Multics Technical Bulletin

TO: Distribution

FROM: R.W.Franklin

SUBJECT: COBOL - MCS

DATE: July 16, 1976

This bulletin describes the preliminary design and implementation of the Message Control System (MCS) for Multics COBOL. It stresses the preliminary design aimed at the terminal user and ends with a proposed interface to the COBOL compiler generated code.

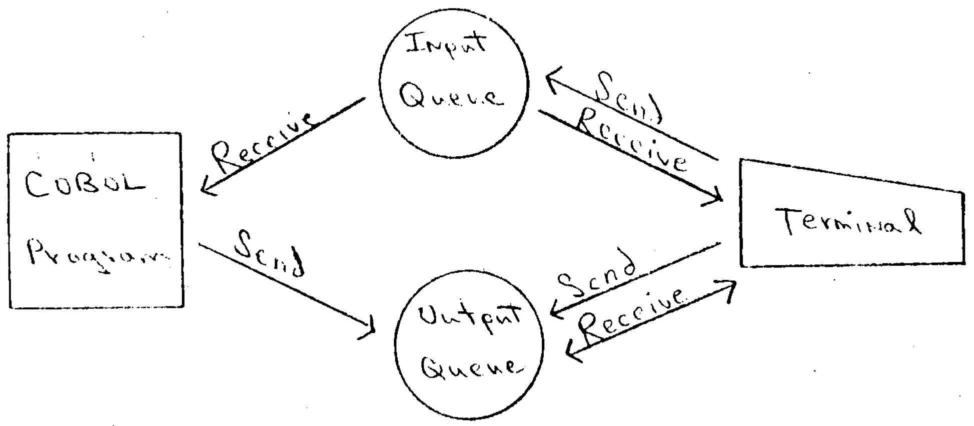
TABLE_OF_CONTENTS

1.0 INTRODUCTION

This MTB describes the COBOL Message Control System (COBOL-MCS) for Multics. For further information please refer to the American National Standards Institute COBOL specifications, ANSI X3.23-1974. COBOL-MCS is defined in the ANSI COBOL specifications, pages XIII-1 to XIII-23 and pages XIV-42 to XIV-48.

1.1 OVERVIEW

COBOL-MCS provides a COBOL program with the ability to access, process, and create messages or portions thereof, and with the ability to communicate with local and remote communication devices. COBOL-MCS is based on message queues. Queues are defined to be holding areas for messages (or portions of messages), and are the "middlemen" between a terminal user and a COBOL object program.



As shown above, the COBOL program sends messages to an output queue and receives messages from an input queue. These constraints are defined in the ANSI COBOL specification. Also shown is a terminal user who can send or receive messages to or from an input or output queue. ANSI COBOL leaves definition of the terminal user interface to the implementor. Our philosophy is to provide a superset of the COBOL-MCS facilities to the terminal user.

The COBOL programmer has five verbs with which to manipulate queues. These are:

ACCEPT	(returns count of messages available)
DISABLE	(put queue on HOLD)
ENABLF	(take queue off HOLD)
RECEIVE	(returns message from queue)
SEND	(put message onto queue)

The same functions are available to the terminal user.

There is also the capability to invoke a COBOL program when a queue becomes non-empty. A utility program is also provided to create a queue hierarchy using a queue description language.

1.2 DESIGN GOALS

A Multics COBOL-MCS design goal has been to make the COBOL user's program interface and the terminal user's interface similar. However, the terminal user's interface can and should be much richer functionally. In fact, the COBOL program user's interface is a proper subset of the terminal user's interface. By adhering strictly to this design goal, the terminal user's interface can be designed, implemented, and checked out prior to any development effort on the COBOL user's side. This provides ease of design, implementation, and maintenance.

Whatever is both transferable and desirable from the GCOS implementation of COBOL-MCS will be used. This includes documentation, design, and interpretation. To this end the Queue Description Language and the Queue Generation Utility, developed by the GCOS implementors, will be adopted as far as is practical.

Contrary to the implementation in GCOS, the Multics Communication System (MCS) is unaffected by the implementation of COBOL-MCS.

The ANSI COBOL specifications do not limit a system to one queue hierarchy in the permanent file system. A desirable design goal is to allow multiple hierarchies to be defined in the same system. Therefore, the Queue Generation Utility may be used to create a queue hierarchy under any directory. The standard Multics file access protection mechanisms are used. A user can specify the set of queues he wishes to manipulate.

The concept of one COBOL-MCS executive controlling all queues and lines in the system does not map readily into the Multics system. A better method appears to be the use of commands that execute within a normal Multics user process.

1.3 DESIGN/IMPLEMENTATION STRATEGY

Queue Hierarchy

A design strategy is to permit multiple queue hierarchies while still retaining the structure defined by the GCOS single queue hierarchy implementation. Placing both the search segment and the queues under the same directory makes cleanup procedures simple -- a single "delete-directory" command eliminates the entire queue hierarchy. Also free access on the queues can now be given to all desired users.

"write" access to the search segment (qh_table) is not required after the queue hierarchy has been defined and is therefore removed after its creation; i.e., the search segment, "qh_table", is given R permission for its creator, with ACL's otherwise preserved analogous to the handling of object segments by language translators. The segment "q_description" must exist in order for the Queue Generation Utility to run and generate the queue hierarchy. After the Queue Generation Utility has run, the queue hierarchy contains:

```
>...>message_queues>qh_table
>...>message_queues>INP_0000
>...>message_queues>INP_0001
                                .
                                .
                                .
>...>message_queues>INP_000N
>...>message_queues>OUT_0000
                                .
                                .
                                .
>...>message_queues>OUT_000M
```

Note that segment "qh_table" is created under the directory "message_queues" as is a segment for each queue defined in "q_description" (see Accept command description in section 3.3.2 for note on naming of physical queues).

Automatic_Invocation
Automatic invocation of a program when a queue goes non-empty is provided by the following mechanism. An optional statement has been defined in the Queue Description Language. This is the COMMAND LINE IS " ..." statement where the double quotes enclose any desired Multics command line. This command line is specified in the "q_description" segment at the level at which it is desired that it be executed. I.e., if it is defined at a level which has multiple queues then any of those queues going non-empty will cause that command line to be executed. E.g. assume that a user wishes the absin file abc to be executed when queue xyz goes non-empty. By placing the statement COMMAND LINE IS "ear abc" in the Queue Description Language at or above the level of queue xyz the command:

```
ear abc -arg xyz
```

will be executed when queue xyz goes non-empty. Note that the -arg queue_name is appended to the command line when it is executed. This permits passing of the queue name to the COBOL program from the absentee job and enables the invoked program to determine which queue going non-empty caused it to be invoked. ANSI-COBOL prohibits the passing of arguments via the USING option to a called COBOL program with the FOR [INITIAL] INPUT phrase. However by having an in-line calling sequence generated by the COBOL compiler to request parameters we can overcome this restriction. The absentee job is written to accept optional arguments (by use of &1 etc.) which are then available to the called program.

Contents of Queues

The queues are maintained in a first-in/first-out (FIFO) fashion. There are two chains of messages in each queue. One is for unprocessed messages and the other is for processed messages. When a message goes from the unprocessed to the processed chain, a "date-time-removed" field is placed with it. At any time, commands can be issued that return histories of the queue and its contents. A mode can be set on a per queue basis that causes the processed message chain to be removed thus freeing the space it previously occupied.

Commands

Commands are used to manipulate the queues on behalf of a terminal user, while subroutine calls to the commands are used to manipulate the queues on behalf of a COBOL program user. These commands execute as part of the user's process in his current ring. The set of functions available to the terminal user is limited only by our imagination and resources.

Calling sequences from the COBOL program can be optimized for efficiency by making use of returned information. For example, a READ-SYMBOLIC-SEGMENT passes the symbolic queue name to the command, which must then resolve it to a physical queue name. This physical queue name can be returned to the caller, and subsequent requests for additional segments can use the READ-QUEUE command.

2.0 QUEUES

2.1 QUEUE CONCEPTS

Queues are fundamental to the operation of the COBOL communications facility. They provide the mechanism by which messages flow between a COBOL program executing in the central computer system and a set of communication devices. From the COBOL viewpoint, a queue contains one or more messages for or to one or more communication devices. As such, the queues serve as communications data buffers for the COBOL program or the terminal user.

The COBOL program, using COBOL-MCS, is independent of the remote devices in the communications network. The COBOL program uses symbolic names with up to four levels of qualification to refer to message sources and destinations. These symbolic names, which can be up to 12 alphanumeric characters long, represent the physical queues. A Queue Generation Utility creates the Queue List Structure which is used in translating the COBOL symbolic queue names to physical queue names.

Input queues are defined in COBOL-MCS in a hierarchical tree structure containing up to four levels. Output queues are described in only one level.

A programmer can reference a specific input queue at the lowest level in a queue hierarchy using the higher levels of the hierarchy as qualifiers for that queue name. By referencing a higher level in the hierarchy, the programmer references all queues subordinate to the level referenced. For example, a reference could be made to a queue name at the 04 level (the lowest possible level in a queue hierarchy) and queue names at levels 01-03 would serve as qualifiers. A reference to the 01 level of a queue hierarchy references all queues defined in the hierarchical structure.

The Queue Generation Utility is an independent program, not a part of either the COBOL compiler or the Message Control System. It performs the following functions for COBOL-MCS:

- Processes a language (the queue description language) that defines the queue hierarchy.
- Generates a queue list structure and places it in a file (qh_table) in the directory hierarchy.

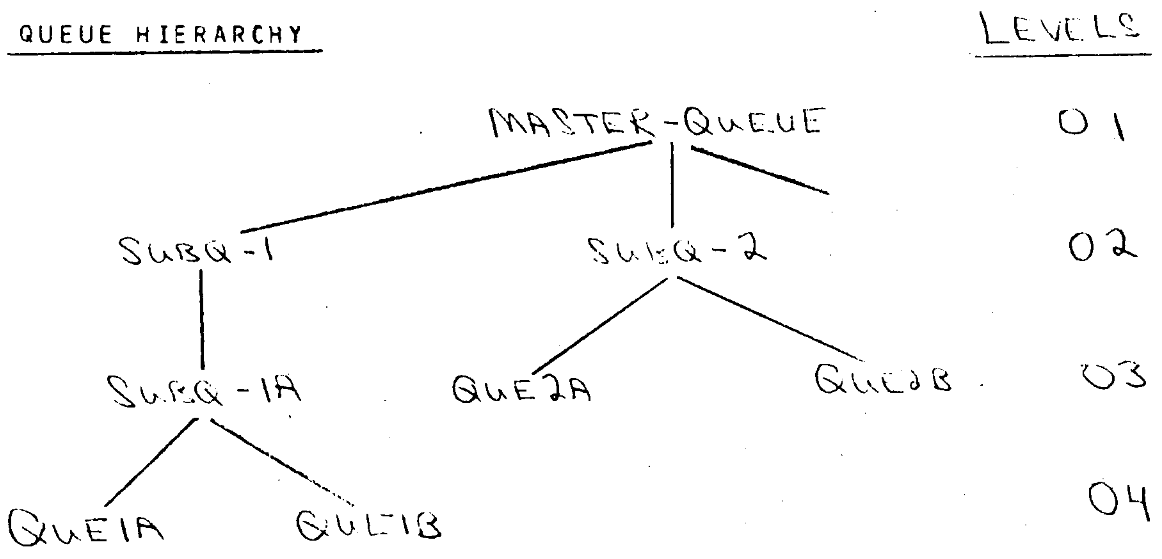
A COBOL-like language is used to describe the queue structure for the Queue Generation Utility. This language permits the queues to be defined using COBOL level numbers (01-04). The relationship between the queue language description of a queue structure and the queue structure itself are illustrated in Figure 2-1. Passwords can be specified at any or all levels of a queue structure and are used for the enable/disable functions.

QUEUE STRUCTURE IN QUEUE DESCRIPTION LANGUAGE

```
01  INPUT-QUEUE (MASTER-QUEUE) PASSWORD IS "MASTER"
    02  SUB-QUEUE-1 (SUBQ-1) PASSWORD IS "SUB1"
        03  SUB-QUEUE-2 (SUBQ-1A) PASSWORD IS "SUB1A"
            04  SUB-QUEUE-3 (QUE1A) PASSWORD IS "QUE1"
            04  SUB-QUEUE-3 (QUE1B) PASSWORD IS "QU1A"
    02  SUB-QUEUE-1 (SUBQ-2) PASSWORD IS "SUB2"
        03  SUB-QUEUE-2 (QUE2A) PASSWORD IS "QUE2"
        03  SUB-QUEUE-2 (QUE2B) PASSWORD IS "QUE3"

01  OUTPUT-QUEUE (OQ3) PASSWORD IS "STA1"
```

QUEUE HIERARCHY



The output structure has only the 01 level.

QUEUE LANGUAGE DESCRIPTION - QUEUE STRUCTURE RELATIONSHIP

FIGURE 2-1

The Queue Generation Utility uses this queue description language as input and generates a queue list structure that describes the queue hierarchy. One queue list structure is generated for each execution of the utility. This structure is built in the specified hierarchy and is placed into segment "message_queues>qh_table". This structure refers to the actual physical queues which are also created as segments in the same directory as "qh_table".

Each queue description is created as a hierarchical list structure which contains all information pertaining to that queue. As the queue language syntax is processed by the Queue Generation Utility, an entry in segment qh_table is built for each level defined in the hierarchy.

There is also a means for interpreting the hierarcny after it has been placed on the permanent file system and for listing its contents.

In COBOL-MCS operation, a verb in a COBOL program references a source/destination by a symbolic name. Using the program's CD area as an interface, this symbolic queue definition is passed to an interpreter where the queue list structure is used to translate the symbolic name to the actual physical queue.

2.2 QUEUE OPERATIONS

The queues can be initialized for either of two modes of operation. An input queue may be initialized to automatically cause COBOL-MCS to invoke a COBOL object program when the queue becomes non-empty. A queue that does not automatically invoke a program holds the message until it is requested by a program or a terminal.

On output, the COBOL program sends data to one or more queues for delivery to specific destinations. If the destination device is not available, the messages are held until the attachment can be made.

In the COBOL procedures, symbolic names can be used to reference various levels of the queue structure. These symbolic names must be moved into the CD-entry before the CD is referenced by the the COBOL-MCS verbs in the program. When the COBOL program "calls" the appropriate subroutine it includes a pointer to the pertinent CD entry. The queue list structure is then used to translate the symbolic name to the actual physical queue name. Thus, by proper construction of the CD-entry in the COBOL program, a programmer can reference one specific queue or, by referencing a level in the queue structure with subordinate levels, can reference all of the queues subordinate to the referenced level.

The symbolic queue/subqueue names used in the COBOL program must be the symbolic queue names used in the generation of the queue list structure.

2.3 QUEUE DESCRIPTION LANGUAGE

The Queue Description Language is a dialect of COBOL that allows a user to define a hierarchical queue structure. This structure is used by the COBOL-MCS commands to translate the symbolic queue/subqueue names used in COBOL to the physical queue names. The queue hierarchy descriptions are a combination of the syntax used to describe the CD-name and the level number concept from the COBOL Data Division.

In using the Queue Description Language, the programmer must use the syntax as described in the following pages (see figure 2-2). Queues are polled in the order that they appear in the queue structure. When a reference is made using all the levels of qualification, only that queue is polled. For a reference to a level in the structure with queues defined at subordinate levels, the queues are polled in the order that they appear in the hierarchical structure, i.e. left to right (see figure 2.3).

SAMPLE QUEUE/SUBQUEUE STRUCTURE

```
01  INPUT-QUEUE (MASTER-QUE) PASSWORD IS "MASTER"
    02  SUB-QUEUE-1   (SUBQ-1)
        03  SUB-QUEUE-2 (QUE1A) PASSWORD IS "QUE"
          COMMAND LINE IS "ear  subq-1"
    02  SUB-QUEUE-1 (SUBQ-2)
        03  SUB-QUEUE-2 (QUE2) PASSWORD IS "QUE2"
        03  SUB-QUEUE-2 (QUE3) PASSWORD IS "QUE3"
        03  SUB-QUEUE-2 (QUE4) PASSWORD IS "QUE4"
    02  SUB-QUEUE-1 (SUBQ-3)
        03  SUB-QUEUE-2 (QUE5) PASSWORD IS "QUE5"
        03  SUB-QUEUE-2 (QUE6) PASSWORD IS "QUE6"
        03  SUB-QUEUE-2 (QUE7) PASSWORD IS "QUE7"
        03  SUB-QUEUE-2 (QUE8) PASSWORD IS "QUE8"

01  OUTPUT-QUEUE (OQ3) PASSWORD IS "STA1"

01  OUTPUT-QUEUE (OQ4) PASSWORD IS "STA2"

99
```

NOTE: The Queue Generation Utility expects a level of 99 at the end of the last entry in the queue structure. The level of 99 terminates the scan of the structure.

FIGURE 2-2. Sample Queue/Subqueue Structure

2.4 INPUT QUEUE DEFINITION

The general format of an input queue structure is as follows:

```
          01  INPUT-QUEUE    (QUEUE-NAME)    [PASSWORD    IS
              literal-1]
              02  SUB-QUEUE-1  (SUB-QUE-NAME)  [PASSWORD  IS
                  literal-2]
                  03  SUB-QUEUE-2 (SUB-QUE-NAME)   [PASSWORD
                      IS literal-3]
                      04  SUB-QUEUE-3          (SUB-QUE-NAME)
                          [PASSWORD IS literal-4]
```

RULES:

1. The INPUT-QUEUE clause is required.
2. The minimum queue structure that can be defined is a single level queue. This is done by defining a 01 level and no subordinate levels. If a single level queue is specified, then the physical queue may be referenced by use of the one symbolic name.
3. The queue (QUEUE-NAME) and subqueue (SUB-QUE-NAME) names can be up to 12 alphanumeric characters in length. A name less than 12 characters in length will be blank-filled on the right.
4. Optional passwords can be specified (PASSWORD IS) at any level in the queue hierarchy. These passwords are optional in the general exchange of messages, however, a password is required for any disabling or enabling of a queue or terminal. Disabling/enabling may be performed at the individual symbolic queue level or, by referencing higher in the queue hierarchy, at a group level (all subordinate symbolic queues referenced). At the group level, only one password can be specified for the whole group. A password must be specified at the level that is to be referenced in the program. A password is composed of up to 10 alphanumeric characters, and if less than 10 characters, it will be blank-filled on the right.
5. When a subqueue is specified as the lowest level in the hierarchy, all higher levels in the hierarchy must be defined. The subqueue definition at the lowest level implies the existence of a physical queue.
6. Optional command lines can be specified (COMMAND LINE IS "...") at any level in the input queue hierarchy. However, once one is specified at a level, it must not be redefined at a lower level (i.e. a level which is encompassed by the higher level definition). When a queue which is defined under this optional statement goes non-empty, the Multics command line contained in

the double quotes will be executed with an -arg queue_name appended to it. This gives the user the capability to automatically invoke a COBOL program when a queue goes non-empty and to pass the pertinent queue name to the invoked program as an argument.

7. A level of 99 must be at the end of the last entry in the queue structure. The 99 terminates the scan of the Queue Description Language.

2.5 OUTPUT QUEUE DEFINITION

The output queue structure has only one level; the 01 level. Thus no levels subordinate to the 01 level can be defined for an output queue. The format of an output queue structure is as follows.

```
01   OUTPUT-QUEUE (QUEUE-NAME) [PASSWORD IS literal-6]
```

RULES:

1. The OUTPUT-QUEUE statement is required.

2. The output queue name (QUEUE-NAME) can contain up to 12 alphanumeric characters. If the name contains fewer than 12 characters, it is space-filled to the right.

3. No levels can be specified subordinate to the 01 level for output queues.

4. A password must be specified (PASSWORD IS) for an output queue if the queue is to be referenced via the Enable/Disable verbs.

SAMPLE QUEUE STRUCTURE

To illustrate the use of the hierarchical queue structure in referencing one or more queues, consider the structure in Figure 2-3. Assume in this example that the queue list structure is organized to describe this queue hierarchy reading from left to right.

If a RECEIVE statement specifies MASTER-QUEUE, this entire queue structure will be traversed to access each queue in the structure until a queue with a message indicator is found. That detailed reference is returned to the program and used to access the queue containing that message. The queue list structure is traversed for each reference. However a second reference to the same queue name causes that queue name to be accessed directly. When using the RECEIVE_SEGMENT command, if a segment is received in the first request to a queue and no change is made in the queue structure reference (i.e., the CD entry), the remaining segments of the message are sent automatically in subsequent RECEIVE_SEGMENT calls. The elements of the queue hierarchy are traversed in the order that they appear in the hierarchy list structure.

Similarly, a RECEIVE statement specifying SUBQ-3 would cause queues QUE5 through QUE8 to be polled for messages.

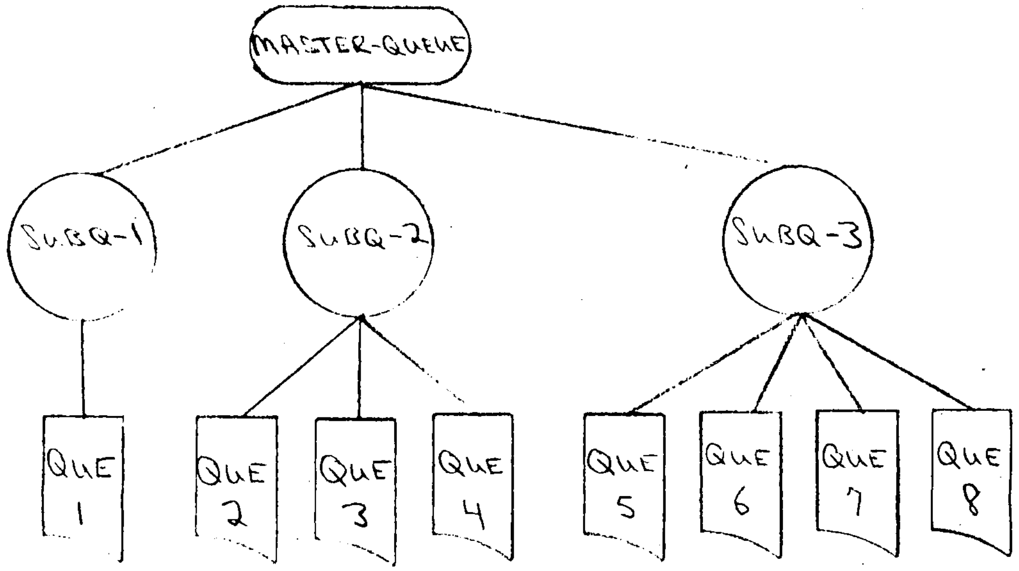


FIGURE 2-3. Hierarchical Queue Structure

A SEND verb (output) references a single level queue structure (the 01 level), thus it can reference one or more queues directly.

2.6 QUEUE_GENERATION

To be used by the COBOL-MCS commands, the queue structure described in the Queue Description Language must be available in the system.

The Queue Generation Utility is a program that operates independently of COBOL. It processes the Queue Description Language (which is in a segment and which defines the queue hierarchy), and creates the search structure in segment "qh_table". This is placed in a directory ("message_queues") in the current hierarchy (see writeup of the Queue Generation Utility program).

During the execution of a COBOL object program, the program sends a request containing a symbolic queue name to the COBOL-MCS command handlers. A search of the queue structure is initiated in a top-down order seeking the detailed queue definition. If a definition is found, it is used by the program in accessing the queue. If the symbolic name is not defined in the queue structure, an error status code is returned to the requesting program.

Based on the queue information supplied in the Queue Description Language, the Queue Generation Utility builds a segment containing threaded lists in a structure suitable for traversing the hierarchies. This segment, "qh_table", is used to resolve all symbolic representations of the actual queues.

2.7 Print_Hierarchy_Command

A command is available to the terminal user to print a formatted version of the queue structure that exists in segment "qh_table". It matches exactly the structure defined by the Queue Description Language. In fact the only difference between the Queue Description Language and the contents of segment "qh_table" is that segment "qh_table" is in a threaded form suitable for fast searching.

3.0 Multics Programs, Segments, and Structures

This section describes the Multics programs, segments, and structures that constitute Multics COBOL-MCS.

3.1 Segment "q_description"

This is the input segment to the Queue Generation Utility. This segment contains the desired input-output queue hierarchy in a COBOL-like free-form structure; i.e. it utilizes the level concept present in the COBOL data division. The overall description of the language matches, almost exactly, that which has been used in the GCOS COBOL-MCS queue generation system. The reader is directed to the GCOS-MCS SITE Manual (DC99) or Design Memo number 878 (revision 1) entitled "COBOL 74 Queue Generation System" written by C.C.Bain dated February 5, 1975. Either gives a detailed description of the format of the language. The exceptions to this language description are:

a. The last line of the segment must contain a level number of 99. The Queue Generation Utility program searches for this to terminate normally.

b. The option [PROCESSOR IS literal-5] is not accepted.

c. The phrase [STATION NAME IS] is not accepted.

d. The new optional phrase [COMMAND LINE IS " ... "] is accepted.

3.2 The Queue Generation Utility

USAGE qgu [-pn arg1 -dr arg2]
where
1. arg1 designates the relative or absolute pathname where segment "q_description" is to be found. If -pn is missing the default is the current working directory.
2. arg2 specifies the relative or absolute pathname where directory "message_queues" is to be placed. if -dr is missing the default is the current working directory.

The Queue Generation Utility program:

a. guarantees that segment "q_description" is present in the directory hierarchy specified in the command which invokes the Queue Generation Utility.

b. truncates all segments under directory "message_queues" so as to preserve any ACL's which may already be present. This also leaves all initial ACL's on directory "message_queues" still applicable.

c. creates the directory "message_queues" in the specified directory, if it does not already exist.

d. creates the segment "qh_table" (if it does not already exist) in directory "message_queues". i.e.,

```
>...>message_queues>qh_table
```

e. processes the rest of segment "q_description" building up segment "qh_table" with a linked structure based on the free-form structure given in "q_description".

f. whenever a lowest level is encountered in "q_description" file, a segment is created (if it does not already exist) in the directory "message_queues" to contain the messages for that queue.

```
>...>message_queues>INP_....      OR
>...>message_queues>OUT_....
```

In order to use the queue generation utility you must have a permission level high enough to delete/create directories/segments off of the specified directory. You must also have read permission on segment "q_description".

The results of a successfully completed queue generation utility run are...

1. A directory "message_queues" which is created off of the specified directory.
2. A segment "qh_table" which is created in the "message_queues" directory.
3. Segments (queues) created for each lowest level in the "q_description" file. These segments are created in the directory "message_queues".

3.3 Program "command"

3.3.1 GENERAL

This program contains the terminal users interfaces to the COBOL-MCS functions. i.e., it contains the entry points that the terminal user calls to accomplish the ANSI defined COBOL-MCS functions of...

1. ACCEPT
2. DISABLE
3. ENABLE
4. RECEIVE
5. SEND

The above are the ANSI defined COBOL-MCS functions that are available to a COBOL program. COBOL-MCS provides the capabilities itemized below to the terminal user. Many of the functions are not applicable to the COBOL program but are desirable from a terminal user's point of view.

3.3.2 Command_Processors

1. NAME accept

   Executing this command yields a printout of the number of unprocessed and processed messages currently existant in queue arg1.

   USAGE accept arg1

   where:
      arg1 must be present and is a queue name; i.e., a name which has been uniquely generated by the Queue Generation Utility. NOTE: The method employed to name the queue is as follows. Input queues are named with 8 characters in the form INP_XXXX where XXXX is a 4 digit number starting with 0000. e.g., the 3rd input queue would be named INP_C002. Output queues are similarly named in the form of OUT_XXXX.

2. NAME symbolic_accept

   Executing this command yields a printout of the number of processed and unprocessed messages for ALL queues described by the hierarchy given in the argument list. This may include one or many actual queues.

   USAGE symbolic_accept arg2 [arg3 arg4 arg5]

   where:
      arg2 and the optional arguments describe the symbolic hierarchy desired starting with the highest level and skipping no levels.

3. NAME info

   Executing this command yields a printout of all unprocessed messages currently residing in queue arg1.

   USAGE info arg1

   where:
      arg1 is as described in command 1.

4. NAME symbolic_info

   Executing this command yields a printout of all unprocessed messages for all queues described by the hierarchy given by the argument list. This may include one or many actual queues.

<u>USAGE</u> symbolic_info arg2 [arg3 arg4 arg5]

where:
the arguments are as described in command 2.

5. <u>NAME</u> history

Executing this command yields a printout of all processed messages residing in queue arg1. This includes the date/time created and the date/time processed for each message.

<u>USAGE</u> history arg1

where:
arg1 is as described in command 1.

6. <u>NAME</u> receive

Executing this command yields a printout and a release of the next available message from queue arg1. The message now becomes a processed message and its date/time processed field is set. If there are no unprocessed messages in queue arg1 then a message stating this is given.

NOTE: If the queue contains an unprocessed message but happens to be locked (see section 5) then,

a. If it has been locked by this user (see writeup on commands 9 and 10) then this queue is used.
b. If it has been locked by another user then a temporary lock-loop (see section 5) is entered.

<u>USAGE</u> receive arg1

where:
arg1 is as described in command 1.

7. <u>NAME</u> symbolic_receive

Executing this command yields a printout and a release of the next available message from the given hierarchy described by the argument list. The queues will be interrogated in the hierarchical order until either an unprocessed message is found and released or until all queues defined by the hierarchy of the argument list have been interrogated. A printout specifying that no unprocessed message could be found in a particular queue is given prior to interrogating the next queue in line. As in the receive command (command 6), the date/time processed is set when a message is released.

NOTE: If a queue is found which contains an unprocessed message but is locked, then the note on command 6 above is applicable.

USAGE symbolic_receive arg2 [arg3 arg4 arg5]

where:
the arguments are as described in command 2.

8. NAME send

Executing this command causes a request for input to appear on the users console. Multiple lines of input (multiple segments) are accepted. Once the entire message has been typed in, the data input mode is terminated by typing in a line where the first character is a ".". This line itself is not considered part of the message. The data is then placed onto queue arg1 and its date/time created is set.

USAGE send arg1

where:
arg1 is as described in command 1.

9. NAME receive_segment

Executing this command yields a printout and a release (possibly temporary) of the next available segment from the next available message in queue arg1. A segment is a line and a message may consist of multiple segments. This type of receive command releases a maximum of one segment from the next available message of queue arg1. The queue arg1 remains locked upon return to the user if there were any remaining segments in the message; i.e., no other user can obtain access to queue arg1 once this command has been executed until the entire message in queue arg1 has been released. The date-processed field is not set until the release of the last segment of the message. At this time an End-of Message indication is also returned to the user.

NOTE: Some unusual results can occur when this command is interleaved with other receive (message) commands. It is allowable to do so and the results are consistent but may not be what was desired. E.g., assume that a message consists of 5 segments and the first 3 segments have been returned, one at a time, via use of the "receive-segment" command. The queue is locked to all other users at this time. Assume that the user now executes a "receive" (message) command. We note that the queue is locked but since it is locked by this user we allow the command. Now the entire message (all 5 segments) is released and returned to the user and the queue is unlocked. i.e. the first 3 segments have now been released twice.

The only time that a queue can remain locked upon exit from a command is when a command of type 9 or 10 has been executed and at least one unprocessed segment still remains in the message. If a user program terminates and any queue remains locked by that user then that queue is unlocked and the beginning of the current message is reset to point to its first segment. Note also that the only commands which will allow entry to a locked queue (and this only providing that this user initially locked the queue) are the four receive commands.

<u>USAGE</u> receive_segment arg1

where:
   arg1 is as described in command 1.

10. <u>NAME</u> symbolic_receive_segment

Executing this command is the same as executing command 9 above except that the queue to be used is decided by a hierarchical search using the argument list as the desired hierarchy. A search is made to find the first non-empty queue in the given hierarchy. This includes already locked queues. If a non-empty queue is found and it has already been locked by this user then it is used. If it has been already locked by someone else then a temporary lock-loop is entered (see section 5).

<u>USAGE</u> symbolic_receive_segment arg1 [arg3 arg4 arg5]

where:
   the arguments are as described in command 2.

11. <u>NAME</u> print_hierarchy

Executing this command yields a printout of the hierarchy present in segment "qh_table". This hierarchy has been created by the Queue Generation Utility program using segment "q_description" as input.

<u>USAGE</u> print_hierarchy -pathname-

where:
   pathname is a relative or absolute pathname leading to segment "qh_table".

12. <u>NAME</u> disable_q

This command puts a physical queue in the "HOLD" state. If the queue is already in the HOLD state then a message stating this is given and the command is considered finished. If not then a password is requested and validated. If the password is given incorrectly then the command is aborted. If the password is correctly given then the queue is put onto HOLD meaning it cannot accept or release any messages until it is ENABLED via the enable command.

USAGE disable_q arg1

where:
  arg1 is as described in command 1.

13. NAME enable_q

This command takes a physical queue off of HOLD. If the queue is already off of HOLD then a message stating this is given and the command is considered complete. If not then a password is requested and validated. If the password is given incorrectly then the command is aborted. If the password given correctly then the queue is taken off of HOLD; i.e. it is ENABLED.

USAGE enable_q arg1

where:
  arg1 is as described in command 1.

3.4 Directory "message_queues"

When the Queue Generation Utility program runs, it uses segment "q_description" as input and creates a hierarchy. It places this hierarchy in a separate directory (message_queues) created off of the specified hierarchy. It does this for ease of management of the queues and the search segment. In order to further protect the hierarchy, only READ permission is placed on the search segment (qh_table) while REW permissions are placed onto the queue segments.

3.5 Structure Definitions

The following structures are currently used either in the creation of segment "qh_table" (the search segment) or in the creation of the physical message queue segments. Both of these are under the directory "message_queues". The structures are given as they currently exist but are very much subject to change.

```
/*   BEGIN INCLUDE FILE ... qhtbl.incl.pl1  */

dcl 1 qhtbl based (qhp) aligned,
    2 max_size fixed bin,                              /* maximum number of entries in q_hierarchy(qh
    2 num_entries fixed bin,                           /* number of active entries */
    2 filler (14) bit (36),                            /* 16 word header */
    2 entry (1000),                                    /* followed by x 32 word entries */
      3 filler1 (32) fixed bin;

/* Entries in qhtbl are 40 octal words long and contiguous */
dcl  qhe_size fixed bin init (32);

dcl 1 qhe based (qhep) unal,
                                                       /*  Word 0 */
    2 used bit (1) unal,
    2 filler2 bit (17) unal,
    2 type fixed bin (17) unal,                        /* level number or 5 = station */
                                                       /* words 1, 2, 3 */
    2 name char (12) aligned,
                                                       /* words 4, 5, 6 */
    2 password char (10) unal,
    2 flags,
      3 input bit (1) unal,
      3 filler22 bit (17) unal,
                                                       /* word 7 */
    2 preceding_entry fixed bin (17) unal,             /* entry number of last entry */
    2 parent fixed bin (17) unal,                      /* entry number of parent */
                                                       /* word 10 */
    2 preceding_station fixed bin (17) unal,           /* entry number of previous station */
    2 next_station fixed bin (17) unal,                /* entry number of next station */
                                                       /* word 11 */
    2 first_station fixed bin (17) unal,               /* entry number of 1st station */
    2 max_station fixed bin (17) unal,                 /* max station entry number to use */
                                                       /* word 12 */
    2 prev_entry_at_same_level fixed bin (17) unal,
    2 next_entry_at_same_level fixed bin (17) unal,    /* entry number of next same level */
                                                       /* words 13, 14, 15, 16  */
    2 filler3 (5) bit (36),                            /* for expansion */
                                                       /* words 20 to 37 octal */
    2 access_entry char (32);

/* END INCLUDE FILE ... qhtbl.incl.pl1   */
/*    BEGIN INCLUDE FILE ... mqtbl.incl.pl1    */

dcl  mqh_header_length fixed bin init (8);

dcl 1 mqh based (mqp) aligned,

    2 num_entries fixed bin (17) unal,
    2 num_entries_processed fixed bin (17) unal,

    2 lock bit (36) aligned,                                   /* when set locks the segment */

    2 first_message fixed bin (17) unal,                       /* rel. ptr to first message in segment */
    2 next_message fixed bin (17) unal,                        /* rel. ptr to where next message should go */

    2 flags,
      3 active bit (1) unal,                                   /* 1 = active now */
      3 disabled bit (1) unal,                                 /* 1 = disabled */
      3 filler3 bit (16) unal,                                 /* room for expansion */
    2 filler4 fixed bin (17) unal,                             /* room for more flags */

    2 filler02 (4) fixed bin,                                  /* header is 8 words in length */

    2 entry (1000),
      3 filler1 (16) fixed bin;

/* entries in queues are variable in length */

dcl  mqep ptr;
dcl  message_header_length fixed bin init (9);
dcl  message_header_length_char fixed bin init (36);

dcl 1 mhe based (mqep) aligned,

    2 num_chars fixed bin (17) unal,
    2 current_char fixed bin (17) unal,

    2 date_time_created char (16),                             /* date time message placed onto queue */

    2 date_time_removed char (16),                             /* date time message removed from queue */

    2 data char (32);

/*    END INCLUDE FILE ... mqtbl.incl.pl1    */
```

4.0 COBOL Object Program Interface

4.1 GENERAL

The COBOL Object Program interfaces to the command routines via calling sequences. In general, the entry point called specifies the particular function being performed. In all probability the COBOL object program calling sequences will be generated in-line to a run-time subroutine (called the Object Program Interface - OPI).

Each of the five COBOL-MCS verbs (ACCEPT, DISABLE, ENABLE, RECEIVE, SEND) refer to either an input or an output Communications Descriptor (CD). The ACCEPT and RECEIVE verbs always refer to an Input CD while the SEND verb always refers to an output CD. The DISABLE and ENABLE verbs refer to either an input or an output CD dependent upon whether the INPUT or OUTPUT phrase is specified.

The COBOL program itself uses the structures described in section 4.3 for the input and output CD's. The following information is based on the premise that there will exist a COBOL Object Program Interface run-time routine. This OPI run-time routine will communicate between the commands and the COBOL program itself. The COBOL compiler will translate the communication verbs into calling sequences to the OPI run-time routine which in turn will call the command. Note that this is not necessarily the way in which it will be implemented but is merely a template on which to base examples. The in-line calling sequences will contain pointers to the pertinent input or output CD structures. The OPI run-time routine will use calling sequences to the commands which contain enough information for the command to perform its function. The original CD structure contains only symbolic information, some of which is resolved down to actual queues by the command, but some of which must be resolved to pointer addresses prior to the transfer of control to the command. This resolution of symbolic names to pointers can be done either by the in-line code or by some run-time subroutine (OPI). The interpretation of returned error-codes and the decision as to what to do when they occur is probably best handled in a run-time subroutine. This decision (all in-line code or a run-time subroutine) may be left up to the COBOL compiler people.

A COBOL program refers to a queue hierarchy someplace in the permanent file system. Multiple queue hierarchies are allowed so a method is required to determine which hierarchy the program is to manipulate. ANSI specifications deny us the option of defining new syntax in the COBOL language (such as a "SET HIERARCHY" statement) so another method must be used. This is not bad as we would like to be able to execute a COBOL program and have it manipulate different hierarchies in different executions without requiring a re-compilation. One possibility is to have the COBOL calling sequences to COBOL-MCS pass the current working directory. A search is then made for directory "message_queues" which may exist off of the current working directory or may be linked to from within the current working directory. At any particular execution of a COBOL program the Job Control Language can set the working directory as desired.

4.2 SAMPLE_CALLING_SEQUENCES

The following calling sequence and structures are given as an example of what is needed by the commands. This is given as an example only and may not bear any resemblance to what is ultimately implemented by the compiler people.

NAME: RECEIVE

The RECEIVE entry point returns one message or portion of a message to the caller.

USAGE:

```
   declare RECEIVE entry (pointer,pointer,fixed bin(35));

   call RECEIVE(CD_ptr,OPI_ptr,code);
```

where:

1. CD_ptr is a pointer to the input_CD structure
2. OPI_ptr is a pointer to the OPI_input_CD structure
3. code is for the returned error code

4.3 STRUCTURES

```
        DCL 01 Input_CD based unaligned,
                02 sq char(12),
                02 ssq1 char(12),
                02 ssq2 char(12),
                02 ssq3 char(12),
                02 message_date pic "999999",
                02 message_time pic "99999999",
                02 symbolic_source char(12),
                02 text_length pic "9999",
                02 end_key char(1),
                02 status_key char(2),
                02 message_count pic "999999";
```

```
DCL 01 Output_CD based unaligned,
      02 destination_count pic "9999",
      02 text_length pic "9999",
      02 status_key char(2),
      02 destination_table (1),
         03 error_key char(1),
         03 symbolic_destination char(12);


DCL 01 OPI_input_CD based aligned,
      02 q_ptr ptr,
      02 date_ptr ptr,
      02 time_ptr ptr,
      02 source_ptr ptr,
      02 length_ptr ptr,
      02 end_key_ptr ptr,
      02 status_key_ptr ptr,
      02 count_ptr ptr;


DCL 01 OPI_output_CD based aligned,
      02 dest_count_ptr ptr,
      02 length_ptr ptr,
      02 status_key_ptr ptr,
      02 dest_table_ptr ptr;
```