Once the trajectory of a spacecraft in flight has been predicted, control measures are required to ensure that the trajectory data is applied in a consistent manner when calculating the many trajectory-related parameters required by the flight controllers of the space flight.

This paper describes the queue control techniques used in generating the predicted trajectory and the subsequent use of the trajectory data. The queue control logic that is discussed requires a minimum amount of main storage while a task is waiting to be performed.

Trajectory control programs in support of Apollo missions

by D. R. Quarles

The real-time computer programs[1] used in support of the Apollo missions are composed of highly complex mathematical programs, various control programs, and a sophisticated operating system. This paper describes a set of control programs, referred to as the trajectory control programs, that make use of features provided by the operating system to enhance the integrity and consistency of the output from some of the mathematical programs.

A large percentage of the data viewed by the flight controllers of a space flight is calculated from a table of position and velocity vectors that describe the predicted spacecraft trajectory for the next several hours.[2] The predicted trajectory, or *ephemeris*, is used to calculate such items as the time to fire retro-rockets to land in a given place, the direction in which a given radar antenna should be pointed to make contact with the spacecraft as soon as it crosses the horizon, which tracking stations will sight the spacecraft in a given revolution, and how a planned maneuver will change the current trajectory. Since many critical decisions are based on ephemeris data, great care must be taken to ensure that the ephemeris is updated in a timely manner and that other programs that might be affected by a changing ephemeris are locked out while the ephemeris is being generated. For example, data needed for determining the positions of contact stations might be rather disjointed if part of it were

based on an old ephemeris and part on a new ephemeris. The flight controller must be assured of receiving up-to-date and consistent information from the real-time computer program. The purpose of the trajectory control programs is to provide the necessary control logic to ensure that the ephemeris is successfully generated and to prevent simultaneous processing that could result in inconsistent or misleading information.

In this paper, we first describe the procedure for updating the trajectory and then indicate the operation of the trajectory control programs within a real-time operating system. The chief topic discussed is the queue control necessary for the operation. Several examples are used to illustrate the concepts of trajectory control.

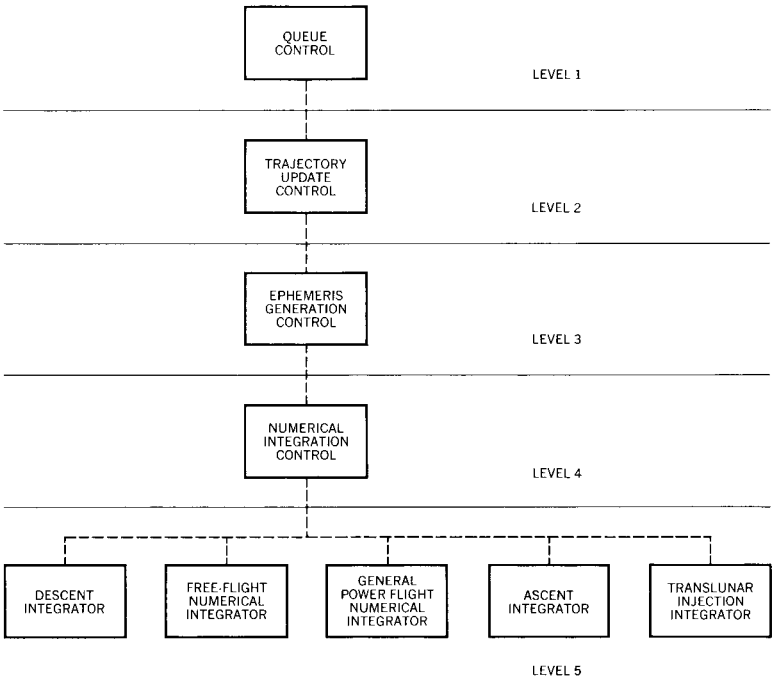
A particularly useful characteristic of the queue control logic is that a minimum amount of main storage is utilized while a task is waiting to be carried out. Even when the available main storage is very large, deadlock is a potential problem in a large multitasking system.[3] Use of the real-time queue control blocks and associated queue elements described herein reduces the required amount of main storage, thus lessening this problem. A control block requires only 32 bytes, each queue element about 60 bytes, and the message length varies according to the application.

Trajectory update

The process of generating the ephemeris and storing it for use by other programs is called a *trajectory update*. The trajectory is updated any time new information is available that increases the accuracy of the predicted trajectory. This includes a better determination of the current spacecraft position and velocity as well as any change in plans for maneuvers.

Figure 1 shows four levels of control programs that are used in the trajectory-update process. The queue control program in Level 1 is by far the most complicated of the trajectory control programs and is discussed at length in this paper. The trajectory-update control program in Level 2 is used to sequence the trajectory-update events which include, in addition to the ephemeris generation, generating revolution times and informing other tasks that the trajectory has been updated. The ephemeris generation program in Level 3 is used to control the process of generating and storing the ephemeris. Since the ephemeris is first generated in main storage and then stored on an auxiliary storage device, it is generated in segments. This process is repeated until the allocated space on the auxiliary storage device has been filled with segments. For this process, still another control program is used: the numerical integration program in Level 4. This program communicates directly with the mathematical routines that are numerical integrators.

Figure 1 Orbit trajectory-update control program levels



Selection of the proper numerical integrator depends on whether the trajectory segment is free-flight or whether it contains a maneuver by the spacecraft. If a maneuver is scheduled, the control program must determine the type of maneuver and select the proper maneuver integrator.

The trajectory-update procedure just described shows the use of control programs in the generation of the predicted trajectory, or ephemeris. By merely scheduling events sequentially, the control programs can ensure that there are no complications within the trajectory update itself. However, conflicts with other tasks cannot be prevented by sequencing events within the trajectory update. The queue control program is used to prevent conflicting processing from being initiated while the trajectory update is in process. A complete description of the queue control logic is presented later, but first some features of the operating system are discussed.

Real-time operating system

The Real-Time Operating System (RTOS) used in the computers that monitor and control space flights was developed from the IBM System/360 Operating System that provides multiprogramming with a variable number of tasks[4] (OS/360 MVT). Many new capabilities and concepts were added to OS/360 MVT, but only those

concepts used by the queue control program are discussed in this paper. Additional information on RTOS may be found in the reference material.[1,5–7]

RTOS provides a task structure that makes use of named *independent tasks*. As the name suggests, these tasks are completely independent of each other, and several tasks may be performed at any time. Communication between independent tasks is achieved by the RTATTACH macroinstruction. When an RTATTACH is issued by the control program, a queue element with an associated message is generated, and the message is passed to the specified task and load module when conditions permit. The program issuing the RTATTACH continues normal processing with no further concern for the message that it has passed to the other task. The format of the RTATTACH macroinstruction follows:

RTATTACH TASK=task name, EP=load module name,
QNAME=RTQCB name, QRANK=*n*, ID=*n*,
PARAM=(message, message length)

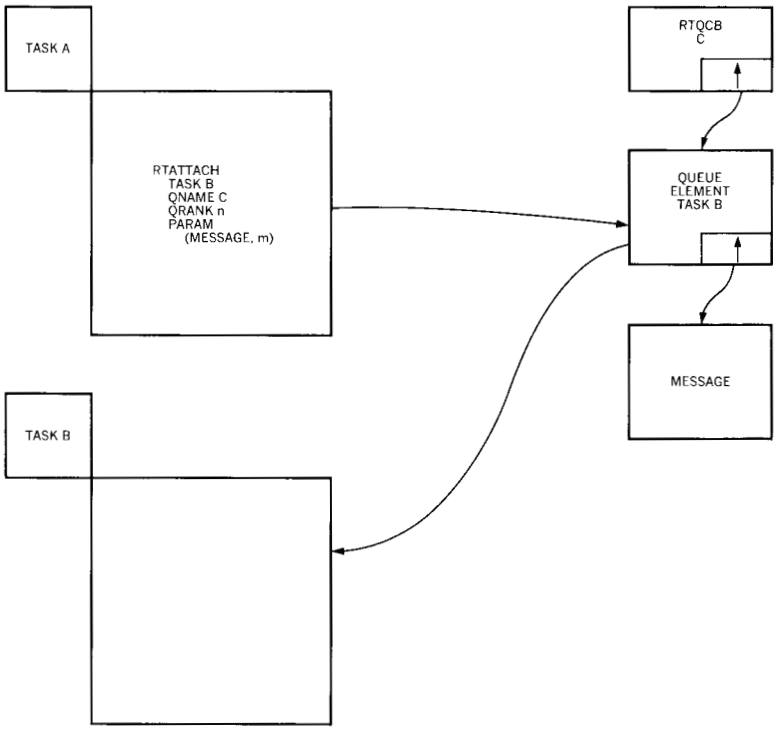
The parameters are as follows:

TASK	The name of the independent task that is to receive the message
EP	The load module name
QNAME	The real-time queue control block through which the queue element is to be chained
QRANK	The relative location of the queue chain in which the newly created queue element is to be inserted
ID	A number used to identify the message
PARAM	The address and length of the message that is to be passed

The queue element created by an RTATTACH is placed in a queue chain controlled by the real-time queue control block (RTQCB) that was specified by the QNAME parameter. Figure 2 illustrates such a situation.

The real-time queue control block has associated with it certain attributes that define how its queue chain is to be processed. These attributes may be dynamically changed by the RTQCNTRL macroinstruction. Among these attributes is a *length* which determines the maximum number of queue elements that may be entered in its queue chain before purging takes place, a *mode* which defines whether the control block is enabled or disabled, and an *order* which determines which queue element should be serviced first. The order may be first-in-first-out (FIFO), last-in-first-out (LIFO), or priority. If priority is the assigned order attribute, the QRANK parameter on the RTATTACH macroinstruction determines the ordering of the queue elements.

Figure 2 RTATTACH communication



RTOS rules

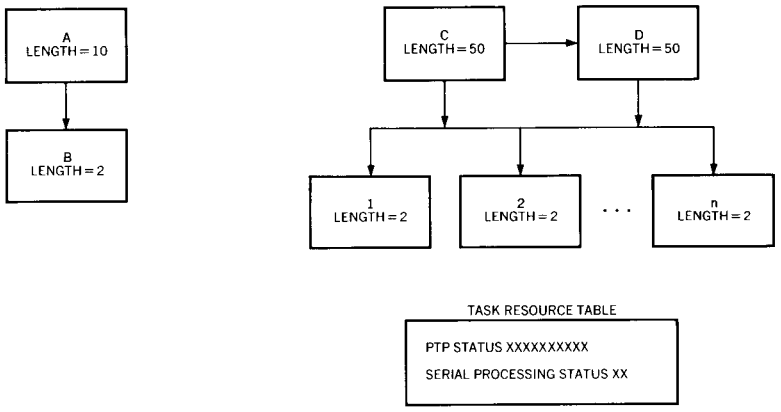
RTOS has the following rules for governing the servicing of queue elements from any real-time queue control block chain:

1. Only one queue element from a control block may be active at any time. That is, when a queue element is removed from a control block chain and passed to an independent task, no further queue elements may be serviced from that chain until the independent task receiving the first queue element is completed.
2. No queue elements will be serviced from a control block chain if the independent task specified for the first eligible queue element in the chain is currently being carried out; that is, it is considered active.
3. No queue elements from a control block chain will be serviced if the control block is disabled. However, additional queue elements may be entered into the chain.

These rules are referred to in subsequent examples.

The independent task and the real-time queue control block are the primary tools of the queue control logic employed by the trajectory control programs. Techniques for using these facilities are discussed in the following section.

Figure 3 Basic queue control logic



Queue control logic

The most sophisticated program in the trajectory control programs group is the queue control supervisor. Its function is to prevent simultaneous performing of independent tasks that can adversely affect each other. This includes protection of the trajectory update from tasks that might inhibit its processing, protection of users of the ephemeris from the trajectory update, and protection from each other of tasks that require a large amount of main storage.

The queue control technique employed by the queue control supervisor is basically one of enabling and disabling real-time queue control blocks. In addition, it takes full advantage of the RTOS queue-element servicing rules outlined previously. Figure 3 shows a group of control blocks that are used by the queue control supervisor. Note that each control block has a designated length which indicates the number of queue elements that it can contain.

Block A is used to chain up to ten incoming trajectory-update requests. It is assumed that it should never be necessary to stack more than ten trajectory-update requests. Block B is used to reroute the requests to the queue control supervisor and to dispatch them for processing. The reason for rerouting these requests is given in the examples. Block C is used for incoming requests from other tasks that require protection. These requests are known as "permission-to-process" (PTP) requests and are rerouted by Block D when necessary. Again, an explanation is given in the examples. The control blocks numbered 1 to *n* are used to dispatch such requests when the necessary protection has been provided. Each task that uses the protection feature provided by the queue control supervisor has been preassigned at least one unique queue control block. When the task requests permission to process (that is, to be performed), it passes the name of the control block to be used as

part of the message. Note that each of these control blocks has a length of two. This is very important in the queue control logic.

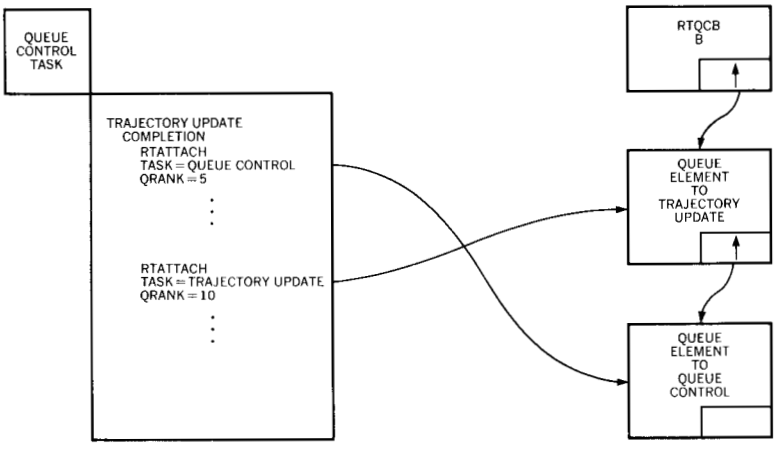
Figure 3 also shows two indicators that are maintained by the queue control supervisor. The first (PTP status) is a bit pattern which uniquely identifies each task that has been granted permission to process. The second indicator is the serial-processing status, use of which is shown in the following examples. The complexity of the examples increases according to their order of presentation.

Example 1

In this first example, Block A controls the queue chain through which all queue elements that cause a trajectory update must be routed. When the queue control supervisor receives a trajectory-update request, it immediately disables Block A so no more requests can be serviced from its queue chain. The queue control supervisor then uses the RTATTACH macroinstruction on itself through Block B, as shown in Figure 4, indicating that the trajectory update is complete; that is, it creates a queue element for itself in the queue chain controlled by Block B. This may seem strange since the trajectory-update control program has not yet been called. However, the newly created queue element for the queue control supervisor cannot be serviced since the task is currently active (RTOS Rule 2). Now, the queue control supervisor applies the RTATTACH macroinstruction to the trajectory-update control program, again creating a queue element in the queue chain controlled by Block B. However, the second queue element is assigned a higher queue rank and thus is placed ahead of the first queue element in the queue chain as in Figure 4.

The queue chain controlled by Block B contains two queue elements which is the maximum length of the chain. Whether or not a queue element can be serviced from this queue chain depends on whether Block B is currently enabled or disabled (RTOS Rule 3). When the

Figure 4 Trajectory-update completion



status of Block B becomes enabled, the trajectory-update queue element will be serviced first because it was assigned a higher queue rank. In the meantime, the queue control supervisor has completed its work and is inactive. It is eligible for further requests and may receive requests from other queue control block chains. It cannot, however, receive the queue element that it has waiting in the Block B queue chain because a queue element from that chain is currently active (RTOS Rule 1). When the trajectory update completes its function and its task becomes inactive, the trajectory-update completion queue element in the Block B queue chain can be serviced. When the completion queue element is received by the queue control supervisor, it enables Block A, which makes any additional trajectory-update requests eligible for servicing.

Example 2

This second example considers Block C, which is used to chain the queue elements for those tasks that cannot be performed when a trajectory update is in process, in comparison to Example 1 which was oversimplified for illustrative purposes. Actually, when the queue control supervisor disables Block A to prevent further trajectory-update requests, it also disables Block C to prevent the servicing of any queue elements from its queue chain.

As mentioned earlier, passing requests through the queue control supervisor is known as requesting permission to process. Since the permission-to-process concept will be used several times in this and subsequent examples, some abbreviations are necessary. A queue element passed to the queue control supervisor requesting permission to process is identified as an RPTP queue element. The queue element passed back to the requesting task is a "grant-permission-to-process" element, identified as a GPTP queue element.

The permission-to-process procedure is as follows:

- When a task in this group receives a request from some other source, it generates an RPTP queue element to the queue control supervisor through Block C.
- The requesting task then returns to the operating system.
- Block C is disabled if a trajectory update is in process. If so, the block is enabled by the queue control supervisor when the trajectory update completes its function.
- When the queue control supervisor finally receives the RPTP queue element, it disables Block B. Remember from Example 1 that Block B is used for trajectory-update queue elements.
- In Figure 3, each of the control blocks numbered 1 to n has a length of two. Each of these blocks is assigned to a unique independent task to be used for the GPTP queue elements.
- The queue control supervisor first generates a queue element to itself through the queue control block assigned to the requesting task. This queue element cannot be serviced since its requested task is still busy.

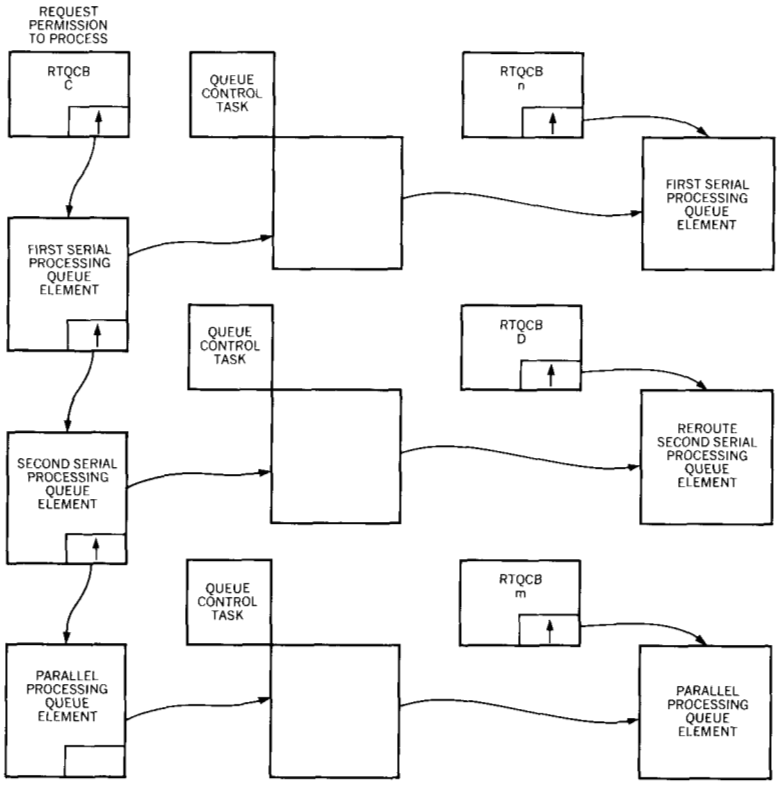
- The queue control supervisor then generates the GPTP queue element through the same control block with a higher queue rank. This means that the GPTP queue element will be serviced before the queue element to the queue control supervisor. Furthermore, the second queue element cannot be serviced until the task receiving the GPTP queue element is completed and returns to the operating system.
- In addition to generating the necessary queue elements, the queue control supervisor sets a unique bit in its task resource table to indicate that the particular task has been granted permission to process.
- When the task receiving the GPTP queue element is completed, the completion queue element is passed to the queue control supervisor. The indicator for the completed task is then turned off.
- There can be many GPTP queue elements outstanding at the same time. Each has a unique bit in the task resource table. As each completion queue element is received by the queue control supervisor, the appropriate indicator is turned off. The queue control supervisor then checks to see if all indicators are off, and if so, Block B is enabled to allow any waiting trajectory-update request to be serviced.

The completion queue logic for the permission-to-process requests is identical to the trajectory-update completion logic mentioned in Example 1 and outlined in Figure 4. Again, the completion queue element is generated first, but it cannot be serviced because the specified task is the queue control supervisor that is still active. The queue element to grant permission to process is then generated and placed in the same queue chain as the completion queue element, but is assigned a higher queue rank. This ordering requires that the queue element granting permission to process will be serviced first, and that the completion queue element must wait until the task receiving the first queue element has been completed. The completion queue element then returns to the queue control supervisor.

The completion logic is extremely valuable since the queue control supervisor maintains complete control over queuing logic. It does not have to depend upon a return queue element being generated by the task that was granted permission to process.

Example 3 As mentioned earlier, some of the tasks in the permission-to-process group cannot be carried out simultaneously. These tasks have been identified and are performed in a serial-processing stream. Here in this third example, as in Example 2, an RPTP queue element is received by the queue control supervisor through control block C as shown in Figure 5. If the task requesting permission to process is not in the serial class, it is performed exactly as illustrated in Example 2. If it is in the serial class, the serial-processing indicator is checked to see if a task from the serial-processing group is currently

Figure 5 Serial processing



active. If not, control block D is disabled, the serial-processing indicator is turned on, and the RPTP queue element is processed the same as a nonserial queue element. If, however, the serial-processing indicator is on and a queue element for another task in the serial group is received, the queue element is rerouted to the queue control supervisor through Block D, which will be disabled. When a completion queue element for a serial GPTP task is received by the queue control supervisor, the serial-processing indicator is turned off, and Block D is enabled to release queue elements that might have been rerouted.

Example 4

Because the permission-to-process logic had not yet been discussed, the first example was oversimplified. Remember that when a GPTP queue element was dispatched, Block B was disabled. However, Block A was not disabled so a trajectory-update request could be passed to the queue control supervisor as in this fourth example. When the queue control supervisor receives the trajectory-update request through Block A, it immediately disables Block C to prevent any more RPTP queue elements from being passed to it. However, there may be GPTP queue elements already active. The queue control supervisor then disables Block A to prevent additional trajectory-

update requests and then reroutes the trajectory-update request to itself through Block B. If there are GPTP queue elements still active, Block B will be disabled and will remain so until all completion queue elements have returned to the queue control supervisor. If there are no GPTP queue elements active, Block B will be enabled, and the trajectory-update request will return to the queue control supervisor as soon as it is inactive. The trajectory update will then be processed as outlined in Example 1; that is, first the associated completion queue element and then the trajectory-update queue element will be created in the queue chain controlled by Block B.

There are other techniques that can be used by an independent task to prevent other independent tasks from interfering with its data usage. One such technique is to lock data tables while they are being used. This prevents other tasks from accessing the data until the lock is removed. For example, the trajectory-update control program could lock the ephemeris table until all of the ephemeris is generated. Similarly, a user of the ephemeris could lock the ephemeris table while it is accessing the ephemeris.

A data-table lock is useful if the table is to be locked for a brief period of time. However, if a data table is used by several independent tasks, and if one task locks the table for a long period of time, the other tasks are required to wait in main storage for the lock to be removed. Another disadvantage of locking data tables is that it denies simultaneous access of data tables by independent tasks that normally can be performed simultaneously.

Summary

The trajectory control programs ensure the integrity and consistency of all trajectory data that is used by the flight controllers on space flights. This means that the ephemeris, which is used for many of the calculations of trajectory data, must be updated in a timely manner with no interference from other tasks. It also means that users of the ephemeris must be protected from trajectory updates that might make their data inconsistent. The objective is achieved by an elaborate queue control process that continually enables and disables various real-time queue control blocks to regulate the independent-task processing.

ACKNOWLEDGMENT

The technique of generating a completion queue element for a task *prior* to actually invoking that task was developed by James Summers.

REFERENCES

1. J. L. Johnstone, "RTOS—Extending OS/360 for real-time space flight control," *AFIPS Conference Proceedings, Spring Joint Computer Conference* **34,** 15–27 (1969).

2. F. Ditto, "Nonlinear trajectory estimation in real time for Project Gemini," *Proceedings of the Real-Time Systems Seminar,* Houston, Texas, International Business Machines Corporation, 1–2 (November 1966).
3. J. W. Havender, "Avoiding deadlock in multitasking systems," *IBM Systems Journal* **7,** No. 2, 74–84 (1968).
4. B. I. Witt, "The functional structure of OS/360, Part II, Job and task management," *IBM Systems Journal* **5,** No. 1, 12–29 (1966).
5. R. Hoffman, "Managing the design, development, and implementation of large-scale generalized real-time systems," *Proceedings of the Real-Time Systems Seminar,* Houston, Texas, International Business Machines Corporation, 13–14 (November 1966).
6. J. H. Mueller, "Philosophy of the RTCC control programs," *ibid.,* 32–40.
7. W. I. Stanley and H. F. Hertel, "Statistics gathering and simulation for the Apollo real-time operating system," *IBM Systems Journal* **7,** No. 2, 85–102 (1968).