11 FEB 1976

Burroughs Corporation INTER-OFFICE CORRESPONDENCE

TO:

CORPORATE UNIT Computer Systems Group	LOCATION Santa Barbara Plant	DEPT.
NAME Distribution		DATE 11 February 1976
FROM D. Stearns	DEPT. & LOCATION Advanced System Development Section	

SUBJECT: C.C.

QUEUE DESIGN IN 5.1

The document following is a major rewrite of "MCPII 5.1 Queue System and Interfaces", which originally came out as the code was being written. Changes were made during development and this documents the revised design. In particular, your attention is called to Section 2.0 "QUEUE DATA STRUCTURES", which more accurately describes buffer management.

[illegible]

D. Stearns
Advanced System Development Section

gp

Attachment

MCPII 5.1 QUEUE SYSTEM AND INTERFACES

CONTENTS

MCPII 5.1 QUEUE SYSTEM AND INTERFACES

1.0 INTRODUCTION

A message queue system has existed in MCPII since Release III.4, but maintenance, speed, and working-set considerations have caused a re-design for 5.1. This note describes the current plan for new queues. It is intended for members of the Programming Activity and TIO, not for users; the aim is to specify the interfaces between the queue system within the MCP and other system software -- NDL, compilers, etc. It does not cover remote files, the NDL/MCS interface, or complex WAIT.

The term "queue" refers to the actual data structure maintained by the MCP as a means of inter-process communication. Queues have various attributes: a 20-character name, user count, message count, etc. Most important, a queue may contain a list of messages (possibly empty). A queue user may add to the back or remove from the front of this list. The queue may be shared -- one or more processes may put messages in the list and one or more processes may remove messages. Only the MCP may access a queue directly. User programs must use other interfaces built upon queues, such as queue files or remote files.

Major changes have been in the following areas:

1. Maximum queue length may be explicitly set at queue create time (file open for user programs).

2. The "ON INCOMPLETE.IO" branch may be used to prevent a process from being blocked (a) during a READ on an empty queue or (b) during a WRITE on a full queue.

3. Message storage in S-memory is minimized by buffer pooling for all queues.

4. Long queues will be optionally maintained on disk. Disk I/O will be overlapped with user processing whenever possible, like disk files.

5. The EOF branch on read/write will be defined for queue files in such a way to permit label equation with, say, card files. New queue files will not report EOF on queue-empty.

6. FPB format will change. New fields are required for new queue parameters, like maximum queue length. Recompilation of existing programs using queue files will not be required however -- a new device type will be used to discriminate new and old queue file FPB's.

7. Queue files may no longer be blocked. Records written to a queue file are immediately put into the queue.

These changes are described below and in the appendices. COBOL and SDL queue file syntax is used for examples.

2.0 QUEUE DATA STRUCTURES

These structures are transparent to the user. They are the implementation of queues in 5.1, but they may change in future releases. For this reason, details should not be relied upon in any application.

2.1 DESIGN PHILOSOPHY

The design of the data structure (Figure 1) was strongly affected by the need to reduce the S-memory needs of queues. Reusable structures like message buffers and message descriptors are pooled for the use of the whole queue system. Empty message buffers and descriptors are not released to FORGETSPACE - the Q-driver retains them for later use. This has two good effects: 1) quicker allocation by avoiding GETSPACE, and 2) less disturbance of the code working set of the system. Since queue files and remote files are unblocked, their FIB's need not have buffers; this makes them the smallest of any FIB's (500-1000 bits).

2.2 QUEUE DESCRIPTORS

For a given queue, the queue name, maximum length, pointers to first and last messages, etc. are stored in the queue descriptor. The descriptor must be in S-memory during the existence of the queue. Users of the queue are given "Q-keys" which serve as capabilities for the queue. For a queue file, the Q-key is stored in the FIB. If the queue is empty, the 360-bit descriptor is the only S-memory structure dedicated solely to the queue.

2.3 QUEUE DISK

Messages stored in a queue may reside on disk or in S-memory. At queue creation (an area of system disk is obtained for the queue large enough to hold Q.MAX.MESSAGES of size Q.MAX.MESSAGE.SIZE. For example, a queue of max length 255 and max message size 200 bytes will require 255 *((200+179)MOD 180) = 510 disk segments. Pre-allocation guarantees 1) trivial assignment of disk space to a queued message and 2) low S-memory requirements to remember the state of the queue disks. Users with cartridge systems may have to limit Q.MAX.MESSAGES to avoid disk space problems, however. Queue disk is not locked in the directory - if the system fails while queues are active, the disk is returned to the available list during clear/start. Note that even queues with Q.BUFFERS = Q.MAX.MESSAGES are assigned disk.

The algorithm for putting messages on disk is as follows. If a message being put in a queue makes the count of messages in <u>memory</u> equal to Q.BUFFERS, then the tail-most message in memory is started out to disk. This should free a buffer for a following queue insert. Exception: See Section 2.5 for queues where Q.BUFFERS≥Q.MAX.MESSAGES.

2.3 QUEUE DISK (Continued)

GETSPACE may also roll messages to disk if it requires the space. This ensures that messages left in infrequently-read queues will be cleared out of main memory.

When the first message is removed from the queue, the next message is checked to see if it is on disk. If so, a look-ahead disk read is initiated to help the reader.

Disk I/O descriptors reside in the queue file FIB's for queue disk I/O. For each mode of use - input or output - a program opening a queue file is given 1 I/O descriptor. A file opened input and output is given 2. I/O descriptors are shared among all members of a queue file family, so no FIB will have more than 1 or 2.

2.4 MESSAGE DESCRIPTORS

The means of storing messages in the queue is by means of a linked list of message descriptors. Each message descriptor (MD) consists of an 80-bit system descriptor and two link fields, for a total of 128 bits each. The system descriptor actually describes the message text, according to normal MCP conventions.

To reduce checkerboarding, MD's are allocated in blocks of 10. Allocation is done by searching the block(s) of 10 for a zeroed-out MD. If none is found, an additional block is obtained via GETSPACE. Blocks are surveyed periodically to FORGETSPACE any spares. At least one block is retained while any queues exist.

2.5 MESSAGE BUFFERS

If a queued message is in S-memory, it is stored in a memory link called a message buffer (MB). No queue may have more than Q.BUFFERS messages in MB's, including those in-process between disk and S-memory. Note that because of the method of handling queue disk, the queue driver cannot guarantee that any given message will be kept in S-memory, even for a queue where Q.BUFFERS ≥ Q.MAX.MESSAGES. For this type queue the queue driver will not start an I/O to write a message to disk, but GETSPACE may do so if space is tight.

MB's are allocated from a common pool of empty buffers. The pool is implemented as a linked list ordered by size so allocation is both best fit and first fit. If the pool does not contain a large enough MB, one is obtained from GETSPACE. When a message is removed from an MB, it is put into the pool. GETSPACE may recapture MB's for other uses by simply delinking them from the queue driver's list.

3.0 QUEUE FILE OPEN AND PARAMETER SPECIFICATION

In addition to parameters common to all files, the user may specify three parameters whose interpretation has special meaning for queue files:

1. Q.MAX.MESSAGES - the maximum number of messages a queue can store, at which point it is full (max 255).

2. Q.BUFFERS - the highest number of messages (max 255) the user wishes to allow in S-memory.

3. Q.FAMILY.SIZE - the number of sub-queues in a multiple queue file.

In "?FILE" notation, for example:

```
?FI MY.Q QUEUE   Q.MAX.MESSAGES=20  BUFFERS=3  RECORD.SIZE=80;
?FILE MY.QFF QUEUE  Q.MAX.MESSAGES=10  Q.FAMILY.SIZE=3 BUFFERS=2
                                                        RECORD.SIZE=80
```

In SDL notation the same files might look like this:

```
FILE MY.Q (DEVICE=QUEUE(20)                   BUFFERS=3, RECORDS=80);
FILE.MY.QFF(DEVICE=QUEUE(10)  FAMILY(3),      BUFFERS=2, RECORDS=80);
```

In COBOL notation:

```
SELECT MY.Q  ASSIGN TO QUEUE.
  .
  .
  .
FD MY.Q  VALUE OF Q.MAX.MESSAGES IS 20
  RESERVE 3 ALTERNATE AREAS.
01 MY.Q.BUF   PIC X(80).

SELECT MY.QFF  ASSIGN TO QUEUE.
  .
  .
  .
FD MY.QFF  FILE CONTAINS 3 QUEUES
  VALUE OF Q.MAX.MESSAGES IS 10
  RESERVE 2 ALTERNATE AREAS.
01 MY.QFF.BUF  PIC X(80).
```

If a queue file family (QFF") is opened (See Figure 2), the same parameters apply to every member individually. In MY.QFF above, for example, all 3 members may hold 10 messages each with 2 each in memory.

Naming is as in old queues - the MFID/FID is the name of the queue for a single queue file. For a QFF, the MFID is the first 10 characters, and a FID is synthesized from the member number for each queue in the family (e.g. the first member of MY.QFF would be "MY.QFF"/"##00000001").

3.0 QUEUE FILE OPEN AND PARAMETER SPECIFICATION (Continued)

When a Q-file is opened, the Q-driver compares the 20-character name with the names of all queues currently in existence. If a queue of that name is found, the opener is linked to the existing queue and the queue's user count is incremented. If the queue does not exist, a new queue is created with the parameters provided in the FPB.

Queue parameter binding occurs when the queue is first created -- that is, by the first process to open the queue file. If two programs share a queue (e.g., both agree on the name), the first program to open his queue file binds the parameters.

Blocking of records is not used in new queue files. Record size sets an upper limit on the length of a message stored in a queue file.

4.0 SUPPORTING BOTH OLD & NEW QUEUE FILES IN 5.1 MCP

Existing code using Q-files need not be recompiled to run on the 5.1 MCP. This causes one problem, handling queue empty on READ; old queue files gave the "EOF" branch (if present); new queue files will give the "INCOMPLETE.IO" branch (if present). If the MCP can discriminate between old and new queue FIB's at run time, this processing is possible. We use a new device-type for new queue files to allow this:

```
FPB.HDWR   61 = New Queue File
FIB.TYPE   62 = Old Queue File
```

<u>QUEUE files and the FILE card</u> - Label equation of any file to the symbolic device "QUEUE" will result in FPB.HDWR set to 61 (new queue). Two new keywords have been added to the FILE card: "Q.FAMILY.SIZE" and "Q.MAX.MESSAGES".

Note that the structures created at run time are always new queues. The FIB-type of old queue-file merely changes the ON-branch used at queue-empty on read.

5.0 FPB FORMAT FOR QUEUE FILES

Two new fields "FPB.Q.FAMILY.SIZE" and "FPB.Q.MAX.MESSAGES", have been added to the currently defined FPB items:

```
01  FPB

    02  FPB.HDWR                BIT       ( 6),     % 61: new    62: old

    02  FPB.BUFFERS             BIT       (24),     % number queue
                                                    messages allowed
                                                    in memory

    02  FPB.INV.CHARS           BIT       ( 2),
    02  FPB.SERIAL              CHAR      ( 6),     % last item in
                                                    V.0 FPB
    02  FPB.Q.FAMILY.SIZE       BIT       ( 8),     % number members
```

5.0 FPB FORMAT FOR QUEUE FILES (Continued)

Mapping SDL file syntax onto the FPB is as follows (uncapitalized letters stand for user-supplied integers):

```
FILE QFILE (DEVICE=QUEUE(n),  BUFFERS=b,  RECORDS=r);
    % A SINGLE QUEUE    QUEUE FILE

FILE QFF1 (DEVICE=QUEUE(n)  FAMILY(f),  BUFFERS=b,  RECORDS=r);
    % A QUEUE FILE FAMILY

FPB.HDWR    :=  61  % NEW QUEUE
FPB.BUFFERS:=  b    % MESSAGES ALLOWED IN S-MEMORY
FPB.Q.MAX.MESSAGES   := n  % QUEUE MAXIMUM NUMBER OF MESSAGES
FPB.Q.FAMILY.SIZE    := f  % 0 FOR SINGLE QUEUE FILE
FPB.QUEUE.FILE       := 1  IF "FAMILY" appears after "DEVICE=QUEUE"
```

The only queue parameter that can be defaulted is BUFFERS, which will be set to 2 by the compiler.

6.0 QUEUE FILE READ/WRITE

The meaning of the three ON-branches is given on the following table:

		EOF	EXCEPTION	INCOMPLETE.IO
NEW	READ	No writers, & queue empty	Invalid key (on QFF only)	Queue empty but writers still exist
	WRITE	Not defined	Invalid key (on QFF only)	Queue full
OLD	READ	Queue empty	Invalid key QFF	Not defined
	WRITE	Not defined	Invalid key QFF	Not defined

6.1 QUEUE FILE FAMILY READ/WRITE

A key must be included to identify the specific queue in a QFF to read/write, just as in random disk files. QFF members are logically numbered 1 to n. Giving a key of zero on a read is defined as an unspecific read. The members will be searched beginning with number 1 and the first queue member found not empty will be read. A key of zero on a write is invalid.

Note: SDL is peculiar in this regard. From an SDL program, QFF members are given key numbers from 0 to n-1, even though the external names of the queues are numbered 1 to n. An SDL unspecific read with a key can be obtained by using a key of -1 (or not zero).

See the MESSAGE.COUNT communicate for another QFF facility.

6.2 WRITE TO TOP OF QUEUE FILE

This service is provided for NDL primarily, but it may be of general use for private stack-files for instance. A record written to a queue file normally goes on the bottom of the queue, but a record may be inserted at the top with this WRITE. The function is invoked in the communicate by setting bit 7 of CT.ADVERB (i.e., SUBBIT(CT.ADVERB, 6, 1)).

SDL syntax is:

```
WRITE filename TOP (where);
```

For example:

```
WRITE MY.Q TOP (BUF);
WRITE QF.FAMILY [1] TOP (BUF);
```

Write to top is not implemented in COBOL.

6.3 COBOL SYNTAX FOR Q-EMPTY/Q-FULL

COBOL syntax applies the "USE procedure" to handle incomplete I/O on queue files.

For example:

```
PROCEDURE DIVISION.
DECLARATIVES.
Q SECTION.
USE FOR Q-EMPTY ON MY.Q.IN
PROC-MY-Q-IN-EMPTY.

  ~ COBOL procedure code

USE FOR Q-FULL ON MY-Q-OUT
PROC-MY-Q-OUT-FULL.

  ~ COBOL procedure code
```

7.0 END OF FILE FOR QUEUE FILES

End of file is defined for new queue files in a way similar to EOF on all other devices. It is now possible to label-equate an input file of device type CARD to QUEUE, for instance.

READ EOF

The precise meaning of EOF on READ queue file is that (a) the last writer on this queue has closed his queue file, and (b) the queue is empty. EOF is treated as a pseudo-message in the queue. That is, when the last message has been read from the queue file, the queue remains "not empty" for WAIT purposes. A subsequent read will result in the EOF branch being taken. The queue is then empty, but still in EOF status, so if yet another read is issued on the queue file, the reader will again take the EOF branch. The EOF can be cleared by either the reader closing and reopening the file or by the opening of the queue by a new writer.

7.0 END OF FILE FOR QUEUE FILES (Continued)

QFF READ EOF

Reads to specific members of a QFF are treated exactly like reads on single queue files. An unspecific read on a QFF will return EOF only if all members of the family are at EOF (i.e., empty, no writers). When the last writer closes any member queue of a QFF, the event Q.INSERT.OCCURRED will be caused for the QFF; this will put the reader in the READY.Q when he WAITS on this event. A subsequent MESSAGE.COUNT communicate will show the EOF as a pseudo-message -- the count for that member will be one more than the count of real messages. When the reader executes a specific read on the member queue which is at EOF, the EOF branch will be given. The next MESSAGE.COUNT will show the member queue as containing no messages. Another read on the member will result in the EOF branch given again (as with a single queue file).

WRITE EOF

Not presently defined but under consideration.

8.0 MESSAGE.COUNT COMMUNICATE

This communicate returns the count of messages of the queue file specified. If a queue file family is specified, the count of each member will be returned in an array (member 1 in the first position, member 2 in the second, etc.), up to the limit of the result field. Counts will be returned either in decimal (COBOL "PICTURE 999") or binary (SDL "BIT(24)") depending on the value of the first bit of CT.ADVERB. COBOL does not implement MESSAGE.COUNT.

Format:

CT.VERB	48 (HEX @30@)
CT.OBJECT	File Number
CT.ADVERB BIT 1	Decimal format results if true
CT.1	Result field length in bits
CT.2	Result field address

SDL/UPL syntax:

```
     MESSAGE.COUNT (file-name, address-generator  );
```

8.0 MESSAGE.COUNT COMMUNICATE (Continued)

MESSAGE.COUNT SDL example:

```
FILE MY.QFF (DEVICE=QUEUE(10) FAMILY (5)
            ,BUFFERS=2, RECORDS=80);

DECLARE X  FIXED

       ,01  PUT.IT.HERE
           ,05  Q.MSG.COUNT(5)  BIT(24)
           ;

OPEN MY.QFF (INPUT);
    ∫
    process
    ∫
WAIT (TIME.TENTHS(100), Q.INSERT.OCCURRED (MY.QFF));

CASE X;

    TIME.OUT.PROCEDURE;

    DO; MESSAGE.COUNT   (MY.QFF, PUT.IT.HERE);
        ∫
        Process messages
        ∫
    END;

END CASE;
```

9.0 RECOMMENDATIONS FOR USERS OF QUEUE FILES

a) Declare short queues (Q.MAX.MESSAGES=1) unless the extra length is needed for "burst mode" operation. Long queues tie up resources and incur long delays for each message, while providing no additional function. Disk queues are discouraged for the same reason. Note, however, that if the queue runs empty, the cost of a disk queue is limited to the system disk required to hold the queue. Therefore, if you must have the potential for occasional long queues, try to ensure that the reader has high enough priority to keep the queue emptied. It will then run as fast and in the same real memory as a memory queue.

b) Use the INCOMPLETE.IO branch on READ/WRITE only if you have real time difficulties with another data-stream (for example, datacomm I/O driving or a reader-sorter). Your program may as well be blocked on queue empty or full and release the processor to other tasks. Be careful of deadlock here though.

c) Use the EOF branch on READ's - it notifies the reader of termination of the Q-writer. It should serve as a termination condition, possibly for ABEND, for the reader.

Figure 1: Two Programs Communicating via a Queue-File called "MY.Q". The Queue contains three messages.

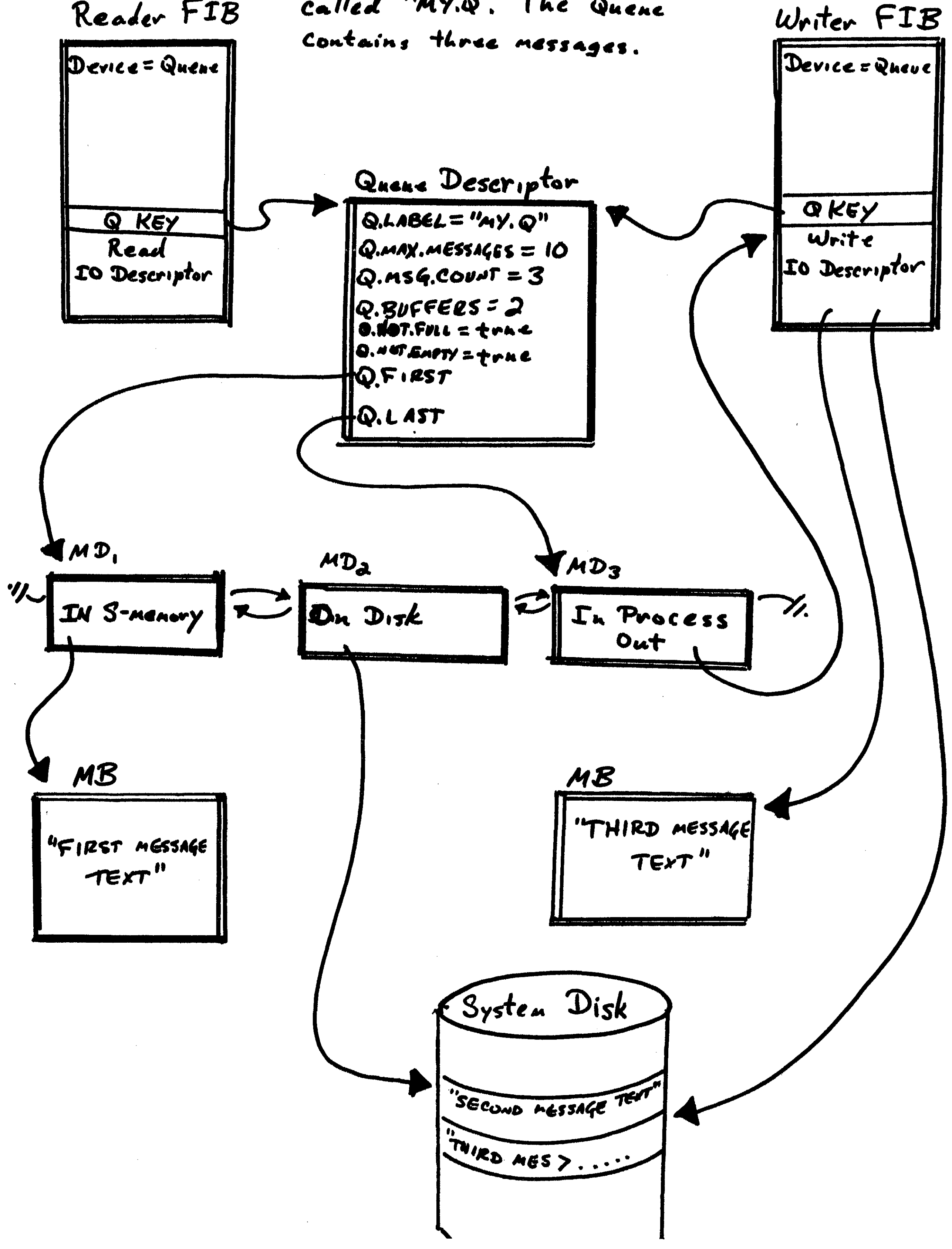
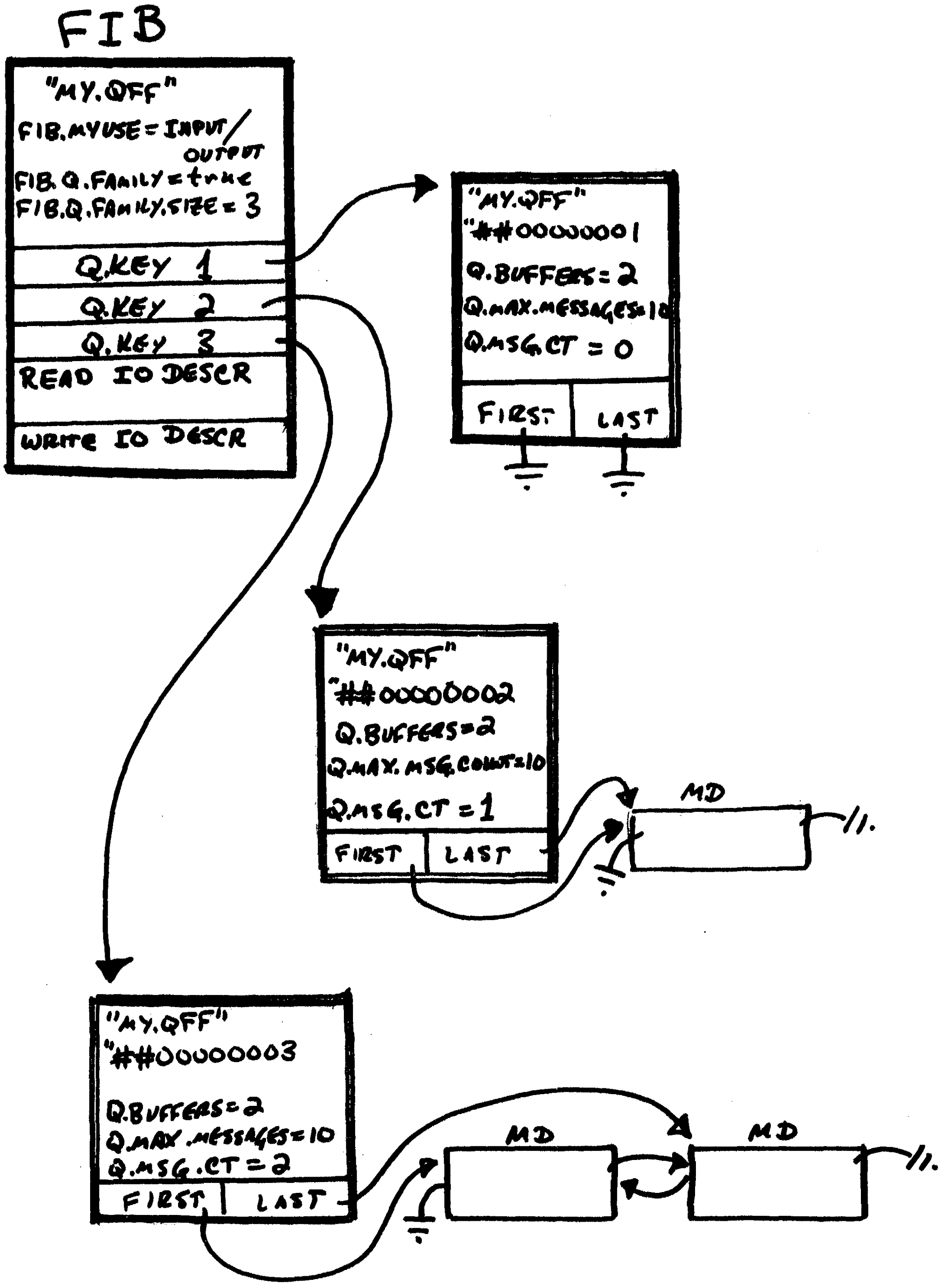


Figure 2: A Queue-file Family with three members.


FIB
"MY.QFF"
FIB.MYUSE = INPUT/OUTPUT
FIB.Q.FAMILY = true
FIB.Q.FAMILY.SIZE = 3
Q.KEY 1
Q.KEY 2
Q.KEY 3
READ IO DESCR
WRITE IO DESCR
"MY.QFF"
"##00000001
Q.BUFFERS = 2
Q.MAX.MESSAGES = 10
Q.MSG.CT = 0
FIRST
LAST
"MY.QFF"
"##00000002
Q.BUFFERS = 2
Q.MAX.MSG.COUNT = 10
Q.MSG.CT = 1
FIRST
LAST
MD
"MY.QFF"
"##00000003
Q.BUFFERS = 2
Q.MAX.MESSAGES = 10
Q.MSG.CT = 2
FIRST
LAST
MD
MD