MTB-341

To: Distribution

From: Robert M. May

Date: May 13, 1977

Subject: COBOL-74 Message Control System (CMCS) for MR6.0

INTRODUCTION

This MTB gives the proposed design for the runtime package to support the full Level-2 functional requirements of the ANSI COBOL-74 Communications Module. This facility is required for MR6.0 shipment.

Multics COBOL is being extended to process the ANSI COBOL-74 Message Control System (CMCS) syntax. Full Level-2 functions are provided for the SEND, RECEIVE, ENABLE, DISABLE, ACCEPT (MESSAGE COUNT) verbs of CMCS. In addition, the PURGE verb from the CODASYL JOD is supported.

The primary purpose of the CMCS facility is to enable Multics COBOL to fulfill the functional requirement imposed by the Navy Audit Routines. Tests for COBOL MCS do not yet exist; however, they are known to be under development and we must be ready to run and pass them when they become available.

Please send all comments on this proposal to the author.

Send U.S. mail to: Robert M. May
Honeywell
P.O. Box 6000, M.S. K-28
Phoenix, AZ 85005

or send Multics mail on System M in Phoenix to:

May.Multics

or call me at: (602) 249-7295
HVN 341-7295

REFERENCES

It is recommended that the reader be familiar with the description of the Communications Module in the ANS COBOL-74 Standard.

1. HIOC - Preliminary MCS Subroutine Specification, Otto Newman, 12/26/76. Defines the object program interfaces (OPI) necessary for the runtime package to support the full level-2, ANSI COBOL-74 Communications Module.

2. ANSI COBOL-74 Standard Definition, ANSI X3.23-1974

3. Preliminary MTB, COBOL MCS, R. W. Franklin

4. CODASYL JOD, 1976, for a description of the PURGE verb.

To assist the reader in understanding the basic functional needs imposed by the ANSI COBOL Standard, a copy of the descriptive narration from the Standard is attached.

TERMINOLOGY

- queue hierarchy
  The tree structure used by COBOL programs to access messages for (COBOL) Communications processing. There can be up to four levels in any subtree. Each level is identified by a level number and a level name. In the Multics implementation, the level names are completely logical; that is, the physical message queues are identified with a separate name associated with each terminating branch (tree path) in the hierarchy definition.

- level number
  The number, from 1-4, associated with each level in a queue hierarchy definition. It is not necessary to use all four levels.

- level name
  The logical name, from 1-12 characters, associated with each level in a queue hierarchy definition.

- tree path
  The concatenation of the level names in a particular branch of the subtree is called the tree path. It is the tree path by which COBOL application programs identify which particular physical message queue or queue hierarchy they wish to access. A tree path will have one of two forms. Internally, it will always be a 48 character string, consisting of the concatenation of the four, 12-character level names. The level names are blank filled to a maximum of 12 characters, and trailing, unused level names must be supplied and blank.

  Externally, the tree path can be a quoted string of the internal form or it can be a variable length string of characters with up to four period-delimited level names, similar to the components of an entryname in the storage system. In this form, the level names are not blank filled. Note that it is possible to have a tree path (in this form) that is 51 characters in length if all four level names are given and they are each 12 characters long.

  Examples of the two formats of tree paths would be:

  ```
  "orders.cloth.shirts.dress"
  "orders      cloth       shirts      dress       "
  ```

- absolute tree path
  An absolute tree path is a tree path that specifies all levels of a subtree necessary to identify a specific physical message queue.

- command line
  When it is desired that a command be executed when a physical message queue goes non-empty (with a message available for processing), a command line is specified for the queue in the source for cmcs_tree_ctl.control. Rules for constructing command lines are given in the description of the cv_cmcs_tree_ctl command.

- absin line
  The absin line is similar to the command line and is used when an absentee is submitted to process a non-empty queue.

- physical queue name
  In all subsequent discussions, the word "physical" will be omitted from "physical queue name"; however, the meaning is the same. The physical queue name defines the name of physical message queue. The actual entryname assigned will always have a suffix of "cmcs_queue".

DESIGN REQUIREMENTS

The COBOL-74 Communications Module is an ambiguous title for the description of the COBOL Message Control System, hereafter called CMCS, or COBOL MCS. COBOL MCS is a general facility for the writing and reading of messages in message queues, invoking application routines to process messages, and controlling access to the terminals and queues. (Terminal access in the Multics implementation is controlled only as it relates to COBOL MCS; outside the context of CMCS, no CMCS controls are imposed.)

1. Full Level 2 COBOL MCS functions must be delivered for MR6.0. "Full Level 2" dictates that all functions provided must adhere strictly to the ANSI COBOL-74 definition of CMCS. (Some functions as defined in the CODASYL JOD are also provided; however, it is nearly certain that ANSI will incorporate these extensions into the next definition of standard COBOL).

2. Performance is secondary to complete functionality.

3. No changes to existing system software are allowed.

4. No metering or accounting data is necessary to pass (as yet undefined) audit routines and therefore is not required for the first release.

5. Final implementation of COBOL MCS may drastically change, depending on the implementation of Multics transaction processing. In the meantime, it must be able to function as an independent subsystem.

6. A four-level hierarchy (not including the root) is set on top of the actual message queues. COBOL programs can request a message from any point in the hierarchy, thus causing the runtime package to look for a message in all queues defined by that subtree. They can wait (go blocked) for any message that becomes available in the specified subtree.

7. Messages can be any length, and can be written and read in any number of pieces, thus allowing the possibility of intermixed messages in the queues. Delimiters for the pieces are specified, but cannot be imbedded in the data.

8. Terminals and queues can be enabled and disabled (as mentioned above) by any user process with the right password; however, any messages in process must be allowed to complete.

DESIGN CONCEPTS

1. The Multics implementation of COBOL MCS is based on the concept of a "station." A station is a logical entity, having controls imposed by the system, that can be attached by a process. Its primary reason for existence is to provide a uniform mechanism for identifying sources and destinations of messages. Thus, the facility is independent from terminals, user-ids (including anonymous users), or constraints placed upon interactive or absentee users).

   In this usage, the term "attach" means only that an available resource becomes solely owned for current use, by a specific process. The connotation of a Multics I/O attachment does not apply.

   Specific stations can be attached to individuals; by default, they are assigned to users dynamically on the basis of terminal subchannels.

2. A process can attach one station. This can be extended after MR6.0 to allow a process to attach more than one station, and to allow multiple processes to share a single station.

3. A process can receive or send from/to any (authorized) queue. The difference between the station queue and all others is that it is specifically dedicated for input to that station.

4. Associated with every communications terminal (communications subchannel), there is one and only one station. Output _to_ a given terminal will always be written first to the station's queue. (If, in the future, a user's terminal can be shared among the user's process and other processes, it may be possible to eliminate the double buffering.)

   CMCS will use the standard Multics interfaces for terminal I/O. The only control imposed by CMCS is that the runtime package will check to verify that a queue or terminal is enabled before it attempts to do I/O with that target.

5. COBOL application programs will be invoked either implicitly or explicitly. Implicit invocation (from a queue going non-empty) requires one input argument, the COBOL hierarchy tree path.

   Explicit invocation of a COBOL MCS program uses no arguments because the program must define the specific queue it wishes to access.

6. Every process must attach its station before proceeding further. The first attachment initializes the user's environment for CMCS processing. At that point, the user can perform any authorized CMCS operation.

7. All message queues and system control tables for a given set of users will be contained in a single, user-specified directory. If desired, a different set of users can operate in a different directory.

8. The initial COBOL MCS facility is oriented strictly to COBOL; other languages may be used but they must interface to the COBOL runtime software. Integration with the forthcoming transaction processing system will occur, wherever possible, after user interfaces become defined; however, it is a goal that no user source program changes or recompilations will be necessary. (Integration is subject to the time constraints of the MR6.0 release.)

9. The physical integrity of all CMCS queues and control segments will be protected from indiscriminate user QUITs by appropriate use of IPS masking.

   In addition, a cleanup handler in cobol_mcs_, the single user (object program) interface, will perform appropriate cleanup of all messages in process.

10. The COBOL language specification requires user programs to specify a password for the ENABLE and DISABLE verbs. User documentation will stress the need to avoid putting literal passwords into program source.

DESIGN_ASSUMPTIONS

o All COBOL application programs are benign, i.e., they will always access CMCS queues and control segments through the CMCS runtime interface. If this policy is violated, correct operation is highly unlikely.

o Most messages will be short. Thus, a message sent to multiple destinations causes a copy of the message to be placed in each destination's queue.

o When a user sends only a portion of a message, it is likely that it is either a long report or a file data copy. Thus, a maximum size holding buffer must be used. For this reason, a temporary segment is assigned to each queue when the user sends a partial message to that queue.

o vfile_ recovery is coming for MR7.0. By using indexed vfiles for the message queues, CMCS will be able to take advantage of the recovery features.

o The meaning of message length (to a COBOL program) becomes ambiguous if the slew controls are imbedded in the data. For this reason, the slew control information is kept separate from the message text until the message is to be sent to an output device.

o The COBOL-74 Communications Module description is vague about the number of passwords needed from CMCS. Thus, only one password, at the CMCS system level, will be used. This one password must be matched by all users wishing to perform enable/disable functions.

o All users are benign, in that they will access CMCS only through the proper interfaces. Thus, there is no use of lower rings for database protection nor are there any special gates developed for CMCS. (This change can be made in the future without affecting user programs.)

o Since the primary goal of this implementation is to satisfy the (undefined) audit routines, tools are not planned for administrative functions that can be performed manually with other Multics commands, i.e., ACL setting.

o Tree Path Policy

It is implied but not required in the COBOL-74 Standard, that the COBOL Application Program use the absolute tree path of the target queue to receive subsequent pieces of a message. To eliminate ambiguities in the processing of receives with tree paths that are subsets of an absolute tree path for a receive in process (not all segments of the message have been read), the following rule is established:

A receive request with a tree path of a.b, that is resolved with a message from a.b.c, must be completed before any other message request from a or a.b. can be processed. Any attempt to use just a.b will be rejected.

Continuation receives for a.b.c are valid (and appropriate), as will be a receive request addressed to a.b.d or a.b.f, where a.b.d and a.b.f are also absolute tree paths.

As an example,

Tree Paths	In-process	Queue Name
a		
a b		
a b c	yes	queue_1
a b d	yes	queue_2
a b f	yes	queue_3

given that requests for absolute tree paths a.b.d and a.b.f are in-process, a new request for tree path a.b would be rejected with cmcs_error_table_$ambiguous_tree_path. This would cause a status key of "20" to be returned to the requesting COBOL application program.

RESPONSES TO OTTO NEWMAN QUESTIONS

The following are responses to questions raised by Otto Newman in his "Preliminary MCS Subroutine Specifications." The parenthesized numbers indicate the corresponding numbers of the questions.

(1) There is one type of message queue, but it will be used in two different ways. One use will be to hold the output messages for terminals until they are written to the output device. The other use will be to hold input messages until they are read by the COBOL application programs.

Because there is minimal distinction between the the queue uses, it would be very little trouble to generalize this capability and allow COBOL application programs to create output messages that would in turn be read by other COBOL application programs. This concept is basic to a transaction processing capability.

(2) In the Multics implementation, there is no physical distinction between queues accessed for receives and queues used to hold messages for destinations.

Because of this, the CMCS queue hierarchy definition must include the specification of all queues, both application queues and destination queues.

In send operations, the destination is translated into a station name, and thus has a specific queue assigned to hold the messages for output to a terminal device.

(3) In the initial implementation, the user can attach ("own") only one station (to receive output as a destination). The process does not "own" the queues it accesses for normal receives.

The definition of password usage in the COBOL Standard is somewhat ambiguous. Thus, for the present, only one password will be used to validate all enable and disable requests. If there is a future clarification that requires multiple passwords, this can be changed.

(4) EMI and EGI message delimiters are processed identically by the runtime package. Differentiation in the meaning of these two (logical) delimiters is left totally to the user software.

(5) Yes. However, attempting to do a send and a receive on the same tree path in the same process is not allowed in the Multics implementation. Once either of the operations is completed, the other can be started without any constraints. Note that this implies a one-level tree path; otherwise,

there would be no way to match up the input source queue with the output destination queue.

For multiple processes, locks are used for critical areas of queue manipulation. Locking is kept to a minimium to reduce inter-process interference.

(6) Only the particular message being received is locked on an extended basis. The entire queue is locked only long enough to accomplish the message lock and changing the status lists.

(There is no distinction made between EMI and EGI in the Multics implementation.)

(7) A design choice was made to require a process to specify its CMCS directory explicitly. This is done with the cobol_mcs command. This constraint could be relaxed in the future.

UNRESOLVED DESIGN ITEMS

- How much integrity protection should be built into the system to protect benign, but careless, users from themselves? (The facility will perform appropriate processing of user QUITs with IPS masking.)

- Does this design preclude extensions for security, extension, or accounting?

- In the standard definition of COBOL MCS, the entering of messages into the system and printing output are defined to be terminal activities, with the implication that the processing of input messages is independent from the terminal activity. Is it legitimate to have the same Multics process that enters a message into the system to also process that message and generate its own output?

REQUIRED_QUEUE_OPERATIONS

The following items are my interpretation of the requirements of the COBOL standard:

1. Write (SEND) message data with 0, ESI, EMI, or EGI. 0, ESI, EMI, and EGI are logical trailing delimiters, corresponding to end of partial message, end of (message) segment, end of message, and end of (message) group, respectively. A message group consists of one or more messages and a message consists of one or more message segments. When writing the message to the queue, only the highest level delimiter specified.

   Only when message data is written with EMI or EGI is the message to be made available in the queue to readers. (The implementation will assume that the complete message will be good, and that message segments are written directly to the queue, rather than holding them in a temporary buffer until they are known to be good. Correspondingly, some means must be provided by which queues can be "purged" of truly invalid, or incomplete, messages.)

   Messages and also message segments from multiple writers can be written to a queue concurrently. Thus, message segments for a given message are not necessarily contiguous in the queue.

   when the COBOL application program issues a send of a partial message (zero value delimiter), the system will create a temporary buffer in which to concatenate subsequent pieces. Only when the message (or message segment) is complete, will it be written to its queue(s).

2. Read (RECEIVE) message or message segment data from a queue. If the receiving buffer is smaller than the string of message data, the system will cause only that sized portion of the message entity to be written into the buffer and will provide the next increment, up to the proper (logical) delimiter, upon subsequent calls to the same queue. (The system must maintain the necessary record locks and pointers; by ANSI definition, these functions are external to the user program.

   Another implication is that the system must be able to retrieve discontinuous portions of message data. For example, assume that a program sent a _message_ to a queue as two _message segments_ of 100 characters each. Another program did a receive message to the queue and got that message. However, its buffer was only 75 characters long. Thus, the system would move data into the receiving buffer in the following manner:

```
Step     (message segment, character positions)
  1      (1, 1-75)
  2      (1, 76-100) (2, 1-50)
  3      (2, 51-100)
```

The receiving program would continue to issue receives until its message delimiter status went from zero to EMI.

3. Enable a queue and disable a queue. This is nothing more than allowing or disallowing user access to a queue. In COBOL, this a user-program function, requiring a password.

   There is a corresponding facility to enable and disable a station or stations from access to all or specific queues.

   The setting of a disable flag does not necessarily immediately prevent a terminal from accessing a queue for input or output. Operations that were begun before the disable flag was set are allowed to continue to completion.

4. Get counts of messages in a queue hierarchy or a single queue. This is the momentary count of all valid (complete) messages in the queue. It specifically does not include any partial messages. (It also does not include any messages that are locked to other processes. This is my inference of the ANSI standard; it will be clarified.)

5. Purge partially sent messages. This is not in ANS Standard COBOL; however, it is expected that ANSI will adopt the currently defined PURGE verb from the CODASYL JOD in the next update of the Standard.

   In the Multics implementation, procedures to perform this function already exist to enable users to clean up the queues.

COBOL_MCS_QUEUE_ORGANIZATION

QUEUE-XXX

RCD-0/1	RCD 1/1	RCD 2/1	RCD 2/2	RCD 3/1
CONTROL	MESSAGE	MESSAGE	SEGMENT	MESSAGE
INFO	HEADER	HEADER	HEADER	HEADER
	SEGMENT	SEGMENT	DATA	SEGMENT
	HEADER	HEADER		HEADER
	DATA	DATA		DATA

Notes:

1. Record zero is always reserved to contain control information used in the processing of messages in the given queue.

2. Any given message can span multiple non-contiguous records of the file. Only the first record for a given message will contain a message header. Subsequent pieces (segments) contain just the segment header and data.

3. Keys for the vfile records consist of two adjacent fields of fixed bin (35) values. The first field contains the ordinal number of the CMCS message. The second field holds the ordinal number for message segments within the given message. Message numbers and message segment numbers always begin with the value one. When the physical queue is created, a record with key values of 0/1 is stored. This record will contain global control information for that queue. It is called the queue control record.

4. Part of the header record for each message will be a pair of forward and backward pointers. The message header of a message will be linked into a list of pointer pairs, based upon the status of the record.

   Current status codes are the following:

   1 - send in process (message being built)
   2 - send complete (available for processing)
   3 - receive in process
   4 - receive complete (ready for deletion)

QUEUE DESIGN CONSIDERATIONS

The choice of an indexed vfile_ file structure for the message queues is based on the following considerations:

1. Message activity can be such that it may not be possible to store all messages in a single segment. The vfile_ organization eliminates this problem. A related requirement is that the sum of the pieces of any given message can be larger than a single segment. (The Multics CMCS implementation currently has a design constraint that any one piece not exceed one segment.)

2. The pieces of a message may not be contiguous. This can easily occur if there are multiple concurrent writers to the file. This problem is eliminated by the indexed file organization. A two-level key of message_number, piece_number, provides nearly direct access to each piece as needed.

3. Messages are variable in length. The internal vfile_ space management facilities eliminate the need for any similar code in CMCS procedures.

4. Linking of specific messages into any one of the message status lists (incomplete, complete, in process, etc.) requires an addressing mechanism that is process independent. The vfile_ record_status control operation solves this problem by returning an address consisting of MSF component number and offset.

5. The forthcoming file integrity extensions to vfile_ will also be available to COBOL MCS.

OVERVIEW_OF_CMCS_DATA_BASES

For a given set of users, the databases described and the associated message queues must reside in a single directory. A different set of users can have their control segments and queues in a different directory.

It is apparent that some databases contain very little data. This is undesirable from a performance standpoint, but that is not an issue for MR6.0, and it does simplify the implementation.

- cmcs_terminal_ctl.control
  This database provides the default station_id for interactive users (based on user-device channel).

- cmcs_tree_ctl.control
  This database contains the template definitions of all CMCS queue hierarchies for a given set of users. This segment is copied to the process_dir during the user's CMCS initialization and is then dynamically updated with user-specific information for each entry used.

- cmcs_station_ctl.control
  This database defines all legitimate stations and contains per-station flags to indicate enable/disable conditions.

- cmcs_wait_ctl.control
  This database is shared by all processes performing a receive with wait. Entries are searched by queue hierarchy on a first come, first served basis.

- cmcs_system_ctl.control
  Initially, this segment will contain only the CMCS-wide password (up to 10 characters), used in granting permission to perform the enable and disable functions as given in the language.

- cmcs_queue_ctl.control
  This database contains the flags for enable/disable functions on a per-queue basis. Additionally, it holds the message status counters and linked-list ptrs for each queue. Entries in this table are searched to find occurrences of available messages before the actual queues are accessed.

- cmcs_user_ctl.control
  This per-process, external static database contains all the common parameters used by the various CMCS subroutines.

USER_COMMANDS_SUMMARY

The following commands are provided as part of the COBOL MCS runtime support facility.

cobol_mcs

This command serves to establish the environment for further CMCS processing. Users specifying the terminal option can perform any of the functions of the COBOL send, receive, enable, disable, accept message count, and purge verbs. If the station option is used, the command will perform the process initialization and then return to command level. This would normally be in preparation for the execution of a COBOL application program.

cobol_mcs_admin

This administrative command currently has only two functions. The first is to create necessary control segments and message queues for a given CMCS directory, using the cmcs_tree_ctl.control segment as input. The second is to set or change the system-wide password, used to validate enable or disable requests.

cv_cmcs_station_ctl
cv_cmcs_terminal_ctl
cv_cmcs_tree_ctl

These three commands are all control segment compilers. They are based on the reduction_compiler tool. They each read a source file called cmcs_XXX_ctl.src, where XXX is station, terminal, or tree, respectively, and generate a binary control segment of the name cmcs_XXX_ctl.control. Subroutines that access the binary control segments have a name of the form cmcs_XXX_ctl_.

Name: cobol_mcs, cmcs

The cobol_mcs command provides a command interface to the COBOL Message Control System (CMCS) functions in a manner similar to that used inside a COBOL program. Refer to the *Multics COBOL Reference Manual*, AS44, for a complete description of the Communications Module.

The first time this command is invoked in the user's process, the process will be initialized for the execution of all subsequent CMCS operations. If the process is to operate as a CMCS terminal, the command will read subcommands from the user_input switch.

Usage

```
     cobol_mcs cmcs_dir {control_arg}
```

where:

1. cmcs_dir
   is the path of the directory containing the desired CMCS message queues and control segments.

2. control_arg
   is one of two control_args. It must have the value -station and be followed by a valid CMCS station_name, or, it must have the value -destination (-ds) and optionally be followed by a station_name. If the station_name is not given for the -destination control_arg, a default station_id will be used.

   The use of -destination will cause the process to be initialized to act as a CMCS terminal/destination. The use of -station will cause the process to be initialized for subsequent use by a COBOL application program.

Notes

For MR6.0, this command must be invoked to initialize the user's process before any other COBOL MCS functions are performed.

Once cobol_mcs is invoked for terminal operations, the command will read requests from the user_input switch. The requests supported are receive, send, enable, disable, accept, purge, and quit. They are identical in function, although slightly different in format, to the corresponding verbs described in the *Multics COBOL Reference Manual*, Order No. AS44, in the Communications Module. (purge is described in the CODASYL JOD.) Full ANSI COBOL-74 Level 2 support is provided.

Request Summary

The requests and their abbreviated forms are listed here; detailed descriptions follow the list, in the same order.

accept_message_count, amc
: prints a count of all messages available in the specified queue hierarchy

receive, r
: prints and deletes all messages available in the specified queue hierarchy

send, s
: sends input lines as messages (or message segments) to the destinations specified (partial messages are first accumulated until they are complete, before writing to the destinations).

enable_input, ei
enable_input_terminal, eit
enable_output, eo
: enables a queue hierarchy, a station, or a set of stations, respectively, for further activity.

disable_xxx, dxxx (same format as enable)
: disables a queue hierarchy, a station, or a set of stations, respectively, from further activity, after the currently active messages have been processed.

The distinction is made because the COBOL definition requires that messages currently being processed must be completed before the queue or the terminal is disabled.

purge, p
: causes all partially sent messages to be deleted, and all partially received messages to be marked again as available.

quit, q

causes the cobol_mcs command to purge any incomplete send and/or receive messages and then return to command level.

Request

accept_message_count tree_path

Where tree_path is a character string of the form a.b.c.d. The components, a, b, c, and d represent the four levels of a CMCS queue hierarchy (maximum). The level names must be alphanumeric (including underscore), and can be from 1-12 characters in length; trailing blanks are appended internally when appropriate. At least the first component is required; trailing components are necessary only to define the desired level in the queue hierarchy.

This request will cause the command to search the appropriate queues for a count of all messages currently available for processing (send complete) and print the sum on the user_output switch. Unless there are no other CMCS users on the system, the availability of messages may change between the accept_message_count and any subsequent receive requests.

Request

receive delim tree_path

The delim argument is either esi or emi, to indicate that either a mesage segment, or an entire message is desired.

The tree_path is as defined above.

The receive request will cause the appropriate queue or queues to be searched for the first available message. If none is found, a "No messages" comment will be generated and the command will return to request level. (The cobol_mcs command interface will never "wait" for a message to become available.) If one is found, the message will be printed on the user-output switch, along with any appropriate slew-control data.

After the complete message is printed, it is deleted from the queue. When performing a receive_segment request, it will find the first available message and print the first segment of that message. Subsequent invocation of the request with either a null or the same tree_path will cause the following segments of the message to be printed. The message will be deleted after the last segment of the message is printed.

Request

```
send delim dest1 {dest2 ... destn}
```

Where delim is as described above.

dest1 ... destn is a list of one or more destinations to which the message will be sent. The list is required at least for the first send operation. If the list is omitted on subsequent sends, the previous list will be used.

The send request duplicates the function of the send verb with one of the four logical delimiters (0, ESI, EMI, EGI), respectively.

when any of the send requests are issued, the command will enter an input mode, similar to that in the EDM. Lines of input are accumulated until a line is input with just a single period. At this point, the accumulated data will be sent to the appropriate destination(s).

Request

```
disable_input, di tree_path
disable_input_terminal, dit station_name
disable_output, do dest1 {dest2 ... destn}
```

Where tree_path and desti are as described above.

The disable requests will first request a password, using the terminal's non-print function or a mask to avoid the printing of the password. The password response is encoded and then compared to the CMCS system password. If equal, the command will then process the arguments on the request line. The result is identical in function to that of the disable verb in the COBOL language.

Request

```
enable_input, ei  tree_path
enable_input_terminal, eit  station_name
enable_output, eo  dest1 {dest2 ... destn}
```

The enable requests operate identically to that of the disable requests, except that they enable the specified queues or terminals.

Request

```
purge s {dest1 ... destn}
purge r {tree_path}
purge all
```

The purge s request will cause all partially sent messages (for that process) being sent to the listed destinations to be deleted. If no destination list is given, all partially sent messages will be deleted.

The purge r request will cause all partially received messages in the specified tree_path to be returned to available status. If no tree_path is given then all partially processed receive messages (for that process) will be returned.

The purge all request will perform a purge of all partially processed send and receive messages (for that process).

Request

```
quit
```

The quit request will cause an implied purge all request and then cause the cobol_mcs command to return to command_level.

ADMINISTRATIVE FUNCTIONS

The CMCS administrator must define and generate the system databases (and their containing directories). The following databases require a source segment for compilation by CMCS compilers:

cmcs_station_ctl.control
cmcs_terminal_ctl.control
cmcs_tree_ctl.control

The station control segment must be generated before the terminal and tree control segments. After the tree control segment is generated, the cobol_mcs_admin command is used to create all queues and additional control segments:

cmcs_queue_ctl.control
cmcs_wait_ctl.control
cmcs_system_ctl.control

After the system control segment is created, the administrator must use the set_system_psw request of the cobol_mcs_admin command to set the initial password for the CMCS system.

Additionally, the administrator must manually set the ACLs on all CMCS segments, as appropriate for the given set of users. All segments, with the exception of the cmcs_system_ctl.control and cmcs_tree_ctl.control segments, must have read and write access for all users. Only the administrator need have write access on the cmcs_system_ctl.control segment.

The cmcs_tree_ctl.control segment MUST have only read access for all users and have the copy switch set ON. This segment is copied to the user's process_dir, so that it can be updated with process-specific information.

Functions, such as accounting, recovery, metering, etc., are not provided in MR6.0 CMCS.

Name: cobol_mcs_admin, cmcsa

The cobol_mcs_admin command is used to perform the software functions involved in COBOL MCS administration. When the command is invoked, it will then read request lines from the user_input switch. Command usage is terminated when the user enters the quit request.

Usage

```
     cobol_mcs_admin   cmcs_dir
```

where cmcs_dir is the path of the desired CMCS directory.

Notes

As mentioned above, the command will read request lines from the user_input switches until it encounters the quit request. At that point, the cobol_mcs_admin command will return to command level.

Requests

set_system_psw, sspsw

The user will be asked twice for a new password, the second time to verify correctness of the first, where password is a character string of 1-10 characters in length.

The password given will be encoded and written on top of the old password without verification of the old password.

change_system_psw, cspsw

The password is as defined above.

The user will be asked for the old password. If the response is correct, the command will request the new password. It will request it a second time to verify that the first typein was correct. The password given will then be encoded and written on top of the old password.

create_cmcs_queues, ccq

This request uses no arguments. The command will read the cmcs_tree_ctl.control segment in the cmcs_dir directory to obtain the defined queue names and the command_line control information associated with each queue. It will create these queues if they do not already exist; if a given queue already exists, the queue will be truncated, a warning to that effect will be printed, and the command will continue. In addition, the command will create (or recreate) the cmcs_queue_ctl.control segment. This segment will contain a list of the queues and, for each queue, the message status lists and flags for enabling/disabling input and output to and from the queues, respectively.

CONSTRUCTING_COMMAND_AND_ABSIN_LINES

TO BE SUPPLIED

Name: cv_cmcs_station_ctl

This COBOL MCS administrative command compiles a source file named cmcs_station_ctl.src into a binary control file that will be accessed by the CMCS runtime subroutines.

This particular command converts a source file that contains a list of all valid stations. This list will be the master file for all station names.

The source file has the following format:

```
<station_name>;
     .
     .
     .
end;
```

NOTES

The binary control file is set up with the standard CMCS header, which gives information about the compilation and the table sizes.

This compilation must be done before either the cmcs_tree_ctl or cmcs_terminal_ctl compilations are done because the cmcs_station_ctl.control segment is used to validate the station names used in the other source files.

Name: cv_cmcs_terminal_ctl

This COBOL MCS administrative command compiles a source file named cmcs_terminal_ctl.src into a binary control file that will be accessed by the CMCS runtime subroutines.

This particular command converts a source file giving pairs of terminal subchannels (tty_ device_channels) and their default station names.

The source file has the following format:

```
<terminal_subchannel>:        <default_station_name>;
     .
     .
     .
end;
```

NOTES

The binary control file is set up with the standard CMCS header, which gives information about the compilation and the table sizes.

This compilation must be done after cmcs_station_ctl.control is generated; it uses information in that segment to validate the station names.

Name: cv_cmcs_tree_ctl

This COBOL MCS administrative command compiles a source file named cmcs_tree_ctl.src into a binary control file that will be accessed by the CMCS runtime subroutines.

This particular command converts a source file that defines the COBOL MCS queue hierarchy, along with controls to be associated with each message queue.

The source file has the following format:

(See below for a lengthy description.)

NOTES

The binary control file is set up with the standard CMCS header, which gives information about the compilation and the table sizes.

This compilation must be done after cmcs_station_ctl.control is generated; it uses information in that segment to validate the station names.

Source Format for the Tree Definition

1. There can be any number of subtrees, and each subtree can have from 1-4 levels. The "root" for all subtrees is an implied one, allowing the CMCS administrator complete flexibility in the hierarchy definition.

2. The queue names specified in the queue hierarchy are entirely logical. For the terminating queue name in any branch, there is an associated name for the physical message queue. The physical queue name defined in the source will be suffixed with the string ".cmcs_queue", when the actual queue is created.

3. The source file will consist of one or more PL/I-like structure declaration statements. Each statement must begin with a "declare" or "dcl" and end with a semicolon (;). The structure can be up to four levels deep. All lines but the last line of a statement are terminated with a comma (,). The source file is terminated with a statement of "end;". PL/I-style comments may be included.

4. Immediately following the level indicator is the hierarchy level name; this is the only required argument. An optional control for any line is the "command <command_line>" (or "absin <absin_line>"). This control is in effect for the current level and all subordinate levels, unless overridden at a subordinate level. The <command_line> argument will be executed, if appropriate, when the associated queue goes non-empty.

5. When a particular level is the final level of a given tree-path (terminal node), the "queue <queue_name>" control must be given. This will identify the desired name for the given physical message queue.

6. Since all physical queues will exist in the same directory (for the one source declaration), the associated names should be unique. Since they will have a suffix appended (.cmcs_queue), the names must be fewer than 22 characters in length.

Syntax_Example

```
declare 1 <level_name>,  /* for RECEIVES */
        2 <level_name> command "<command_line>",
        3 <level_name> queue <queue_name>,
        3 <level_name> queue <queue_name>,
        2 <level_name> queue <queue_name>;

dcl     1 <level_name> queue <queue_name>
          absin "<absin line>";

dcl     1 <level_name> queue <queue_name>
           command "<command_line>";

/* declaration of a one-level send queue */
dcl     1 <level_name> queue <queue_name>;

end;
```

In this example, the command control given in the first level 2 line causes the <command_line> to be set for both of the level 3 queues immediately following. Effect of the command control is terminated by the following level 2 line.

MAJOR_SUBROUTINE_SUMMARY

It should be noted that none of the CMCS subroutines are intended to be called directly by user source code. The compiler generates the object code calls according to use of COBOL MCS verbs in the source.

```
cobol_mcs_$(receive, receive_wait, send, enable_input_queue,
     enable_input_terminal, enable_output,
     disable_input_queue, disable_input_terminal,
     disable_output, purge, stop_run)
```

This subroutine is the one interface module for the COBOL application programs. In general, it serves as a transfer vector to the function-specific cmcs_XXX_ctl_ subroutines. It does do some reformatting of the data received from the COBOL programs to a more efficient internal form.

```
cmcs_tree_ctl_$(find_tree_path, find_destination,
     find_index)
```

This subroutine uses the tree control segment in the process_dir. It will search for the first occurrence of an entry that matches the tree_path given in the input CD structure, or the entry that matches a given output destination. The find_index entrypoint just returns a pointer to a specific entry.

```
cmcs_terminal_ctl_$(find)
```

This subroutine does nothing more than to return a default station name for a given terminal device_channel.

```
cmcs_status_list_ctl_$(add move delete)
```

This subroutine maintains the linked lists that group together all messages of a particular processing status. Separate lists are maintained for each queue.

```
cmcs_queue_ctl_$(send, receive, enable, disable, purge)
```

This subroutine performs the functions of message addition and deletion to/from the message queues (thru vfile_). For message reading, it locks the next available message to the process and returns message location and length information to the reader

```
cmcs_station_ctl_$(find, lock)
```

This subroutine searches the cmcs_station_ctl segment. Its current use is to find a given station and return a pointer to that entry. If desired, the entry can be locked to the calling process.

cmcs_wait_ctl_$(add, delete, find)

This subroutine maintains the list of processes who have done a receive with wait and no message was available for processing. The processes are added to the wait list and put to sleep until a message becomes available. When a queue becomes non-empty, the list is checked to see if any process is waiting on that queue and if so, the process is awakened, etc.

cmcs_station_ctl_$(disable_input_terminal, disable_output_terminal, enable_input_terminal, enable_output_terminal, attach, detach, validate, input_disabled, output_disabled)

This subroutine performs all functions related to the "attaching" and "detaching" of stations and the processing of enable and disable functions.

cmcs_fillin_hdr_

This subroutine performs the general initialization of the common header used in most CMCS files.

I/O_TYPES_AND_SUBTYPES

Throughout all CMCS software, the following codes will be used to identify major I/O types and subtypes:

Major Type (VERB)	Subtype
1 - Send	1 - Partial
	2 - Segment
	3 - Message
	4 - Group
2 - Receive	1 - Segment (No Wait)
	2 - Message (No Wait)
	3 - Segment (Wait)
	4 - Message (Wait)
3 - Enable	1 - Input (to a queue)
	2 - Input Terminal
	3 - Output
4 - Disable	1 - Input (to a queue)
	2 - Input Terminal
	3 - Output
5 - Accept Message Count	1 - Single Subtype
6 - Purge	1 - Sends Only (CODASYL)
	2 - Receives Only (Multics)
	3 - Both (Multics)

Name: cobol_mcs_

This COBOL MCS subroutine serves as the single interface module between the COBOL object program and the CMCS runtime package.

When it is called the first time in a process, it will ensure that the proper environment has been established for subsequent CMCS processing.

All entrypoints and their usages are described in Otto Newman's memo, Preliminary MCS Subroutine Specifications. Information from this document will be integrated into the release documentation. (It is available in BCO from Otto Newman, in CISL from Betsy Kerr, and in Phoenix from Bob May.)

Name: cmcs_tree_ctl_

This COBOL MCS subroutine is used to find entries in the copy of cmcs_tree_ctl.control, contained in the user's process_dir. Given an input CD pointer or an output station name, it returns an index and a pointer to the entry matching the given tree_path or destination station name, respectively. The last entry returns a pointer to the entry corresponding to a given index.

Entry: cmcs_tree_ctl_$find_tree_path

This entrypoint finds the first entry in tree control that matches the tree path specified in the input CD structure. This may be the top of a subtree and if so, the count returned will indicate the number of entries in the subtree below. If the count is zero, the entry found matches an absolute tree path, and is for a message queue.

Usage

```
     dcl cmcs_tree_ctl_$find_tree_path entry (ptr,fixed bin,fixed
          bin,ptr,fixed bin (35));

     call cmcs_tree_ctl_$find_tree_path
          (input_cdptr,entry_index,count,entry_ptr,code);
```

Where:

1. input_cdptr (Input)
   is a pointer to the input CD structure in the COBOL program.

2. entry_index (Output)
   is the index into the tree control segment for the entry matching the tree path.

3. count (Output)
   is the count of entries in a subtree for the matching entry. If zero, it indicates the entry is for a message queue.

4. entry_ptr (Output)
   is a pointer to the matching entry.

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

Notes

----Notes for find_tree_path

Entry: cmcs_tree_ctl_$find_destination

This entrypoint is used to find the entry that matches the given (single) station name. It is used to process send requests.

Usage

```
dcl cmcs_tree_ctl_$find_destination entry (char (12),fixed
     bin,ptr,fixed bin (35));

call cmcs_tree_ctl_$find_destination
     (station_name,entry_index,entry_ptr,code);
```

where:

1. station_name (Input)
   is the name of a destination station.

2. entry_index (Output)
   is the index of the matching entry.

3. entry_ptr (Output)
   is a pointer to the matching entry.

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

Notes

----Notes for find_destination

Entry: cmcs_tree_ctl_$find_index

This entrypoint is used to find subsequent entries after a find_tree_path.

Usage

```
     dcl cmcs_tree_ctl_$find_index entry (fixed bin,ptr,fixed bin
          (35));

     call cmcs_tree_ctl_$find_index (index,entry_ptr,code);
```

Where:

1. index (Input)
   is the index of the desired entry.

2. entry_ptr (Output)
   is as described above.

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

Notes

----Notes for find_index

Name: cmcs_queue_ctl_

This COBOL MCS subroutine is called by cobol_mcs_ to manipulate the message queues and the control information contained in cmcs_queue_ctl.control. It accesses messages in the indexed files through vfile_ IO module, using a two-level key. This subroutine also performs the actual movement of data to and from the buffers of the COBOL programs.

Global locking of queues is done by vfile_ for record addition and deletion; additional locks are used to manipulate the control information associated with the queues. These locks are essentially independent from one another.

Entry: cmcs_queue_ctl_$send

This entry is the major procedure for all send functions. It uses the output CD pointer supplied by the COBOL application program to obtain the specific send controls to be used.

Usage

```
     dcl cmcs_queue_ctl_$send entry (ptr, fixed bin, ptr, fixed
          bin, bit (36), fixed bin (35));

     call cmcs_queue_ctl_$send (output_cdptr, io_subtype,
          buffer_ptr, buffer_len, final_delim, slew_ctl, code);
```

Where:

output_cdptr
: is the pointer to the COBOL program output CD structure (Input).

io_subtype
: is the specific type of send operation required (Input).

buffer_ptr
: is the pointer to the COBOL program buffer space (Input).

buffer_len
: is the length of the message in the program buffer (Input).

final_delim
: is the logical delimiter to be assigned to the current message portion (Input).

slew_ctl
: is a field defining the slew control operations to be performed when the message is sent to an output device (Input).

code
: is a standard status return (Output).

Notes

The io_subtype mentioned above indicates the type of message delimiter to be used for the current portion of the output message. It can have a value of 0-3.

0 This means that the COBOL program is sending out only a piece of a message. Message pieces will be accumulated in the process space (a temporary segment) until a delimiter of 1, 2, or 3 is specified to terminate the current piece.

1 This delimiter specifies that the message buffer is to be sent out as a message segment.

2 This delimiter specifies the the message buffer is to be written out as a message.

3 This delimiter specifies that the buffer is to be written with a message group delimiter.

Entry: cmcs_queue_ctl_$receive

This entry is the major procedure to perform the receive functions. It finds an available message and moves it to the COBOL program buffer. It manages the movement of partial messages to the user buffer and controls the subsequent processing of receives to move subsequent pieces of a message into the user buffer. If a wait function is required, it will initiate this.

Usage

```
     dcl cmcs_queue_ctl_$receive entry (ptr, fixed bin, ptr,
          fixed bin, fixed bin (35));

     call cmcs_queue_ctl_$receive (input_cdptr, io_subtype,
          buffer_ptr, buffer_len, code);
```

Where:

input_cdptr
: is a pointer to the COBOL program input CD structure (Input).

io_subtype
: is the type of receive function required (Input).

buffer_ptr
: is a pointer to the user input buffer (Input).

buffer_len
: is the maximum number of characters that can be stored into the buffer (Input).

code
: is a standard status code (Output).

Notes

1. If I/O is already in process, and it is also for a receive, it will continue that I/O. If the new request is for a send, it will be aborted.

2. If I/O is not in process, cmcs_queue_ctl.control will be checked for available messages in the desired subtree. If the function is a receive no-wait, and no messages are available, it will set a status key and return.

3. If there is a message, the file will be attached, opened (if not already), and the record locked to the given process. The message will then be moved into the user buffer.

4. If the entire message could be copied into the user buffer in one pass, the message status will be changed from available to complete. If not, the message status will be set to receive-in-process.

5. If no message is available and the receive specified a wait, a wait request will be set into cmcs_wait_ctl.control, and the process will be put to sleep, waiting for a new message.

Entry: cmcs_queue_ctl_$enable

This entrypoint causes all queues specified by the tree_path given in the input CD to be enabled for input.

Usage

```
dcl cmcs_queue_ctl_$enable entry (ptr, fixed bin, char (10),
     fixed bin (35));

call cmcs_queue_ctl_$enable (input_cdptr, io_subtype,
     password, code);
```

Where:

input_cdptr
: is as described above (Input).

io_subtype
: must be 1 (Input).

password
: is a password to be encoded and matched with the CMCS system-wide password (Input).

code
: is a standard status return (Output).

Notes

----notes for enable

Entry: cmcs_queue_ctl_$disable

This entry is the main procedure to perform queue disable functions.

Usage

```
dcl cmcs_queue_ctl_$disable entry (ptr, fixed bin, char
     (10), fixed bin (35));
     call cmcs_queue_ctl_$disable (input_cdptr, io_subtype,
          password, code);
```

Where:

input_cdptr
: is as described above (Input).

io_subtype
: is the disable function required. It must have a value of 1 (Input).

password
: is as described above (Input).

code
: is a standard status return (Output).

Notes

----notes for disable

Entry: cmcs_queue_ctl_$accept_message_count

This entrypoint is the main procedure to obtain the count of all available messages in the tree_path subtree specified in the input CD.

Usage

```
     dcl cmcs_queue_ctl_$accept_message_count entry (ptr, fixed
          bin, fixed bin (35));

     call cmcs_queue_ctl_$accept_message_count (input_cdptr,
          io_subtype, code);
```

Where:

input_cdptr
: is as described above (Input).

io_subtype
: is as described above. It must always have the value of 1 (Input).

code
: is a standard status return (Output).

Notes

The total count is stored back into the input CD structure.

Entry: cmcs_queue_ctl_$purge

This entrypoint is the main procedure to perform all purge functions. In addition to the purge of partially sent messages as defined by the CODASYL JOD, it also is used to purge partially received messages, or both.

Usage

```
     dcl cmcs_queue_ctl_$purge entry (ptr, fixed bin, fixed bin
          (35));

     call cmcs_queue_ctl_$purge (cd_ptr, io_subtype, code);
```

Where:

cd_ptr
: is a pointer to either an input CD structure, an output CD structure, or null. The pointer definition must correspond to the function defined by the io_subtype. (Input).

io_subtype
: defines the type of purge to be performed (Input). It must have a value of 1-3, for specifying a purge of partially sent messages, partially received messages, or all partially processed messages, respectively.

Notes

In addition to the purge function called for by the COBOL application program, the system uses this procedure to clean up any unprocessed messages of either type. The purge "all" function is invoked when the COBOL program does a stop run command.

Name: cmcs_status_list_ctl_

This COBOL MCS subroutine is used to manage the message status lists in each of the message queues. It is called by cmcs_queue_ctl_ when it needs to change the status of a particular message.

Entry: cmcs_status_list_ctl_$add

```
dcl cmcs_status_list_ctl_$add  (ptr, ptr, fixed bin, code);

call cmcs_status_list_ctl_$add (rcd_loc_ptr, iocb_ptr,
     status, code);
```

rcd_loc_ptr points to the record locator. The record locator is a one-word structure giving the record location in terms of file component number and word offset (half_word values). This location is process-independent.

iocb_ptr points to the iocb of the target file. Status defines the particular status list for insertion. It must have the value of 1 or 2. Code is the standard return code.

Notes

This entrypoint must be used only to insert the locator for a new message into the status list, either in the list for send_incomplete (1), if only a message segment was written, or send_complete (2) if a complete message was written as one (vfile_) record.

Entry: cmcs_status_list_ctl_$delete

```
dcl cmcs_status_list_ctl_$delete  (ptr, ptr, fixed bin,
     fixed bin(35));

call cmcs_status_list_ctl_$delete   (rcd_loc_ptr, iocb_ptr,
     status, code);
```

Where all fields are as described above.

Notes

This entrypoint is called by cmcs_queue_ctl_ when a message (and all its segments) is to be deleted from the message queue. The status code will usually be 4, indicating that a message was successfully processed and is now being deleted. However, the status code could be 1, for example, indicating a purge of an incomplete message. The message specified by the rcd_loc_ptr will be deleted from the list.

Entry: cmcs_status_list_ctl_$move

```
dcl cmcs_status_list_ctl_$move  (ptr, ptr, fixed bin, fixed
     bin, fixed bin(35));

call cmcs_status_list_ctl_$move  (rcd_loc_ptr, iocb_ptr,
     old_status, new_status, code);
```

Where rcd_loc_ptr, iocb_ptr, and code are as described above.

The function of the move entrypoint is effectively that of a paired list delete and a list add.

The values given in old_status and new_status define the old status list from which the message is being removed and the new status list to which the message is being added, respectively.

Name: cmcs_wait_ctl_

This COBOL MCS subroutine manages the wait functions for all the processes that are waiting for a message to become available in any specified queue. Entrypoints are provided to add, delete and find wait_list entries in the cmcs_wait_ctl.control segment.

When the COBOL application program issues a receive-with-wait, it can specify any level up in the queue hierarchy. This means that the program wishes to receive the next message that is (or becomes) available anywhere in the specified subtree. If no message is available, the program's request will be added to the wait list and the process put to sleep.

When another process causes a message to become available in a queue, it will check the wait list to determine if any request can be satisfied. If so, the second process will update the specific wait entry with information about the new message and then send a wakeup to the waiting process.

The requesting process will then obtain the specific message controls from the wait entry and then delete the entry.

Entry: cmcs_wait_ctl_$add

This entry will add a process to the list of processes waiting to receive a message.

Usage

```
     dcl cmcs_wait_ctl_$add entry (char(48), fixed bin, fixed
          bin(35));

     call cmcs_wait_ctl_$add (rcv_tree_path, index, code);
```

where:

rcv_tree_path
: defines the hierarchy subtree from which a message is requested (Input).

index (Output)
: is the index of the wait entry. The awakened program will access this entry to obtain information about the available message.

code
: is a standard status code (Output).

Notes

Once generated, the control information needed to do an ipc_$block and hcs_$wakeup is fixed for the process. The subroutine will obtain this data directly from cmcs_user_ctl.control.

Entry: cmcs_wait_ctl_$find

This entry is called by the process that causes the number of available messages in a queue to go nonzero. Its function is to determine if any process is waiting for that message and, if one is found, send a wakeup to that process.

Usage

```
     dcl cmcs_wait_ctl_$find entry (char(48), char(32), fixed
          bin(35));

     call cmcs_wait_ctl_ (abs_tree_path, queue_name, code);
```

where:

abs_tree_path
: is the full tree_path used to define the message queue that went non-empty. This information will be set into the COBOL application program's CD structure (Input).

queue_name
: is the full entryname of the queue containing the message (Input).

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

Entry: cmcs_wait_ctl_$delete

This entry causes the given wait entry to be zeroed and moved to the free list.

Usage

```
     dcl cmcs_wait_ctl_$delete entry (fixed bin, fixed bin(35));

     call cmcs_wait_ctl_$delete (wait_ctl_eindex, code);
```

where:

wait_ctl_eindex
: is the index to the particular wait entry to be moved to the free list (Input).

code
: is a standard status code.

Name: cmcs_terminal_ctl_

This COBOL MCS subroutine performs a search of cmcs_terminal_ctl.control, to find the default station name for a given terminal device_channel.

Entry: cmcs_terminal_ctl_$find

This entrypoint does the work of searching the control segment, looking for the given device_channel. When one is found, the associated default station name is returned to the caller.

Usage

```
     dcl cmcs_terminal_ctl_$find entry (char (8),char(12),fixed
          bin(35));

     call cmcs_terminal_ctl_$find
          (device_channel,station_name,code);
```

Where:

1. device_channel (Input)
   is the name of the terminal subchannel, as found in the Channel Definition Table (CDT).

2. station_name (Output)
   is the default station name to be used by the interactive user.

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

Notes

----Notes for find

Name: cmcs_station_ctl_

This COBOL MCS subroutine controls all functions related to station attach, detach, enable, and disable.

Entry: cmcs_station_ctl_$disable_input_terminal

This entry uses the station_name from the input CD structure and causes that particular station to be disabled for input.

Usage

```
     dcl cmcs_station_ctl_$disable_input_terminal entry (ptr,
          char (10), fixed bin (35));

     call cmcs_station_ctl_$disable_input_terminal (input_cdptr,
          password, code);
```

where:

1. input_cdptr (Input)
   is a pointer to the input CD structure in the COBOL program.

2. password (Input)
   is the CMCS system password.

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

Notes

----Notes for disable_input_terminal

Entry: cmcs_station_ctl_$disable_output_terminal

This entrypoint uses the set of station_names from the output CD structure and causes those stations to be disabled for output.

Usage

```
dcl cmcs_station_ctl_$disable_output_terminal entry (ptr,
     char (10), fixed bin (35));

call cmcs_station_ctl_$disable_output_terminal
     (output_cdptr, password, code);
```

where:

1. output_cdptr (Input)
   is a pointer to the output CD structure in the COBOL program.

2. password (Input)
   is the CMCS system password.

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

Notes

----Notes for disable_output_terminal

Entry: cmcs_station_ctl_$enable_input_terminal

This entry uses the station_name from the input CD structure and causes that particular station to be enabled for input.

Usage

```
dcl cmcs_station_ctl_$enable_input_terminal entry (ptr, char
     (10), fixed bin (35));

call cmcs_station_ctl_$enable_input_terminal (input_cdptr,
     password, code);
```

where:

1. input_cdptr (Input)
   is a pointer to the input CD structure in the COBOL program.

2. password (Input)
   is the CMCS system password.

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

Notes

----Notes for enable_input_terminal

Entry: cmcs_station_ctl_$enable_output_terminal

This entrypoint uses the set of station_names from the output CD structure and causes those stations to be enabled for output.

Usage

```
     dcl cmcs_station_ctl_$enable_output_terminal entry (ptr,
          char (10), fixed bin (35));

     call cmcs_station_ctl_$enable_output_terminal (output_cdptr,
          password, code);
```

where:

1. output_cdptr (Input)
   is a pointer to the output CD structure in the COBOL program.

2. password (Input)
   is the CMCS system password.

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

Notes

----Notes for enable_output_terminal

Entry: cmcs_station_ctl_$attach

This entrypoint is used to attach a particular station by name. It is needed by processes that wish to initialize their environment for CMCS terminal operations.

Usage

```
     dcl cmcs_station_ctl_$attach entry (char (12), fixed bin,
          fixed bin (35));

     call cmcs_station_ctl_$attach (station_name, entry_index,
          code);
```

where:

1. station_name (Input)
   is the name of the desired station.

2. entry_index (Output)
   is the index of the given station in the station_ctl structure.

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

Notes

----Notes for attach

Entry: cmcs_station_ctl_$detach

This entrypoint uses an index into station_ctl to detach a particular station.

Usage

```
     dcl cmcs_station_ctl_$detach entry (fixed bin, fixed bin
          (35));

     call cmcs_station_ctl_$detach (entry_index, code);
```

where:

1. entry_index (Input)
   is the index of the given station in the station_ctl structure.

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

Notes

----Notes for detach

Entry: cmcs_station_ctl_$detach_name

This entrypoint uses the station_name to detach a particular station.

Usage

```
     dcl cmcs_station_ctl_$detach_name entry (char (12), fixed
          bin (35));

     call cmcs_station_ctl_$detach_name (station_name, code);
```

where:

1. station_name (Input)
   is the name of the desired station.

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

Notes

----Notes for detach_name

Entry: cmcs_station_ctl_$validate

This entrypoint is called by CMCS procedures that must check the validity of a given station_name. The only set of station names that can be assumed valid is contained in station_ctl. Other databases containing station names must have verified those names against the names contained in station_ctl.

Usage

```
     dcl cmcs_station_ctl_$validate entry (char (12), fixed bin
          (35));

     call cmcs_station_ctl_$validate (station_name, code);
```

where:

1. station_name (Input)
   is the name of the desired station.

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

Notes

----Notes for validate

Entry: cmcs_station_ctl_$input_disabled

This function entrypoint returns a flag to indicate whether or not the given station is disabled for input.

Usage

```
     dcl cmcs_station_ctl_$input_disabled entry (fixed bin, bit
          (1), fixed bin (35));

     call cmcs_station_ctl_$input_disabled (entry_index, flag,
          code);
```

where:

1. entry_index (Input)
   is the index of the given station in the station_ctl structure.

2. flag (Output)
   indicates the disable status.

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

Notes

----Notes for input_disabled

Entry: cmcs_station_ctl_$output_disabled

This function entrypoint returns a flag to indicate whether or not the given station is disabled for output.

Usage

```
dcl cmcs_station_ctl_$output_disabled entry (fixed bin, bit
     (1), fixed bin (35));

call cmcs_station_ctl_$output_disabled (entry_index, flag,
     code);
```

where:

1. entry_index (Input)
   is the index of the given station in the station_ctl structure.

2. flag (Output)
   indicates the disable status.

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

Notes

----Notes for output_disabled

Name: cmcs_fillin_hdr_

This COBOL MCS subroutine is used to set most of the header information in CMCS control segments and queues.

Usage

```
     dcl cmcs_fillin_hdr_ entry (ptr, fixed bin, fixed bin, fixed
          bin, fixed bin (35));

     call cmcs_fillin_hdr_ (hdr_ptr, version, hdr_len, entry_len,
          code);
```

where:

1. hdr_ptr (Input)
   points to the header of a newly created control segment or queue.

2. version (Input)
   is the version of the given file.

3. hdr_len (Input)
   defines the length of special header data that is unique to the given file.

4. entry_len (Input)
   defines the length of the individual entries in the file.

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

Notes

By convention, all CMCS control segments and message queues are created with a standard header. Additional header information must always follow the standard header. The header is declared in cmcs_control_hdr.incl.pl1.

The hdr_len value is added to the length of the common header data and the sum is subtracted from the maximum possible length of a segment. The result is divided by the length of the individual entry to give the maximum number of entries the segment can contain.

DEVELOPMENT_TASKS

MAJOR_DEVELOPMENT_ITEMS

These items are necessary for a complete implementation. Although several of these items are deferred until after MR6.0, nothing in this design must be allowed to preclude their development in the future.

* 1. Queue Processing

2. Terminal Management

3. Backup, Recovery

4. Accounting

* 5. Testing, Q/A

* 6. Documentation

7. Security

* 8. Assumptions, Philosophy

* 9. Error Processing

* 10. Definitions

* 11. Queue/Hierarchy Creation

* 12. Data Bases

* 13. Subroutine Call Interfaces

14. Metering

* 15. Command Interfaces

* Required for MR6.0, COBOL MCS

CHANGES_FOR_PHASE_2

The following changes and extensions to the Phase 1 implementation are recommended:

o Restructure cmcs_tree_ctl.control. The initial implementation wastes considerable space, for multiple-level hierarchies, in that it reserves space for every level that is needed only at the lowest level.

o Allow the interactive user who wishes to send a message to several destinations, to use a segment to list the destinations.

o Add controls to distinguish between queues used by interactive users as terminals/destinations, and queues used only by the COBOL application programs.

o Integrate the control segments for more efficiency.

o Allow one process to attach more than one station, and allow multiple processes to share a single station.

o Investigate the use of destination lists, to be associated with long messages being sent to several destinations. This facility, if needed, would reduce the amount of system storage used to store output messages until they are processed.

o Move the queues and control segments to ring 3, for security (and integrity).

o Provide a means for the interactive user of cobol_mcs to do simple editing of message data that is already entered into the system, but prior to being sent to the given message queues.

INCLUDE FILES

The include files to support COBOL MCS are given below. They are not in final form. Listings of the include files can be reviewed at CISL (Betsy Kerr), BCO (Otto Newman), and Phoenix (Bob May).

cmcs_control_hdr.incl.pl1

cmcs_message_hdr.incl.pl1

cmcs_message_key.incl.pl1

cmcs_message_segment.incl.pl1

cmcs_queue_ctl.incl.pl1

cmcs_slew_ctl.incl.pl1

cmcs_station_ctl.incl.pl1

cmcs_terminal_ctl.incl.pl1

cmcs_tree_ctl.incl.pl1

cmcs_user_ctl.incl.pl1

cmcs_wait_ctl.incl.pl1