Access Control List (ACL) of the message segment. A call to delete all the entries from the ACL of a message segment is mapped by this procedure into a call to replace the ACL with two entries, one for *.*.* and one for *.SysDaemon.*, with rew real access and null extended access. These ACL entries must be maintained so that the primitives themselves will never be denied real access to the message segments and backup will always be able to dump and reload message segments.

## LOCKING AND UNLOCKING MESSAGE SEGMENTS (ring_1_lock_)

This procedure locks and unlocks message segments to prevent simultaneous access by more than one process. It makes use of the standard file system locking mechanism.

This procedure must be able to validate an existing lock. If a process terminates while it has a message segment locked, the Multics salvager does not unlock the message segment. Therefore, when this locking procedure encounters a locked message segment, it makes a call to a hardcore procedure, which checks the process ID in the lock. If the process ID does not correspond to an existing process, the hardcore procedure relocks the message segment with the process ID of the caller.

ring_1_lock_ makes a specified number of attempts at locking a message segment. After each unsuccessful attempt, it goes blocked for a short while before trying again.

Unlocking is accomplished with the standard unlocking instruction.

ring_1_lock_ turns on guaranteed eligibility when it locks a message segment and turns off guaranteed eligibility when it unlocks a message segment. Cleanup handlers exist to ensure that guaranteed eligibility is always off on an abnormal return.

## MANIPULATING MESSAGES

The module that actually manipulates messages is composed of three procedures: a procedure that checks access and dispatches calls; a procedure that reads and deletes messages; and a procedure that adds messages.

### Checking Access and Dispatching Calls (mseg_)

This procedure is the heart of the message segment facility. It is the lowest level primitive through which all calls to access the message segment are routed. It does the following:

1. Gets a pointer to the message segment.

2. Checks the identity of the segment being referenced.

3. Checks the user's access to the message segment and rejects the call if access is insufficient.

4. Locks the message segment.

5. Converts the message if it is not in the current format.

6. Establishes a cleanup handler to invoke the message segment salvager, return appropriate arguments to the caller, and turn off guaranteed eligibility.

7. If the call is one that reads or sets data in the header, performs the required action.

8. If the call is to read or delete a message, dispatches it to the appropriate procedure.

9. If the call is to add a message, fills in the header of the message segment if empty and dispatches the call.

10. If the operation leaves no information in the message segment, truncates the message segment to zero length.

All the variables for the creation of the message segment header are computed from the block size in the message facility data base. The algorithm for this computation is:

1. Get the maximum size of the message segment.

2. Get the block size from the message facility data base.

3. The number of possible blocks in the message segment is equal to 1 divided by 2. Since the allocation bit string contains one bit for each block, the computed value is the bit length of that string.

4. Compute the number of blocks needed for the header of the message segment. This is done by taking the length of the header, adding the length of the allocation bit string, and dividing by the block size.

5. Compute the number of blocks remaining for messages. This is 3 - 4.

The purpose of this computation is to eliminate dependence on a particular block size.

Whenever mseg_ discovers an inconsistency or possible inconsistency in the message segment, it invokes the message segment salvager and returns.

Reading and Deleting Messages (mseg_util_)

This procedure is one of several called by mseg_. Its function is to return and/or delete messages.

All operations begin by gathering the data necessary to locate a message. This data is taken from the header of the message segment and the message facility data base. Data taken from the data base consists of:

1. The length in words of a message header.

2. The length in words of a message trailer.

If the length of either structure is changed, that value must be updated in the data base.

The next step common to all operations is to locate the desired message. This can be the first message, the last message, or an incremental message. If the message desired is the first or last message, its offset is stored in the message segment header. Two arguments supplied by the caller are used to locate an incremental message: the location and time of the message from which to read incrementally and the direction in which to increment. (The caller obtains the location and time from the previous read operation.) The format of the message is then checked. Information regarding the message is extracted from the trailer.

A final access check is made. An argument determines whether the call is being made from an own entry or from a nonown entry. If the call is own and the user did not send the message (Person_id.Project_id in the message trailer does not match Person_id.Project_id in the group ID), then only the data extracted from the trailer is returned. This sequence of operations allows the own module (see Figure 2-1) to call mseg_ repeatedly and get back only the information needed for another incremental read until mseg_ reads a message sent by the caller.

The message thread is now chased. If the message is being returned, it is copied out to a user area. If it is being deleted, the aip bit is turned on, the appropriate blocks are zeroed out, the header information is updated, and the aip bit is turned off.

Adding Messages (mseg_add_)

To add a message, this procedure:

1. Computes the number of blocks needed for the message.

2. Finds that number of unused blocks in the segment.

3. Threads these blocks together.

4. Fills the message and block header and trailer data into the blocks.

If processing is interrupted during one of the above steps, the message segment is left in a consistent state. The blocks containing the new message are not yet threaded into the segment itself or recorded in the header as in use.

5. Turns on the aip bit in the header of the message segment.

6. Threads the message into the segment.

7. Updates the header.

8. Turns off the aip bit.

This procedure contains an entry to add a message using a sender ID specified by the caller. The entry is used by the conversion module described under the heading "Converting Message Segments" and is not available in the user ring.

MANIPULATING ONLY CALLER MESSAGES (mseg_own_)

This procedure provides controlled access to a message segment by returning to the caller only messages placed there by the caller. It calls the appropriate entry in mseg_ to read a message. If a nonzero length message is returned, mseg_own_ returns this message to the caller. Otherwise it calls to read another message.

Since mseg_own_ does not access the message segment directly, it does not turn on guaranteed eligibility.

## SALVAGING MESSAGE SEGMENTS

The module that salvages message segments is invoked to restore the message segment to a consistent state. The basic approach to salvaging is to save only those messages that are in a consistent state and not to attempt to repair messages. The reason for this approach is twofold:

1. The salvager cannot ensure the correctness of the text of the message. Even if it could restore damaged threads, header, and trailer data, the message itself might be garbled. Furthermore, messages that are contained in damaged blocks are more likely to be garbled than those in undamaged blocks.

2. The salvager is much simplified by this implementation.

The salvager is divided into two procedures, ms_salvager_ and ms_salv_util_. The procedure ms_salvager_ does the following:

1. Establishes a cleanup handler to truncate the message segment. This step prevents repeated crawlouts from the administrative ring.

2. Computes the variables needed in salvaging. These are the same variables used by mseg_ (See the description of mseg_ earlier in this section).

3. Calls ms_salv_util_ to attempt a forward salvage.

4. If all messages were saved, skips the next step.

5. Calls ms_salv_util_ to attempt a backward salvage.

6. If anything was saved, updates the inconsistent information in the message segment.

No changes are made to the message segment until it is determined that some data has been saved. This sequence is necessary because the salvager is used in the conversion module to check the format of a message segment before it is converted. If the message segment salvager wrote into a message segment of an incorrect format, it would render the segment useless.

## CONVERTING MESSAGE SEGMENTS

This module is responsible for converting message segments from one format to another. It consists of the following procedures:

### mseg_

This is the dispatching procedure through which all calls to manipulate message segments are routed. After checking access to the message segment, mseg_ takes the following steps:

1. Checks the version number of the message segment. A version 1 message segment lacks a unique identifier in the second word of the message segment header. All other message segments contain the version number in a fixed location in the header.

2. If a format error is detected, salvages the message segment and returns.

3. If the version number in the message segment header is not equal to the version number stored in the message segment data base, attempts to convert the message segment. The conversion is accomplished by calling a procedure with a standard calling sequence. This code should not have to be changed if another version of message segments is created.

4. If the conversion is successful, continues the operation.

5. If the conversion is unsuccessful because a needed procedure was missing (error_table_$improper_data_format returned from mseg_convert_), returns this code to the user, unlocks the message segment, and returns.

6. If the conversion was unsuccessful for any other reason, assumes a format error in the current version and attempts to salvage.

mseg_add_

This procedure is used to add a message to a message segment. It normally obtains the user ID of the message from get_group_id_. An entry named mseg_add_$convert is provided for conversion purposes to add a message with a user ID specified by the caller.

mseg_convert_

This procedure is called by the mseg_ routine and is responsible for performing the conversion. It accepts as arguments the version number of the message segment to be converted and the version number to which to convert it. The procedure mseg_convert_ executes the following steps:

1. Checks for the existence of procedures needed to convert the message segment to the version just previous to the current version. If these procedures are not present, returns the code error_table_$improper_data_format.

2. If the old version is less than the current version - 1, calls itself recursively, decrementing the current version by 1.

3. If the old version is equal to the current version - 1, then:

   a. Invokes the salvager corresponding to the old version. This step is executed to ensure consistency of the message segment.

   b. Invokes the conversion procedure to convert from old version to old version + 1.

ms_salvager_vn_

This procedure is made from the previous version salvager when a new version of message segments is created. n is the now previous version number. Steps are:

1. Remove code to truncate bad message segments.

2. Change proc: and end statements and rename.

ms_salv_util_vn_

This procedure is created from the previous version procedure by changing the proc: and end statements and renaming.

mseg_util_vn_

This procedure is created from the previous version procedure by changing the proc: and end statements and renaming.

## THE MESSAGE SEGMENT DATA BASE (mseg_data_)

This data base contains all the constants needed by the message segment facility. A copy is included in Appendix A.

## EXECUTION IN THE USER RING

A set of procedures exists that allow the facility to be executed in the user ring for debugging purposes. These procedures enable the caller to set breaks using debug and to run the facility without endangering installed message segments. To use this debugging environment:

1. The user must have access to phcs_.

2. The installed primitives must not have been initiated by the user's process. If uncertain, it is wise to do a new_proc at this point.

3. To set up the environment, the user executes the following commands:

   a. set_exmode_level n;

   b. in dummy_admin_gate_ admin_gate_
      in dummy_message_segment_ message_segment_
      in dummy_mailbox_ mailbox_
      (These commands initiate private versions of the gates used by the message segment facility.)

c. Various procedures that manipulate message segments have entries to set the pathname of the directory containing the message segment. These entries should be called with the pathname of the user's test directory. (See the IO Daemon PLM, Order No. AN58.)

d. Message segments can now be created in the test directory and message segment commands executed on them. For a description of these commands, see the PLM on the IO Daemon and the two sections of this document describing the queue message segment facility and the mailbox message segment facility.

## THE MESSAGE SEGMENT COMMAND UTILITY

The message segment command utility (ms_create) has separate entries for queue message segment and mailbox commands and can be modified to add entries for new types of message segments. Data particular to each type of message segment is kept in an include file called extended_access_data (see Appendix A). The commands check that the segments they work on have certain properties described in extended_access_data. Among these are the name suffix and the number of extended access bits defined.

The commands are described in Section V under "Command Interface" sections for each application. They dispatch appropriate message segment subroutine calls by:

1. Concatenating gate and entrynames for the type of message segment, obtained from extended_access_data.

2. Calling hcs_$make_ptr to get a pointer to the gate entry.

3. Calling cu_$ptr_call to call through the pointer with a uniform set of arguments.

The module has its own name duplication handler to remove names from and to delete message segments. These operations require calls to message segment primitives.

The ACL commands (ms_list_acl, mbx_delete_acl, etc.) call find_common_acl_names_ (described in the Command Implementation PLM, Order No. AN67) to return all the ACL entries that match a given set of access control name arguments. find_common_acl_names_ and its caller share data in the following structure:

```
1 data aligned based (datap),
  2 aclp pointer,
  2 bsp pointer,
  2 acl_count fixed bin(17),
  2 extended_access_bit_length fixed bin(17),
  2 real_access_bit_length fixed bin(17);
```

Each ACL command with its list of access control name arguments does the following:

1. Calls find_common_acl_names_$init to list the ACL of the message segment, setting aclp.

2. Allocates and initializes to zero a string of bits corresponding to the ACL entries. Sets bsp to point to this bit string. Allocates an array big enough to hold all the access names.

3. Calls find_common_acl_names_ for each access control name argument. Gets back an updated bit string and an array of all matching names that were not returned by previous calls. Adds these names to an ACL array that it is building. The ACL array contains no duplicated entries.

4. After all arguments have been processed, calls the appropriate ACL primitive with the new ACL array.

5. Frees the bit string, the ACL arrays, and the array of names.

Access control names are of the form:

<component1>.<component2>.<component3>

The matching strategy used by find_common_acl_names_ can be summarized in three rules:

1. A literal component name, including *, matches only a component of the same name.

2. A missing component name not delimited by a period (eg. *.Multics, in which the third component is missing) is treated the same as a literal *. Missing components on the left must be delimited by periods.

3. A missing component delimited by a period matches any component.

For examples, see "Notes" under the description of the ms_delete_acl command in Section V of this manual.

# SECTION V

# MESSAGE SEGMENT APPLICATIONS

This section describes two specific applications of the message segment facility. These applications are queues and mailboxes.

## EXTENDED ACCESS

The extended access attributes defined for both kinds of message segment are:

a allows a user to add a message.

d allows a user to delete any message.

r allows a user to read any message.

o allows a user to read and/or delete messages sent by him.

s allows a user to find out the number of messages in the message segment and whether or not the message segment has been salvaged.

## THE QUEUE MESSAGE SEGMENT FACILITY

The queue message segment facility is used to implement the I/O Daemon and Absentee Daemon queues.

Topics discussed below include the subroutine calls and commands that work on queues.

## The Queue Message Segment Module

The queue message segment module is that portion of the message segment facility that defines queue message segments. It is composed of:

1. A gate (message_segment_) from the user ring into the administrative ring. It contains all the entries to queue message segment primitives.

2. A defining procedure (queue_mseg_) called through the gate. Before passing the call on to the appropriate primitive, this procedure checks that the name of the segment being referenced ends in the suffix ms. It also checks for any call involving the ACL of a message segment that only the first five bits of extended access (adros) are on.

## The Queue Message Segment Subroutine Interface

The following is a list of standard error codes returned by queue message segment facility subroutines.

| Code | Meaning |
|---|---|
| error_table_$bad_segment | an inconsistency was detected in the message segment causing it to be salvaged. The call should be made again. |
| error_table_$moderr | extended access is insufficient to perform the operation. |
| error_table_$no_message | the requested message was not found. |
| error_table_$improper_data_format | a message segment of different version number was encountered and the software necessary to perform the conversion cannot be found. |

The following is an alphabetized list of arguments used in the described calls.

acl_count (fixed bin) — is the number of entries in the structure pointed to by aclp.

aclp (pointer) — is a pointer to the following structure:

```
declare 1 acl_entries (acl_count) aligned based (aclp),
          2 access_name char(32) aligned,
          2 modes bit(36) aligned,
          2 extended_access bit(36) aligned,
          2 reterr fixed bin(35);
```

where:

access_name — is the access name (in the form Person_id.Project_id.tag) that identifies a class of users.

modes — is the real access for this access name.

extended_access — is the extended access for this access name.

reterr — is a standard Multics status code.

areap (pointer) — is a pointer to a user defined area.

argp (pointer) — is a pointer to the following structure:

```
declare 1 mseg_return_args aligned based (argp),
          2 ms_ptr ptr,
          2 ms_len fixed bin(18),
          2 sender_id char(32) aligned,
          2 level fixed bin,
          2 ms_id bit(72) aligned,
          2 sender_authorization bit(72),
          2 access_class bit(72);
```

where:

| | |
|---|---|
| ms_ptr | is a pointer to the returned message. |
| ms_len | is the bit length of the returned message. |
| sender_id | is the ID of the sender of the message in the form Person_id.Project_id.tag. |
| level | is the validation level of the sender of the message. |
| ms_id | is the ID of the returned message. |
| sender_authorization | is the access authorization of the sender. |
| access_class | is the access class of the message. |
| code (fixed bin(35)) | is a standard file system error code. |
| dir_name (char (*)) | is the pathname of the directory containing the message segment to be referenced. |
| direction (bit(2) aligned) | is a switch indicating the direction of an incremental read. "00"b => current message; "10"b => previous message; "01"b => next message. |
| ent_name (char(*)) | is the entryname of the message segment to be referenced. |
| index (fixed bin) | is an index to the message segment to be referenced. |

| | |
|---|---|
| message_count (fixed bin) | is the number of messages in the message segment. |
| message_id (bit(72) aligned) | is a unique identifier corresponding to a message in the message segment. |
| message_length (fixed bin(18)) | is the length, in bits, of a message. |
| message_wanted (bit(1) aligned) | is a switch indicating which message is wanted. "0"b => first message; "1"b => last message. |
| messagep (pointer) | is a pointer to the message. |
| new_name (char(*)) | is the name to which to rename a message segment. |
| old_name (char(*)) | is the name of the message segment to be renamed. |
| salvaged_bit (bit(1) aligned) | is a switch indicating whether or not the message segment has been salvaged. "0"b => no; "1"b => yes. |
| turn_off (bit(1) aligned) | is a switch indicating whether or not to turn off salvaged_bit. "0"b => no; "1"b => yes. |

The following is a list of subroutine calls to the queue message segment facility. They are grouped by class.

## CREATING AND DELETING QUEUE MESSAGE SEGMENTS

Entry: message_segment_$create

This entry point is used to create a queue message segment.

Usage

```
declare message_segment_$create entry
        (char(*), char(*), fixed bin(35));

call message_segment_$create
        (dir_name, ent_name, code);
```

1. dir_name Input
2. ent_name Input
3. code Output

Entry: message_segment_$delete

This entry point is used to delete a queue message segment.

Usage

```
declare message_segment_$delete
        (char(*), char(*), fixed bin(35));

call message_segment_$delete
        (dir_name, ent_name, code);
```

1. dir_name Input
2. ent_name Input
3. code Output

Entry: message_segment_$ms_acl_add

This entry point is used to add one or more entries to the access control list of a queue message segment.

Usage

```
declare message_segment_$ms_acl_add
        entry (char(*), char(*), ptr, fixed bin,
        fixed bin(35));

call message_segment_$ms_acl_add
        (dir_name, ent_name, aclp, acl_count, code);
```

1. dir_name Input
2. ent_name Input
3. aclp Input
4. acl_count Input
5. code Output

Entry: message_segment_$ms_acl_delete

This entry point is used to delete one or more entries from the access control list of a queue message segment.

Usage

```
declare message_segment_$ms_acl_delete
        entry (char(*), char(*), ptr, fixed bin, ptr,
        fixed bin(35));

call message_segment_$ms_acl_delete
        (dir_name, ent_name, aclp, acl_count, areap, code);
```

1. dir_name Input
2. ent_name Input

3. aclp see "Note"

4. acl_count see "Note"

5. areap see "Note"

6. code Output

Note

If acl_count is equal to -1, areap is assumed to point to a user area. In this case the entire ACL is replaced with two entries, one for *.*.* and one for *.SysDaemon.*, both with null extended access. Information about these entries is returned in the area pointed to by areap. aclp is set to point to the allocated data and acl_count is set to the number of entries. If acl_count is not -1, then aclp is assumed to point to a structure containing the ACL entries enumerated by acl_count. The entries for *.*.* and *.SysDaemon.* cannot be deleted.

Entry: message_segment_$ms_acl_list

This entry point is used to list one or more items from the access control list of a queue message segment.

Usage

```
declare message_segment_$ms_acl_list entry
        (char(*), char(*), ptr, fixed bin, ptr, fixed bin(35));

call message_segment_$ms_acl_list (dir_name, ent_name,
        aclp, acl_count, areap, code);
```

1. dir_name Input

2. ent_name Input

3. aclp see "Note"

4. acl_count see "Note"

5. areap see "Note"

6. code Output

Note

If acl_count is -1, areap is assumed to point to a user defined area. A list of the entire ACL is returned in space allocated in this area and aclp is set to point to the list. If acl_count is not -1, areap may be null and aclp is assumed to point to an ACL data structure in which data is returned.

Entry: message_segment_$ms_acl_replace

This entry point is used to replace the access control list of a queue message segment.

Usage

```
declare message_segment_$ms_acl_replace entry
        (char(*), char(*), ptr, fixed bin, fixed bin(35));

call message_segment_$ms_acl_replace
        (dir_name, ent_name, aclp, acl_count, code);
```

1. dir_name Input
2. ent_name Input
3. aclp Input
4. acl_count Input
5. code Output

Note

The list of entries replacing the current access control list must contain entries for *.*.* and *.SysDaemon.*.

## RENAMING QUEUE MESSAGE SEGMENTS

Entry: message_segment_$chname_file

This entry point is used to rename a queue message segment.

Usage

```
declare message_segment_$chname_file entry
        (char(*), char(*), char(*), char(*) fixed bin(35));

call message_segment_$chname_file
        (dir_name, ent_name, old_name, new_name, code);
```

1. dir_name Input
2. ent_name Input
3. old_name Input
4. new_name Input
5. code Output

The remaining primitives in this section can be called either of two ways:

1. The message segment can be opened using message_segment_$open and calls made with the index thus obtained.
2. The primitive can be called directly using dir_name and ent_name to reference the segment.

The former method is more efficient for repeated calls as it avoids repeated initiations and terminations of the message segment in the administrative ring.

The entries are documented in pairs corresponding to the two modes of use.

## OPENING AND CLOSING A QUEUE MESSAGE SEGMENT

Entry: message_segment_$open

This entry point is called to open and get an index to a queue message segment.

Usage

```
declare message_segment_$open entry
        (char(*), char(*), fixed bin, fixed bin(35));

call message_segment_$open
        (dir_name, ent_name, index, code);
```

1. dir_name Input
2. ent_name Input
3. index Output
4. code Output

Entry: message_segment_$close

This entry point is used to close a queue message segment after it has been opened.

Usage

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

call message_segment_$close
        (index, code);
```

1. index Input
2. code Output

OBTAINING QUEUE MESSAGE SEGMENT HEADER STATUS

Entry: message_segment_$check_salv_bit_index
message_segment_$check_salv_bit_file

This entry point is used to check whether or not a queue message segment has been salvaged.

Usage

```
declare message_segment_$check_salv_bit_index
        entry (fixed bin, bit(1) aligned, bit(1) aligned,
        fixed bin(35));

call message_segment_$check_salv_bit_index
        (index, turn_off, salvaged_bit, code);
```

or:

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

call message_segment_$check_salv_bit_file
        (dir_name, ent_name, turn_off, salvaged_bit, code);
```

1. index Input
2. turn_off Input
3. salvaged_bit Output
4. code Output
5. dir_name Input
6. ent_name Input

Note

The caller must have s extended access to the message segment. If the turn_off bit is on, indicating that the caller wishes to turn off the salvaged bit, the caller must have delete extended access to the message segment.

Entry: message_segment_$get_message_count_index
message_segment_$get_ message_count_file

This entry point is used to obtain the number of messages in a message segment.

Usage

```
declare message_segment_$get_message_count_index
        entry (fixed bin, fixed bin, fixed bin(35));

call message_segment_$get_message_count_index
        (index, message_count, code);
```

or:

```
declare message_segment_$get_message_count_file
        entry (char(*), char(*), fixed bin, fixed bin(35));

call message_segment_$get_message_count_file
        (dir_name, ent_name, message_count, code);
```

| | | |
|---|---|---|
| 1. | index | Input |
| 2. | message_count | Output |
| 3. | code | Output |
| 4. | dir_name | Input |
| 5. | ent_name | Input |

Note

The caller must have status extended access to the message segment.

## OBTAINING EFFECTIVE ACCESS TO A MESSAGE SEGMENT

Entry: message_segment_$get_mode_index
message_segment_$get_mode_file

This entry point is used to find out the user's effective extended access to a message segment.

Usage

```
declare message_segment_$get_mode_index entry
        (fixed bin, fixed bin(5), fixed bin(35));

call message_segment_$get_mode_index
        (index, mode, code);
```

or:

```
declare message_segment_$get_mode_file entry
        (char(*), char(*), fixed bin(5), fixed bin(35));

call message_segment_$get_mode_file
        (dir_name, ent_name, mode, code);
```

1. index Input
2. mode Output
3. code Output
4. dir_name Input
5. ent_name Input

## ADDING MESSAGES TO A QUEUE MESSAGE SEGMENT

Entry: message_segment_$add_index
message_segment_$add_file

This entry point is used to add a message to a message segment.

Usage

```
declare message_segment_$add_index entry
        (fixed bin, ptr, fixed bin, bit(72) aligned,
        fixed bin(35));

call message_segment_$add_index
        (index, messagep, message_length, message_id, code);
```

or:

```
declare message_segment_$add_file entry
        (char(*), char(*), ptr, fixed bin, bit(72) aligned,
        fixed bin(35));

call message_segment_$add_file
        (dir_name, ent_name, messagep, message_length
        message_id, code);
```

| | | |
|---|---|---|
| 1. | index | Input |
| 2. | messagep | Input |
| 3. | message_length | Input |
| 4. | message_id | Input |
| 5. | code | Output |
| 6. | dir_name | Input |
| 7. | ent_name | Input |

Note

The caller must have append extended access to the message segment.

## DELETING MESSAGES FROM A QUEUE MESSAGE SEGMENT

Entry: message_segment_$delete_index
message_segment_$delete_file

This entry point is used to delete a message from a message segment.

Usage

```
declare message_segment_$delete_index entry
        (fixed bin, bit(72) aligned, fixed bin(35));

call message_segment_$delete_index
        (index, message_id, code);
```

or:

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

call message_segment_$delete_file
        (dir_name, ent_name, message_id, code);
```

1. index Input
2. message_id Input
3. code Output
4. dir_name Input
5. ent_name Input

Note

To delete a message sent by the caller, owner and/or delete extended access to the message segment is required. To delete a message sent by someone other than the caller, delete extended access to the message segment is required.

## READING MESSAGES FROM A QUEUE MESSAGE SEGMENT

Entry: message_segment_$read_index
message_segment_$read_file

This entry point is used to read the first or last message from a message segment.

Usage

```
declare message_segment_$read_index entry
        (fixed bin, ptr, bit(1) aligned, ptr, fixed bin(35));

call message_segment_$read_index
        (index, areap, message_wanted, argp, code);
```

or:

```
declare message_segment_$read_file entry
        (char(*), char(*), ptr, bit(1) aligned, ptr,
        fixed bin(35));

call message_segment_$read_file
        (dir_name, ent_name, areap, message_wanted, argp,
        code);
```

| | | |
|---|---|---|
| 1. | index | Input |
| 2. | areap | Input |
| 3. | message_wanted | Input |
| 4. | argp | Input |
| 5. | code | Output |
| 6. | dir_name | Input |
| 7. | ent_name | Input |

Note

The caller must have read extended access to the message segment.

Entry: message_segment_$incremental_read_index
message_segment_$incremental_read_file

This entry point is used to read a message incremental to another message. The message_id of the message from which the incremental read is to take place must be supplied as an argument, therefore the read entry must be called prior to calling this entry.

Usage

```
declare message_segment_$incremental_read_index
        entry(fixed bin, ptr, bit(2) aligned, bit(72) aligned,
        ptr, fixed bin(35));

call message_segment_$incremental_read_index
        (index, areap, direction, message_id, argp, code);
```

or:

```
declare message_segment_$incremental_read_file
        entry (char(*), char(*), ptr, bit(2) aligned,
        bit(72) aligned, ptr, fixed bin(35));

call message_segment_$incremental_read_file
        (dir_name, ent_name, areap, direction, message_id,
        argp, code);
```

1. index Input
2. areap Input
3. direction Input
4. message_id Input
5. argp Input
6. code Output
7. dir_name Input
8. ent_name Input

Note

The caller must have read extended access to the message segment.

COMBINED READ AND DELETE FROM A QUEUE MESSAGE SEGMENT

Entry: message_segment_$read_delete_index
message_segment_$read_delete_file

This entry point is used to read and delete the first or last message from a queue message segment.

Usage

```
declare message_segment_$read_delete_index
        entry (fixed bin, ptr, bit(1) aligned, ptr,
        fixed bin(35));

call message_segment_$read_delete_index
        (index, areap, message_wanted, argp, code);
```

or:

```
declare message_segment_$read_delete_file entry
        (char(*), char(*), ptr, bit(1) aligned, ptr,
        fixed bin(35));

call message_segment_$read_delete_file
        (dir_name, ent_name, areap, message_wanted, argp,
        code);
```

| | | |
|---|---|---|
| 1. | index | Input |
| 2. | areap | Input |
| 3. | message_wanted | Input |
| 4. | argp | Input |
| 5. | code | Output |
| 6. | dir_name | Input |

6. dir_name Input

7. ent_name Input

Note

The caller must have read and delete extended access to the message segment.

REWRITING MESSAGES IN A QUEUE MESSAGE SEGMENT

Entry: message_segment_$update_message_index
message_segment_$update_ message_file

This entry point is used to rewrite an existing message in a queue message segment.

Usage

```
declare message_segment_$update_message_index
        entry  (fixed bin, fixed bin(18), bit(72) aligned, ptr,
        fixed bin(35));

call message_segment_$update_message_index
        (index, message_length, message_id, messagep, code);
```

or:

```
declare message_segment_$update_message_file
        entry (char(*), char(*), fixed bin(18),
        bit(72) aligned, ptr, fixed bin(35));

call message_segment_$update_message_file
        (dir_name, ent_name, message_length, message_id,
        messagep, code);
```

1. index Input

2. message_length Input

3. message_id Input

| | | |
|---|---|---|
| 4. | messagep | Input |
| 5. | code | Output |
| 6. | dir_name | Input |
| 7. | ent_name | Input |

Notes

The caller must have delete extended access to the message segment.

The lengths of the old and new messages must be the same.

## READING CALLER MESSAGES FROM A QUEUE MESSAGE SEGMENT

These entries return a message that was sent by the caller. In contrast to the read and incremental_read entries, which require read extended access, these entries require read and/or owner extended access.

Entry: message_segment_$own_read_index
message_segment_$own_read_file

This entry point is used to read the first or last message placed by the caller in a queue message segment.

Usage

```
declare message_segment_$own_read_index
        entry (fixed bin, ptr, bit(1) aligned, ptr,
        fixed bin(35));

call message_segment_$own_read_index
        (index, areap, message_wanted, arg_ptr, code);
```

or:

```
declare message_segment_$own_read_file
        entry (char(*), char(*), ptr, bit(1) aligned, ptr,
        fixed bin(35));

call message_segment_$own_read_file
        (dir_name, ent_name, areap, message_wanted,
        arg_ptr, code);
```

| | | |
|---|---|---|
| 1. | index | Input |
| 2. | areap | Input |
| 3. | message_wanted | Input |
| 4. | arg_ptr | Input |
| 5. | code | Output |
| 6. | dir_name | Input |
| 7. | ent_name | Input |

Entry: message_segment_$own_incremental_read_index
message_segment_$own_incremental_read_file

This entry point is used to read a message placed by the caller in a queue message segment incremental to another message placed by the caller in that message segment. The message_id of the message from which the incremental read is to take place must be supplied, therefore the own_read entry must be called prior to calling this entry.

Usage

```
declare message_segment_$own_incremental_read_index
        entry (fixed bin, ptr, bit(2) aligned, bit(72) aligned,
        ptr, fixed bin(35));

call message_segment_$own_incremental_read_index
        (index, areap, direction, message_id, argp, code);
```

or:

```
declare message_segment_$own_incremental_read_file
        entry (char(*), char(*), ptr, bit(2) aligned,
        bit(72) aligned, ptr, fixed bin(35));

call message_segment_$own_incremental_read_file
        (dir_name, ent_name, areap, direction, message_id,
        argp, code);
```

1. index Input
2. areap Input
3. direction Input
4. message_id Input
5. argp Input
6. code Output
7. dir_name Input
8. ent_name Input

## The Queue Message Segment Command Interface

The following is an alphabetized list of the commands that manipulate queue message segments.

Name: ms_add_name, msan

The ms_add_name command adds alternate names to the existing name(s) of a message segment.

Usage

```
ms_add_name path entries
```

where:

1. path is the pathname of a message segment. The star convention is allowed.
2. entries are names to be added. The equal convention is allowed.

Notes

If the suffix ms does not appear at the end of path, it is assumed.

entry must be unique in the directory. If there is a name duplication and the old segment has only one name, the user is interrogated as to whether the old segment is to be deleted. If the old segment has other names, the conflicting name is removed and a message is printed to that effect.

Example

```
msan >ddd>idd>io_daemon_3  io_queue_3
```

causes the name io_queue_3.ms to be added to the message segment io_daemon_3.ms in the directory >ddd>idd.

Name: ms_create, mscr

The ms_create command creates a message segment of a given name in a given directory.

Usage

```
ms_create paths
```

where paths are the pathnames of message segments to be created.

Notes

If the suffix ms does not appear at the end of path, it is assumed.

The user must have append (a) access to the containing directory.

Name duplication is handled similarly to ms_add_name.

The user is given adros extended access to the message segment created. Null extended access is given to *.*.*.

Example

```
mscr io_daemon_3.ms >ddd>idd>io_daemon_2
```

causes the message segment io_daemon_3.ms to be created in the working directory and the message segment io_daemon_2.ms to be created in the directory >ddd>idd.

Name: ms_delete, msdl

The ms_delete command deletes message segments.

Usage

```
ms_delete paths
```

where paths are the pathnames of message segments to be deleted. The star convention is allowed.

Notes

If the suffix ms does not appear at the end of path, it is assumed.

The user must have modify (m) access to the containing directory and delete (d) extended access to the message segment. If delete extended access is lacking, the user is interrogated as to whether the message segment is to be deleted.

Examples

```
msdl **
```

deletes all message segments in the working directory.

```
msdl a.ms >udd>m>Doe>b
```

deletes a.ms in the working directory and b.ms in >udd>m>Doe.

Name: ms_delete_acl, msda

This command deletes some or all of the items on the Access Control List (ACL) of a message segment.

Usage

ms_delete_acl path -acnames-

where:

1. path — is the pathname of a message segment. The star convention is allowed.

2. acnames — are access control names. If there is no acname specified, the user's process group ID is assumed; if it is -all (-a), the entire ACL is deleted. Otherwise, acname must be of the form Person_id.Project_id.tag. Every ACL entry whose name matches the access control name is deleted. See "Notes" below for a description of the matching strategy.

Notes

If the suffix ms does not appear at the end of path, it is assumed.

An asterisk (*) in the access control name means the literal star character. Therefore, *.*.* matches only the ACL entry *.*.*. If a missing component is not delimited by a period, it is assumed to be a literal star. *.Multics matches only the ACL entry *.Multics.*. If a missing component is delimited by a period, however, it matches any component.

| | |
|---|---|
| Multics | matches the ACL Multics.*.*. The absence of a leading period makes Multics the first component. |
| .Multics. | matches any entry with middle component Multics |
| .. | matches any ACL entry |
| . | matches any entry whose last component is * |
| null string | matches any entry ending in .*.* |

Examples

```
msda foo .Multics ..a
```

deletes from the ACL of foo.ms all entries ending in .Multics.* and all entries with instance tag a.

```
msda foo.ms *.Multics.* Fred..
```

deletes the entry *.Multics.* or prints an error message and deletes all entries with first component Fred.

Name: ms_delete_name, msdn

This command deletes names from message segments having multiple names.

Usage

```
ms_delete_name paths
```

where paths are the pathnames of message segments. The entryname portion is the name to be deleted. The star convention is allowed.

Notes

If the suffix ms is missing from path, it is assumed.

If deleting a name would leave no names on the message segment, the user is interrogated as to whether the message segment is to be deleted.

Example

```
msdn alpha >udd>Multics>Doe>beta
```

deletes the name alpha.ms from the list of names for the appropriate message segment in the current working directory and deletes the name beta.ms from the list of names for the appropriate message segment in the directory >udd>Multics>Doe.

Name: ms_list_acl, msla

This command lists some or all of the items on the Access Control List (ACL) of a message segment.

Usage

ms_list_acl path -acnames-

where:

1. path — is the pathname of a message segment. The star convention is allowed.

2. acnames — are access control names. If there is no acname specified or if it is -all (-a), the entire ACL of the message segment is listed. Otherwise, acname must be of the form Person_id.Project_id.tag. All ACL entries with names matching the access control name are listed. The strategy for matching is described under "Notes" for ms_delete_acl.

Note

If the suffix ms does not appear at the end of path, it is assumed.

Examples

msla foo .Multics ..a

lists from the ACL of foo.ms all entries ending in .Multics.* and all entries with instance tag a.

Name: ms_rename, msrn

The ms_rename command replaces a specified name on a message segment, without affecting any other names the message segment has.

Usage

ms_rename path1 name1...pathn namen

where:

1. pathi specifies the old name. The star convention is allowed.

2. namei specifies the new name (i.e., replaces the entryname portion of pathi). The equal convention is allowed.

Notes

If the suffix ms does not appear at the end of pathi or namei, it is assumed.

Name duplication is handled similarly to ms_add_name.

Example

msrn alpha beta >udd>Multics>Doe>gamma.ms delta.ms

causes alpha.ms in the current working directory to be renamed beta.ms, and gamma.ms in the directory >udd>Multics>Doe to be renamed delta.ms.

Name: ms_set_acl, mssa

This command adds items to the Access Control List (ACL) of a message segment.

Usage

ms_set_acl path ac1 acname1 ... acn acnamen

where:

1. path — is the pathname of a message segment. The star convention is allowed.

2. aci — is the access to be given. It can consist of any or all of the letters adros or it can be null, n, or "" to denote null access.

3. acnamei — is an access control name. It must be of the form Person_id.Project_id.tag. If all three components are present, the ACL is searched for an entry by that name. If one is found, the access is changed. Otherwise, a new ACL entry is added. If one or more components are missing from the access control name, the access is changed on all entries that match the access control name. The strategy for matching is described under "Notes" for ms_delete_acl.

Note

If the suffix ms does not appear at the end of path, it is assumed.

Examples

mssa n ad Joe.Proj

adds to the ACL of the message segment n.ms an entry for Joe.Proj.* with add and delete access.

mssa ** adros *

gives *.*.* "adros" access to every message segment in the working directory.

## THE MAILBOX MESSAGE SEGMENT FACILITY

The mailbox message segment facility implements user mailboxes for interprocess mail and messages. It resembles the queue message segment facility, with the following differences:

1. The suffix on a mailbox is mbx instead of ms.

2. Two additional extended access bits determine what kinds of wakeups can accompany mailbox messages. These bits are:

   w - allows a user to send a normal wakeup when adding a message.

   u - allows a user to send an urgent wakeup when adding a message.

### The Mailbox Message Segment Module

The gate mailbox_ is similar to the gate message_segment_ in the queue message segment module. It has fewer entries. These are listed under "Subroutine Interface".

The defining procedure mbx_mseg_ is similar to queue_mseg_ in the queue message segment module and makes similar checks before passing on its calls. Unlike queue_mseg_, it reserves the front part of each message for data peculiar to mailbox applications. The include file mail_format.incl.pl1 contains a structure describing this data:

```
dcl 1 mail_format aligned,
      2 version fixed bin(17),
      2 sent_from char(32) aligned,
      2 lines fixed bin(17),
      2 text_len fixed bin(17),
      2 switches aligned,
        3 wakeup bit(1) unaligned,
        3 urgent bit(1) unaligned,
        3 has_been_read bit(1) unaligned,
        3 acknowledge bit(1) unligned,
        3 others bit(32) unaligned,
      2 text aligned
            char (text_length refer(mail_format.text_len));
```

where:

| | | |
|---|---|---|
| 1. | version | is a version number for the mailbox. |
| 2. | sent_from | is a terminal ID or other ID. |
| 3. | lines | is the number of lines in text. |
| 4. | text_len | is the number of characters in text. |
| 5. | wakeup | is on for a wakeup message. |
| 6. | urgent | is on for an urgent wakeup message. |
| 7. | has_been_read | is on for an old message. |
| 8. | acknowledge | is on if acknowledgement is desired. |
| 9. | text | is the actual text of the message. |
| 10. | text_length | is the actual number of characters in text. |

The Mailbox Message Segment Subroutine Interface

mailbox_$entry calls are similar to message_segment_$entry calls and take the same arguments. The following entries are available:

```
create
delete
chname
open
close

add_index
delete_index
read_index
read_delete_index
incremental_read_index
own_read_index
own_incremental_read_index

get_message_count_index
get_mode_index
check_salv_bit_index
```

mailbox_ ACL calls are similar to the corresponding message_segment_ calls, though they recognize seven extended

access bits instead of five. The ACL calls are:

```
mbx_acl_add
mbx_acl_delete
mbx_acl_list
mbx_acl_replace
```

## The Mailbox Message Segment Command Interface

The commands mbx_<command> (mbcc) perform the same functions on mailboxes that ms_<command> (mscc) commands perform on queue message segments. They have the same user interface. Commands that work on ACLs recognize normal wakeup and urgent wakeup extended access. If the pathname of a mailbox does not end in mbx, that suffix is assumed.

# APPENDIX A

## extended_access_data

```
dcl typ_count fixed bin(17) aligned int static init (2),

 1 segdata (2) aligned int static,
  2 gate_name char(32) aligned
               init ("message_segment_", "mailbox_"),
  2 acl_prefix char(8) varying aligned
               init ("ms_", "mbx_"),
  2 segment suffix length fixed bin(17) aligned init (3, 4),
  2 segment_suffix char(8) aligned init (".ms", ".mbx"),
  2 full_extended_access bit(36) aligned
               init ("111110000000000000000000000000000000"b,
                     "111111100000000000000000000000000000"b),
  2 first_extended_access_bit fixed bin(17) aligned init (1, 1),
  2 mode_letter_count fixed bin(17) aligned init (5, 7),
  2 mode_letters (36) char(1) unaligned
               init ("a", "d", "r", "o", "s", (31)"",
                     "a", "d", "r", "o", "s", "w", "u",(29)"");
```

## ms_block_hdr

```
dcl block_ptr pointer,          /* pointer to message block */

    1 ms_block_hdr aligned based (block_ptr),
                                /* message block header structure */
     2 f_offset bit(18) unaligned,
                                /* offset to next block of message */
     2 first_block bit(1) unaligned,
                                /* ON if block is first in message */
     2 block_count bit(17) unaligned;
                                /* number of message bits in block */
```

## ms_block_trailer

```
dcl tr_ptr pointer,       /* pointer to message block trailer */

 1 ms_block_trailer aligned based (tr_ptr),
                          /* message block trailer structure */
  2 tr_pattern bit(36) aligned,
                          /* to identify beginning of trailer */
  2 f_offset bit(18) unaligned,
                          /* offset to next logical message */
  2 b_offset bit(18) unaligned,
                          /* offset to previous logical message */
  2 ms_size bit(18) unaligned,
                          /* bit count of the message */
  2 time bit(54) unaligned,
                          /* time the message was sent */
  2 ring_no bit(18) unaligned,
                          /* validation level */
  2 pad bit(18) unaligned,

  2 sender_id char(32) aligned;
                          /* id of message sender */
```

mseg_data_

```
segdef                  max_message_size
segdef                  block_size
segdef                  version_number
segdef                  block_hdr_data
segdef                  block_trailer_data
segdef                  mseg_b36
segdef                  mseg_tr36

max_message_size:       dec 2048
block_size:             dec 32
version_number:         dec 2
block_hdr_data:         dec 1
block_trailer_data:     dec 13
mseg_b36:               oct 252525252525
mseg_tr36:              oct 777777777777

end
```

## mseg_hdr

```
dcl mptr pointer,          /* pointer to message segment */

 1 mseg_hdr aligned based (mptr), /* message segment header */
  2 lock bit(36) aligned, /* standard file system lock */
  2 mseg_pattern bit(36) aligned, /* to identify a message seg */
  2 pad (6) fixed bin(17) aligned,
  2 first_ms_offset bit(18) aligned, /* offset to 1st message */
  2 last_ms_offset bit(18) aligned, /* offset to last message */
  2 alloc_len fixed bin, /* length of allocaion bit string */
  2 space_left fixed bin, /* number of empty blocks */
  2 number_of_messages fixed bin, /* message count */
  2 block_size fixed bin, /* message block size */
  2 switches unaligned,
   3 aip bit(1), /* ON if allocation is in progress */
   3 os bit(1), /* ON if message segment was salvaged */
   3 ms_in_hdr bit(1), /* ON if there is a header message */
   3 pad2 bit(33),
  2 version_number fixed bin, /* version  of message seg */
  2 hdr_ms_len fixed bin, /* length of header message */
  2 hdr_ms char(hdr_ms_len), /* space for header message */
  2 alloc_bits bit(alloc_len) aligned;
                          /* allocaion bit string */
```