Spring 1988

A Simulation Study of a Space-Borne Optical Disk Mass Memory System

Kenny G. Chen
Old Dominion University

A SIMULATION STUDY OF A SPACE-BORNE OPTICAL DISK MASS MEMORY SYSTEM

by

Kenny G. Chen
B.S. May 1986, Old Dominion University

A Thesis Submitted to the Faculty of
Old Dominion University in Partial Fulfillment
of the Requirements for the Degree of

MASTER OF ENGINEERING

OLD DOMINION UNIVERSITY
April, 1988

Approved by:

_____
David L. Livingston

_____

_____

_____

ABSTRACT

A SIMULATION STUDY OF A SPACE-BORNE
OPTICAL DISK MASS MEMORY SYSTEM

Kenny G. Chen
Old Dominion University, 1988
Director: Dr. David L. Livingston

Issues concerning space-borne applications of an optical disk mass memory system (ODMMS) are investigated through computer simulation. The simulation model is developed according to a current description of the ODMMS with certain application constraints. The results are examined in terms of system modularity, multi-user data rate buffering, disk module access, and read/write file management. The conditions and requirements for future disk controller designs are indicated in the simulation results.

ACKNOWLEDGEMENT

I would like to acknowledge Dr. D. L. Livingston for his persistence and inspiration and Dr. T. A. Shull for his technical help. This research was sponsored by the National Aeronautics and Space Administration under contract NAS1-17993-62.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

In future space-borne applications, there will be increasing demands for data storage and retrieval systems exhibiting high density, large capacity, fast access, long storage life, and low bit-error rates. Magnetic recording technology is not adequate in providing these capabilities. In order to achieve the projected high density, large storage capacity, and other operational requirements, newly available erasable laser recording technology is currently being integrated into conventional multi-disk storage systems. The result is a new type of disk storage, called an optical disk mass memory system (ODMMS).

The reason for selecting a multi-disk form is that the systems will be able to accept continuous data at various rates and play back at a predetermined rate, or to simultaneously accept input data on some of its surfaces while reading previously recorded data from other surfaces. Earlier studies on optical memory system architectures [1], [2], [3] indicate that the architecture of the ODMMS will have many similarities to conventional disk storage; however, the requirements and constraints imposed by space-borne applications will add more complexity and special features into the memory system design. Issues such

as I/O rate buffering, disk module access, read/write data format, system synchronization, and system modularity will all have a great impact on the structure of the storage systems. The purpose of this study is to investigate the overall performance of the ODMMS within these constraints and suggest possible improvements.

One way to estimate the performance of a mass memory architecture is to build a hardware model which is proportional to the actual size of the ODMMS. The model can be run under various test conditions to provide an insight into the system operation. However, the idea is unrealistic due to the unavailability of technologies to be employed in the system design and the expensive cost. An alternative cost-effective approach is through computer simulation.

Simulation has been widely used in computer system evaluation. Examples include main memory partitioning, CPU scheduling in a multiprogramming system, computer system design optimization, and resource management in a time-sharing system [4]. The case in this study is an example of memory management and I/O service under a typical workload. Running the computer model by using sampled data from projected real-time operation, we are able to obtain information about the system at any time instant during its simulated operation. Thus, the real-time performance of the system can be analyzed from this information.

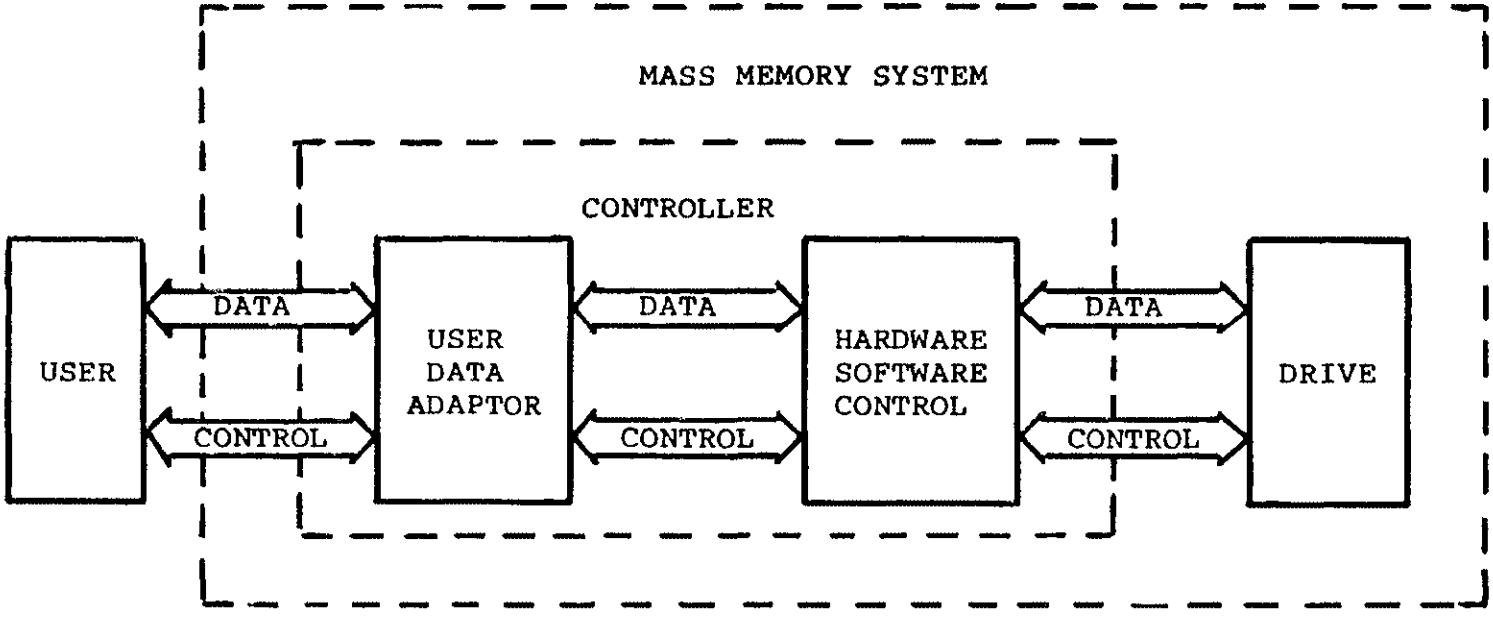


FIGURE 2.1 HIGH LEVEL ODMMS ARCHITECTURE

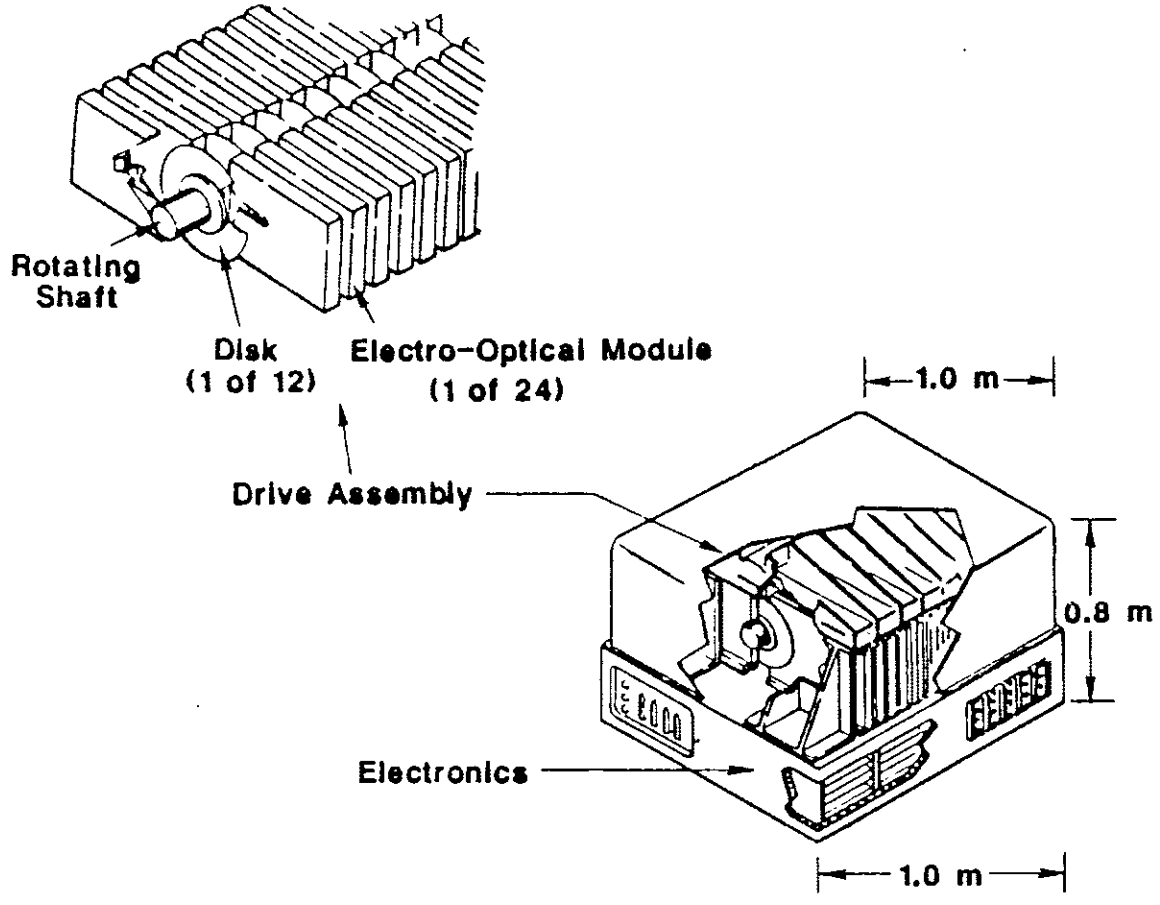


FIGURE 2.2 CONCEPTUAL CONFIGURATION OF THE ODMMS
REPRODUCED WITH PERMISSION FROM SHULL [5]

read/write head on each disk surface. To avoid possible confusion, we define a track as consisting of eight data sub-tracks and one permanent pilot sub-track as illustrated in Figure 2.3 [6]. The pilot sub-track contains track number identification data, which could be processed by the controller at a separate low rate, to provide optical head control information. Instead of using a conventional concentric track format, it has been changed to a spiral track format. As will be discussed in the next chapter, the spiral track format requires less amount of disk access time than the concentric track format in processing continuous and long stream type of data. Solid state lasers, implemented in a nine-diode array positioned on a read/write head, can read or write eight data sub-tracks simultaneously to achieve a desired surface I/O rate of 150 megabits per second. The projected data storage capacity (24 surfaces) is on the order of 10^{12} bits [5].

The magneto-optical recording process used for disk-write and disk-read is illustrated in Figure 2.4 [6]. At normal ambient temperature, the bias magnetic field has no effect on disk medium. An increase of temperature caused by a focused laser beam results in a local reversal of magnetization; information is thus written on the medium surface. The read operation is performed via a phenomenon known as the Kerr effect, i.e., the polarization angle of the reflected laser beam is rotated according to the orientation of the magnetic field on the medium. To erase

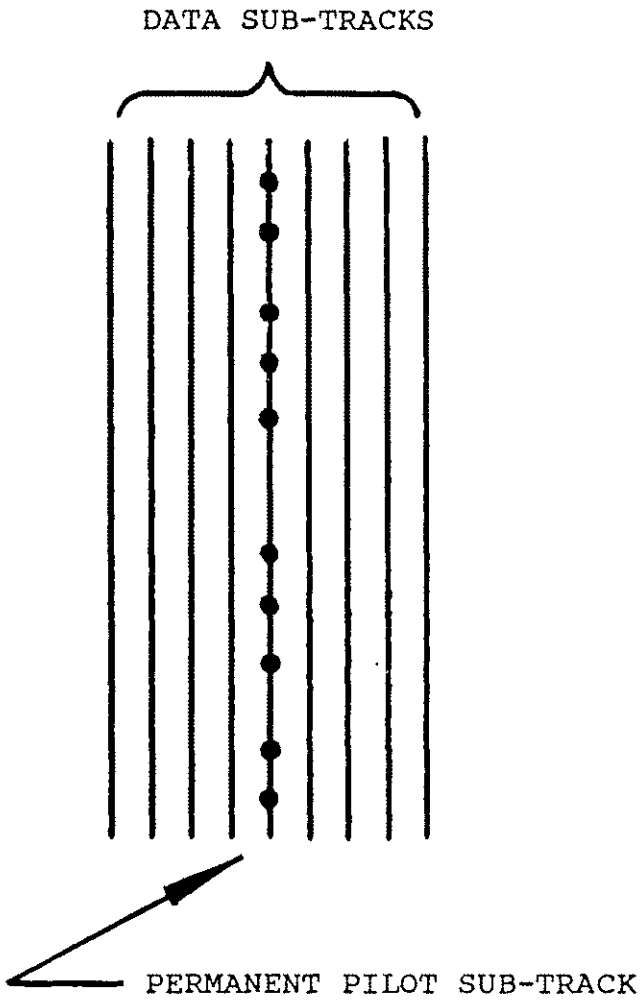


FIGURE 2.3 ILLUSTRATION OF DISK TRACK

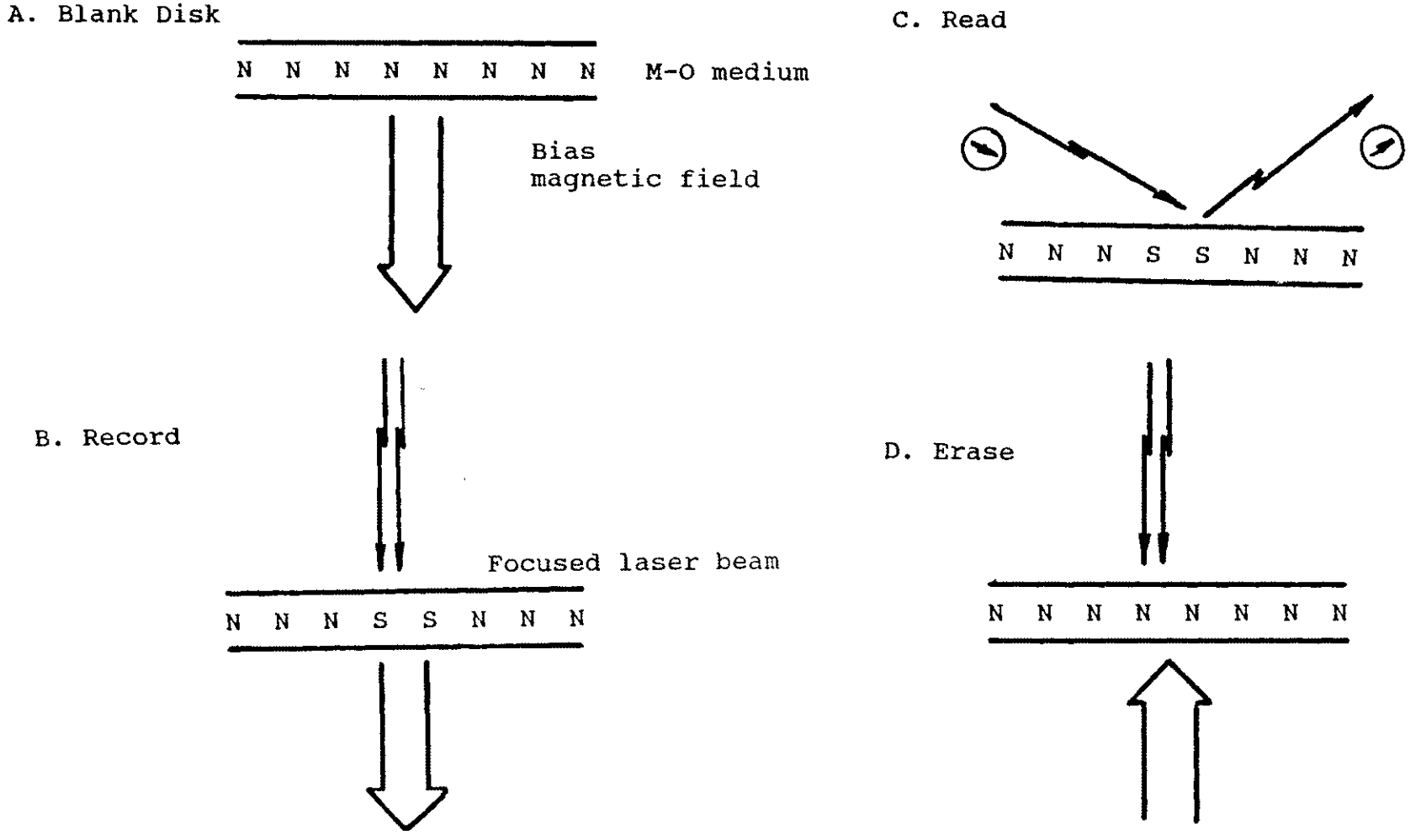


FIGURE 2.4 MAGNETO-OPTIC RECORDING PROCESS

the information, or to recover the original uniform magnetic pattern, the bias field is reversed and the vicinity temperature is raised by the laser. The erase operation has to be executed before the medium can be used for further recording.

As shown in later simulation results, the dynamic capacity of the ODMMS is limited primarily by the transmission channels available for reading data from the memory. It is desired to read as much data off the disk surfaces as possible during the limited transmission time; meanwhile, the disk system can accept more input data. This will increase the dynamic capacity of the ODMMS. However, the concurrence of erase and read operations on a same disk surface would cause the read/write heads to move back and forth and increase the disk access time for the read operation. The actual time available for data reading would be reduced. To avoid the access conflict, a proper scheduling of erase operations is required. We assume that an erase operation can occur only after all the data on a disk surface has been read. The erase operation takes the same amount of time as a disk read.

A possible space-borne application of the ODMMS is aboard an Earth Observing System (EOS), where the memory system will function as a data reservoir [5]. This is illustrated in Figure 2.5. The EOS is used to collect a large quantity of image data and transmit it down to earth stations through the Tracking and Data Relay Satellite

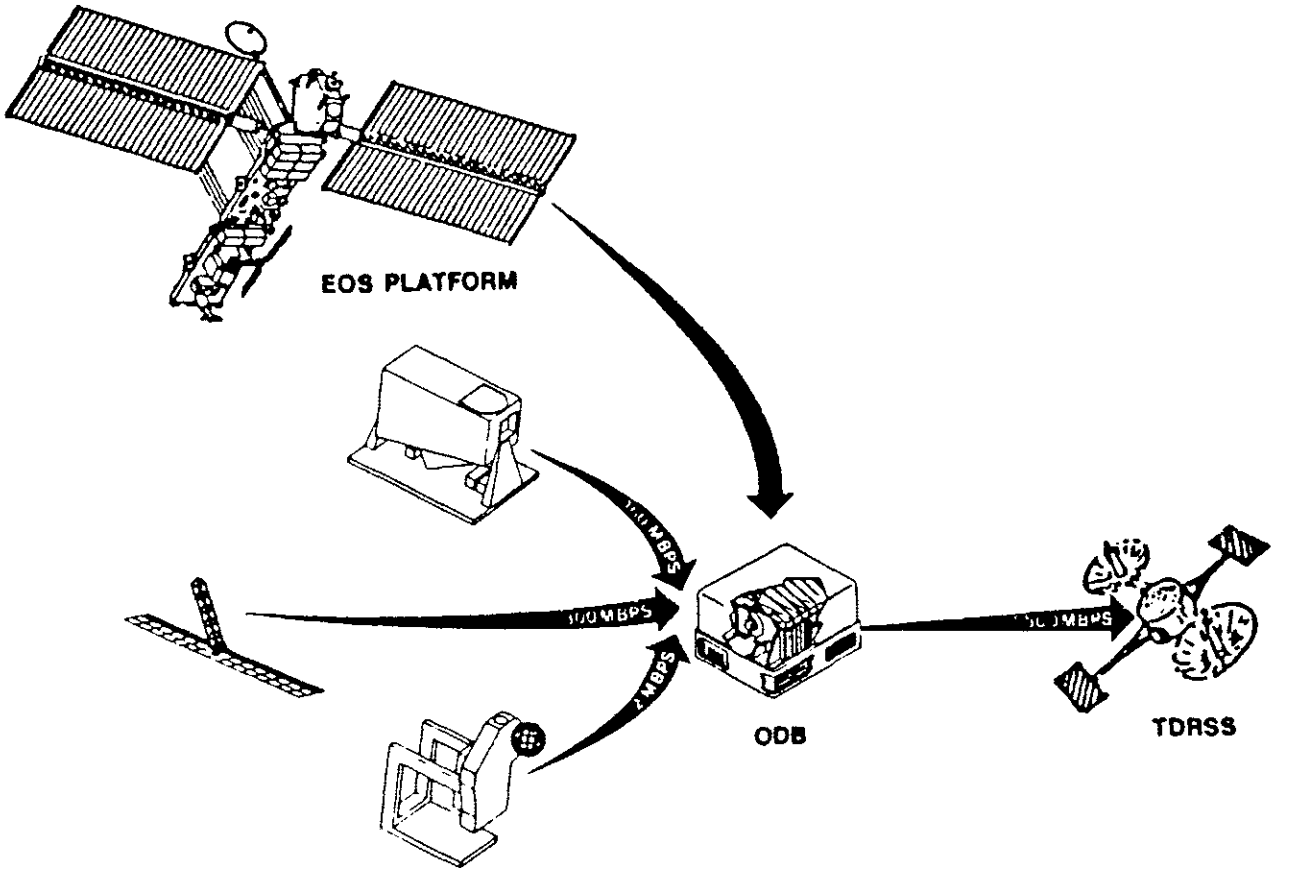


FIGURE 2.5 ODMMS ABOARD EOS PLATFORM
REPRODUCED WITH PERMISSION FROM SHULL [5]

System (TDRSS). The EOS moves about the earth orbit and transmits data only when it is in line of sight with the TDRSS, which is in a geostationary orbit. During the time when the TDRSS is out of transmission sight, data from different sources have to be temporarily stored in the memory system. A scenario provided by NASA Langley and used in this study makes the available time for data transmission using the TDRSS, 33 minutes out of an 100-minute EOS orbit time. If the simulation data is run over an eight-hour period, the TDRSS will be available approximately four times, which accounts for a total data-read time of 132 minutes. This puts a significant constraint on the dynamic storage capacity of the ODMMS.

So far, the discussion of the ODMMS architectures has been limited to the conceptual level. In estimating the performance of a memory system that does not yet exist, Lucas [7] points out that the most potentially powerful and flexible evaluation techniques are computer simulations, which provide a testing ground for, and insight into, the functioning of the system. This has been verified through various simulation studies, and their results are consistent with later actual system measurements [7]. In general, there are two basic types of simulation that have been used to evaluate computer systems. One type of simulation uses empirically-derived data which are manipulated to correspond with a specific system configuration and workload; the other type models the

actual operations of a system for which a schedule of events is maintained. Probability distributions are generally used to describe the system performance. The boundaries between these two types of modeling are often overlapping. Because systems differ so much in their organization and operation, there is no standard way of modeling a complex system for either type of simulation. A common practice is to design a simulation model following the individual needs and particular structure of a system.

Since a simulation model is never a complete representation of the real system, we have to validate the model after it has been established. Several standard validation procedures, such as verifying the program, comparing model data with real system data, and sensitivity analysis, are commonly used in practice [8]. The purpose of verifying the program is to determine whether or not the program implements the model as intended. This is simply part of what is commonly known as the "program-debugging" process.

Comparing model data with real system data is one of the primary approaches used in the model-validation process. The usual condition for this type of testing is that both the test data from the real system and the model-generated data will be stochastic variables. Thus, the problem of comparing this data corresponds to so-called two-sample statistical testing. A further discussion on this procedure is presented and illustrated by Payne [8].

Sensitivity analysis views the model as an input-output process. The basic idea is to vary input variables to the model, by using incremental changes, and observe the incremental changes in the output variables. The ratio of these changes is referred to as the sensitivity of the outputs to the specified inputs. It is reasonable for the sensitivity of a model to be approximately constant for small changes in the input variables. In this study, unfortunately, no real data from the ODMMS can be measured to compare with the model-generated data. Thus, we have to mainly rely on the two validation procedures: program verification and sensitivity analysis.

Through the above discussion of the configurations of optical disk mass memory systems and possible applications, we intend to provide an overall picture of the systems. This leads us to further investigate the design considerations of such systems.

CHAPTER 3
DISCUSSION OF ISSUES

In the use of disk storage memory, the disk access time is always one of the main drawbacks. As defined by Baer [9], the disk access time T, in the case of movable head disks, is of the form:

$$T = Ts + Tr + Tt ,$$

where Tr is the rotation time, Tt is the data transfer time, and Ts is the seek time, or time required to reach the right track. The rotation time and data transfer time are determined by the physical constraints of the systems, and are not related to the disk format. However, the amount of track seek time can be reduced by changing a conventional concentric track format to a spiral one. In the case of concentric track format, extra seek time is required when heads move from one track to the next. The exact amount of this time is determined by the mechanical structure of the drive and varies from system to system, usually between 10 to 20 milliseconds for magnetic disk systems [10]. If thousands of tracks need to be accessed continuously, the amount of delay could reach the order of tens of seconds. Under the circumstances, a huge data buffer would be required to temporarily store the accumulated data. This situation is more likely to occur

when a long stream of high-rate data needs to be retrieved or written onto disk surfaces. With the spiral track format, heads can move from track to track in a smooth manner so that the delay between the consecutive tracks is actually eliminated.

Tracks, in a conventional disk storage system, are further divided into sectors. The size of a sector is determined by the type of data the memory system is accommodating, i.e., small sectors work best with small data files but require more overhead information; and large sectors require less overhead but leave large portions of sectors unused when data files are small. The question is often resolved through simulation. Conceptually, the ODMMS uses one track (one complete revolution of the disk) as a sector. As shown later in the simulation results, this is a reasonable choice for the long stream type of image data processed by the EOS platform.

In addition to the physical configuration of the ODMMS, its environmental and operational requirements also need to be considered. In a space platform, the mass memory system usually works over extended periods with infrequent visitation for maintenance. This requires that the memory system be able to detect failures and, if possible, reconfigure or map out failed elements. Reconfiguration may reduce the system capacity, resulting in a degraded performance [5]. This suggests the idea of disk modularity. By partitioning the memory drive into several

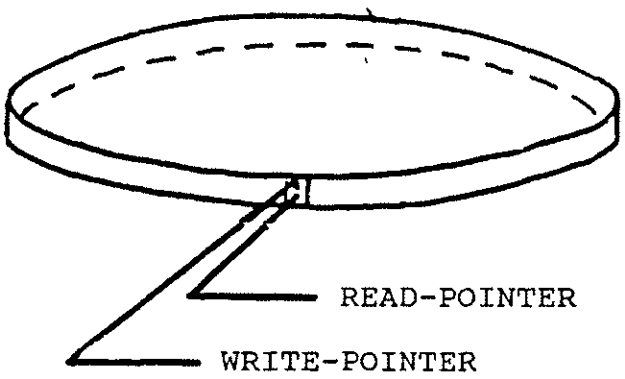
independent modules, a malfunctioned module can be easily mapped out, or be replaced by a spare module. We consider one double-sided disk as a module. There are three main reasons behind this consideration. First, it combines two disk surface I/O rates of 150 megabits per second together to produce a net modular I/O rate of 300 megabits per second, which matches the projected maximum user data rate. The condition of modular I/O rate greater than the input user rate ensures that data will not overflow into a circular buffer, which is defined later in this chapter. Second, it requires a minimum size circular buffer. Third, a module containing a small number of disk surfaces has less effect on the overall dynamic capacity of the memory system after a malfunctioned disk module is mapped out. This is justified in the simulation results.

In the applications to the Earth Observing System described in Chapter 2, incoming data from different sources is usually large quantities of image data, contained in a long stream file at a relatively high rate. Any transmission delay, including hand-shaking and disk access, could imply the loss of information unless a temporary buffer is used. On the other hand, physical constraints on the ODMMS require the size of this buffer be kept at a minimum. To reduce the amount of delay, it becomes the responsibility of users to notify the disk controller, so the controller has enough time to complete the hand-shaking process and place the optical heads on the

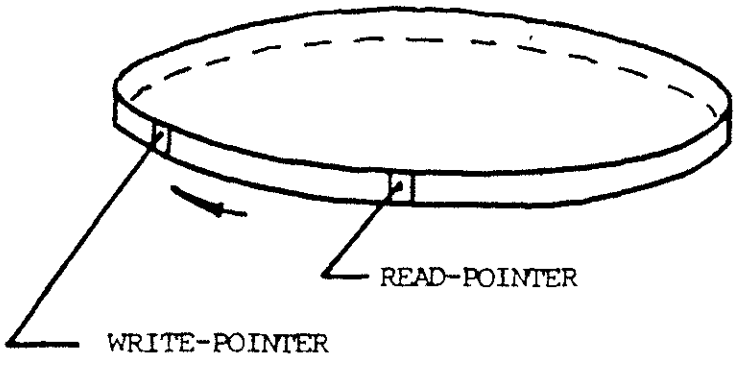
right tracks before an actual I/O operation takes place. In any case, the disk rotation delay, which occurs when writing a new data file or writing data on a blank disk module, cannot be eliminated. We will consider disk rotation delay as the only disk access time for the write operation under the condition of the controller being notified beforehand.

The actual amount of rotation delay (usually a random parameter) is determined by the optical head position on a track. We assume that the head position is represented by a uniform distribution. The maximum delay is then one disk rotation with an average of a half disk rotation. Certainly other types of distributions can also be used for analysis, but we will see that the difference is insignificant in the way the rotation delay is actually handled.

From the view of system modularity, the ODMMS should maintain a modular I/O rate of 300 megabits per second, providing that the disk rotation rate is fixed. Different user data rates have to be converted to the rate of 300 megabits per second. A circular buffer which is twice the size of one modular track (two tracks) is suggested to perform the rate conversion, as shown in Figure 3.1. The circular buffer uses two pointers; one for the read operation and one for the write operation. Initially the pointers address the same position. Incoming user data is first written into the buffer and causes the write-pointer to move away from the read-pointer. Once the difference

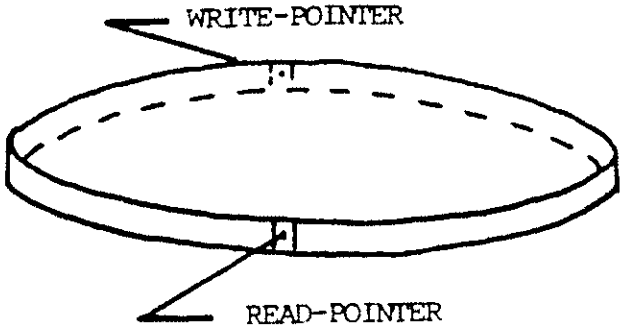


A. INITIAL POSITION OF READ-, WRITE-POINTERS.

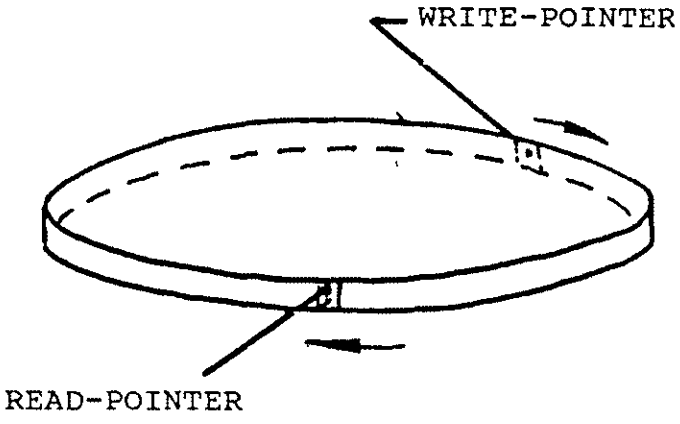


B. WRITE-POINTER MOVES AWAY FROM READ-POINTER.

FIGURE 3.1 CIRCULAR BUFFER CONCEPT



C. READ-POINTER STARTS MOVING WHEN THEY ARE 180° APART.



D. READ-, WRITE-POINTERS BOTH MOVE AROUND THE BUFFER.

FIGURE 3.1 (CONTINUED)

between two pointers indicates a full modular track size of data (equivalent to 16 data sub-tracks), information is written to the disk module after the read/write heads reach the beginning of a pilot sub-track. Meanwhile, the read-pointer moves toward the write-pointer. Since simultaneous I/O is allowed, the write-pointer keeps moving away from the read-pointer. At the end of a disk track, the controller again needs to examine the difference between two pointers. If the difference is larger than, or equal to, the size of one modular track, data is written on the next consecutive disk track without delay; otherwise, the optical heads will rotate above the next consecutive track waiting for the controller's go-ahead signal.

When the distance between two pointers reaches the size of one modular track, the optical heads may or may not be at the beginning of a pilot sub-track. That is, the heads are not synchronized with the tracks. Additional buffer space is required to postpone the data transmission until the optical heads again reach the beginning of the tracks. In the worst case, it requires a full modular track. For this purpose, the size of the circular buffer is maintained as two modular tracks. We eventually overcome part of the disk rotation delay problem.

We have seen that the disk module access conflict between the read and erase operations increases the access time. The other disk module access conflict caused by read and write operations can also have significant effects on

the access time. Because user data has to be stored immediately after being transmitted in order to keep the circular buffer to a minimum size, we assume that the write operation has a high priority over the read operation. To keep the seek time at a minimum, the optical heads should be maintained on the last tracks they served. This is not always the case if read and write operations are allowed on the same disk module. For example, while reading data off a disk module, a disk module write request could occur. The optical heads then have to be moved to the current available tracks. At the worst, the optical heads would step from the innermost track to the outermost track, or vice versa. If extra disk module access time has to be included, one way of handling this is to increase the circular buffer size. We are aware that a trade-off between the physical size of the buffers and the complexity of the controller always exists.

The file management of a memory system takes various forms depending on design choices. The foremost criteria are whether a file is to be partitioned into fixed or variable memory areas, and whether replacement algorithms are to be local; i.e., involving only the memory allocated to a particular file; or global, i.e., taking into account the history of all resident files [9]. A replacement algorithm is the policy used to determine which file stored earlier in the memory system needs to be erased when the full capacity of the memory is reached. The newly

available memory space is reserved for accepting new data files. Some of the most often used replacement algorithms are first-in, first-out (FIFO) replacement, first-in-not-used-first-out (FINUFO) replacement, and least-recently-used (LRU). Originally, these concepts were discussed in reference to paging memory systems. At this early stage of discussion we have no intentions to restrict the ODMMS to a paging organization. Thus, we consider files as basic memory blocks.

The FIFO policy stores files in a sequential order. The space allocated to the first file becomes the first one to be replaced. FINUFO keeps a FIFO queue as before, but it associates with each file in the queue a "use" tag which will be turned on when the corresponding file is referenced after its initial loading. When memory space is full, the new file will replace the first file not referenced in the queue. This algorithm has been used in MULTICS and the IBM System/360 Model 67 [9]. The disadvantages of FIFO and FINUFO are that they both have no reordering of files according to the number or recentness of references. This constraint is implemented in the LRU algorithm, which uses a stack to record the references instead of using a queue and reference tag. If a file is referenced most recently, it will be placed at the top of the stack, and the file at the bottom of stack is replaced first.

After examining the potential applications of the ODMMS, we conclude that the FIFO algorithm is the most

simple and effective memory organization technique as compared to the others. As indicated earlier, the primary function of the ODMMS is to serve as a temporary buffer. Data files stored earlier on a disk module are most likely to be read and transmitted down to an earth station for further processing. There is no potential need to reference a particular file. If such occasions do happen in the future, our circular buffer structure could become inappropriate or inadequate, and we have to look for other alternatives.

CHAPTER 4
SIMULATION MODEL STRUCTURE

The simulation model of the ODMMS needs to be implemented to represent the functions of memory storage capacities relative to a first-in-first-out read/write file management system under a given data-load. It also needs to reflect the condition of I/O requests and input data rate buffering. A proper representation of these operational relations is the type of simulation that uses empirically derived data which are manipulated to correspond to a specific system configuration and workload as previously mentioned in Chapter 2. There are no certain rules as to how to establish such a model. The actual modeling is dependent on our knowledge and earlier discussions on the ODMMS. The following conditions and assumptions are also integrated into the model establishment.

1. The disk module write has higher priority over the disk module read to eliminate the track-seek time.
2. Disk rotation delay is considered as the only disk access time for the disk module write. It has a uniform distribution with a maximum delay of one disk rotation and an average of a half-disk rotation.

3. Track-seek time is included for the disk module read. The heads are assumed to be positioned at the last tracks served from the previous operation.
4. Disk module reads and writes cannot occur on the same module. This eliminates a possible access conflict between these two operations.
5. A disk module is erased only after all the data on that module has been read. A disk module erase takes the same amount of time as the disk module read.

Figure 4.1 illustrates the overall program algorithm. The initialization routine at the beginning establishes system parameters, such as the number of disk modules, the starting time of the TDRSS for reading data from the disk memory system, the number of input files, etc. These are the basic elements which will affect the overall system performance once the test ground (simulation model) is established. The simulation is then started by the arrival time of the first input file. In Appendix A the program listing is presented, which consists of five subroutines: data file schedule, disk module write, disk module read, disk module access, and rate buffering. The main program corresponds to the initial routine in Figure 4.1, and also controls the program flow.

NASA Langley has provided a sample data distribution over an eight-hour time period from a typical EOS application, as shown in Figure 4.2. Figure 4.3 is a

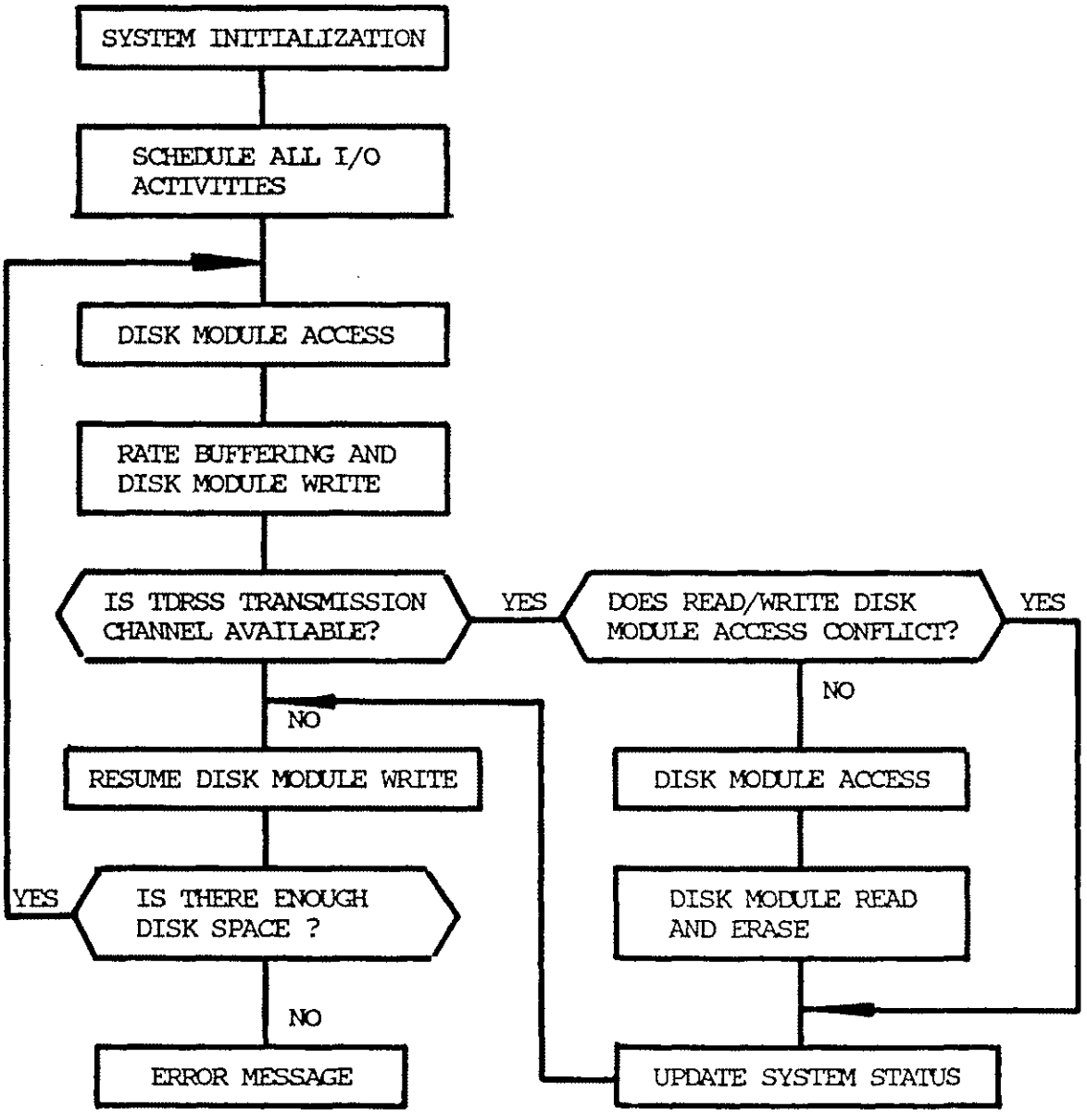


FIGURE 4.1 PROGRAM ALGORITHM

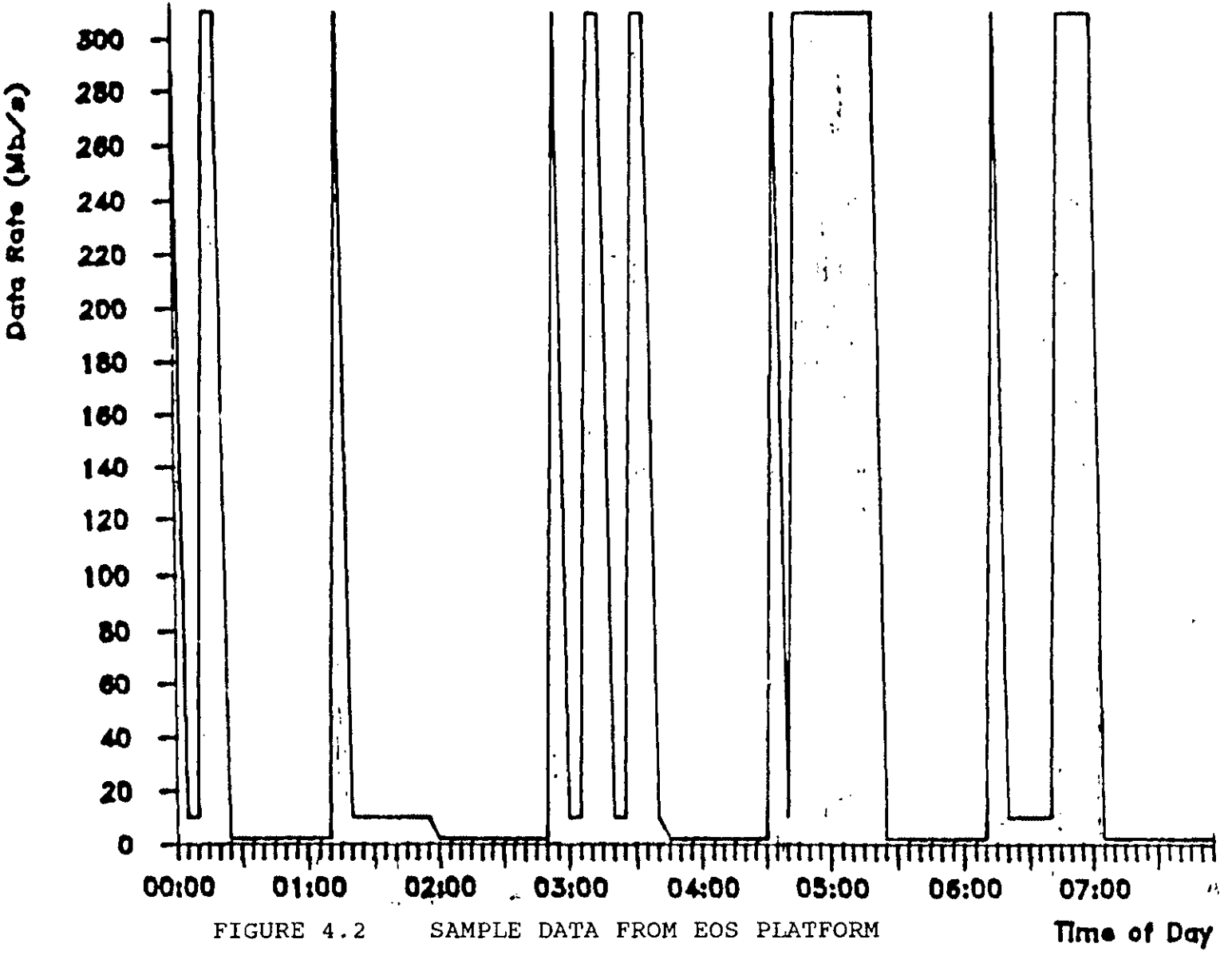


FIGURE 4.2 SAMPLE DATA FROM EOS PLATFORM

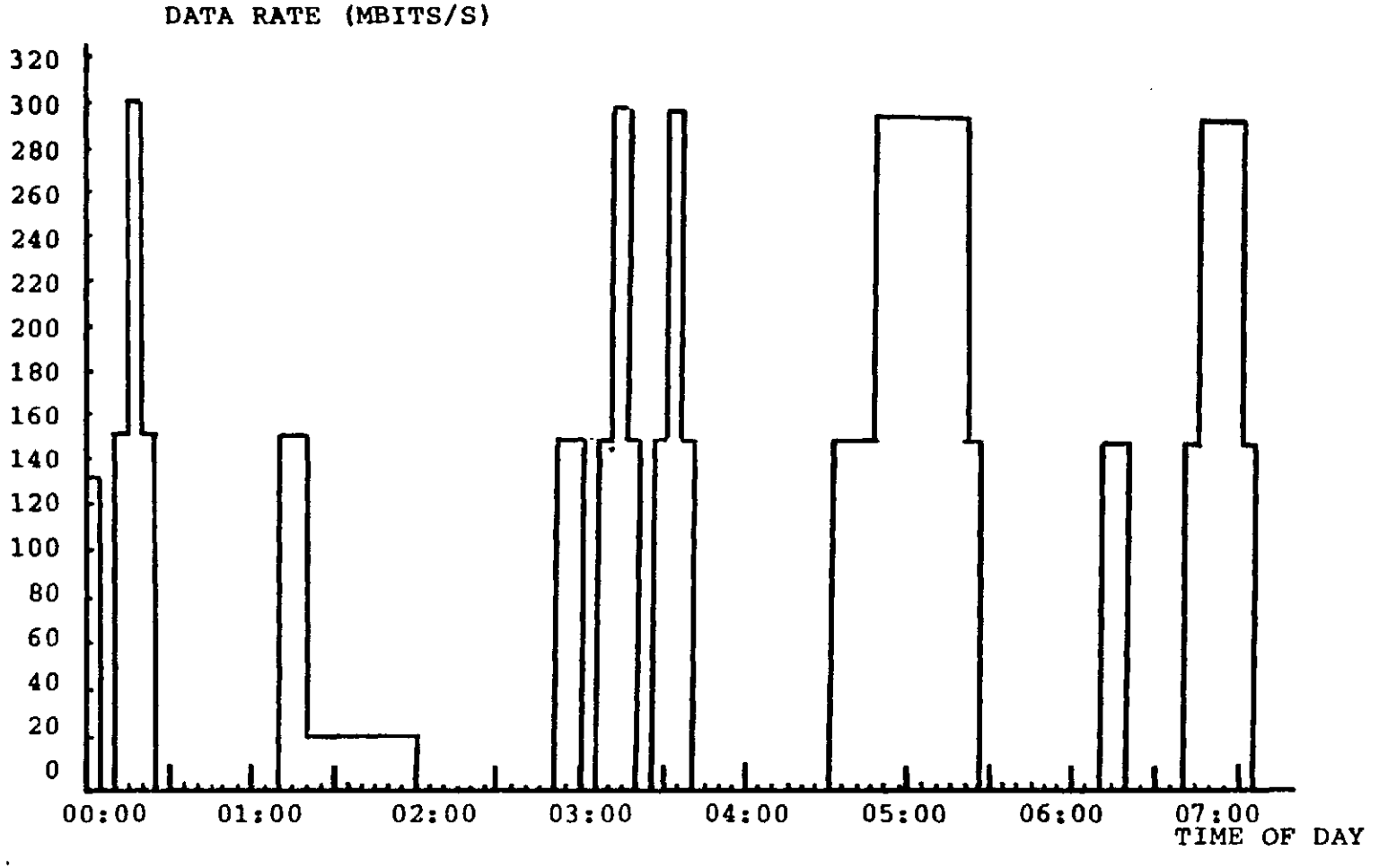


FIGURE 4.3 APPROXIMATION OF EOS DATA DISTRIBUTION

numerical interpretation of this data, where it is packed into approximate discrete data blocks to match the discrete simulation structure. By doing so we assume that the dynamic capacity of the disk systems depends on the amount of data transmitted during certain time periods, not the data rates, since different data rates can be converted to a constant rate through the circular buffers. Each constant rate data-block is considered as a file. Data rates below ten megabits per second have been either disregarded or included into previous high-rate data blocks; however, they could also be separated and considered as individual data files. This information is then used in the data-file schedule subroutine to schedule the occurrences of all the disk-module writes.

The overall simulation timing is controlled by the disk writing time. A separate read-time pointer is used to control the execution of disk reads. Upon the completion of writing a file or the condition of disk module full, the read-time pointer is checked against the schedule of the TDRSS channel availability. If the system is available for data transmission, the disk-module read subroutine is initiated and data written on previous disk modules can be read and erased; otherwise, the read-time pointer is elapsed to the point equal to the overall system time, which implies a disk module access conflict. The I/O operations are then resumed.

The detailed block diagrams of disk-module read and disk-module write are illustrated in Figures 4.4 and 4.5. At the beginning of a disk-module write, the current disk space availability is checked against the required data space. If the condition is false, a data overflow message is issued and the simulation halts. The disk-module access subroutine and rate buffering subroutine are executed next; one determines the current head position above the tracks and the disk access time, and the other determines the size of the circular buffer and describes rate buffering.

After each disk-module write or disk-module read, the overall system status is updated. We have used the scheme shown in Figure 4.6 to keep tracking the system status, where WR-pointer points to the current writing disk module, Length 1 is the number of blank disk modules at the right of WR-pointer, RD-pointer points the first blank module at the left of WR-pointer, and Length 0 is the number of blank modules at the right of RD-pointer. For example, in Figure 4.6, the values of WR-pointer, Length 1, RD-pointer, and Length 0 are equal to 6, 5, 0, and 4, respectively. The numbers also imply that module 6 is the current writing module, 5 blank modules are at the right of the WR-pointer, module 0 is the first blank module at the left of the WR-pointer, and module 4 is the current reading module. Pointers move from left to the right. When the furthermost right module is full, the values of each pointer are updated: WR-pointer becomes zero (writing module 0); Length

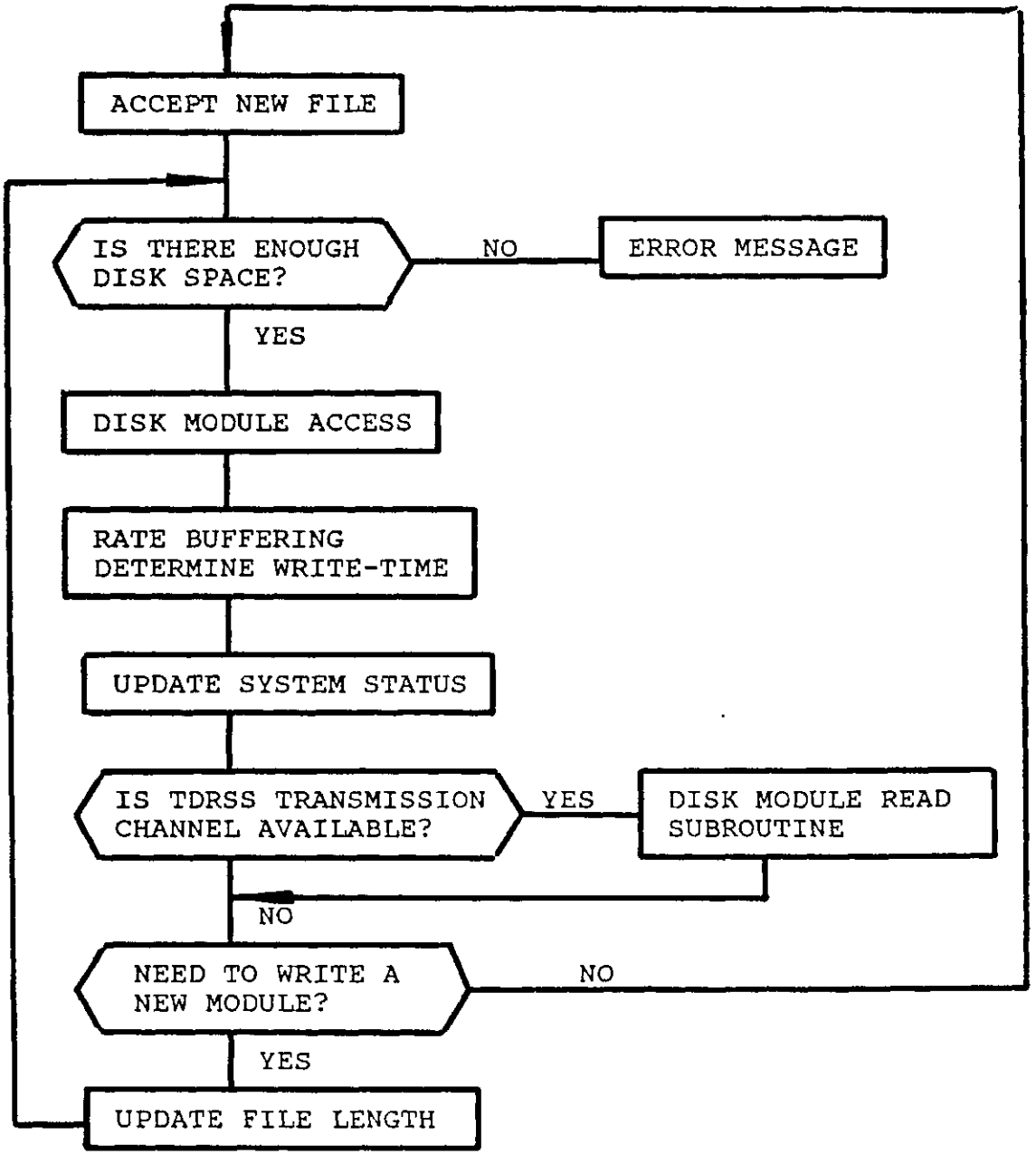


FIGURE 4.4 DISK MODULE WRITE ALGORITHM

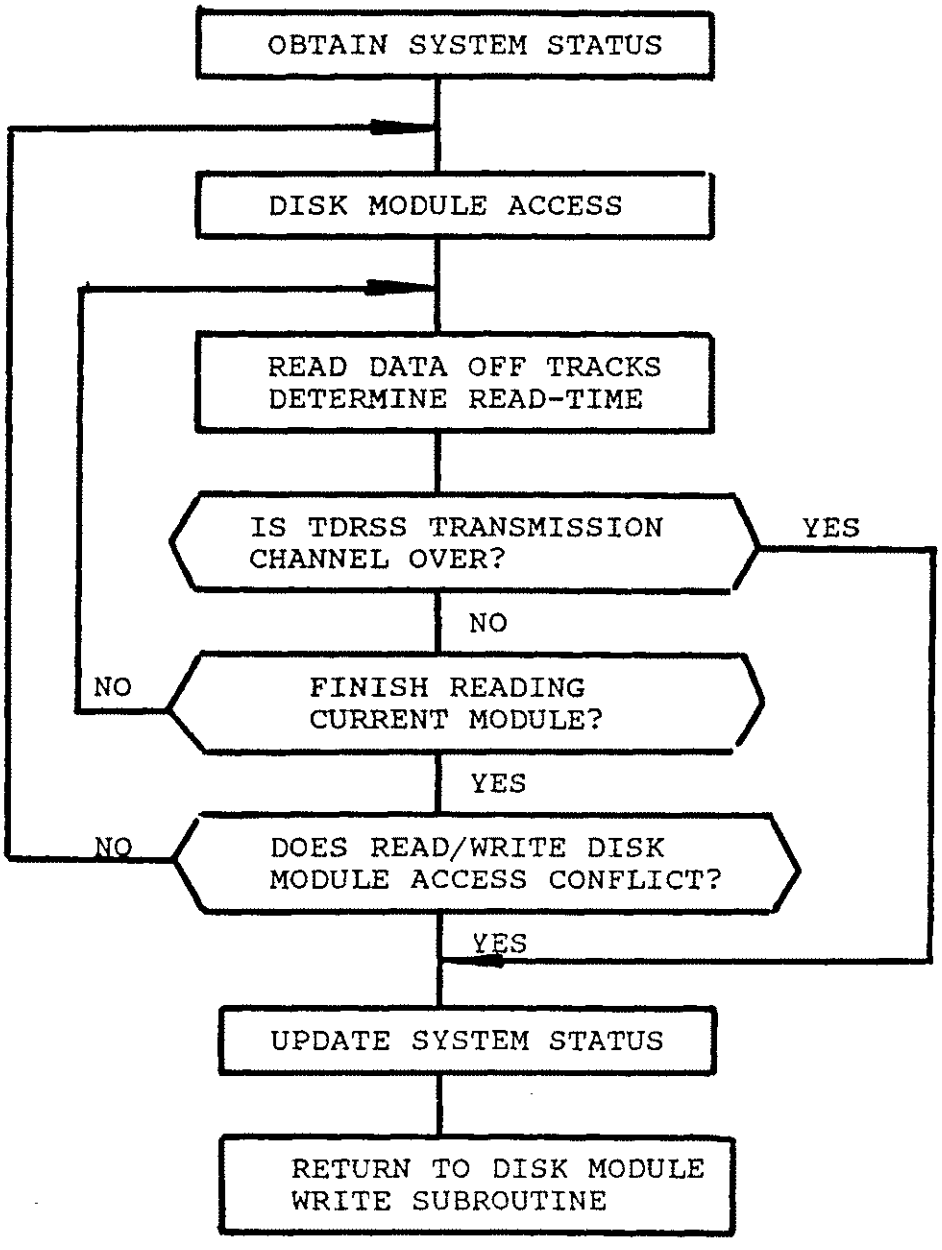


FIGURE 4.5 DISK MODULE READ ALGORITHM

1 takes the value of Length 0 minus one (number of blank modules at the right of WR-pointer); RD-pointer takes value of Length 0 (current reading module); and Length 0 becomes zero (no blank module at the right of RD-pointer).

Another scheme, used to define the status of blank tracks on a particular disk module, is shown in Figure 4.7. Free-track points to the first blank track, and #-track is the total number of blank tracks, assuming the innermost track is track 0. Initially, free-track is equal to zero and #-track is equal to the total number of tracks on one disk surface. The pointers rotate from in to out.

The block diagram of disk-module access subroutine is shown in Figure 4.8. The uniform distribution of head positions assumed earlier is produced by a pseudorandom number generator. By assumption, the disk-access time for disk-module reads also includes the track-seek time. Usually, the seek time is a non-linear function of the number of the tracks the heads have crossed as shown in Figure 4.9. For a small number of tracks crossed, the seek time consists of linear increments of the step-time from track to track. The slope of the curve decreases when the number of tracks traversed becomes larger. The upper bound of this function is close to the dashed line in Figure 4.9. Since it is reasonable to evaluate the system capacity as the access time reaching its upper bound, we assume that the access time consists of linear increments of the step-time of one millisecond from track to track.

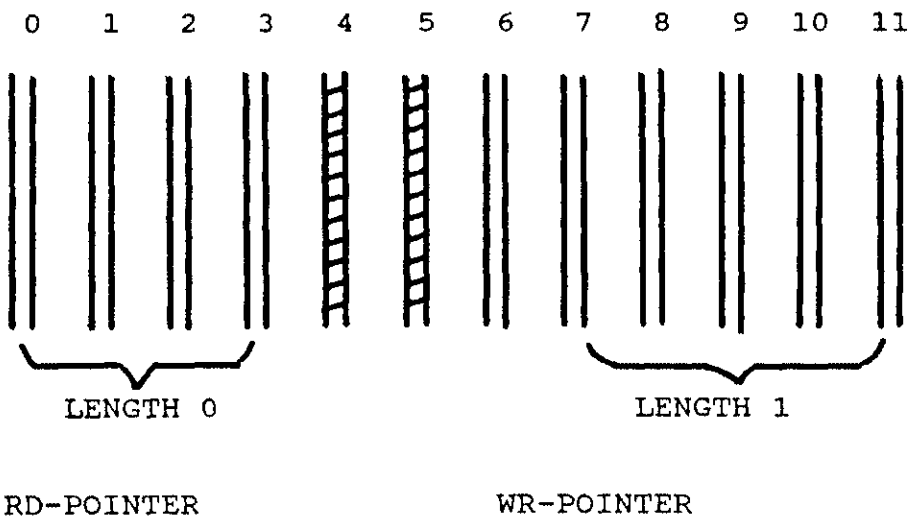


FIGURE 4.6 DISK MODULE STATUS SCHEME

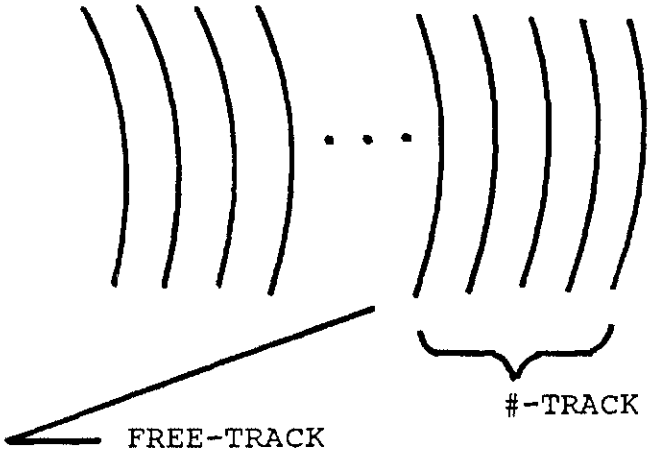


FIGURE 4.7 DISK SURFACE STATUS SCHEME

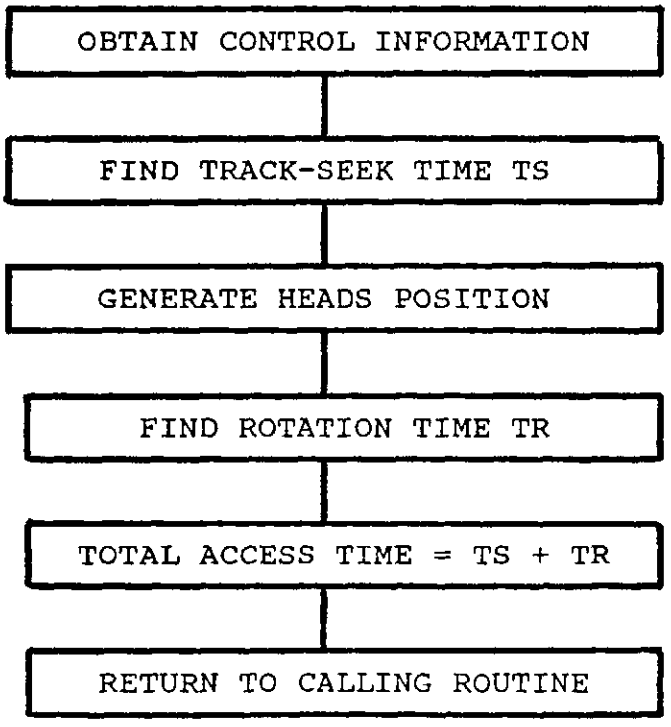


FIGURE 4.8 DISK MODULE ACCESS ALGORITHM

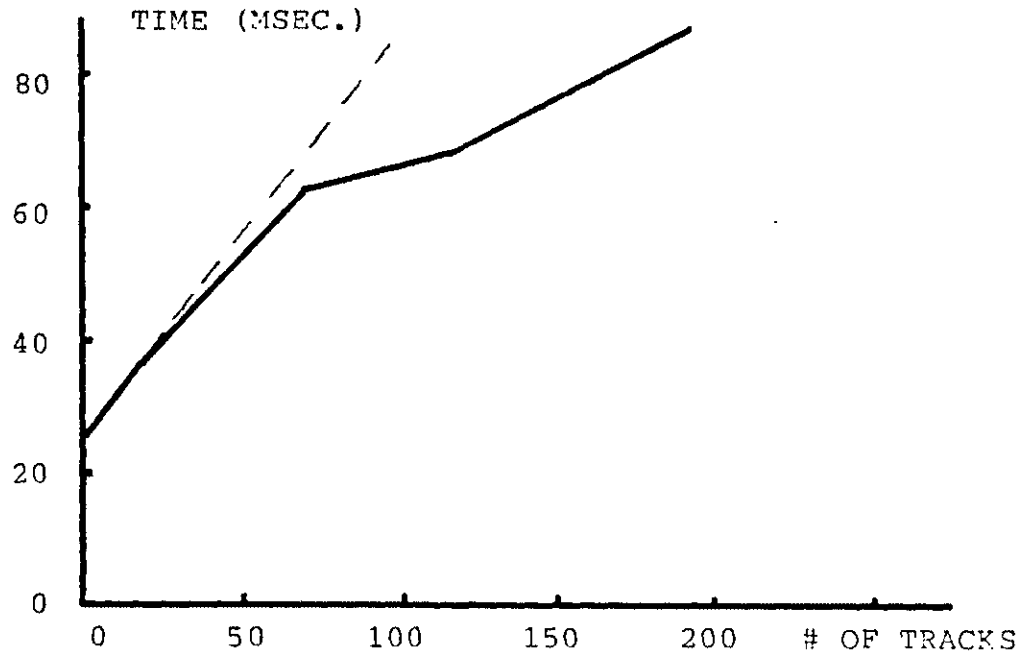


FIGURE 4.9 SEEK TIME VS. NUMBER OF TRACKS TRAVERSED

Figure 4.10 shows the block diagram of the rate buffering subroutine. The size of the buffer is determined by the amount of data accumulated due to disk-module access delay plus the current buffer status. The rate buffering is implemented according to the previously discussed circular buffer concepts. After each disk rotation, a separate pointer indicates the amount of data in the buffer, and a time increment equivalent to one disk rotation is added to the total disk-module write time. If the pointer exceeds the size of a modular track, the amount of data equivalent to one modular track is subtracted from the pointer. This process continues until the data transmission is completed. Since the input rate is less than or equal to the disk-module write rate, the write time is always behind the user data transmission time.

The level of detail at which the simulation operates is a macro-level, i.e., the effects of processing complete jobs are simulated. The simulation model is written in the C language [11] because it has the capability of hardware manipulation, an efficient utility for the possible micro-level simulation of the optical memory system. For instance, to determine the format of individual instructions, the number of bits needed as overhead, or the number and type of redundancy bits used in information coding, a bit-level simulation may be required.

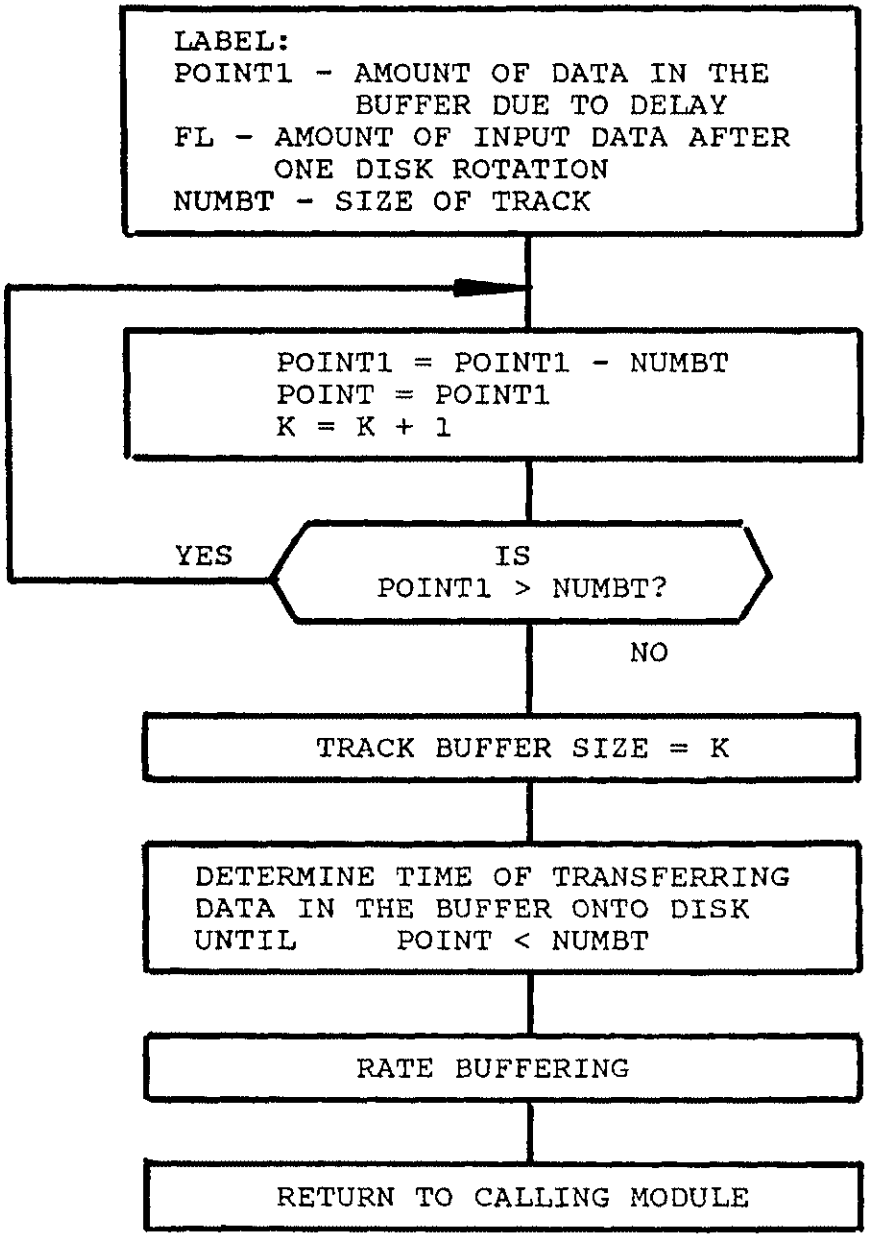


FIGURE 4.10 RATE BUFFERING ALGORITHM

CHAPTER 5
SIMULATION RESULTS

Before analyzing the simulation results, it is necessary to validate the simulation model. Efforts have been made in the program debugging to ensure an error-free program. As mentioned in Chapter 2, due to the lack of actual system measurements, the usual probability analysis procedure does not seem applicable. We have to rely on the sensitivity analysis. In Figure 5.1a, the function of system efficiency vs. time is shown for TDRSS transmission channel starts at 0, where the system efficiency E is defined as:

$$E = \frac{\text{Number of Disk Modules Used}}{\text{Total Number of Modules}} \times 100\ \% .$$

If considering the number of disk modules as input to the model and the system efficiency as output, from the point of sensitivity analysis, we expect the ratio of output vs. input to be constant. To reflect this condition in the graph, for a fewer number of disk modules, the curves in Figure 5.1a will keep approximately the same shape but be shifted upward. That is, reducing the system capacity causes an increase of system efficiency under a constant data-load. In Figure 5.1b, the curve is plotted

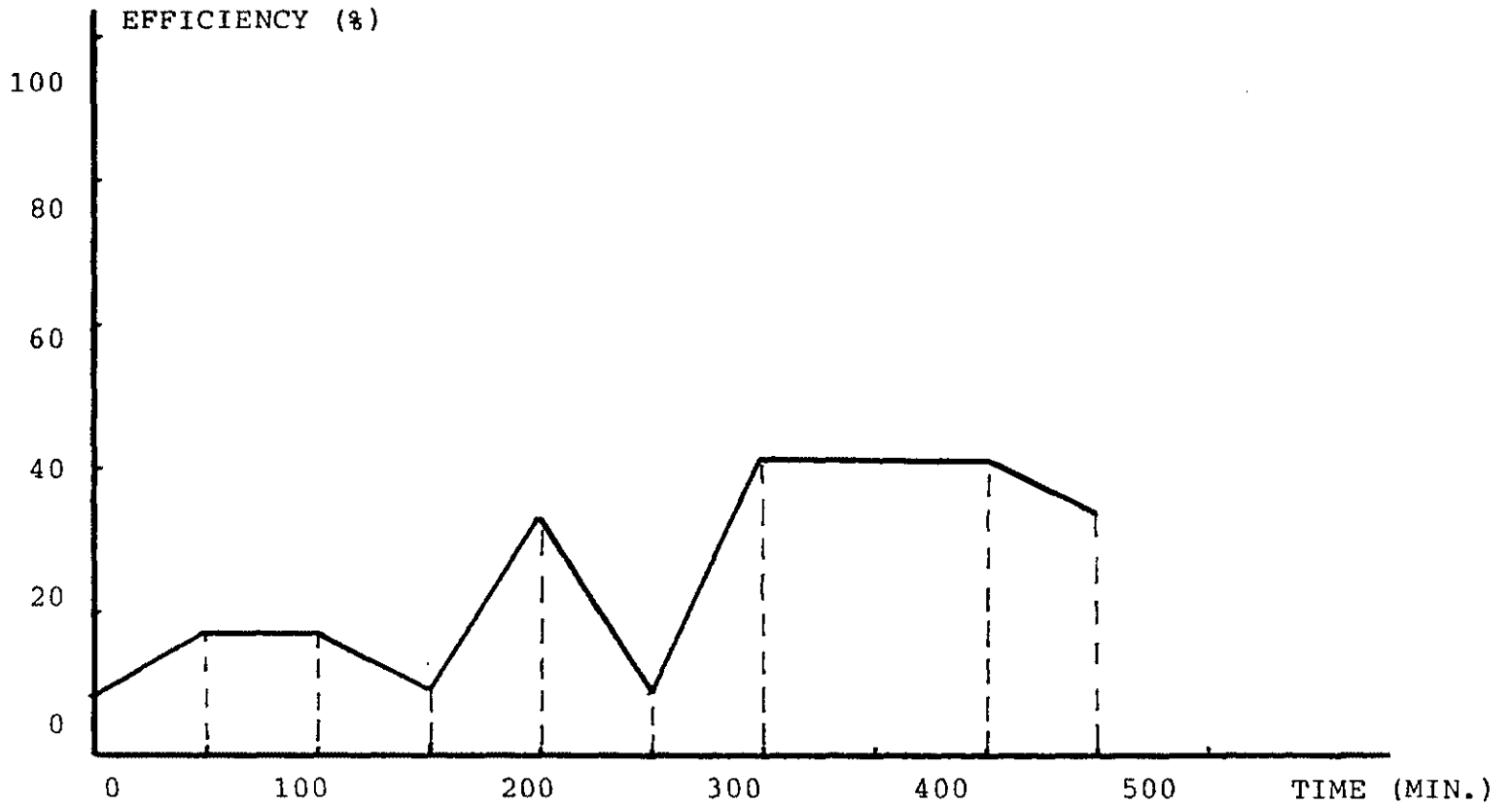


FIGURE 5.1A EFFICIENCY VS. TIME (12 DISK MODULES)

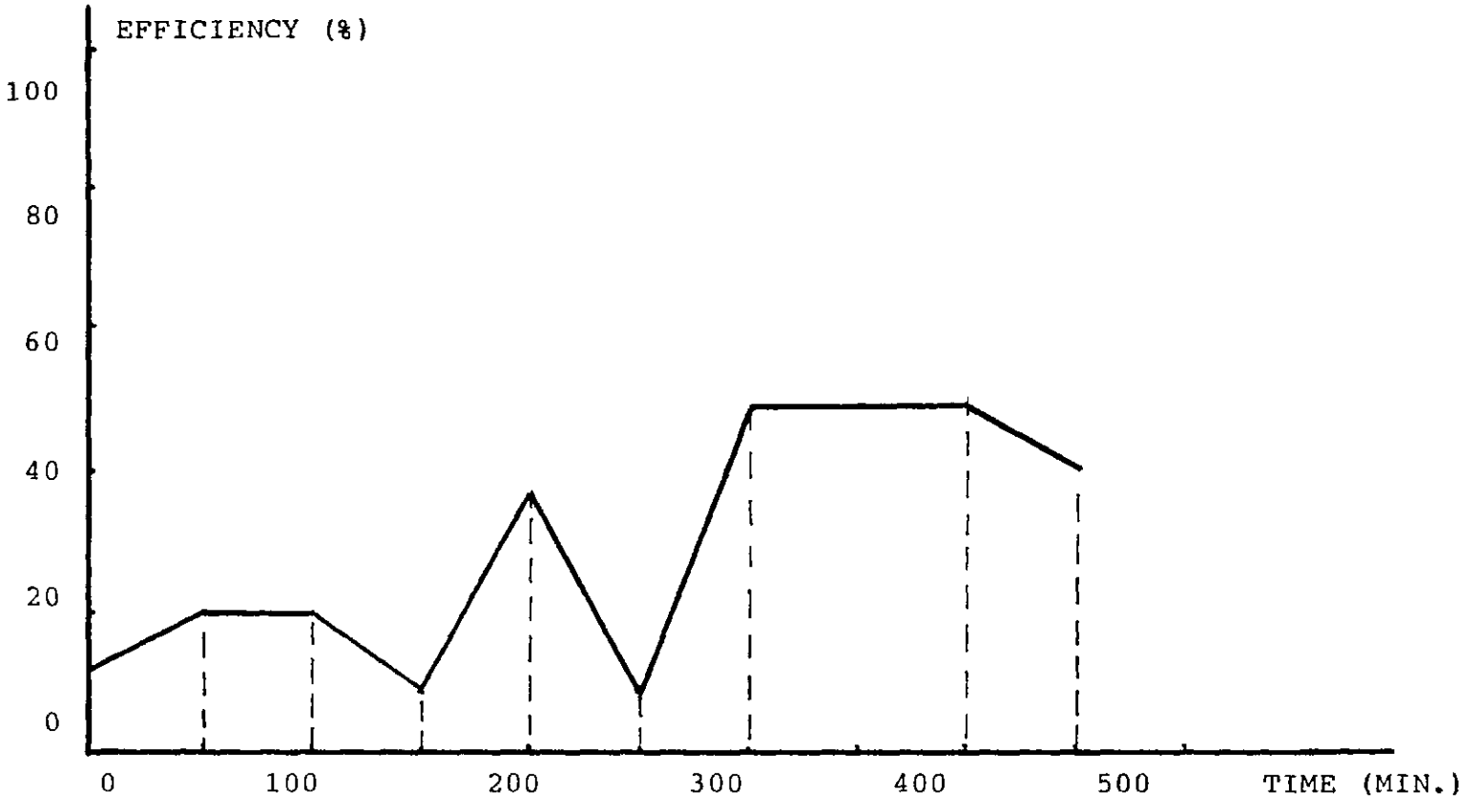


FIGURE 5.1B EFFICIENCY VS. TIME (10 DISK MODULES)

for the total number of modules equal to ten. The results meet our prediction.

After the model verification, the following results need to be drawn from the simulation:

1. the number of disk modules required without data loss,
2. the size of the circular buffer for different input rates, and
3. the effects of TDRSS starting transmission times on the system capacity.

Figure 5.2 shows the number of disk modules required for different available starting transmission times. Naturally, the memory system can start when the TDRSS transmission channel has already been available. The system status of this situation is examined by extending the same data distribution used in Figure 4.3 to a 24-hour period. In this case, the curve tends to flatten. Under normal operation ten disk modules are required in active mode.

The effect of missing the TDRSS transmission window once during an 100-minute EOS orbit can be seen in Figure 5.3. The performance degradation of the ODMMS is severe if missing the TDRSS channel occurs after 200 minutes. Because of a heavy data-load, the ODMMS reaches its full storage capacity before the next consecutive TDRSS transmission channel is available, and data overflows. The problem cannot be overcome by just simply increasing the

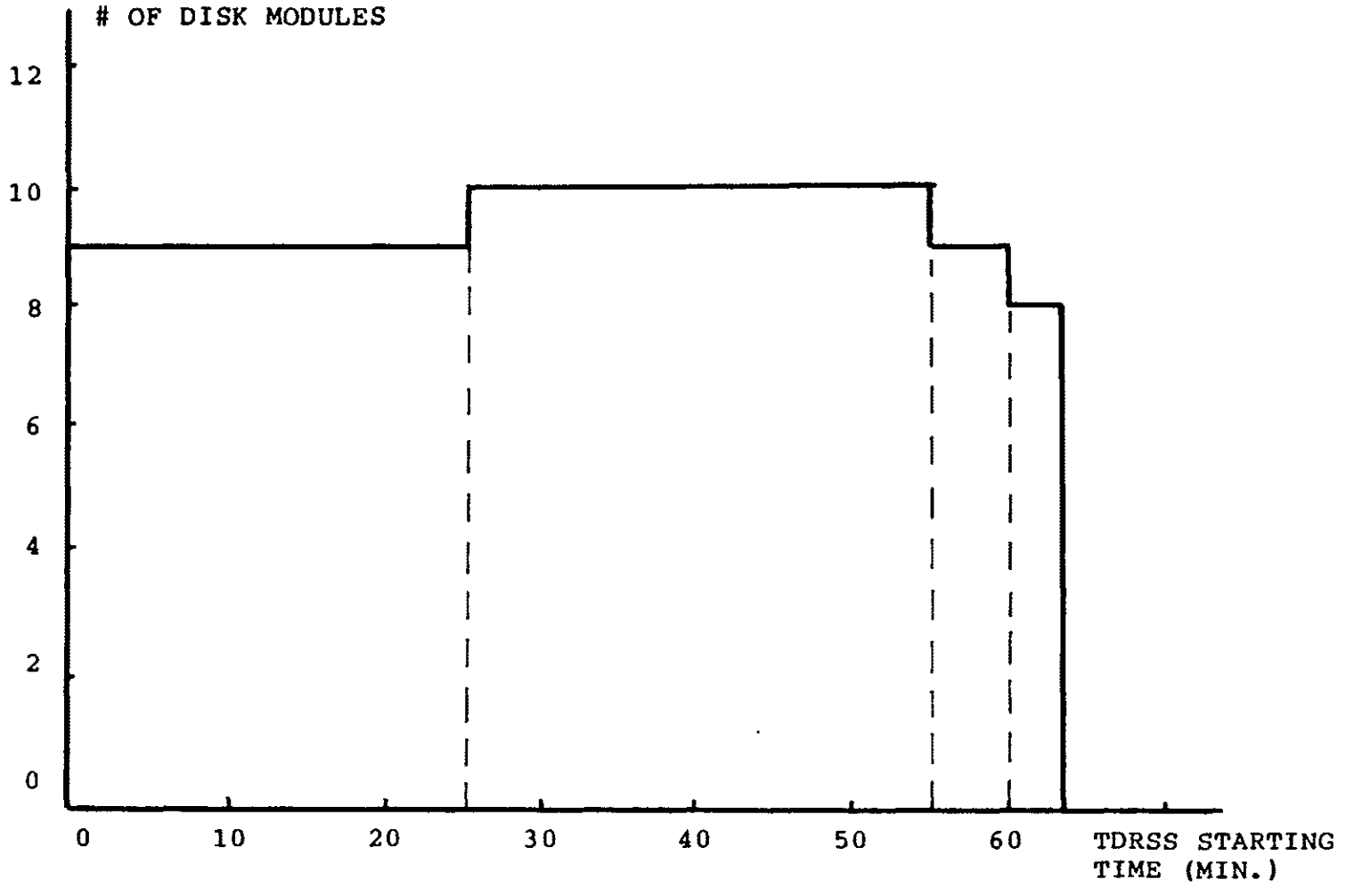


FIGURE 5.2 DISK MODULES REQUIRED VS. INITIAL TDRSS TIME

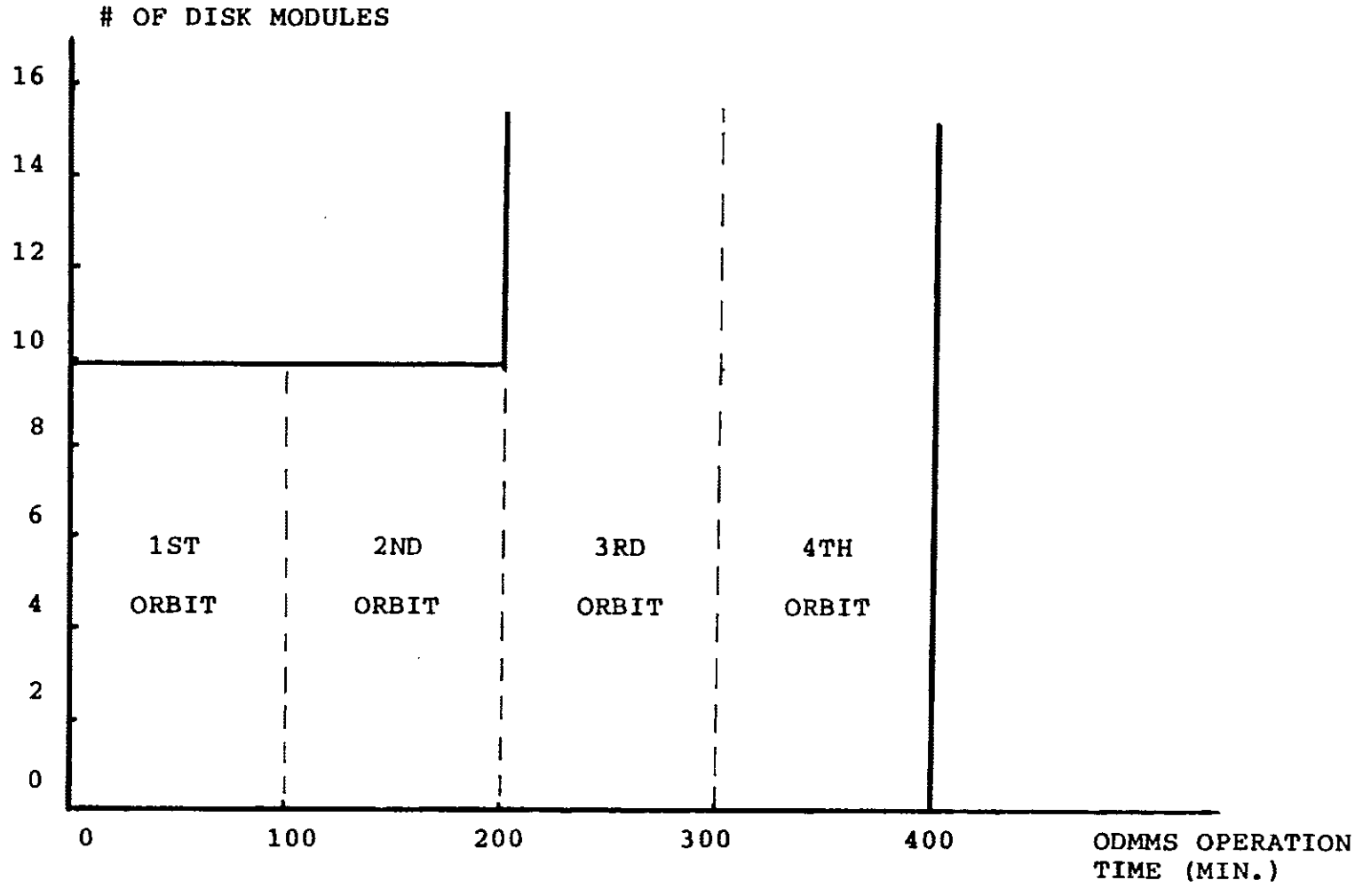


FIGURE 5.3 EFFECT OF TDRSS CHANNEL UNAVAILABLE

number of disk surfaces in each disk module. The increase of the number of disk surfaces in each disk module creates a gain in I/O rate but not in the overall system capacity. Examining the detailed simulation results, we find that the earlier two TDRSS orbits have not been fully utilized; i.e., input data-load is very light and the TDRSS channel availability is used at a low percentage (refer to Figure 5.1). Thus the capacity of the ODMMS is limited mainly by the I/O activities occurring during the later two TDRSS orbits.

The above discussion does not favor a module consisting of more than two disk surfaces. We next examine the system modularity from a different aspect. We have seen that ten disk modules are required under normal operation. For a total of 12 disk modules, there are two extra modules which can be considered as redundant. If the number of disk surfaces in each module is increased to three, for example, the disk system is partitioned into eight modules. In order to reach the capacity of 20 disk surfaces (ten modules) in the previous case, seven modules (consisting of 21 disk surfaces) are required, leaving one redundant module. To leave two modules as redundant, the system capacity will be reduced to six modules (18 disk surfaces), which obviously cannot meet the storage capacity required. Again, from the point of system modularity, a system partition containing more than two disk surfaces is not favorable.

As was previously discussed, a circular buffer is used for user rate buffering and disk module access. When the user rate is below 300 megabits per second, the amount of data accumulated in this buffer due to disk rotation delay will decrease with a disk-module write. For example, assume the incoming data rate is 280 megabits per second and the amount of data in the circular buffer is 1.9 modular tracks. After writing one modular track of data onto the current disk module, the amount of data left in the buffer will be reduced to approximately 1.833 modular tracks. It must continuously write about 14 modular tracks before the amount of data accumulated in the buffer can be reduced to less than one modular track. However, if the space left in the current disk module is less than 14 modular tracks, a new disk module needs to be initiated. In the worst case, a full modular track of data due to the new rotation delay could be added into the circular buffer. Thus, the total amount of data accumulated in the buffer would exceed two modular tracks. From discussions with other research groups, it has been suggested to increase the size of the circular buffer to three modular tracks.

However, examining the simulation results, we find that the size of the circular buffer is also determined by the following. In writing a long stream of data with the input rate of 300 megabits per second, the I/O status of the buffer is in a static state; i.e., data accumulation will not decrease with disk-module writes. Again, a full

modular track of data could be added into the buffer when starting to write a blank disk module. With the continuation of disk-module writes, the data accumulation would then increase with the number of disk modules accessed. Therefore, a large size circular buffer depending on the length of data files would be required. To circumvent these problems, we suggest the use of two circular buffers. The two buffers operate in an interleaving manner; one starts accepting data when the data left in the other fits into the remaining space of the current disk module. In any case, the data left in a circular buffer is less than or equal to the size of two modular tracks. It becomes the responsibility of the controller to monitor the status of the current circular buffer and initiate the second one on time.

Since it is impossible to start writing data anywhere other than the start of a track, one question concerning the track-space utilization is whether data left in a circular buffer should be dumped onto the disk module at the end of a file, maintained in the buffer waiting for the next data file, or a percentage transferred onto the disk module, resulting in less waste of track space. The problem is integrated into the disk-module write subroutine, where we consider 20, 40, 60, and 80 percent of track full as one track. As seen from the simulation results, the effects on total system capacity are insignificant or minor due to the nature of user data

files, which are often long and continuous. Therefore, data left in a buffer should be written onto the disk module. From the practical point of view, any unnecessary data transmission delay is undesirable. This confirms the earlier choice of selecting one track as a sector for a better utilization of disk space.

CHAPTER 6
CONCLUSIONS

Issues concerning system modularity, multi-user rate buffering, overall read/write file management, disk module access, and other details have been discussed with the simulation results. Most of the solutions are specially targeted to the space-borne applications, and are different from the conventional methods adopted for magnetic disk storages. It is worthwhile to emphasize them again as our recommendations on the future architecture of the optical disk controller. Some assumptions, made for the system modeling, are essential to the evaluation of the overall system performance and will also have considerable impact on the memory system design.

A fundamental assumption in this study is the first-in, first-out (FIFO) read/write file management. It is the most straightforward file management policy when considered either from the complexity of the controller's hardware or the practical applications on the EOS platform. Under the assumptions that there are no disk-module access conflicts and an optical head is maintained on the last track it served, the FIFO method requires no track-seek time for disk-module writes. The circular buffers, which have the size of two modular tracks, are shown to be adequate to

resolve the disk rotation delay and optical heads synchronization.

The concept of circular buffers, proposed here in handling the rate buffering and disk-module access, is an unique feature contributed to the controller's architecture. Their operation will be under the control of the disk controller's hardware/software, and constitute a major element of the data adaptor mentioned in Chapter 2.

We have further indicated that two circular buffers must operate in a manner of interleaving to avoid the increase of data accumulation inside a buffer. That is, the controller needs to monitor the track status of a current writing module and initiate a second buffer when a new disk module needs to be accessed. This is a new requirement imposed on the controller. As direct outcomes, the size of the circular buffers can be maintained at a minimum and multi-user inputs are allowed.

A trade-off between the operational requirements and disk access time is the completion of a hand-shaking process before an I/O operation can take place. To eliminate the track-seek delay, we have suggested that the optical heads need to be placed on the right tracks beforehand; however, it requires the controller to have a pre-notification time period from the users.

When partitioning the ODMMS into independent modules, we considered two major factors: the system capacity degradation and the modular I/O rate. The effect of system

degradation dictates that a disk module should contain a minimum number of disk surfaces. On the other hand, in accommodating the circular buffer structure, the modular I/O rate must be larger than, or equal to, the projected user rate of 300 megabits per second. We recommend one double-sided disk as a disk module. Technically speaking, if possible, the optical head on each disk surface can then be hard-wired to operate simultaneously. Thus, the synchronization of both heads is accomplished automatically.

Finally, to release the burden on the controller, data left in the circular buffer at the end of writing a file, is recommended to be transferred to the current disk module. Due to the nature of large data files, the wasting of disk space is very limited. This also makes a further partition of tracks unnecessary.

Our discussion up to this point has relied on the assumption of FIFO read/write file management. If the functional requirements of the ODMMS are changed, or files stored earlier on the memory need to be further referenced, then other file management policies could be feasible. Under the circumstances, the rate buffering strategy may vary correspondingly. Similarly, the relationship between the system storage capacity and data-load may also change. This leaves an interesting problem for further research study, and our simulation program could be further modified or referenced.

APPENDIX A
SIMULATION PROGRAM LISTING

```
/*    freet              Free-track pointer associated with write  */
/*    freet1             Free-track pointer associated with read   */
/*    ldisk[0]           Number of free disk sides with side[0]    */
/*    ldisk[1]           Number of free disk sides with side[1]    */
/*    MAXINT             Maximum integer                           */
/*    NDISK              Number of disk sides                      */
/*    NUMBT              Number of bits / track                    */
/*    NUMTRK             Total number of tracks / side             */
/*    ptrk               Current number of free tracks / side      */
/*    RATEROT            Disk rotation rate  r/s                   */
/*    sched.filenum      Data file number                          */
/*    sched.flength      Data file length   bits                   */
/*    sched.frate        User data rate  bits/s                    */
/*    sched.tarive       Data arriving time                        */
/*    sched.tlen         User file transmission time               */
/*    side[0]            Right free-disk-side pointer              */
/*    side[1]            Left free-disk-side pointer               */
/*                       side[1] --) (W/R) --) side[0]             */
/*                          ldisk[1]        ldisk[0]               */
/*    tread              Time-pointer of read operation            */
/*    trkp               Current heads position                    */
/*    trkq               Track buffer                              */
/*    TSTEP              Seek-time from track to track             */
/*    twindow            Time window of read operation             */
/*    WRATE              Disk module write rate   bits/s           */

#include(stdio.h)
#define TSTEP     0.001
#define RATEROT   15.413
#define MAXINT    32767.0
#define NUMBT     9.732045e06
#define NUMTRK    4727
#define WRATE     1.5e08
#define NDISK     20

struct {
        float  tarive[100];
        float  frate[100];
        float  tlen[100];
        float  flength[100];
        int    filenum[100];
       } sched;
```

```
int freet[30],freet1[30],ptrk[30],trkp[30],lmax,count;
int side[2],ldisk[2];
float twindow[20],tread,trkq;

void  event_schedule();
void  op_write(int lcount, float *timept);
void  op_read();
float seek_track(int trk, int trks);
float track_queue(int lcount, int trk, float tdel);

                /****************************/
                /**                        **/
                /**      MAIN PROGRAM      **/
                /**                        **/
                /****************************/

/*    LABEL DEFINING:                                                  */
/*      lcount            Data file number                             */
/*      lmax              Total number of data file                    */
/*      tcount            Overall system time clock                    */

main()
{
  int lcount,k;
  float tcount;

                /**  INITIALIZE  SYSTEM  **/

  srand(1);                          /* Seed random number generator */
  event_schedule();
  lcount = 0;
  count = 0;
  tread = 0.0;
  trkq = 0.0;
  tcount = sched.tarive[0];
  twindow[0] = 4020;             /* Read-time window starts anytime
                                          between 0 to 67 min. */
  for (k = 1; k (= 15; k++)
    twindow[k] = twindow[k-1] + 6000.0;
  for (k = 0; k ( 30; k = k + 2) {
    ptrk[k] = NUMTRK;
    trkp[k] = 0;
    freet[k] = 0;
    freet1[k] = 0;
    }
  side[0] = side[1] = 0;
  ldisk[0] = NDISK - 2;
  ldisk[1] = 0;

                /**  EXECUTE  EVENTS  **/

  for (k = 0; k ( lmax; k++)  {
    printf("FILE  #%d\n",lcount);
```

```
      op_write(lcount, &tcount);
      lcount = lcount + 1;
      }
  }
        /*********************************************/
        /**                                         **/
        /**     DATA FILES SCHEDULE SUBROUTINE      **/
        /**                                         **/
        /*********************************************/

void event_schedule()
{
  float tem2, tem3, tem4;
  int i, tem0, tem1;
  FILE *fp;

          /**  READ DATA FROM FILE  " OPTICAL.IN "  **/

  fp = fopen("optical.in", "r");
  fscanf(fp,"%d\n",&tem0);
  lmax = tem0;
  for(i = 0; i < lmax; i++)  {
    fscanf(fp,"%d %e %f %f\n", &tem1, &tem2, &tem3, &tem4);
    sched.filenum[i] = tem1;
    sched.frate[i] = tem2;
    sched.tarive[i] = tem3;
    sched.tlen[i] = tem4;
    sched.flength[i] = sched.frate[i] * sched.tlen[i];
    }
  fclose(fp);
}
          /*********************************************/
          /**                                         **/
          /**     DISK MODULE WRITE SUBROUTINE        **/
          /**                                         **/
          /*********************************************/

 /*    LABEL DEFINING:                                                */
 /*      flag              Flag is set if system clock is greater     */
 /*                        than data arriving time                    */
 /*      flag1             Flag1 is set after update track buffer     */
 /*      ntrk              Number of tracks needed on one disk side   */
 /*      ntrkc             Number of tracks needed besides tracks     */
 /*                        available on the current disk module       */
 /*      ntrki             Lower integer of ntrk                      */
 /*      t                 Disk module access time                    */
 /*      tdel              Delay time used to determine buffer size   */
 /*      timept            System clock                               */
 /*      trkq1             Current size of track buffer               */
 /*      twrite            Disk module write time                     */
```

```
void op_write(int lcount, float *timept)
{
  int k, ki, flag, flag1;
  long ntrki, ntrkc;
  float t, tdel, twrite, ntrk, temp, trkq1;

  k = side[0];
  flag = 0;
  flag1 = 0;
  ntrk = sched.flength[lcount] / (2.0 * NUMBT);
  ntrki = ntrk;
  trkq1 = trkq + NUMBT * (ntrk - ntrki);
  if (trkq1 )= 0.1 * NUMBT) {
    ntrki = ntrki + 1;              /* Empty trkq if it is 10% full */
    flag1 = 1;
    }
  ntrkc = ntrki - ptrk[k];
  printf("FILE LENGTH(TRACKS):    %ld\n",2*ntrki);
  if (*timept (= sched.tarive[lcount])
    *timept = sched.tarive[lcount];
  else
    flag = 1;
  other_disk:
  if (ntrkc ( 0) {                /* Enough space on one disk module */
    if (flag == 1)
      t = 0.0;                       /* Continue to write new file */
    else
      t = seek_track(freet[k], trkp[k]);
    tdel = t;
    if (tdel )= sched.tlen[lcount])         /* Data transmission is
                                       completed within delay time */
      tdel = sched.tlen[lcount];
    twrite = track_queue(lcount, ntrki, tdel);
    *timept = *timept + t + twrite;
    if (flag1 == 1)
      trkq = 0.0;                            /* Update track-buffer */
    trkp[k] = freet[k] + ntrki;      /* Update current head position */
    ptrk[k] = ptrk[k] - ntrki;      /* Update number of free tracks  */
    freet[k] = freet[k] + ntrki;       /* Update free track location */
    if (ptrk[k] (= 0)   {              /* Start writing to next disk */
      ldisk[0] = ldisk[0] - 2;
      side[0] = side[0] + 2;
      freet[k] = 0;
      ptrk[k] = 0;
      trkp[k] = NUMTRK;
      }
    if (side[0] )= NDISK) {              /* Disk allocation diagram: */
      ldisk[0] = ldisk[1] - 2;      /* Before:  | [1]->| data [0]->| */
      side[0] = 0;                   /* After:  | [0]->|[1]-> data | */
      side[1] = ldisk[1];
      ldisk[1] = 0;
      }
```

```
    printf("SYSTEM STATUS BEFORE READING:\n");
    printf("SYSTEM_TIME:%10.3f  READ_TIME:%10.3f  R_LENGTH:%10.3f\n",
                                        *timept,twindow[count],tread);
    printf("RD_PTR: %d    LENGTH: %d    WR_PTR: %d    LENGTH: %d\n",
                         side[1]/2,ldisk[1]/2,side[0]/2,ldisk[0]/2);
    k1 = 0;
    for(k=0; k<NDISK; k=k+2)    {
      printf("DISK%2d:    %d\n",k1,2*ptrk[k]);
      k1 = k1 + 1;
      }
    printf("\n");

    if (*timept >= twindow[count])  {              /* Starts reading */
      if ((side[1] + ldisk[1]) == side[0])  {
        tread = tread + *timept - twindow[count]; /* Unable to read */
        twindow[count] = *timept;
        }
      else {
        temp = tread;
        op_read();
        twindow[count] = twindow[count] + tread - temp;
        }

      printf("SYSTEM STATUS AFTER READING:\n");
      printf("R_LENGTH: %10.3f\n", tread);
      printf("RD_PTR: %d    LENGTH: %d    WT_PTR: %d    LENGTH: %d\n",
                        side[1]/2,ldisk[1]/2, side[0]/2, ldisk[0]/2);
      k1 = 0;
      for (k=0; k<NDISK; k=k+2)      {
        printf("DISK%2d:    %d\n",k1,2*ptrk[k]);
        k1 = k1 + 1;
        }
      printf("\n");

      if (tread >= 1980.0)  {          /* Read-time window is over */
        tread = 0.0;
        count = count + 1;     /* Advance to next read-time window */
        }
      }
  return;
  }
else {                           /* Need more than one disk module */
  if (flag == 1)
    t = 0;
  else
    t = seek_track(freet[k], trkp[k]);
  flag = 0;                                          /* Reset flag */
  ntrki = ptrk[k];
  tdel = t;
  if(tdel >= sched.tlen[lcount])
    tdel = sched.tlen[lcount];
  twrite = track_queue(lcount, ntrki, tdel);
  sched.tlen[lcount] = sched.tlen[lcount] - twrite;
```

```
if (sched.tlen[lcount] < 0.0)
  sched.tlen[lcount] = 0.0;
*timept = *timept + t + twrite;
trkp[k] = NUMTRK;
freet[k] = 0;
ptrk[k] = 0;
side[0] = side[0] + 2;
ldisk[0] = ldisk[0] - 2;
if (side[0] >= NDISK)  {            /* Disk allocation diagram:  */
  side[0] = 0;                    /* Before:  | [1]->| data [0]->| */
  ldisk[0] = ldisk[1] - 2;         /* After:  | [0]->|[1]-> data | */
  side[1] = ldisk[1];
  ldisk[1] = 0;
  }

printf("SYSTEM STATUS BEFORE READING:\n");
printf("TIME:%10.3f  READ_TIME:%10.3f  R_LENGTH:%10.3f\n",
                                  *timept, twindow[count],tread);
printf("RD_PTR: %d    LENGTH: %d    WR_PTR: %d    LENGTH: %d\n",
                       side[1]/2,ldisk[1]/2,side[0]/2,ldisk[0]/2);
k1 =0;
for (k=0; k<NDISK; k=k+2)       {
  printf("DISK%2d:    %d\n",k1,2*ptrk[k]);
  k1 = k1 + 1;
  }
printf("\n");

if (*timept >= twindow[count])  {
  if (side[1] + ldisk[1] == side[0] - 2)  {
    tread = tread + *timept - twindow[count];
                                  /* Wait till write is finished */
    temp = tread;
    op_read();
    twindow[count] = *timept + tread - temp;
    }
  else {
    temp = tread;
    op_read();
    twindow[count] = twindow[count] + tread - temp;
    }

  printf("SYSTEM STATUS AFTER READING:\n");
  printf("R_LENGTH: %10.3f\n", tread);
  printf("RD_PTR: %d    LENGTH: %d    WR_PTR: %d    LENGTH: %d\n",
                       side[1]/2,ldisk[1]/2,side[0]/2,ldisk[0]/2);
  k1 = 0;

  for(k=0; k<NDISK; k=k+2)    {
    printf("DISK%2d:    %d\n",k1,2*ptrk[k]);
    k1 = k1 + 1;
    }
  printf("\n");
```

```
      if (tread )= 1980.0)  {
        tread = 0.0;
        count = count + 1;
        }
      }
    k = side[0];
    ntrki = ntrkc;                              /* Update file length */
    ntrkc = ntrkc - ptrk[k];
    if ((ptrk[k] (= 0)&&(ntrkc ) 0))    {
      printf("WRITE OPERATION IS INCOMPLETE\n");
      printf("\n");
      return;
      }
    goto other_disk;
    }
}

            /*******************************************/
            /**                                       **/
            /**      DISK MODULE READ SUBROUTINE      **/
            /**                                       **/
            /*******************************************/

void op_read()
{
  int trk, dsk;

  if (tread )= 1980.0)                  /* Terminate read operation */
    return;

  loop2:
  dsk = side[1] + ldisk[1];
  trk = freet1[dsk];
  if (side[0] ) dsk)  {                 /* Disk allocation diagram: */
                                        /* | [1] ->| data ->| [0] | */
    tread = tread + seek_track(freet1[dsk], trkp[dsk]);
    do {
      tread = tread + 1.0 / RATEROT;            /* Read one track */
      trk = trk + 1;
      if (trk )= NUMTRK)      {   /* Completion of one disk reading */
        ptrk[dsk] = NUMTRK;
        trkp[dsk] = 0.0;                /* Head position after read */
        freet1[dsk] = 0;
        ldisk[1] = ldisk[1] + 2;
        dsk = dsk + 2;                      /* Advance pointer [1] */
        trk = 0;
        if (dsk )= side[0])                  /* Two pointers crash */
          goto loop1;
        tread = tread + seek_track(freet1[dsk], trkp[dsk]);
        }
      } while(tread ( 1980.0);          /* Read-time slot is 33 min. */
    freet1[dsk] = freet1[dsk] + trk;
    ptrk[dsk] = freet1[dsk];
```

```
    trkp[dsk] = 0.0;
    loop1:
    return;
    }
  else {                                  /* Disk allocation diagram: */
                                            /* | [0]-)| [1] -) data | */
    tread = tread + seek_track(freet1[dsk], trkp[dsk]);
    do {
      tread = tread + 1.0 / RATEROT;              /* Read one track */
      trk = trk + 1;
      if (trk )= NUMTRK)  {
        freet1[dsk] = 0;
        ptrk[dsk] = NUMTRK;
        trkp[dsk] = 0.0;
        dsk = dsk + 2;
        side[1] = side[1] + 2;
        ldisk[0] = ldisk[0] + 2;
        trk = 0;
        if (dsk )= NDISK) {/* Change pointers, back to the original */
          side[1] = 0;
          if (side[0] == 0)
            return;
          goto loop2;
          }
        tread = tread + seek_track(freet1[dsk], trkp[dsk]);
        }
     } while(tread ( 1980.0);
    freet1[dsk] = freet1[dsk] + trk;
    ptrk[dsk] = freet1[dsk];
    trkp[dsk] = 0.0;
    return;
    }
}
          /*****************************************/
          /**                                     **/
          /**     DISK MODULE ACCESS SUBROUTINE   **/
          /**                                     **/
          /*****************************************/

/*    LABEL DEFINING:                                                  */
/*      phead             Heads' position over the right tracks      */
/*      tjump             Time to move heads to the right tracks     */
/*      trk               Tracks needs to be found                   */
/*      trks              Current heads position                     */
/*      trot              Disk rotation delay                        */
/*      tseek             Total disk module access time              */

float seek_track(int trk, int trks)
{
  float tjump, tseek, phead, trot;

  /* Position head to the nth track
     assuming innermost track as 0th track */
```

```
  if (trk == trks)
    tjump = 0.0;
  else if (trk ( trks)                          /* Move R/W inward */
         tjump = TSTEP * (trks - trk);
       else                                     /* Move R/W outward */
         tjump = TSTEP * (trk - trks);

  /* Position head to the start of track */

  phead = rand() / MAXINT;              /* Position R/W at nth track */
  trot = (1 - phead) / RATEROT;                 /* Rotation delay */
  tseek = tjump + trot;
  return(tseek);
}
          /****************************************/
          /**                                    **/
          /**     RATE BUFFERING SUBROUTINE      **/
          /**                                    **/
          /****************************************/

/*    LABEL DEFINING:                                                */
/*      point1          Number of bits in the buffer due to delay */
/*      trk             Number of tracks to be written            */
/*      fl              Number of bits accumulated after one disk */
/*                      rotation                                  */

float track_queue(int lcount, int trk, float tdel)
{
  float point, point1, fl, t;
  int k, k1;

  k = 0;
  k1 = 1;
  t = 0.0;
  fl = sched.frate[lcount] / (2.0 * RATEROT);
                /* Advance pointer by fl bits after one rotation */

         /**   DETERMINE BUFFER SIZE  **/

  point1 = trkq + sched.frate[lcount] * tdel / 2.0;
  point = point1;
  if (point1 )= NUMBT)
    do {
      point1 = point1 - NUMBT;
      k1 = k1 + 1;
      } while(point1 )= NUMBT);
  printf("TRACK BUFFER SIZE(TRACK): %d\n",k1);

           /**   RATE CHANGE   **/

  if (point )= NUMBT)
```

```
    do {
      point = point - NUMBT + fl;
      k = k + 1;
      t = t + 1.0 / RATEROT;
      if (k )= trk)
        goto loop;
      } while(point )= NUMBT);
  do {
    point = point + fl;
    if (point )= NUMBT)  {
      point = point - NUMBT;
      k = k + 1;
      }
    t = t + 1.0 / RATEROT;
    } while(k ( trk);
  loop:
  trkq = point;
  return(t);
}
```

APPENDIX B
PROGRAM INPUT DESCRIPTION

The input file is determined corresponding to the diagram shown in Figure 4.3 where each constant-rate block is considered as a data file. To make the input file structure more understandable, a detailed explanation is provided here (referring to the listing on the next page).

The following instructions are used in the program to read data from the input file:

```
fp = fopen("optical.in", "r");
```

It opens the file named "optical.in" as the input file. Each following instruction reads one line of data from the file "optical.in", and the instructions are repeated in a loop. No data field specification is required.

```
fscanf(fp,"%d\n",lmax);
```

lmax -- total number of files, integer.

```
for(i = 0; i < lmax; i++)  {
  fscanf(fp,"%d %e %f %f\n", filenum, frate, tarive,
        tlen);
  }
```

filenum -- file number, integer;

frate -- data transmission rate, exponential, Mb/sec.;

tarive -- starting transmission time, real, second;

tlen -- data transmission time, real, second.

INPUT FILE: "optical.in"

```
21
0     1.3e08          0.0      300.0
1     1.5e08        600.0      300.0
2     3.0e08        900.0      300.0
3     1.5e08       1200.0      300.0
4     1.5e08       4200.0      600.0
5     1.0e07       4800.0     2400.0
6     1.5e08      10200.0      600.0
7     1.5e08      11100.0      300.0
8     3.0e08      11400.0      300.0
9     1.5e08      11700.0      300.0
10    1.5e08      12300.0      300.0
11    3.0e08      12600.0      300.0
12    1.5e08      12900.0      300.0
13    1.5e08      16200.0      600.0
14    1.5e08      16800.0      300.0
15    3.0e08      17100.0     2100.0
16    1.5e08      19200.0      300.0
17    1.5e08      22200.0      600.0
18    1.5e08      24000.0      300.0
19    3.0e08      24300.0      900.0
20    1.5e08      25200.0      300.0
```

For example, after reading the very first two lines of data from the file "optical.in", each variable in the instructions will be assigned the following value:

lmax = 21

filenum = 0

frate = 1.3e08

tarive = 0.0 and

tlen = 300.0.

The file 0 corresponds to the left-most, constant-rate data block in Figure 4.3, where the left edge of the block determines the value of frate, and the width of the block determines the value of tlen. The determination of the rest files follows the same reasoning.

APPENDIX C
PROGRAM OUTPUT DESCRIPTION

The output is printed directly on a screen during a program run. We intend to provide the readers a better understanding by explaining a typical output segment listed below (numbers at the beginning of lines are for illustration only).

```
0   FILE #4
1   FILE LENGTH(TRACKS):  9248
2   TRACK BUFFER SIZE(TRACK):  1
3   SYSTEM STATUS BEFORE READING:
4   TIME:  4580.111   READ_TIME:  4020.0  R_LENGTH:  0.00
5   RD_PTR:  0   LENGTH:  0  WR_PTR:  3   LENGTH:  6
6   DISK 0:     0
7   DISK 1:     0
8   DISK 2:     0
9   DISK 3:  9454
10  DISK 4:  9454
11  DISK 5:  9454
12  DISK 6:  9454
13  DISK 7:  9454
14  DISK 8:  9454
15  DISK 9:  9454
```

```
16  SYSTEM STATUS AFTER READING:
17  R_LENGTH:  934.383
18  RD_PTR:  0   LENGTH:  3  WR_PTR:  3   LENGTH:  6
19  DISK 0:  9454
20  DISK 1:  9454
21  DISK 2:  9454
22  DISK 3:  9454
23  DISK 4:  9454
24  DISK 5:  9454
25  DISK 6:  9454
26  DISK 7:  9454
27  DISK 8:  9454
28  DISK 9:  9454

29  TRACK BUFFER SIZE(TRACK):  1
30  SYSTEM STATUS BEFORE READING:
31  TIME:  4800.075 READ_TIME:  4954.382 R_LENGTH:  934.383
32  RD_PTR:  0   LENGTH:  3   WR_PTR:  3   LENGTH:  6
33  DISK 0:  9454
34  DISK 1:  9454
35  DISK 2:  9454
36  DISK 3:  3598
37  DISK 4:  9454
38  DISK 5:  9454
39  DISK 6:  9454
40  DISK 7:  9454
41  DISK 8:  9454
```

42 DISK 9: 9454

Line 0 - line 2 provides the following information: the number of current data file, its length (in terms of modular tracks), and the size of buffer (in terms of modular track) for disk-module access.

Line 4 provides the timing information: TIME is the system time after writing the current module, READ_TIME is the TDRSS available time, and R_LENGTH is the amount of TDRSS time that has been used. The unit of time is in seconds.

Line 5 provides the capacity status of the system after writing the current module. RD_PTR, LENGTH, WR_PTR, and LENGTH correspond to the status scheme defined previously in Figure 4.6.

Line 6 - line 15 lists the number of modular tracks available on each module. Zero means a module is full, and 9454 is the total number of modular tracks available on a module (two disk surfaces).

As assumed, the system time-pointer is checked against the TDRSS availability at the end of writing a file or a disk module (end of a module in this case), to reflect a simultaneous I/O. Since the value of TIME is larger than the value of READ_TIME, the disk-module read subroutine is called. The output then reflects the system status after the disk-module read (line 16 - line 28). Meanwhile, the system time-pointer is frozen until the disk-module write

subroutine is resumed. To read module 0 through module 2, it requires about 934.38 seconds (equal to the value of R_LENGTH). The disk-module read cannot continue on module 3 because it is the current writing module. The program control is then switched to the disk-module write. The output format is similar as before.

Line 36 indicates that part of the data file is written on module 3.

At the end of writing file #4, the value of the system time-pointer is smaller than the value of READ_TIME, which means that data-read on module 2 has not yet been completed. Therefore, no disk-module read can be initiated at this time instant, and the disk memory starts accepting new data files (it is the case of simultaneous I/O).

The above output segment is part of a computer printout from a sample run (refer to the next few pages). Following the same reasoning, the readers will be able to trace the rest of the outputs.

```
FILE  #0
FILE LENGTH(TRACKS):    4008
TRACK BUFFER SIZE(TRACK): 1
SYSTEM STATUS BEFORE READING:
SYSTEM_TIME:   300.071  READ_TIME:  4020.000  R_LENGTH:      0.000
RD_PTR: 0    LENGTH: 0    WR_PTR: 0    LENGTH: 9
DISK 0:    5446
DISK 1:    9454
DISK 2:    9454
DISK 3:    9454
DISK 4:    9454
DISK 5:    9454
DISK 6:    9454
DISK 7:    9454
DISK 8:    9454
DISK 9:    9454

FILE  #1
FILE LENGTH(TRACKS):    4624
TRACK BUFFER SIZE(TRACK): 1
SYSTEM STATUS BEFORE READING:
SYSTEM_TIME:   900.071  READ_TIME:  4020.000  R_LENGTH:      0.000
RD_PTR: 0    LENGTH: 0    WR_PTR: 0    LENGTH: 9
DISK 0:    822
DISK 1:    9454
DISK 2:    9454
DISK 3:    9454
DISK 4:    9454
DISK 5:    9454
DISK 6:    9454
DISK 7:    9454
DISK 8:    9454
DISK 9:    9454

FILE  #2
FILE LENGTH(TRACKS):    9248
TRACK BUFFER SIZE(TRACK): 1
SYSTEM STATUS BEFORE READING:
TIME:   926.802  READ_TIME:  4020.000  R_LENGTH:      0.000
RD_PTR: 0    LENGTH: 0    WR_PTR: 1    LENGTH: 8
DISK 0:    0
DISK 1:    9454
DISK 2:    9454
DISK 3:    9454
DISK 4:    9454
DISK 5:    9454
DISK 6:    9454
DISK 7:    9454
DISK 8:    9454
DISK 9:    9454

TRACK BUFFER SIZE(TRACK): 2
SYSTEM STATUS BEFORE READING:
```

```
SYSTEM_TIME:  1200.186  READ_TIME:  4020.000  R_LENGTH:      0.000
RD_PTR: 0    LENGTH: 0    WR_PTR: 1    LENGTH: 8
DISK 0:    0
DISK 1:    1028
DISK 2:    9454
DISK 3:    9454
DISK 4:    9454
DISK 5:    9454
DISK 6:    9454
DISK 7:    9454
DISK 8:    9454
DISK 9:    9454

FILE  #3
FILE LENGTH(TRACKS):    4624
TRACK BUFFER SIZE(TRACK): 1
SYSTEM STATUS BEFORE READING:
TIME:  1266.948  READ_TIME:  4020.000  R_LENGTH:      0.000
RD_PTR: 0    LENGTH: 0    WR_PTR: 2    LENGTH: 7
DISK 0:    0
DISK 1:    0
DISK 2:    9454
DISK 3:    9454
DISK 4:    9454
DISK 5:    9454
DISK 6:    9454
DISK 7:    9454
DISK 8:    9454
DISK 9:    9454

TRACK BUFFER SIZE(TRACK): 1
SYSTEM STATUS BEFORE READING:
SYSTEM_TIME:  1500.256  READ_TIME:  4020.000  R_LENGTH:      0.000
RD_PTR: 0    LENGTH: 0    WR_PTR: 2    LENGTH: 7
DISK 0:    0
DISK 1:    0
DISK 2:    5858
DISK 3:    9454
DISK 4:    9454
DISK 5:    9454
DISK 6:    9454
DISK 7:    9454
DISK 8:    9454
DISK 9:    9454

FILE  #4
FILE LENGTH(TRACKS):    9248
TRACK BUFFER SIZE(TRACK): 1
SYSTEM STATUS BEFORE READING:
TIME:  4580.111  READ_TIME:  4020.000  R_LENGTH:      0.000
RD_PTR: 0    LENGTH: 0    WR_PTR: 3    LENGTH: 6
DISK 0:    0
DISK 1:    0
```

```
DISK 2:     0
DISK 3:     9454
DISK 4:     9454
DISK 5:     9454
DISK 6:     9454
DISK 7:     9454
DISK 8:     9454
DISK 9:     9454

SYSTEM STATUS AFTER READING:
R_LENGTH:     934.383
RD_PTR: 0     LENGTH: 3     WR_PTR: 3     LENGTH: 6
DISK 0:     9454
DISK 1:     9454
DISK 2:     9454
DISK 3:     9454
DISK 4:     9454
DISK 5:     9454
DISK 6:     9454
DISK 7:     9454
DISK 8:     9454
DISK 9:     9454

TRACK BUFFER SIZE(TRACK): 1
SYSTEM STATUS BEFORE READING:
SYSTEM_TIME:  4800.075  READ_TIME:  4954.382  R_LENGTH:    934.383
RD_PTR: 0     LENGTH: 3     WR_PTR: 3     LENGTH: 6
DISK 0:     9454
DISK 1:     9454
DISK 2:     9454
DISK 3:     6064
DISK 4:     9454
DISK 5:     9454
DISK 6:     9454
DISK 7:     9454
DISK 8:     9454
DISK 9:     9454

FILE  #5
FILE LENGTH(TRACKS):     2466
TRACK BUFFER SIZE(TRACK): 1
SYSTEM STATUS BEFORE READING:
SYSTEM_TIME:  7199.431  READ_TIME:  4954.382  R_LENGTH:    934.383
RD_PTR: 0     LENGTH: 3     WR_PTR: 3     LENGTH: 6
DISK 0:     9454
DISK 1:     9454
DISK 2:     9454
DISK 3:     3598
DISK 4:     9454
DISK 5:     9454
DISK 6:     9454
DISK 7:     9454
DISK 8:     9454
```

```
DISK 9:     9454

SYSTEM STATUS AFTER READING:
R_LENGTH:    3179.431
RD_PTR: 0     LENGTH: 3     WT_PTR: 3     LENGTH: 6
DISK 0:     9454
DISK 1:     9454
DISK 2:     9454
DISK 3:     3598
DISK 4:     9454
DISK 5:     9454
DISK 6:     9454
DISK 7:     9454
DISK 8:     9454
DISK 9:     9454

FILE  #6
FILE LENGTH(TRACKS):     9248
TRACK BUFFER SIZE(TRACK): 1
SYSTEM STATUS BEFORE READING:
TIME: 10433.442  READ_TIME: 10020.000  R_LENGTH:      0.000
RD_PTR: 0     LENGTH: 3     WR_PTR: 4     LENGTH: 5
DISK 0:     9454
DISK 1:     9454
DISK 2:     9454
DISK 3:     0
DISK 4:     9454
DISK 5:     9454
DISK 6:     9454
DISK 7:     9454
DISK 8:     9454
DISK 9:     9454

SYSTEM STATUS AFTER READING:
R_LENGTH:     724.906
RD_PTR: 0     LENGTH: 4     WR_PTR: 4     LENGTH: 5
DISK 0:     9454
DISK 1:     9454
DISK 2:     9454
DISK 3:     9454
DISK 4:     9454
DISK 5:     9454
DISK 6:     9454
DISK 7:     9454
DISK 8:     9454
DISK 9:     9454

TRACK BUFFER SIZE(TRACK): 1
SYSTEM STATUS BEFORE READING:
SYSTEM_TIME: 10800.053  READ_TIME: 10744.906  R_LENGTH:    724.906
RD_PTR: 0     LENGTH: 4     WR_PTR: 4     LENGTH: 5
DISK 0:     9454
DISK 1:     9454
```

```
DISK 2:     9454
DISK 3:     9454
DISK 4:     3804
DISK 5:     9454
DISK 6:     9454
DISK 7:     9454
DISK 8:     9454
DISK 9:     9454

SYSTEM STATUS AFTER READING:
R_LENGTH:     780.052
RD_PTR: 0     LENGTH: 4     WT_PTR: 4     LENGTH: 5
DISK 0:     9454
DISK 1:     9454
DISK 2:     9454
DISK 3:     9454
DISK 4:     3804
DISK 5:     9454
DISK 6:     9454
DISK 7:     9454
DISK 8:     9454
DISK 9:     9454

FILE  #7
FILE LENGTH(TRACKS):     4624
TRACK BUFFER SIZE(TRACK): 1
SYSTEM STATUS BEFORE READING:
TIME: 11346.862  READ_TIME: 10800.053  R_LENGTH:    780.052
RD_PTR: 0     LENGTH: 4     WR_PTR: 5     LENGTH: 4
DISK 0:     9454
DISK 1:     9454
DISK 2:     9454
DISK 3:     9454
DISK 4:     0
DISK 5:     9454
DISK 6:     9454
DISK 7:     9454
DISK 8:     9454
DISK 9:     9454

SYSTEM STATUS AFTER READING:
R_LENGTH:    1638.010
RD_PTR: 0     LENGTH: 5     WR_PTR: 5     LENGTH: 4
DISK 0:     9454
DISK 1:     9454
DISK 2:     9454
DISK 3:     9454
DISK 4:     9454
DISK 5:     9454
DISK 6:     9454
DISK 7:     9454
DISK 8:     9454
DISK 9:     9454
```

```
TRACK BUFFER SIZE(TRACK): 1
SYSTEM STATUS BEFORE READING:
SYSTEM_TIME: 11400.027  READ_TIME: 11658.010  R_LENGTH:  1638.010
RD_PTR: 0    LENGTH: 5    WR_PTR: 5    LENGTH: 4
DISK 0:    9454
DISK 1:    9454
DISK 2:    9454
DISK 3:    9454
DISK 4:    9454
DISK 5:    8634
DISK 6:    9454
DISK 7:    9454
DISK 8:    9454
DISK 9:    9454

FILE  #8
FILE LENGTH(TRACKS):    9248
TRACK BUFFER SIZE(TRACK): 1
SYSTEM STATUS BEFORE READING:
TIME: 11680.181  READ_TIME: 11658.010  R_LENGTH:  1638.010
RD_PTR: 0    LENGTH: 5    WR_PTR: 6    LENGTH: 3
DISK 0:    9454
DISK 1:    9454
DISK 2:    9454
DISK 3:    9454
DISK 4:    9454
DISK 5:    0
DISK 6:    9454
DISK 7:    9454
DISK 8:    9454
DISK 9:    9454

SYSTEM STATUS AFTER READING:
R_LENGTH:   1971.371
RD_PTR: 0    LENGTH: 6    WR_PTR: 6    LENGTH: 3
DISK 0:    9454
DISK 1:    9454
DISK 2:    9454
DISK 3:    9454
DISK 4:    9454
DISK 5:    9454
DISK 6:    9454
DISK 7:    9454
DISK 8:    9454
DISK 9:    9454

TRACK BUFFER SIZE(TRACK): 2
SYSTEM STATUS BEFORE READING:
SYSTEM_TIME: 11700.153  READ_TIME: 11991.371  R_LENGTH:  1971.371
RD_PTR: 0    LENGTH: 6    WR_PTR: 6    LENGTH: 3
DISK 0:    9454
DISK 1:    9454
DISK 2:    9454
```

```
DISK 3:     9454
DISK 4:     9454
DISK 5:     9454
DISK 6:     8840
DISK 7:     9454
DISK 8:     9454
DISK 9:     9454

FILE  #9
FILE LENGTH(TRACKS):     4624
TRACK BUFFER SIZE(TRACK): 1
SYSTEM STATUS BEFORE READING:
SYSTEM_TIME: 12000.225  READ_TIME: 11991.371  R_LENGTH:  1971.371
RD_PTR: 0    LENGTH: 6    WR_PTR: 6    LENGTH: 3
DISK 0:     9454
DISK 1:     9454
DISK 2:     9454
DISK 3:     9454
DISK 4:     9454
DISK 5:     9454
DISK 6:     4216
DISK 7:     9454
DISK 8:     9454
DISK 9:     9454

SYSTEM STATUS AFTER READING:
R_LENGTH:    1980.224
RD_PTR: 0    LENGTH: 6    WT_PTR: 6    LENGTH: 3
DISK 0:     9454
DISK 1:     9454
DISK 2:     9454
DISK 3:     9454
DISK 4:     9454
DISK 5:     9454
DISK 6:     4216
DISK 7:     9454
DISK 8:     9454
DISK 9:     9454

FILE  #10
FILE LENGTH(TRACKS):     4624
TRACK BUFFER SIZE(TRACK): 1
SYSTEM STATUS BEFORE READING:
TIME: 12573.548  READ_TIME: 16020.000  R_LENGTH:      0.000
RD_PTR: 0    LENGTH: 6    WR_PTR: 7    LENGTH: 2
DISK 0:     9454
DISK 1:     9454
DISK 2:     9454
DISK 3:     9454
DISK 4:     9454
DISK 5:     9454
DISK 6:     0
```

```
DISK 7:    9454
DISK 8:    9454
DISK 9:    9454

TRACK BUFFER SIZE(TRACK): 1
SYSTEM STATUS BEFORE READING:
SYSTEM_TIME: 12600.039  READ_TIME: 16020.000  R_LENGTH:      0.000
RD_PTR: 0    LENGTH: 6    WR_PTR: 7    LENGTH: 2
DISK 0:    9454
DISK 1:    9454
DISK 2:    9454
DISK 3:    9454
DISK 4:    9454
DISK 5:    9454
DISK 6:    0
DISK 7:    9046
DISK 8:    9454
DISK 9:    9454

FILE  #11
FILE LENGTH(TRACKS):    9248
TRACK BUFFER SIZE(TRACK): 1
SYSTEM STATUS BEFORE READING:
TIME: 12893.558  READ_TIME: 16020.000  R_LENGTH:      0.000
RD_PTR: 0    LENGTH: 6    WR_PTR: 8    LENGTH: 1
DISK 0:    9454
DISK 1:    9454
DISK 2:    9454
DISK 3:    9454
DISK 4:    9454
DISK 5:    9454
DISK 6:    0
DISK 7:    0
DISK 8:    9454
DISK 9:    9454

TRACK BUFFER SIZE(TRACK): 2
SYSTEM STATUS BEFORE READING:
SYSTEM_TIME: 12900.126  READ_TIME: 16020.000  R_LENGTH:      0.000
RD_PTR: 0    LENGTH: 6    WR_PTR: 8    LENGTH: 1
DISK 0:    9454
DISK 1:    9454
DISK 2:    9454
DISK 3:    9454
DISK 4:    9454
DISK 5:    9454
DISK 6:    0
DISK 7:    0
DISK 8:    9252
DISK 9:    9454

FILE  #12
FILE LENGTH(TRACKS):    4624
```

```
DISK 3:    9454
DISK 4:    9454
DISK 5:    9454
DISK 6:    8840
DISK 7:    9454
DISK 8:    9454
DISK 9:    9454

FILE  #9
FILE LENGTH(TRACKS):    4624
TRACK BUFFER SIZE(TRACK): 1
SYSTEM STATUS BEFORE READING:
SYSTEM_TIME: 12000.225  READ_TIME: 11991.371  R_LENGTH:  1971.371
RD_PTR: 0    LENGTH: 6    WR_PTR: 6    LENGTH: 3
DISK 0:    9454
DISK 1:    9454
DISK 2:    9454
DISK 3:    9454
DISK 4:    9454
DISK 5:    9454
DISK 6:    4216
DISK 7:    9454
DISK 8:    9454
DISK 9:    9454

SYSTEM STATUS AFTER READING:
R_LENGTH:   1980.224
RD_PTR: 0    LENGTH: 6    WT_PTR: 6    LENGTH: 3
DISK 0:    9454
DISK 1:    9454
DISK 2:    9454
DISK 3:    9454
DISK 4:    9454
DISK 5:    9454
DISK 6:    4216
DISK 7:    9454
DISK 8:    9454
DISK 9:    9454

FILE  #10
FILE LENGTH(TRACKS):    4624
TRACK BUFFER SIZE(TRACK): 1
SYSTEM STATUS BEFORE READING:
TIME: 12573.548  READ_TIME: 16020.000  R_LENGTH:      0.000
RD_PTR: 0    LENGTH: 6    WR_PTR: 7    LENGTH: 2
DISK 0:    9454
DISK 1:    9454
DISK 2:    9454
DISK 3:    9454
DISK 4:    9454
DISK 5:    9454
DISK 6:    0
```

```
DISK 7:    9454
DISK 8:    9454
DISK 9:    9454

TRACK BUFFER SIZE(TRACK): 1
SYSTEM STATUS BEFORE READING:
SYSTEM_TIME: 12600.039  READ_TIME: 16020.000  R_LENGTH:     0.000
RD_PTR: 0    LENGTH: 6    WR_PTR: 7    LENGTH: 2
DISK 0:    9454
DISK 1:    9454
DISK 2:    9454
DISK 3:    9454
DISK 4:    9454
DISK 5:    9454
DISK 6:    0
DISK 7:    9046
DISK 8:    9454
DISK 9:    9454

FILE  #11
FILE LENGTH(TRACKS):    9248
TRACK BUFFER SIZE(TRACK): 1
SYSTEM STATUS BEFORE READING:
TIME: 12893.558  READ_TIME: 16020.000  R_LENGTH:     0.000
RD_PTR: 0    LENGTH: 6    WR_PTR: 8    LENGTH: 1
DISK 0:    9454
DISK 1:    9454
DISK 2:    9454
DISK 3:    9454
DISK 4:    9454
DISK 5:    9454
DISK 6:    0
DISK 7:    0
DISK 8:    9454
DISK 9:    9454

TRACK BUFFER SIZE(TRACK): 2
SYSTEM STATUS BEFORE READING:
SYSTEM_TIME: 12900.126  READ_TIME: 16020.000  R_LENGTH:     0.000
RD_PTR: 0    LENGTH: 6    WR_PTR: 8    LENGTH: 1
DISK 0:    9454
DISK 1:    9454
DISK 2:    9454
DISK 3:    9454
DISK 4:    9454
DISK 5:    9454
DISK 6:    0
DISK 7:    0
DISK 8:    9252
DISK 9:    9454

FILE  #12
FILE LENGTH(TRACKS):    4624
```

```
TRACK BUFFER SIZE(TRACK): 1
SYSTEM STATUS BEFORE READING:
SYSTEM_TIME: 13200.197  READ_TIME: 16020.000  R_LENGTH:      0.000
RD_PTR: 0    LENGTH: 6    WR_PTR: 8    LENGTH: 1
DISK 0:    9454
DISK 1:    9454
DISK 2:    9454
DISK 3:    9454
DISK 4:    9454
DISK 5:    9454
DISK 6:    0
DISK 7:    0
DISK 8:    4628
DISK 9:    9454

FILE  #13
FILE LENGTH(TRACKS):    9248
TRACK BUFFER SIZE(TRACK): 1
SYSTEM STATUS BEFORE READING:
TIME: 16500.277  READ_TIME: 16020.000  R_LENGTH:      0.000
RD_PTR: 0    LENGTH: 6    WR_PTR: 9    LENGTH: 0
DISK 0:    9454
DISK 1:    9454
DISK 2:    9454
DISK 3:    9454
DISK 4:    9454
DISK 5:    9454
DISK 6:    0
DISK 7:    0
DISK 8:    0
DISK 9:    9454

SYSTEM STATUS AFTER READING:
R_LENGTH:    934.340
RD_PTR: 0    LENGTH: 9    WR_PTR: 9    LENGTH: 0
DISK 0:    9454
DISK 1:    9454
DISK 2:    9454
DISK 3:    9454
DISK 4:    9454
DISK 5:    9454
DISK 6:    9454
DISK 7:    9454
DISK 8:    9454
DISK 9:    9454

TRACK BUFFER SIZE(TRACK): 1
SYSTEM STATUS BEFORE READING:
SYSTEM_TIME: 16800.027  READ_TIME: 16954.340  R_LENGTH:    934.340
RD_PTR: 0    LENGTH: 9    WR_PTR: 9    LENGTH: 0
DISK 0:    9454
DISK 1:    9454
DISK 2:    9454
```

```
DISK 3:    9454
DISK 4:    9454
DISK 5:    9454
DISK 6:    9454
DISK 7:    9454
DISK 8:    9454
DISK 9:    4834

FILE  #14
FILE LENGTH(TRACKS):    4624
TRACK BUFFER SIZE(TRACK): 1
SYSTEM STATUS BEFORE READING:
SYSTEM_TIME: 17100.100  READ_TIME: 16954.340  R_LENGTH:   934.340
RD_PTR: 0    LENGTH: 9    WR_PTR: 9    LENGTH: 0
DISK 0:    9454
DISK 1:    9454
DISK 2:    9454
DISK 3:    9454
DISK 4:    9454
DISK 5:    9454
DISK 6:    9454
DISK 7:    9454
DISK 8:    9454
DISK 9:    210

SYSTEM STATUS AFTER READING:
R_LENGTH:   1080.100
RD_PTR: 0    LENGTH: 9    WT_PTR: 9    LENGTH: 0
DISK 0:    9454
DISK 1:    9454
DISK 2:    9454
DISK 3:    9454
DISK 4:    9454
DISK 5:    9454
DISK 6:    9454
DISK 7:    9454
DISK 8:    9454
DISK 9:    210

FILE  #15
FILE LENGTH(TRACKS):    64736
TRACK BUFFER SIZE(TRACK): 1
SYSTEM STATUS BEFORE READING:
TIME: 17106.977  READ_TIME: 17100.100  R_LENGTH:  1080.100
RD_PTR: 9    LENGTH: 0    WR_PTR: 0    LENGTH: 8
DISK 0:    9454
DISK 1:    9454
DISK 2:    9454
DISK 3:    9454
DISK 4:    9454
DISK 5:    9454
DISK 6:    9454
DISK 7:    9454
```

```
DISK 8:     9454
DISK 9:     0

SYSTEM STATUS AFTER READING:
R_LENGTH:    1391.238
RD_PTR: 0     LENGTH: 0     WR_PTR: 0     LENGTH: 9
DISK 0:     9454
DISK 1:     9454
DISK 2:     9454
DISK 3:     9454
DISK 4:     9454
DISK 5:     9454
DISK 6:     9454
DISK 7:     9454
DISK 8:     9454
DISK 9:     9454

TRACK BUFFER SIZE(TRACK): 2
SYSTEM STATUS BEFORE READING:
TIME: 17413.672  READ_TIME: 17411.238  R_LENGTH:  1391.238
RD_PTR: 0     LENGTH: 0     WR_PTR: 1     LENGTH: 8
DISK 0:     0
DISK 1:     9454
DISK 2:     9454
DISK 3:     9454
DISK 4:     9454
DISK 5:     9454
DISK 6:     9454
DISK 7:     9454
DISK 8:     9454
DISK 9:     9454

SYSTEM STATUS AFTER READING:
R_LENGTH:    1704.812
RD_PTR: 0     LENGTH: 1     WR_PTR: 1     LENGTH: 8
DISK 0:     9454
DISK 1:     9454
DISK 2:     9454
DISK 3:     9454
DISK 4:     9454
DISK 5:     9454
DISK 6:     9454
DISK 7:     9454
DISK 8:     9454
DISK 9:     9454

TRACK BUFFER SIZE(TRACK): 2
SYSTEM STATUS BEFORE READING:
TIME: 17720.396  READ_TIME: 17724.812  R_LENGTH:  1704.812
RD_PTR: 0     LENGTH: 1     WR_PTR: 2     LENGTH: 7
DISK 0:     9454
DISK 1:     0
DISK 2:     9454
```

```
DISK 3:     9454
DISK 4:     9454
DISK 5:     9454
DISK 6:     9454
DISK 7:     9454
DISK 8:     9454
DISK 9:     9454

TRACK BUFFER SIZE(TRACK): 3
SYSTEM STATUS BEFORE READING:
TIME: 18027.111  READ_TIME: 17724.812  R_LENGTH:  1704.812
RD_PTR: 0    LENGTH: 1    WR_PTR: 3    LENGTH: 6
DISK 0:     9454
DISK 1:     0
DISK 2:     0
DISK 3:     9454
DISK 4:     9454
DISK 5:     9454
DISK 6:     9454
DISK 7:     9454
DISK 8:     9454
DISK 9:     9454

SYSTEM STATUS AFTER READING:
R_LENGTH:    1980.052
RD_PTR: 0    LENGTH: 1    WR_PTR: 3    LENGTH: 6
DISK 0:     9454
DISK 1:     8346
DISK 2:     0
DISK 3:     9454
DISK 4:     9454
DISK 5:     9454
DISK 6:     9454
DISK 7:     9454
DISK 8:     9454
DISK 9:     9454

TRACK BUFFER SIZE(TRACK): 3
SYSTEM STATUS BEFORE READING:
TIME: 18333.826  READ_TIME: 22020.000  R_LENGTH:      0.000
RD_PTR: 0    LENGTH: 1    WR_PTR: 4    LENGTH: 5
DISK 0:     9454
DISK 1:     8346
DISK 2:     0
DISK 3:     0
DISK 4:     9454
DISK 5:     9454
DISK 6:     9454
DISK 7:     9454
DISK 8:     9454
DISK 9:     9454

TRACK BUFFER SIZE(TRACK): 3
```

```
SYSTEM STATUS BEFORE READING:
TIME: 18640.545  READ_TIME: 22020.000  R_LENGTH:      0.000
RD_PTR: 0    LENGTH: 1    WR_PTR: 5    LENGTH: 4
DISK 0:    9454
DISK 1:    8346
DISK 2:    0
DISK 3:    0
DISK 4:    0
DISK 5:    9454
DISK 6:    9454
DISK 7:    9454
DISK 8:    9454
DISK 9:    9454

TRACK BUFFER SIZE(TRACK): 4
SYSTEM STATUS BEFORE READING:
TIME: 18947.252  READ_TIME: 22020.000  R_LENGTH:      0.000
RD_PTR: 0    LENGTH: 1    WR_PTR: 6    LENGTH: 3
DISK 0:    9454
DISK 1:    8346
DISK 2:    0
DISK 3:    0
DISK 4:    0
DISK 5:    0
DISK 6:    9454
DISK 7:    9454
DISK 8:    9454
DISK 9:    9454

TRACK BUFFER SIZE(TRACK): 5
SYSTEM STATUS BEFORE READING:
SYSTEM_TIME: 19200.408  READ_TIME: 22020.000  R_LENGTH:      0.000
RD_PTR: 0    LENGTH: 1    WR_PTR: 6    LENGTH: 3
DISK 0:    9454
DISK 1:    8346
DISK 2:    0
DISK 3:    0
DISK 4:    0
DISK 5:    0
DISK 6:    1652
DISK 7:    9454
DISK 8:    9454
DISK 9:    9454

FILE  #16
FILE LENGTH(TRACKS):    4624
TRACK BUFFER SIZE(TRACK): 1
SYSTEM STATUS BEFORE READING:
TIME: 19307.656  READ_TIME: 22020.000  R_LENGTH:      0.000
RD_PTR: 0    LENGTH: 1    WR_PTR: 7    LENGTH: 2
DISK 0:    9454
DISK 1:    8346
DISK 2:    0
```

```
DISK 3:     0
DISK 4:     0
DISK 5:     0
DISK 6:     0
DISK 7:     9454
DISK 8:     9454
DISK 9:     9454

TRACK BUFFER SIZE(TRACK): 1
SYSTEM STATUS BEFORE READING:
SYSTEM_TIME: 19500.455  READ_TIME: 22020.000  R_LENGTH:      0.000
RD_PTR: 0    LENGTH: 1    WR_PTR: 7    LENGTH: 2
DISK 0:     9454
DISK 1:     8346
DISK 2:     0
DISK 3:     0
DISK 4:     0
DISK 5:     0
DISK 6:     0
DISK 7:     6482
DISK 8:     9454
DISK 9:     9454

FILE  #17
FILE LENGTH(TRACKS):     9248
TRACK BUFFER SIZE(TRACK): 1
SYSTEM STATUS BEFORE READING:
TIME: 22620.611  READ_TIME: 22020.000  R_LENGTH:      0.000
RD_PTR: 0    LENGTH: 1    WR_PTR: 8    LENGTH: 1
DISK 0:     9454
DISK 1:     8346
DISK 2:     0
DISK 3:     0
DISK 4:     0
DISK 5:     0
DISK 6:     0
DISK 7:     0
DISK 8:     9454
DISK 9:     9454

SYSTEM STATUS AFTER READING:
R_LENGTH:    1908.008
RD_PTR: 0    LENGTH: 8    WR_PTR: 8    LENGTH: 1
DISK 0:     9454
DISK 1:     9454
DISK 2:     9454
DISK 3:     9454
DISK 4:     9454
DISK 5:     9454
DISK 6:     9454
DISK 7:     9454
DISK 8:     9454
DISK 9:     9454
```

```
TRACK BUFFER SIZE(TRACK): 1
SYSTEM STATUS BEFORE READING:
SYSTEM_TIME: 22800.059  READ_TIME: 23928.008  R_LENGTH:  1908.008
RD_PTR: 0    LENGTH: 8    WR_PTR: 8    LENGTH: 1
DISK 0:    9454
DISK 1:    9454
DISK 2:    9454
DISK 3:    9454
DISK 4:    9454
DISK 5:    9454
DISK 6:    9454
DISK 7:    9454
DISK 8:    6688
DISK 9:    9454

FILE  #18
FILE LENGTH(TRACKS):    4624
TRACK BUFFER SIZE(TRACK): 1
SYSTEM STATUS BEFORE READING:
SYSTEM_TIME: 24300.045  READ_TIME: 23928.008  R_LENGTH:  1908.008
RD_PTR: 0    LENGTH: 8    WR_PTR: 8    LENGTH: 1
DISK 0:    9454
DISK 1:    9454
DISK 2:    9454
DISK 3:    9454
DISK 4:    9454
DISK 5:    9454
DISK 6:    9454
DISK 7:    9454
DISK 8:    2064
DISK 9:    9454

SYSTEM STATUS AFTER READING:
R_LENGTH:   2280.045
RD_PTR: 0    LENGTH: 8    WT_PTR: 8    LENGTH: 1
DISK 0:    9454
DISK 1:    9454
DISK 2:    9454
DISK 3:    9454
DISK 4:    9454
DISK 5:    9454
DISK 6:    9454
DISK 7:    9454
DISK 8:    2064
DISK 9:    9454

FILE  #19
FILE LENGTH(TRACKS):    27744
TRACK BUFFER SIZE(TRACK): 1
SYSTEM STATUS BEFORE READING:
TIME: 24367.066  READ_TIME: 28020.000  R_LENGTH:      0.000
RD_PTR: 0    LENGTH: 8    WR_PTR: 9    LENGTH: 0
DISK 0:    9454
```

```
DISK 1:     9454
DISK 2:     9454
DISK 3:     9454
DISK 4:     9454
DISK 5:     9454
DISK 6:     9454
DISK 7:     9454
DISK 8:     0
DISK 9:     9454

TRACK BUFFER SIZE(TRACK): 2
SYSTEM STATUS BEFORE READING:
TIME: 24673.775  READ_TIME: 28020.000  R_LENGTH:      0.000
RD_PTR: 8    LENGTH: 0    WR_PTR: 0    LENGTH: 7
DISK 0:     9454
DISK 1:     9454
DISK 2:     9454
DISK 3:     9454
DISK 4:     9454
DISK 5:     9454
DISK 6:     9454
DISK 7:     9454
DISK 8:     0
DISK 9:     0

TRACK BUFFER SIZE(TRACK): 3
SYSTEM STATUS BEFORE READING:
TIME: 24980.520  READ_TIME: 28020.000  R_LENGTH:      0.000
RD_PTR: 8    LENGTH: 0    WR_PTR: 1    LENGTH: 6
DISK 0:     0
DISK 1:     9454
DISK 2:     9454
DISK 3:     9454
DISK 4:     9454
DISK 5:     9454
DISK 6:     9454
DISK 7:     9454
DISK 8:     0
DISK 9:     0

TRACK BUFFER SIZE(TRACK): 3
SYSTEM STATUS BEFORE READING:
SYSTEM_TIME: 25200.234  READ_TIME: 28020.000  R_LENGTH:      0.000
RD_PTR: 8    LENGTH: 0    WR_PTR: 1    LENGTH: 6
DISK 0:     0
DISK 1:     2682
DISK 2:     9454
DISK 3:     9454
DISK 4:     9454
DISK 5:     9454
DISK 6:     9454
DISK 7:     9454
DISK 8:     0
DISK 9:     0
```

```
FILE  #20
FILE LENGTH(TRACKS):    4624
TRACK BUFFER SIZE(TRACK): 1
SYSTEM STATUS BEFORE READING:
TIME: 25374.309  READ_TIME: 28020.000  R_LENGTH:     0.000
RD_PTR: 8    LENGTH: 0    WR_PTR: 2    LENGTH: 5
DISK 0:    0
DISK 1:    0
DISK 2:    9454
DISK 3:    9454
DISK 4:    9454
DISK 5:    9454
DISK 6:    9454
DISK 7:    9454
DISK 8:    0
DISK 9:    0

TRACK BUFFER SIZE(TRACK): 1
SYSTEM STATUS BEFORE READING:
SYSTEM_TIME: 25500.264  READ_TIME: 28020.000  R_LENGTH:     0.000
RD_PTR: 8    LENGTH: 0    WR_PTR: 2    LENGTH: 5
DISK 0:    0
DISK 1:    0
DISK 2:    7512
DISK 3:    9454
DISK 4:    9454
DISK 5:    9454
DISK 6:    9454
DISK 7:    9454
DISK 8:    0
DISK 9:    0
```

.

SELECTED BIBLIOGRAPHY

[1] G. J. Ammon and J. A. Calabria, "Operational Performance of Optical Disk Systems," SPIE, Vol. 529, Optical Mass Data Storage, 1985, pp.131-137.

[2] G. J. Ammon, "An Optical Disk Jukebox Mass Memory System," private communication.

[3] G. M. Claffie, "Optical Disk Recorders for Operationally Demanding Mass Storage Applications," private communication.

[4] C. M. Weng and Y. H. Yaun, "CDC Cyber 170/172 System Performance Evaluation," 17th Annual Simulation Symposium, 1984, pp.193-208.

[5] T. A. Shull and B. A. Conway, "Spaceborne Optical Disk Controller Development," SPIE, Vol. 695, Optical Mass Data Storage, 1986.

[6] Notes from "Photonics-Based Flight System Workshop," NASA Langley Research Center, May 1987.

[7] H. C. Lucas, Jr., "Performance Evaluation and Monitoring," Computing Surveys, Vol. 3, No. 3, 1971, pp.79-91.

[8] J. A. Payne, Introduction to Simulation: Programming Techniques and Methods of Analysis, McGraw-Hill Book Company, New York, N.Y., 1982.

[9] J. L. Baer, Computer Systems Architecture, Computer Science Press, Inc., Potomac, Md., 1980.

[10] D. J. Kuck, The Structure of Computers and Computations, Vol. 1, John Wiley & Sons, Inc., New York, N.Y., 1978.

[11] B. W. Kernigham and D. M. Ritchie, The C Programming Language, Prentice-Hall, Inc., Englewood Cliff, N.J., 1978.

Date Due

DEC 2 0 [illegible]			