Z77-7007

TECHNICAL INFORMATION EXCHANGE

IBM®

January 30, 1967

OPERATING SYSTEM/360 PRINCIPLES

Mr. J. K. Boggs, Jr.
IBM Corporation
6900 Fannin Street
Houston, Texas 77025

Two principles are basic to understanding Operating System/360 (OS/360). The first is that of a queue. The important function to be accomplished is resource allocation. In order to improve the throughput of the system, maximum use of limited resources must be accomplished. A method to accomplish this is the queue. The queue is a way of requesting a resource so that my request is filled at the earliest possibility.

The second principle is that of events. A dynamic system must manage multiple requests for resources. When resources later become available some way to grant unrelated requests must be accomplished. The event is a way to accomplish this .

For IBM Internal Use Only

IBM Corporation, Technical Publications Dept., 112 E. Post Road, White Plains, N.Y. 10601

Z77-7007

Operating System/360 Principles

Two principles are basic to understanding Operating System/360 (OS/360). The first is that of a *queue*. The important function to be accomplished is resource allocation. Inactive resources waste time and, thus, cost money. In order to improve the efficiency (throughput) of the system, maximum use of limited resources may be accomplished through use of the queue. The queue is a way of requesting a resource so that my request is filled at the earliest possible moment. A familiar example of the queue is a waiting line at a ticket booth. The resource to be allocated is the ticket seller's time and, one by one, those who wait get the resource.

The second principle is that of *events*. A dynamic system must manage multiple requests for resources. When resources later become available some way to grant unrelated requests must be accomplished. The event is a way to accomplish this wait. We have events occurring daily. "Has the mail come yet?" is a question about an event which must occur before we can pay the bills or read over our favorite magazine.

Waiting must be implemented in a complex system using multiprogramming. Two ways to accomplish this are queueing and eventing. This paper discusses how both apply to OS/360.

1. INTRODUCTION

The advent of very powerful hardware computing systems has placed a new emphasis of the Programming Systems — the limitations have become less hardware oriented. Rather, performance of a system, as measured by throughput, has become dependent upon the software monitor which coordinates activities. It has become apparent that it is necessary to use a multiprogramming environment in order to get the most out of the system. But multiprogramming places a new demand upon the software; that of waiting without stopping.

"Waiting without stopping" means that Program-1 can be put into a software wait state while the CPU switches to Program-2 which is ready to execute. To determine the magnitude of the problem, consider the event that placed Program-1 in the software wait state. Program-1 had control first for some good reason and should assume control again. How can this be done? (Event means only that "something happens." This something could be an I/O interrupt or a normal program termination.) Multiprogrammed systems must have a way of waiting on resources which are not available and a way of responding to events (things that happen).

IBM
EDUC 10-1
J. K. Boggs 12/8/66

The two principles basic to multiprogramming are also basic to OS/360. These principles are Queues and Events.

2. WHAT IS A QUEUE?

The queue is a method whereby one resource may be requested and/or used. When the resource becomes available, the queue can be used to decide what request should be satisfied next.

A familiar queue is a waiting line at a ticket booth. The line of people waiting is itself the queue: the ticket booth is the resource.

There are three types of resources:

1. Immediately usable (re-entrant)
2. One-time usable (self modifying/destroying)
3. Serially reusable (self-initializing)

For the approach to be consistent and flexible, it will be convenient to create a piece of control information which can "represent" or "stand for" the resource to be queued upon. In OS/360, this control information is called a *Queue Control Block* (QCB). The QCB is one full word of core. Bit 0 of the QCB is a one when the resource it represents is being used (i.e., Bit 0 is the busy bit).

The type of the resource affects how many QCB's are required, Only one QCB is required for a re-entrant resource because any request can immediately be satisfied. However, a non-reusable resource requires a "new copy" of the resource for every request. For example, when a rocket is fired, the rocket is destroyed. If we require two rockets to meet in space, then we need two resources even though they might be identical twins. Every request for a non-reusable resource will thus require a new, additional QCB.

Finally, the serially-reusable resource brings us back to the ticket booth. This is a resource upon which we can really queue and only one QCB is necessary for any number of requests.

2.1 HOW ARE REQUESTS MADE?

A queue requires not only a resource (Fig. 1) which can be allocated, but also requests for the resource. The basic function of a request is to specify what resource is being requested. This function can be implemented using a full word in core to address the resource being requested. The physical implementation of a request is called a *Simple Queue Element* (QEL) i.e., a part of a queue. (Fig. 2)

QEL's can be located in core in two different ways: (1) the elements can be fixed in number and in consecutive core locations in a contiguous set of QEL's (Fig. 3); or (2) variable in number and distributed in core in a *distributed* set of QEL's (Fig. 4 and 5).

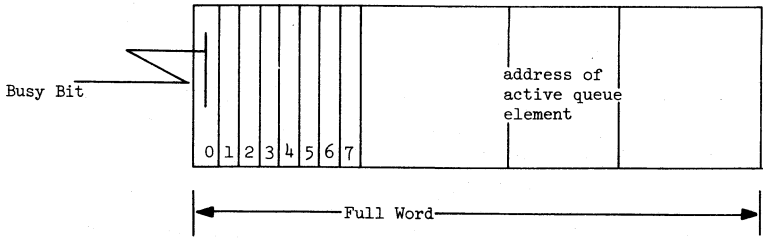


Figure 1. Queue Control Block (QCB)

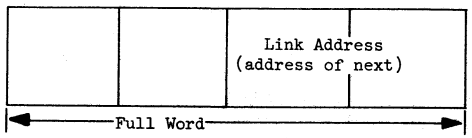


Figure 2. Simple Queue Element (QEL)

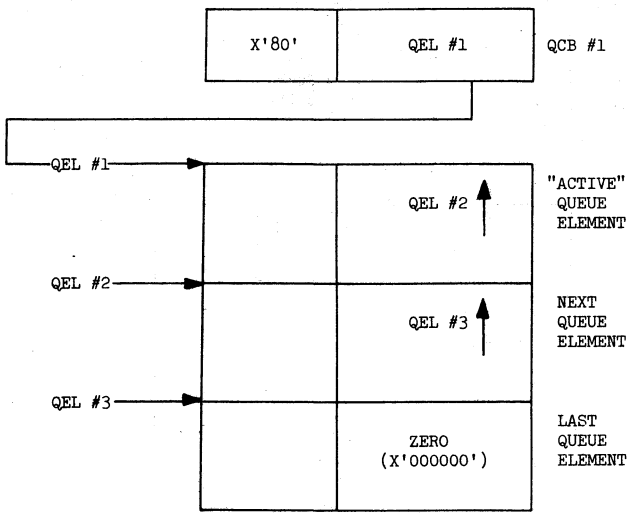


Figure 3. Contiguous Queue

IBM
EDUC 10-1
J. K. Boggs 12/8/66

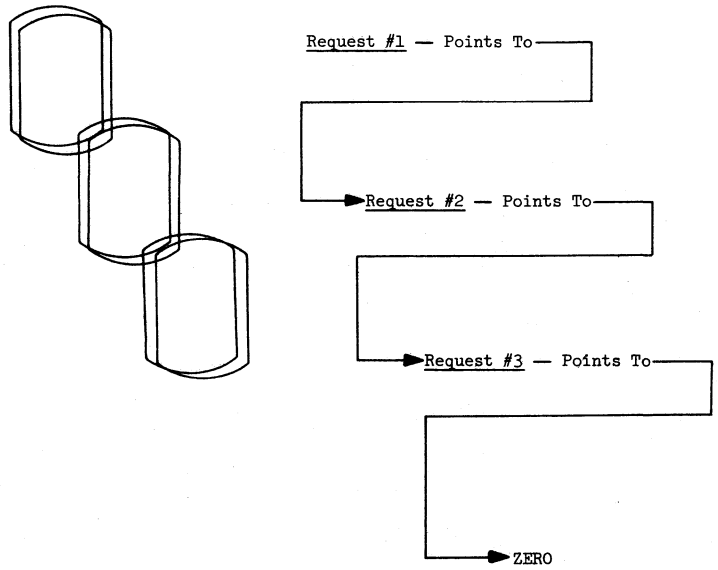


Figure 4. Chain of Requests

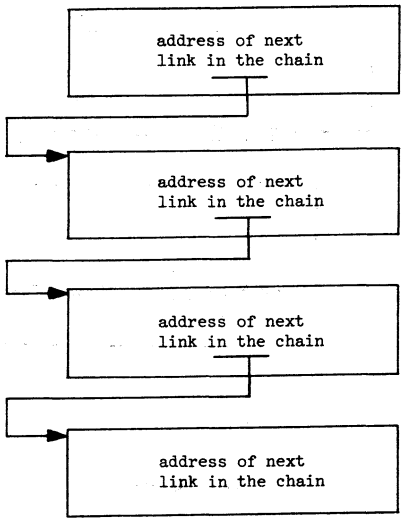


Figure 5. Forward Chaining

IBM
EDUC 10-1
J. K. Boggs 12/8/66

In addition to the forward (Fig. 5) chain of the QEL's mentioned, one can implement requests so that a search can be made in either direction. To implement this, a second word is required to address the request "behind" each request. This QEL is now called an *Expanded* QEL. (Fig. 6 and 7)

2.3 HOW IS QUEUE BUILT? SERVICED?

It is important to note the difference between the way the queue is built and the way that requests are granted (Fig. 8). The way that a queue is built refers to how addresses are placed into the QEL's. There are three common rules used to order a queue: a. LIFO (last in, first out) — the latest request is placed ahead of all other requests. This is like "cutting in" at the head of the ticket line, b. FIFO (first in, first out) — the latest requested is placed behind all other requests. This is the normal way of adding to the ticket line, c. PRIO (priority) — the order of requests is according to some numeric priority regardless of chronological order. There is a parallel between this method and the mayor walking into a movie while the rest of us wait at the ticket booth.

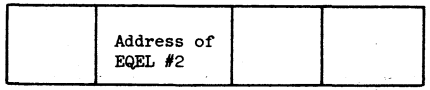
The manner in which requests are satisfied in a queue depends upon the resource. The QCB will contain, in bytes 2, 3, and 4, the address of the request being satisfied (or the request which presently has the resource).

When the present request is finished with the resource, the address of the next request in the queue is obtained from the then-finshed QEL and the chained QEL is made active. From this viewpoint, the queue is being serviced in a manner. Simple QEL's will always be serviced on a first in, first out basis.

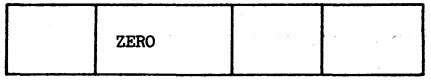
Expanded QEL's may use the second word to allow requests to be serviced LIFO. Using this strategy, word Number 2 of the QEL is used to service the queue and word Number 1 is used to build the queue.

For example:

EQEL #1



EQEL #2



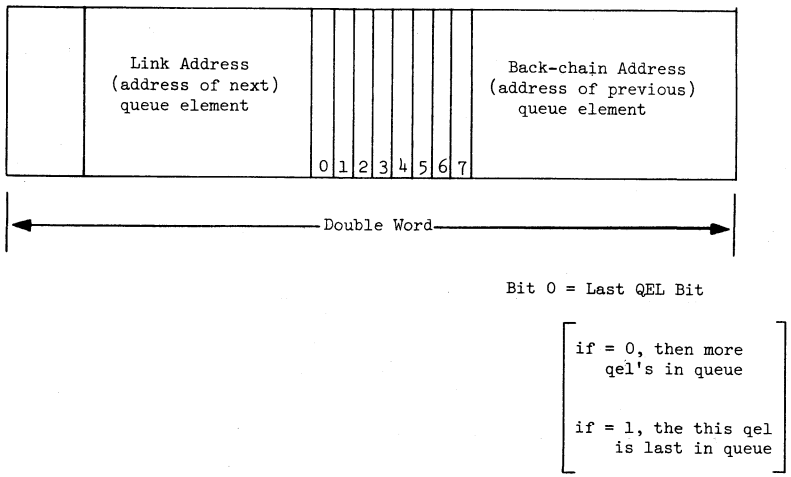


Figure 6. Expanded Queue Element (EQEL)

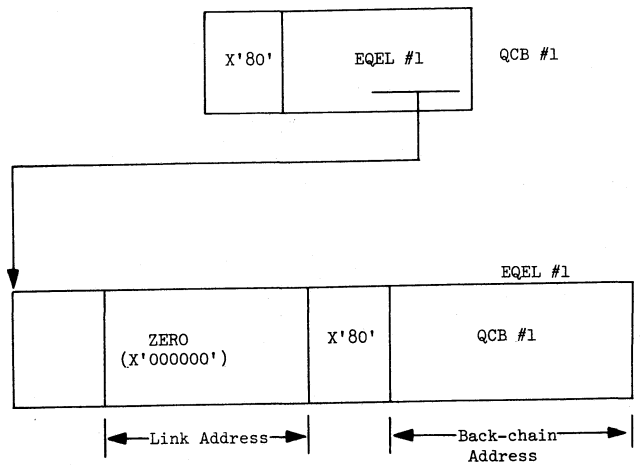


Figure 7. Backward Chaining

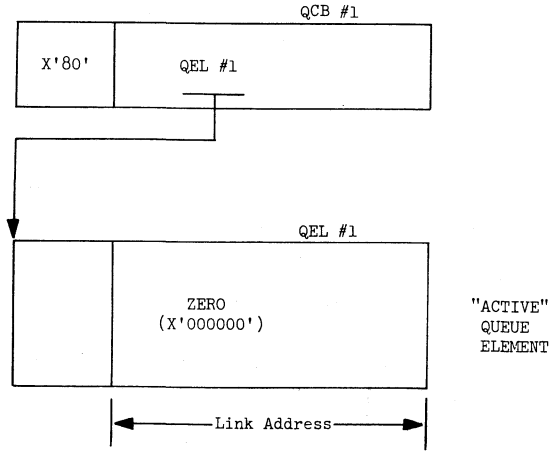


Figure 8. One-Request Queue

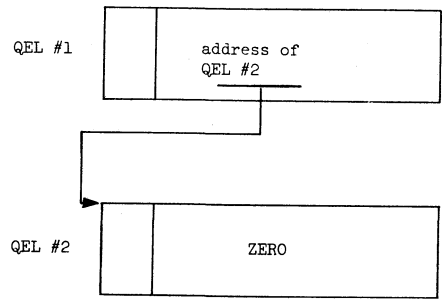


Figure 9. Basic Queue

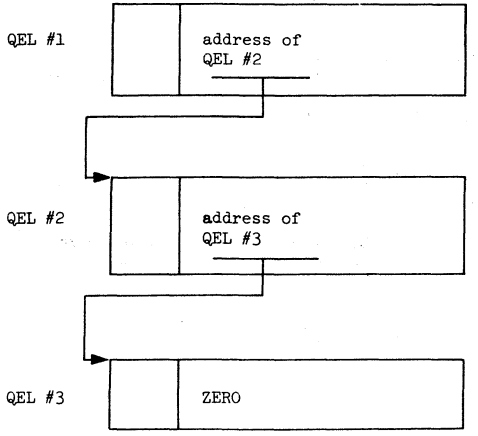
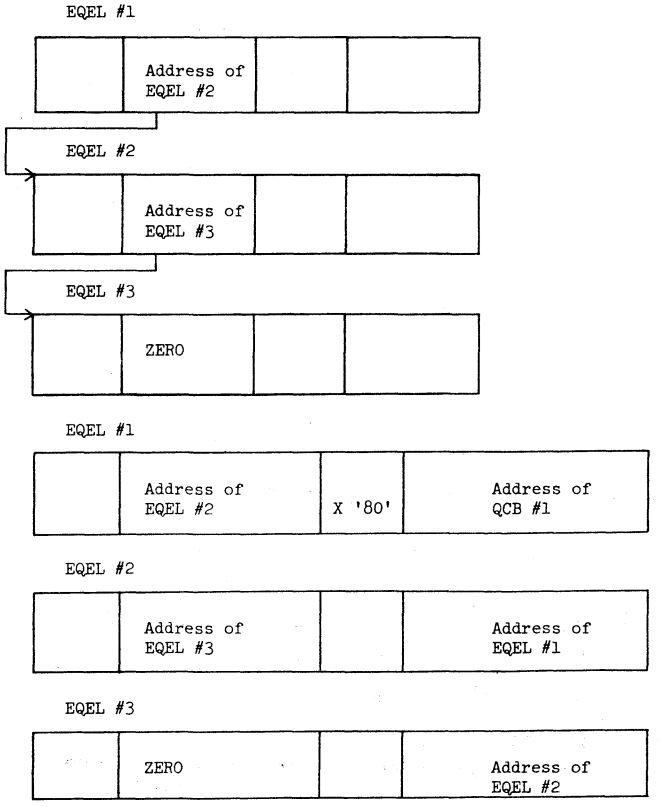


Figure 10. Basic Queue With Added Request

IBM
EDUC 10-1
J. K. Boggs 12/8/66

When another request is made, add it on in a FIFO manner.



2.4 ADVANTAGES

Contiguous QEL's are less flexible than *distributed* QEL's and require that core be permanently allocated. The major reason for using *contiguous* QEL's is that it is time consuming to allocate and de-allocate core. For input/output operations, the time demands are *very* heavy (literally thousands of records per minutes) and thus *contiguous* QEL's are used. For queues that are less demanding, *distributed* QEL's allow more freedom to accomplish new approaches (Fig. 11).

2.5 A NOTE ON USE

The QEL's and QCB's discussed in the preceding section are directly involved in the queueing function. In order to really "do something", additional information is required (Fig. 12). The convenient method seems to be to add additional storage to the control block.

The total of the information in the control block can now be made useful by way of a control program. The next topics describe how queueing is accomplished under OS/360 for the following queueing control blocks:

TASK CONTROL BLOCK - TCB
CPU QUEUE

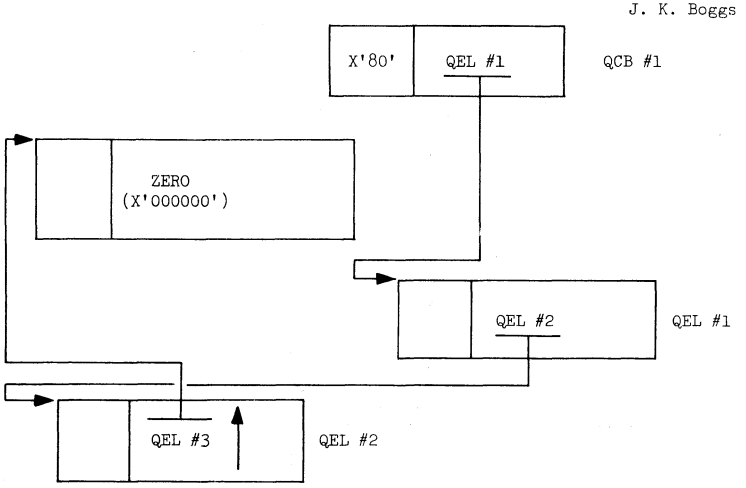
REQUEST BLOCK - RB
TASK QUEUE
PROGRAM QUEUE

3. PROGRAM MANAGEMENT

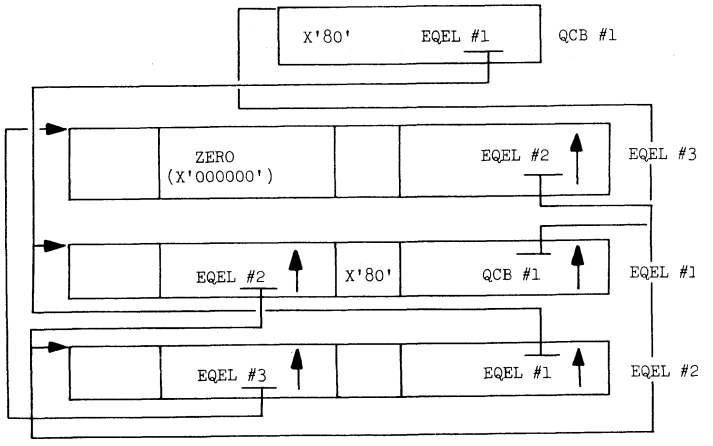
Requests for load modules are numerous in any computing system. In an environment where many assignments can be given the CPU, the multiple use of load modules becomes very attractive in order to conserve core. For example: in the FORTRAN shop, the square root routine (SQRT) is called upon. However, if ten FORTRAN programs are simultaneously executed, it is possible to have ten requests for the SQRT routine. If the routine is 100 words, we can, through multiple copies, find ourselves with 1000 words (10 x 100) allocated to SQRT (Fig. 13). If, however, requests are queued according to some rule, we can reduce the core overhead. Notice that this is a trade-off, because it will require CPU time to do management functions.

Consider three queuing rules: (1) first in, first out (FIFO); (2) last in, first out, (LIFO); (3) priority, highest priority first (PULL). The rule itself will not affect our queueing technique (QCB and QEL). In a FIFO queue for SQRT:

IBM
EDUC 10-1
J. K. Boggs 12/8/66



(A) Simple Queue Element



(B) Expanded Queue Elements

Figure 11. Distributed Queues

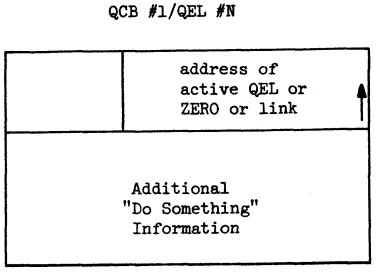


Figure 12. Relation Between Queue and Resource

IBM
EDUC 10-1
J. K. Boggs 12/8/66

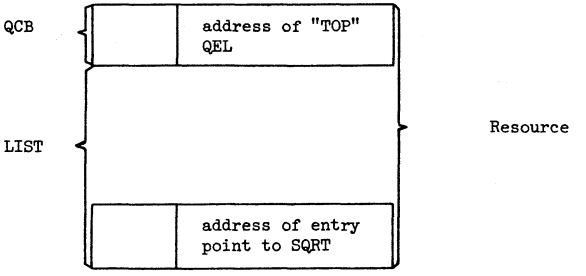
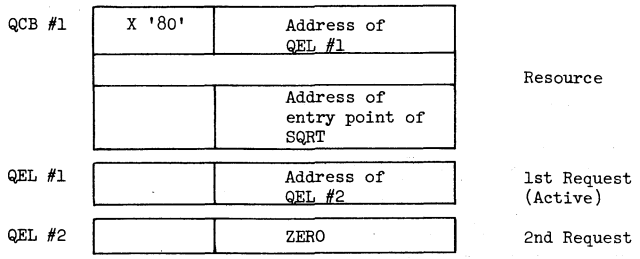


Figure 13. Program Management Queing Control Block

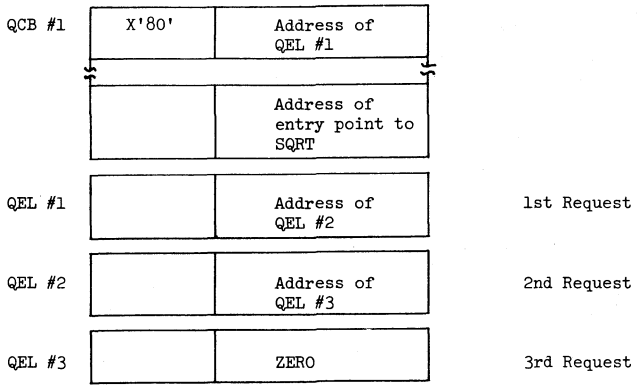


The rule which is used to order the queue does not affect our activation of the top queue element. En-queing then is just placing our request (QEL) in its position in the QEL chain. De-queing is just changing the addresses in the QEL's so that the chain no longer knows my QEL exists (Fig. 14).

A request is granted, usually, by placing the QEL address into the QCB; thus activating that request. We can talk about a queue describing the manner in which requests are granted, not the way the queue is built, e.g., even though the RB's are serviced LIFO, the RB queue on a task in built FIFO.

Returning to the management of our square root route (SQRT), the QCB and a list are established as shown in Figure 13.

With this kind of arrangement, we can easily add or delete requests in our queue. Suppose a third request occurs. We must scan the queue for the last QEL and insert in the second word the address of QEL 3. We must then create QEL 3 as follows:



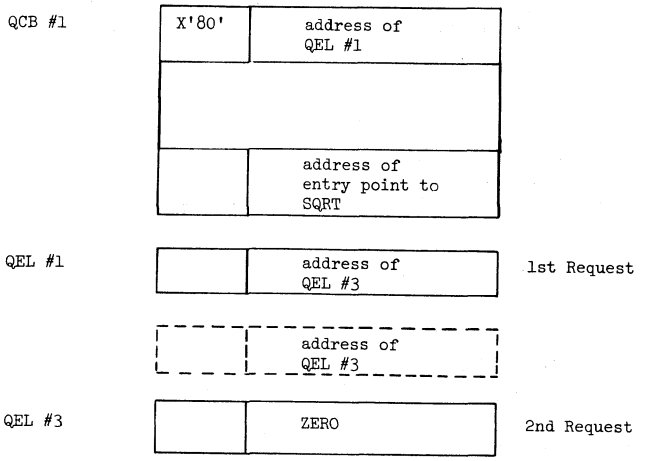


Figure 14. Program Management Queing

Suppose that now we find that we do not need to satisfy our second request because the job abnormally ended. To de-enqueue, all that needs to be done is to look at request 2 and "chain" around it. This process is simplified since, in fact, *expanded QEL's* are used.

BEFORE DE-ENQUEING

Address of QEL 2	Address of QCB	QEL 1 Request 1
Address of QEL 3	Address of QEL 1	QEL 2 Request 2
ZERO	Address of QEL 2	QEL 3 Request 3

AFTER DE-ENQUEING

Address of QEL 3	Address of QCB	QEL 1 Request 1
Address of QEL 3	Address of QEL 1	QEL 2 Request 2
ZERO	Address of QEL 1	QEL 3 Request 3

Now, if we start at QEL 1, the chain ends on the second element — QEL 3. QEL 2 has been chained around. The advantage of this approach includes some fast-executing codes.

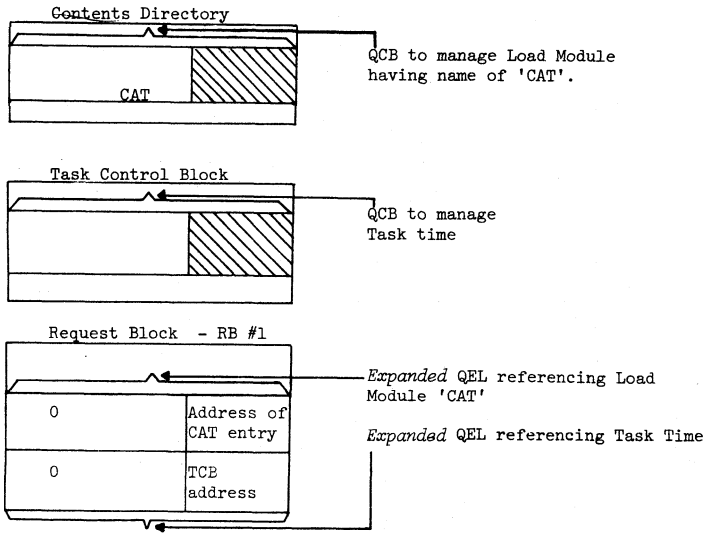
Program management can thus be implemented using QEL information and QCB information. Under OS/360, the list of load module QCB's is called the Contents Directory — the QEL is part of the request block (RB). The contents directory stands for the load modules. The RB consists of a double queue with two resources: the load module via the contents directory, and task time via the TCB.

The Task is a set of control information, the resource for which is CPU time. (See Section 4, Task Management)

To get some useful work done by the computing system, we need at least two resources: Task time (TCB), and a program (Load Module-Contents Directory). The queue elements, which request these resources, are in the RB.

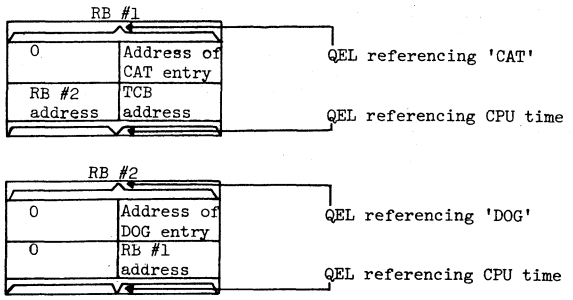
IBM
EDUC 10-1
J. K. Boggs 12/8/66

Let's see how this might appear in core:



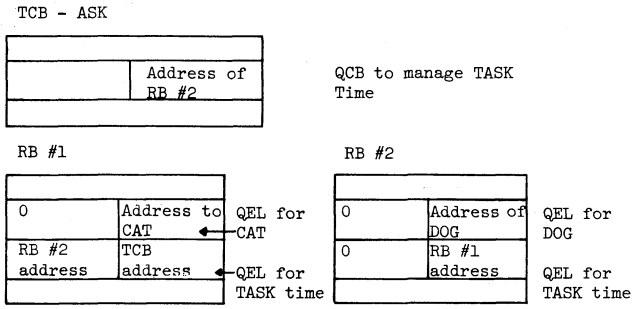
As you might imagine, it is possible for a given Load Module to request the services of another. All that must be done is to create another RB containing appropriate QEL's.

However, we know that only one load module at a time can have CPU time. We must have a rule, therefore, for building our QEL's that reference the TCB. A good choice appears to be FIFO. Therefore, a request by CAT for DOG would appear as:

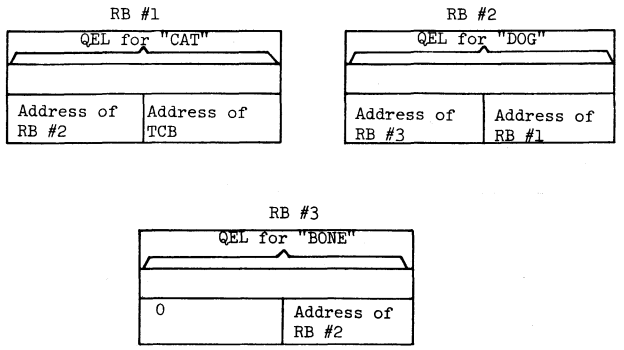


Note that, for load modules, though the RB queue is built FIFO; it is serviced LIFO.

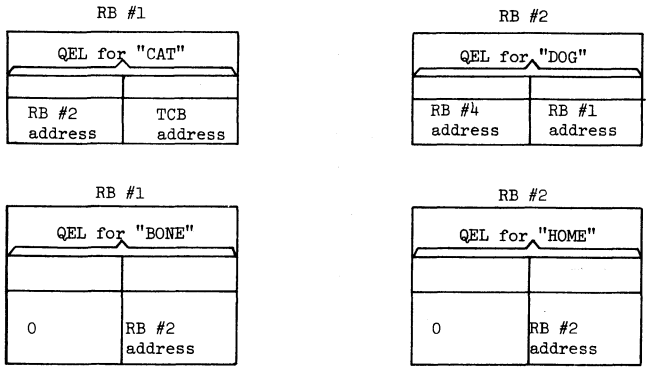
With this queue of RB's, we can now decide quickly which load module to activate. All we need to do is maintain, in the TCB-QCB, the address of the top RB. There:



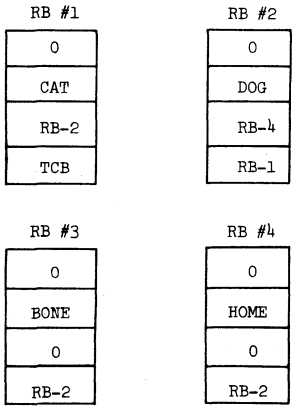
To add and delete RB's, we issue in our problem program macro statements LINK or XCTL. The LINK macro adds another RB (RB #3) to our present chain making it the top RB, i.e., the program that will have the next task time. The XCTL macro creates an RB to replace the RB for the load module that issued the macro. For example, if load module "DOG" issued a LINK to "BONE" we will have a third RB. And the three would appear as:



If "BONE" now issues the XCTL to "HOME," a fourth RB is created and replaces the the third RB. It would appear as:

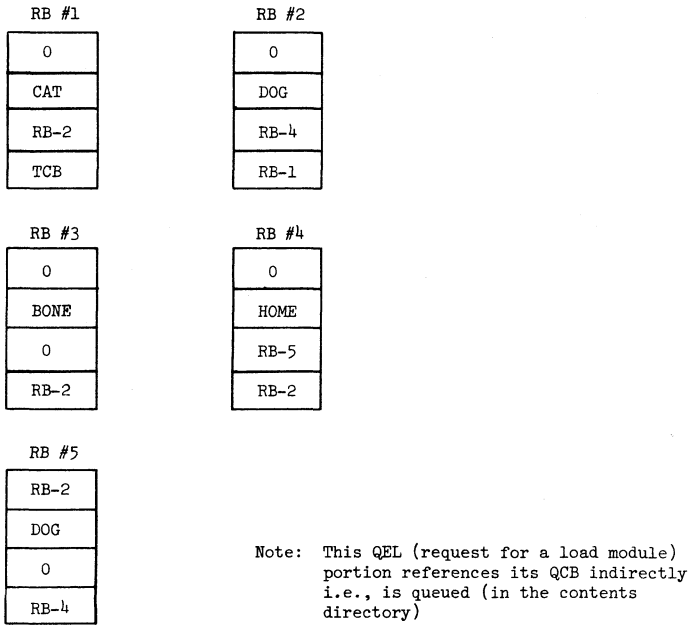


A short cut method for describing this sequence in the RB que is:



(This notation will be used in referring to RB's)

Notice that if "HOME" were to have a LINK to DOG, and if DOG is serially reusable, we can queue up this request. In fact, OS/360 will queue in a FIFO queue requests for serially reusable load modules (within a JOB). We now have:



Note: This QEL (request for a load module) portion references its QCB indirectly i.e., is queued (in the contents directory)

OS/360 has two macros which will give the problem programmer the ability to create his own FIFO queues. The programmer need only define a full word as a QCB and allocate a double word as a QEL every time he uses the macro. Thus:

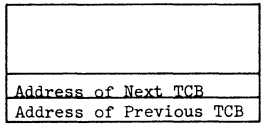
```
ENQ      QCB = NAME,QEL = FIRST

DEQUE    QCB = NAME,QEL = FIRST

NAME  DC   F'0'

FIRST DS   D
```

The problem programmer can use this ability if the address of the QCB and QEL are known.

IBM
EDUC 10-1
J. K. Boggs 12/8/66

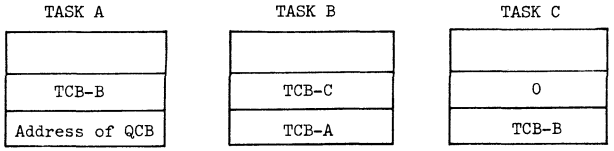
4. TASK MANAGEMENT

The task is that control block which controls the CPU. In previous systems this was called multiprogramming. For instance, the 7090 and 1410 Operating Systems both had options which included SPOOL (Simultaneous Peripheral Operations On-Line). This option allowed two tasks in the system — problem program, and utility operations (card-to-tape, etc.). When the problem program was not using CPU time, the operating system switched tasks until the problem program was again ready to execute. Under OS/360 there may be many tasks in the system at any given time. To efficiently manage these tasks, remember our old friends — the QCB and QEL's. The task control block is itself a QEL which looks like this:

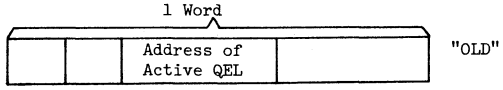


Expanded QEL
(Double Word)

So we might consider three tasks in our queue:



In this case, the QCB is located from the Communication Vector Table (CVT). The CVT is a scratch pad the system uses to communicate with itself. The QCB for task management looks like any other QCB:



"OLD"

All that needs to be done to switch tasks is to change the address in the task QCB which is called "OLD." It turns out that we use this location to tell us one of two things: (1) if an interrupt occurs and the TASK which requested the interrupt is different, and a higher priority, then the higher priority TCB address is placed in OLD and activated. Consider this decision table:

OLD = TCB address in Task QCB

NEW = TCB address requesting interrupt.

Decision Table #1

	Give control to "OLD"	Consult Table #2
OLD = NEW	X	
OLD ≠ NEW		X

Decision Table #2

	Scans TCB's	Give control to "OLD"	Give control to "NEW"
OLD = ZERO	X		
Priority OLD > NEW		X	
Priority NEW > OLD			X

When the operating system scans the TCB's, it gives control to the highest priority TCB which is ready. If none are ready, the system enters the WAIT state and sets "OLD" equal to ZERO.

Just a brief review of what Ready means. Since the TCB controls the CPU, it must also reflect several states. A TCB can be in the READY, WAIT, or ACTIVE state, one at a time.

ACTIVE: The task has CPU control.

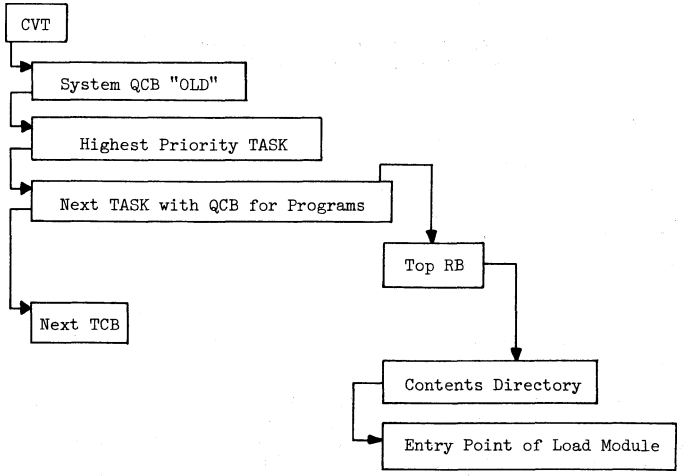
READY: The task is prepared to immediately use CPU time when it becomes available.

WAIT: The task is waiting for an event to occur and cannot use CPU time until the event does occur.

These several states have to do with the rules which activate or dispatch a task. These states do not affect the queueing technique. OS/360 builds the TCB queue by priority. This priority is called Dispatching Priority because it is used in dispatching a TASK. There is also a limit Priority established for the task which cannot be changed by the task itself. The task may; however, change its own Dispatching Priority at any time. (Dispatching Priority $\leq$ Limit Priority).

IBM
EDUC 10-1
J. K. Boggs 12/8/66

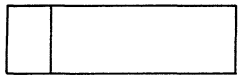
Let's review for a moment....the queues for task and program management are:



The management of resources is based on the concept of a control QCB which is referenced by QEL's. The system is built this way, and the user has available a control program service to create queues. Now let's consider events.

5. EVENTS

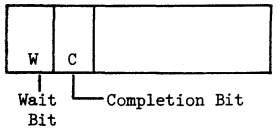
An event is simple from the viewpoint that the event has either occurred or it hasn't. Considering an event as the completion of the reading of a card record into an input area, it either has been done or it hasn't. Thus, logically, an event is binary and can be considered a logical switch. In addition, when an event occurs, the control program must know what CPU job was concerned with the event. Events don't just happen, they are caused. Thus, we must know who caused it...and who is waiting on it. The WHO is OS/360; the requester of CPU time is a task. Thus an event must be associated with a task. The implementation of an event is through an Event Control Block (ECB). The ECB must have a bit to indicate whether the event has occurred and the address of the RB which requested the event. Thus, we have defined:



ECB is one full word of core.

Low-order 3 bytes are the address of the RB or the code posted upon completion of the event.

Lower order byte



(Note that the TCB can be found by way of the back chain in the expanded QEL in the RB.)

There are three types of ECB's that must be considered. First, there is an ECB in supervisor core that is created for every ATTACH done by the Initiator/Terminator. This means that for every Job Step which is initiated in the System, there is an ECB in supervisor core with the I/T address. As a result, the Initiator/Terminator need only WAIT on the completion of that ECB (and my job step) before continuing. When the ECB is placed in the wait state, the wait bit is turned on, and the address of the RB which is waiting is placed in the ECB.

The second type of event is one created by the use of the interval timer. The STIMER and WAIT macro can create an ECB to wait on a time event. The implementation of this is similar to the ATTACH.

We must have ECB's for I/O or Data. The Data Event Control Block (DECB) is created for the set of requests by a job step for a data set. This DECB is then manipulated and used by the access methods that obtain the data.

Three types of events were discussed separately because they are separate logical chains which the control program services. Their use, however, is logically the same.

IBM
EDUC 10-1
J. K. Boggs 12/8/66

SUMMARY

The many control blocks of OS/360 appear at first to be confusing and illogical. However, two concepts are basic to their understanding and to the appreciation of OS/360.

The first concept is that of a queue. The queue is a management method which allows concurrent but unrelated requests to be satisfied in logical order. Two basic control blocks are used. The QCB represents the resource and QEL's for requests. Flexibility in the use of queues allows a consistent and powerful tool for developing as well as implementing OS/360.

The second basic concept is that of the event. The event, or occurrence of an interrupt, can best be satisfied by making a note in the bits of an ECB to the effect that "something" is going to happen. When that "something" does happen, the binary switch — ECB --- is tripped so that tasks may coordinate their activities.

OS/360 has taken a fresh and logical approach to the multiprogramming problems of data processing. The present consistent approach allows many varied possibilities for compatible growth. The concepts of queues and events will become as basic as the Hollerith Card Code.

APPENDIX A

QUEUE UTILITIES

The following routines perform service functions in the manipulation of queues. These functions are:

(1) Adding simple QEL's in a FIFO manner.

(2) Adding simple QEL's in a LIFO manner.

(3) Adding simple QEL's in a PRIORITY manner.

(4) DELETING the TOP QEL.

(5) DELETING a specific QEL.


```
*        UPON ENTRY
*
*        REGISTER  4 = QEL ADDRESS
*        REGISTER  5 = QCB ADDRESS
*        REGISTER 14 = RETURN ADDRESS
*
*
*        THIS ROUTINE USES REGISTERS 6,7 AND 8.
*
*        REGISTER  6 = WORK
*        REGISTER  7 = WORK AND RETURN INDICATOR
*        REGISTER  8 = BASE REGISTER
*
```

```
*
*       ERROR CONDITIONS AND INDICATIONS
*
*       REG 7= 0        QUEING FUNCTION REQUESTED UNCLEAR.CHECK
*                       BITS 3-7 OF BYTE ADDRESSED BY REG 4. ALL
*                       REGISTERS LEFT UNCHANGED.
*
*       REG 7= 2        ATTEMPT MADE TO DROP A QEL FROM AN INACTIVE
*                       RESOURCE. THE QCB IS ZERO'D. ALL
*                       REGISTERS  LEFT UNCHANGED.
*
*       REG 7= 4        ATTEMPT MADE TO DROP A QEL WHICH IS NOT
*                       CHAINED IN THE QUEUE. ALL REGISTERS
*                       LEFT UNCHANGED.
*
*       REG 7= 8        PREVIOUSLY BUSY RESOURCE HAS NOW BECOME
*                       INACTIVE DUE TO THE DELETION OF A QEL.
*                       (THIS INDICATOR IS NOT ON BY ITSELF.
*
*       REG 7= 16       PREVIOUSLY INACTIVE RESOURCE HAS JUST
*                       BECOME ACTIVE DUE TO SUCESSFUL  ADDITION
*                       OF A QEL. (THIS INDICATOR IS NOT   ON
*                       BY ITSELF.)
*
*       REG 7= 32       INDICATES THE SUCESSFUL ADDITION OF A QEL.
*
*       REG 7= 48       SEE  REG 7=16  AND  REG 7= 32 .
*
*       REG 7= 64       INDICATES  THE SUCESSFUL DELETION OF A QEL.
*
*       REG 7= 72       SEE  REG 7= 8  AND  REG 7= 64 .
*
*
*       REG 7= 128      INDICATES A PREVIOUSLY ACTIVE QEL HAS
*                       BEEN MADE ACTIVE.
*
*
*       REG 7= 192      SEE REG 7= 64 AND REG 7= 128 .
*
```



```
         BALR  3,0
         USING *,8
         LA    7,0                SET REG 7=0 TO INDICATE INVALID
*                                 QUEUE REQUEST.
         TM    0(4),B'00000001'   TEST FOR ADDING FIFO.
         BC    1,ADDFIFO          BRANCH IF YES.
         TM    0(4),B'00000010'   TEST FOR ADDING LIFO.
         BC    1,ADDLIFO          BRANCH IF YES.
         TM    0(4),B'00000100'   TEST FOR ADDING PRIO.
         BC    1,ADDPRIO          BRANCH IF YES.
         TM    0(4),B'00001000'   TEST FOR DELETING TOP QEL.
         BC    1,DELTOP           BRANCH IF YES.
         TM    0(4),B'00010000'   TEST FOR DELETING SPECIFIC QEL.
         BC    1,DELSPEC          BRANCH IF YES.
         BCR   15,14
```

```
ADDFIFO  TM    0(5),B'10000000'   TEST IF QCB IS BUSY.
         BC    8,LASTAD           BRANCH IF NOT BUSY.
FIFO     L     5,0(5)             REG 5= ADDRESS NEXT QEL IN CHAIN.
         L     6,0(5)
         LA    6,0(6)
         LTR   6,6
         BC    7,FIFO             BRANCH IF NOT LAST QEL IN CHAIN.
         IC    6,0(5)             SAVE FIRST BYTE FROM LAST QEL IN
*                                 CHAIN.
         ST    4,0(5)             STORE NEW QELADDRESS INTO LAST QEL.
         STC   6,0(5)             STORE SAVED BYTE.
LAST     SR    6,6                GET A FULL WORD OF ZERO.
         ST    6,0(4)             CLEAR NEW QEL TO ZERO.
         LA    7,32               SET REG 7= 32 TO INDICATE A SUCESSFUL
*                                 ADDITION OF A QEL.
         BCR   15,14
```

```
DELSPEC  TM    0(5),B'10000000'    TEST TO SEE IF QCB IS BUSY.
         BC    8,NOTBUSY           BRANCH IF NOT BUSY.
         L     6,0(5)              GET FIRST QEL ADDRESS.
         LA    6,0(6)              CLEAR HIGH ORDER BYTE.
         CR    6,4                 TEST TO SEE IF DELETING FIRST QEL.
         BC    8,TOPONE            BRANCH IF YES.
         LR    7,5                 SAVE QCB ADDRESS.
SEARCH   LR    5,6
         L     6,0(5)              GET NEXT QEL ADDRESS.
         LA    6,0(6)              CLEAR HIGH ORDER BYTE.
         CR    4,6                 SEE IF FOUND
         BC    8,HITEQUAL
         L     5,0(6)
         LA    5,0(5)
         LTR   5,5
         BC    7,SEARCH
NOTFOUND LR    5,7                 RESTORE ORIGINAL VALUE OF REG5.
         LA    7,4                 SET REG 7=4 TO INDICATE ATTEMPT.
*                                  TO DELETE A QEL NOT IN THE QUEUE.
         BCR   15,14
         BC    15,14               RETURN TO CALLER.
HITEQUAL MVC   0(3,5),0(4)         MOVE DELETED QEL LINK ADDRESS INTO
*                                  PRECEEDING QEL.
         L     7,64                SET REG 7= 64 TO INDICATE A
*                                  SUCESSFUL DELETION
         BCR   15,14
         BC    15,14
         END   TESTDEFD
```

```
ADDLIFO  TM    0(5),B'10000000'    TEST IF QCB IS BUSY.
         BC    8,LASTAD            BRANCH IF NOT BUSY.
ADD2     L     6,0(5)
         ST    6,0(4)              MOVE CONTENTS OF QCB TO NEW QEL.
         LA    4,0(4)                   CLEAR HIGH ORDER BYTE.
         ST    4,0(5)              STORE NEW QEL ADDRESS INTO QCB.
         OI    0(5),X'80'          TURN ON BUSY BIT OF QCB.
         LA    7,32(7)             SET REG 7= 32 TO INDICATE A SUCESSFUL
*                                  ADDITION OF A QEL.
         BCR   15,14
```



```
ADDPRIO  TM    0(5),B'10000000'    TEST IF QCB IS BUSY.
         BC    8,LASTAD                 BRANCH IF NOT BUSY.
         L     6,0(5)              GET NEXT QEL ADDRESS IN CHAIN.
         CLC   4(1,4),4(6)         COMPARE PRIORITIES.
         BC    2,HIGH              BRANCH IF NEW QEL IS HIGHER PRIORITY.
         LR    5,6                 REG 5= PREVIOUS QEL ADDRESS.
         L     7,0(6)
         LA    7,0(7)
         LTR   7,7
         BC    7,PRIO              BRANCH IF NOT LAST QEL IN CHAIN.
HIGH     IC    7,0(5)              SAVE FIRST BYTE FROM HIGHER PRIORITY
*                                  PREVIOUS QEL.
         ST    4,0(5)              STORE NEW QEL ADDRESS INTO PREVIOUS QEL.
         STC   7,0(5)              STORE SAVED BYTE.
         BC    8,LAST              BRANCH IF PREVIOUS QEL WAS LAST ON CHAIN.
         IC    7,0(4)              SAVE FIRST BYTE FROM NEW QEL.
         ST    6,0(4)              STORE ADDRESS OF LOWER PRIORITY QEL
*                                  INTO NEW QEL.
         STC   7,0(4)              STORE SAVED BYTE.
         LA    7,32                SET REG 7= 32 TO INDICATE A SUCESSFUL
*                                  ADDITION OF A QEL.
         BCR   15,14
LASTAD   LA    7,16                     INDICATE RESOURCE HAS JUST BECOME
*                                  BUSY.
         BC    15,ADD2
```

```
DELTOP   TM    0(5),B'10000000'    TEST IF QCB IS BUSY.
         BC    8,NOTBUSY           BRANCH IF NOT BUSY.
         L     4,0(5)              GET TOP QEL ADDRESS.
TOPONE   L     7,0(4)
         LA    7,0(7)
         LTR   7,7
         BC    8,EQUAL             BRANCH IF DELETING LAST QEL IN CHAIN
         L     6,0(4)              GET NEXT QEL ADDRESS.
         L     7,0(6)              MOVE CONTENTS OF QEL INTO QCB.
         ST    7,0(5)
         TM    0(6),B'10000000'    TEST TO SEE IF QEL HAD AT ONE TIME
*                                  BEEN ACTIVATED. IF IT HAD, THEN BITS
*                                  2-7 MAY BE SIGNIFICANT.
         BC    1,WASBUSY           BRANCH IF YES.
         LA    7,64                SET REG 7=64 TO INDICATE A SUCESSFUL
*                                  DELETION OF A QEL.
         OI    0(5),B'10000000'    TURN ON BUSY BIT.
         BCR   15,14
WASBUSY  LA    7,192               SET REG 7=64+128  TO INDICATE A
*                                  SUCESSFUL DELETION AND THAT THE
*                                  NEWLY ACTIVATED QEL HAS PREVIOUSLY
*                                  BEEN ACTIVE.
         BCR   15,14
NOTBUSY  LA    7,2                 SET REG 7=2 TO INDICATE ATTEMPT
*                                  MADE TO DROP A QEL FROM AN
*                                  INACTIVE RESOURCE.
         BC    15,SUB              GO CLEAR QCB TO ZERO.
EQUAL    LA    7,72                SET REG 7=72 TO INDICATE THAT A
*                                  PREVIOUSLY BUSY RESOURCE HAS
*                                  NOW BECOME INACTIVE.
SUB      SR    6,6                 OBTAIN A WORD OF ZERO.
         ST    6,0(5)              CLEAR QCB TO ZERO.
         BCR   15,14
         BC    15,14               RETURN TO CALLER.


*        CONSTANTS
*
CON      DC    X'000000'           CON= THREE ZERO BYTES.
REG4     DC    X'28'               TO BE USED FOR NONSPECIFIC DELETE.
```