A SIMULATION-BASED DECISION SUPPORT SYSTEM FOR FINITE POPULATION MAINTENANCE LOGISTICS

by

Mark Steven Gallion

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Industrial Engineering and Operations Research

APPROVED:

M. H. Agee, Chairman

A. Chandawarkar

R. C. Heterick, Jr.

16 December 1986

Blacksburg, Virginia

A SIMULATION-BASED DECISION SUPPORT SYSTEM FOR FINITE POPULATION MAINTENANCE LOGISTICS

by

Mark Steven Gallion

M. H. Agee, Chairman

Industrial Engineering and Operations Research

(ABSTRACT)

This thesis presents a decision support system, MAINTAIN, that was developed to assist maintenance and logistics planners in the evaluation of alternative strategies that might be used in the maintenance of a finite population of end items. The modeled population must consist of a finite number of identical end items, each at age zero at the beginning of the analysis. End items are modeled at their component level, with the lifetime of each component described by a probability distribution. These components are assumed to fail independently of each other and to cause end item failure if any one component should fail. Several component-level preventive maintenance policies are integrated into the modeling system and include age-based replacement, opportunistic look-ahead replacement, or no replacement until failure.

Also incorporated into the model is the use of two repair facilities, which function as constraints on the resources available to perform maintenance tasks. One repair facility is used under normal conditions,

while the second can be used when the first facility is operating beyond its capacity.

MAINTAIN is a simulation-based modeling system, in that a proposed or actual maintenance system is captured as an input data set and its operation is simulated over the specified planning horizon. Time-series output statistics are produced by the model to allow the analysis of time dependent parameters such as population availability, maintenance costs, spares requirements and repair facility utilization. Output data are available in several forms for use in external analyses.

ACKNOWLEDGEMENTS

Throughout the pursuit of my master's degree, I have experienced many doubting moments as to whether or not continuing on was worth the effort and frustration being experienced at the time. But during these trying times I would relate back to my thoughts prior to returning to Blacksburg that went: "You know you are going to say, 'I have had enough!', many times, but you know you want to do it." Thus, being a person that places learning high on the list of life's goals, I decided to stick it out. I would now like to thank those individuals that made the completion of my studies possible.

I will start off with a "blanket" thank you to cover all of my friends and fellow students. These people provided a type of assistance that cannot be attributed to any single person, yet it was very important in determining the outcome of my stay in Blacksburg.

After that catch-all, the first person on my list would have to be my major advisor, Dr. Marvin Agee. Without his constant "nagging", I would probably have never returned to school full-time in the first place. And he has been more than just an advisor that provides technical guidance, but a friend as well. I would like to thank him for all his support.

Second on my list of contribution makers, is a fellow student, Mr. Chell Roberts. Without one of Chell's suggestions, I might not have discovered

one of the more important data structures used in the MAINTAIN system. He also functioned as an idea sounding board in relation to simulation related matters. Thank you Chell for your time and assistance.

I would also like to thank the other members of my thesis committee, Dr. Aseem Chandawarkar and Dr. Robert Heterick. The outcome of my research work would have certainly been lacking without their suggestions concerning system design elements and implementation style.

Lastly, I cannot forget both of my parents. Without their assistance and encouragement, all of my efforts would have been in vain. Thanks, Mom and Dad.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

1.0 CHAPTER 1. INTRODUCTION

In today's world of increasing productivity and cost awareness, preventive maintenance plays an important role in decreasing long term costs and/or increasing the reliability/readiness of equipment on which it is performed. It is obvious that the preventive maintenance function must be performed correctly for benefits to accrue. That is, excessive preventive maintenance results in high cost and low availability of equipment whereas too little or no preventive maintenance can result in a higher rate of equipment failures with, again, high cost and low availability. Thus, effective preventive maintenance management depends upon determining the components, sub-assemblies, etc. on which preventive maintenance is to be performed and when to perform such maintenance.

1.1 PROBLEM STATEMENT

There have been many attempts to create models with which to predict maintenance requirements for complex systems (or end items). These models have usually addressed a specific system and relied on strict mathematical models. This approach results in a compromise of model flexibility, and the models are only useful to the developers, leaving businesses without skilled development resources to fend for themselves. Logistics planners and maintenance managers of different sized organizations need some tool that allows the prediction and analysis of life-cycle costs and measures for an existing or planned population of complex systems. These costs

and measures consist primarily of spares requirements and system availability, while other costs and measures are derived from these two basic items. The systems can be anything from a fleet of nuclear destroyers to a fleet of fork-lift trucks, but the underlying concepts and requirements are the same.

With the cost of computer memory and power constantly decreasing, it seems reasonable that an alternative (or complementary) method for computing population end item maintenance requirements is a simulation model. Simulation modeling techniques are frequently used when problems become quite complex, beyond the capabilities of mathematical modeling. Simulation gives a model the item-by-item tracking ability required for preventive maintenance modeling (and emergency repair maintenance requirements) of a population of end items. Further, simulation provides the capability to analyze multiple preventive maintenance policies without major restructuring of the model, whereas if a purely mathematical model is used, each different preventive maintenance policy can require major restructuring of the model.

There are other benefits in using a simulation-based model for population maintenance requirements planning. As a stand-alone system, a simulation model can provide assistance in determining the life-cycle cost estimates that are performed before a major population begins operation. Once a population of items is in operation, the simulation model continues to operate as a planning tool. By maintaining a file on the status of end items (and their respective components) in the populations, a simulation

model can be executed using current data, predicting future requirements on the basis of past performance history. This past performance history can include updated information on the failure characteristics of components, replacing the estimates that may have been used in pre-operation runs.

The simulation model can be envisioned as a single module in a "total" maintenance planning system. Since maintenance data is seen as a resource in an increasing number of organizations, it is now being kept more accurately than ever. Systems currently exist which produce maintenance schedules and keep records on actual maintenance actions taken. By providing the means with which to automatically update the data required in the simulation model, the model becomes an important decision support tool for the maintenance planner, allowing "what if?" games to be played using alternative preventive maintenance (PM) policies and component failure characteristics. The drawing in Figure 1 illustrates a possible configuration of such an integrated system.

1.2 OBJECTIVES

The intent of this research was to develop an interactive decision support system (DSS) to aid maintenance managers and logistics planners in determining maintenance requirements and outcomes for some general start-up population of complex end items (analysis begins with each end item at age zero). The end items must be identical in composition (component parts are the same) and have their reliability described by the probabi-

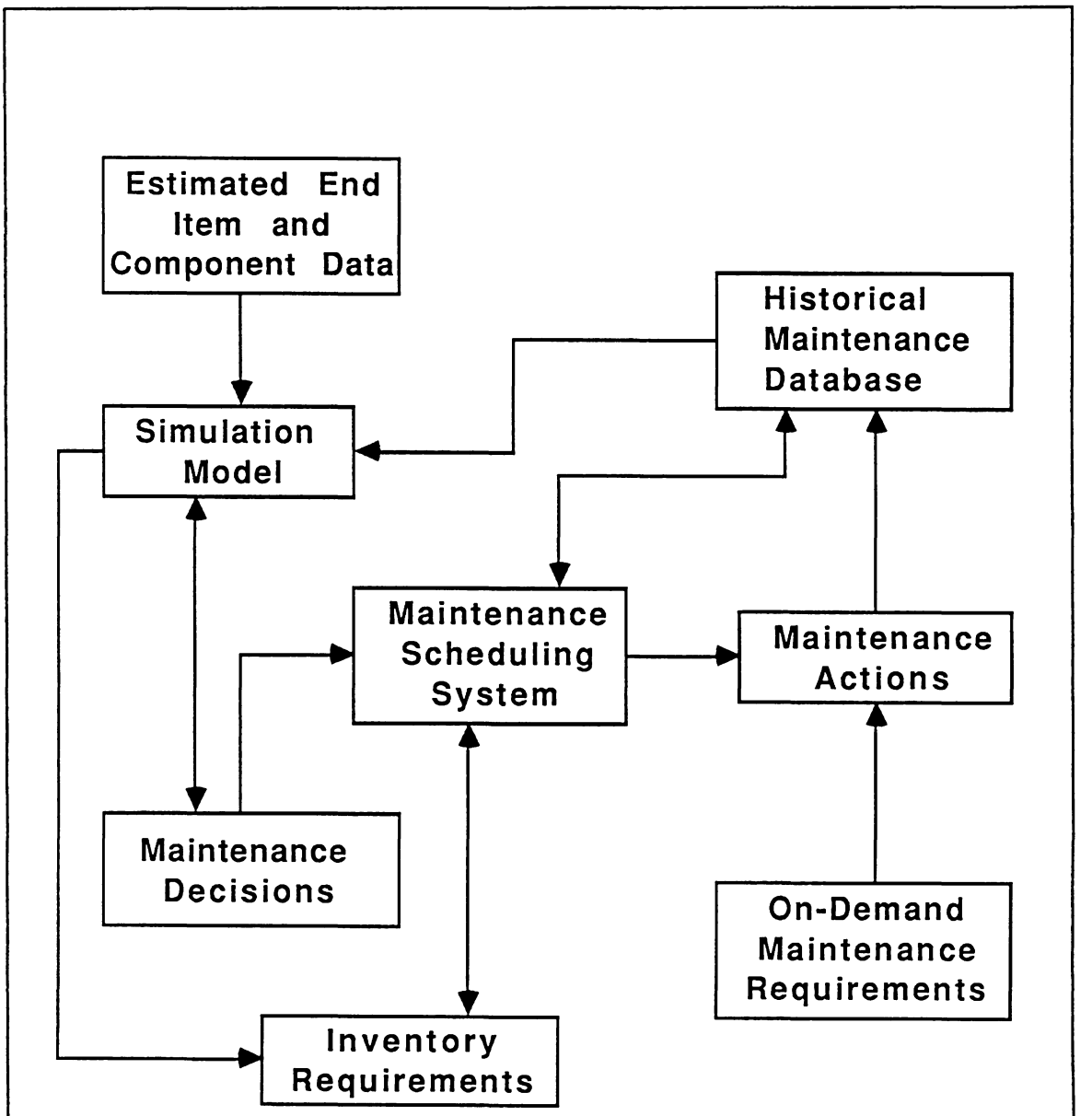

Figure 1. An Integrated Maintenance System

listic failure distributions of the component parts. The population of end items is then modeled at the component level.

The primary component of the DSS is a next-event driven simulation model that describes the population under consideration. There were several levels of objectives considered during the design of the simulation system. The first level of objectives dealt with general system goals. The following list illustrates these objectives:

1. Create a general microcomputer implementation using the C language.
2. Provide a system to work with a general population of end items.
3. Create an "open" system, able to fit the larger schema discussed above.
4. Not to optimize, but to analyze the effect of varying system parameters.

The second level of objectives dealt more with specifics on the type of execution options the system would supply to the user, and included:

1. The ability to perform scan-ahead opportunistic PM on components.
2. A choice of various Failure Distributions for each component.
3. Alternative Levels of Repair (LOR).
4. Limiting of resources available at each LOR.
5. Alternative Modes of Fielding (MOF) the population.

Another, and final, class of objectives concerned the output generated by the modeling system. In a simulation model of this type the most interesting statistic is time-series based. For this reason the statistics presented by the system fit into this category. There are many separate statistics that could have been collected during the simulation and the following list is by no means all inclusive, but represents those time-series statistics that are presented to a user:

1. Instances of Preventive Maintenance and Emergency Repair (ER) for each component.
2. Server utilization at each LOR.
3. Availability of the population.
4. Total cost of the maintenance program.
5. Downtime instances for the population.
6. Other cost drivers.

1.3 DOCUMENT OVERVIEW

Chapter 2 presents a review of pertinent literature that relates simulation modeling to maintenance procedures and analysis. The next chapter, Chapter 3, presents a conceptual description of the developed simulation model. The chapter includes background information on maintenance modeling, important definitions, and discusses the major features of the simulation model, including assumptions and limitations. Chapter 4 follows and presents the underlying technical aspects of the model. It mainly covers the data structures and algorithms used within the model.

Chapter 5 concludes the document and presents a set of recommendations for future research and the thesis conclusions. The closing chapter is followed by the bibliography and two appendices. Appendix A contains a user's guide for the software that was developed to implement the model, and Appendix B contains a listing of the program source code.

2.0 CHAPTER 2. LITERATURE REVIEW

The field of maintenance modeling is very large in scope; a review of the literature revealed hundreds of references. Therefore, this literature review will concentrate on the specific subject under investigation - the simulation of maintenance logistics and requirements. For a thorough review of maintenance modeling in general, the reader is referred to Luxhoj [12], Pierskalla and Voelker [16], and Sherif and Smith [17]. This review will be divided into two parts. The first part of the chapter concerns maintenance modeling efforts that have used simulation methods, while the second part attempts to justify the method of simulation in the maintenance field.

2.1 MAINTENANCE SIMULATION MODELS

An interesting phenomenon surfaced while researching these simulation models - the level of sophistication did not increase in any chronological order. In fact, most of the models seemed to possess about the same level of complexity. For this reason, the models will be presented in chronological order, from 1969 to 1983.

The earliest simulation model discovered also appears to be one of the more sophisticated. Hixson [7] presented a model developed for the U.S. Air Force, in which a combination of discrete and continuous simulation was used to study the effects of alternative maintenance policies. Mul-

tiple units of equipment formed a population, with each unit described by a probabilistic failure distribution. The population age was described by a probability distribution function, and treated as a whole. A continuous simulation generated failures within some time frame, and the number of failures was passed to the discrete portion, which allocated manpower and spares to the repairs necessary. The measure of effectiveness of the maintenance policy under study was considered to be the total cost of maintaining the population. A total of 370 man-hours were used during the development of the model.

Soon after the paper by Hixson, Keeney [10] presented the GOAL modeling system, while Dougherty and Hazlett [3] presented MARS. Both of these models were developed within the Boeing Company and appear to be very similar in construction. In both models, military aircraft sorties were simulated and the entire maintenance cycle was simulated (including the recycling of failed parts). Resource availability was used as a constraining factor in the simulation. Manpower and spares requirements were determined, as was system availability. The maximum time spans of the simulations were approximately 1.5 years for GOAL, and two months for MARS.

Following closely behind the previous three models, Widawsky [18] outlined a next-event simulation model to assist in maintaining high reliability during manned space flight missions. Widawsky's model concerns only a single complex system consisting of several subsystems, and a fixed repair crew size that allowed only one man per repair. A primary con-

sideration in the model was the priority associated with critical subsystems. If a higher priority critical subsystem failed while another was under repair, the repair activity was shifted to the higher priority system. If the total repair downtime during a repair session exceeded the maximum allowable downtime, then the mission was deemed to have failed. The probability distribution of mission success and subsystem reliability was the most important model output.

The above four models were developed in the late 1960's and early 1970's. An interesting aspect to notice about these models is that they were directed toward military and government applications. It seems that little work was done in this area over the next few years and when simulation models began to reappear in the literature (about 1978), they were dominated by commercial applications.

Manpower requirements planning is a continually important topic within the manufacturing environment. Williams [19] developed a GPSS simulation model to evaluate the manpower required in the maintenance of a fiber processing facility. The skill levels and occupations required for various maintenance tasks were defined and the maintenance calls of the facility were simulated, based on empirical distributions of incoming calls per hour for different types of repairs (including scheduled maintenance). The model was executed by specifying a manpower configuration and then simulating the effectiveness of such a configuration. These measures included:

- proportion of call-in jobs serviced immediately
- jobs completed during eight hour shift
- average job waiting time
- utilization of manpower

In another commercial application of simulated maintenance modeling, Czajkiewicz [2] described a next-event model in which maintenance policies for industrial equipment were tested. The simulation took place with a single piece of equipment comprised of multiple critical parts with known failure distributions. A set of maintenance policies were input to the model and the policies which allowed acceptable production levels were output. From the output set, the policy with the lowest cost was selected. The costs used consisted of lost production time, idle time, spare parts, and manpower. Czajkiewicz referred to the resulting output policy as an "optimal" policy. No actual optimization was performed during the simulation, but was dependent on the particular set of input policies used in the analysis.

The maintenance management of vehicle fleets occurs in many diverse organizations. Maze, Dutta, and Kutsal [13] stressed the importance of quantitative analysis in maintenance decision making for an urban bus transportation system. The purpose of the analysis was to provide increased or adequate bus service given less operating funds. In a related publication, Maze, Jackson, and Dutta [14] presented a simulation model to perform that analysis. Model detail was sparse, but the primary objective was to predict future maintenance requirements in the presence

of alternative maintenance forces. This "what if?" game allowed the transportation planner to identify a plan of attack that required a minimum amount of funds while maintaining an acceptable level of bus service. The modeling effort was prompted by an objective stated in the 1980 Fiscal Transportation Appropriation Act, which required life-cycle costing of proposed transportation systems.

Each of these previous modeling efforts were developed in either the FORTRAN or GPSS programming languages, and were implemented on mainframe computers (with the exception of the bus maintenance model, where detail was omitted from the journal articles). The models were developed for a specific application system and do not appear to be general simulation models.

2.2 IN SUPPORT OF MAINTENANCE SIMULATION

In an attempt to justify the use of simulation modeling efforts for the identification of maintenance requirements, several sources are noted. These sources do not provide overwhelming support, but do suggest acceptance of the technique, noting that each model discussed previously was developed in lieu of a strict mathematical model.

The earliest found reference to the use of simulation modeling in the maintenance function came from Kelly and Harris [11]. They suggest that simulation modeling can definitely assist the maintenance manager in a decision support role, providing the tools with which to determine optimum

maintenance policies and analyze the effects of varying parameters and policies on the performance of a system.

McGrath and Henry [15] presented support from the military side of maintenance and logistics planning. The U.S. Department of Defense issued a directive in 1979 that required logistic supportability to be on equal terms with cost during weapon system acquisition. The use of simulation methods has therefore increased in this area due to the timeliness with which an analysis can be prepared. The U.S. Air Force and Navy also routinely use simulation models to determine manpower requirements and the effects of policy changes on logistics support.

Probably the strongest support for the use of simulation modeling in maintenance requirements planning comes from Pierskalla and Voelker [16]. In the conclusion of their article on maintenance and replacement models which surveyed what were essentially strict mathematical models they state:

> "The area requiring the most future work is the study of multiechelon, multipart interaction maintenance models. This area is perhaps the most difficult to handle mathematically ... Often the only way these problems can be handled is by simulation; consequently, few general results will be obtained in the near future."

3.0 CHAPTER 3. METHODOLOGY

This chapter presents the conceptual development and description of the MAINTAIN system. The chapter first covers some background necessary in understanding the concepts of maintenance, then proceeds to discuss the model and its assumptions and limitations.

The primary objective of this research was to develop a simulation model to analyze maintenance requirements for some population of end items. Therefore, the simulation model was the major focus of the research, but a thorough understanding of maintenance (particularly preventive maintenance) was first required, since the model needed to incorporate alternative maintenance policies for the components of an end item.

3.1 MAINTENANCE MODELING

Preventive maintenance is the function of replacing some component(s) of an end item in an attempt to achieve continued performance without unexpected failures and to prolong the economic life of the end item. For example, replacing the oil in a motor at some point when the oil begins to "wear out" or replacing a nut when it begins to loosen (tightening a nut can be considered as replacing the nut in its correct position) are considered typical preventive maintenance actions. Preventive maintenance might also be concerned with inspections and testing to determine the state of some component, but the focus of this thesis is only on the re-

placement actions associated with maintenance requirements planning and preventive maintenance.

3.1.1 Failure Distributions

The time at which replacement or maintenance should occur is determined by the failure distribution of the component. This distribution describes the probability of failure of the component versus its age (it might also be thought of as a "lifetime" or "time between failure" distribution). Failure distributions are typically represented by standard distributions such as the normal, exponential, lognormal, and Weibull distributions, or by an empirical distribution if none of the standard functions work well with available failure data. An example of the cumulative normal failure distribution is shown in Figure 2. F(t) is the probability that component failure occurs before time t.

These failure distributions are the driving data for maintenance requirements planning and are not always easy to determine or obtain. The most common source of this data is from in-house maintenance records or from the original equipment manufacturer (OEM).

3.1.2 Hazard Rates

An important parameter (in relation to preventive maintenance) of any failure distribution function is a measure called the failure rate, or the hazard rate. The hazard rate, as defined by Jardine [8], is the

probability that a component will fail in the next time interval given that it is operating normally at the beginning of the interval. The instantaneous hazard rate is defined as $r(t)=f(t)/[1-F(t)]$, where $f(t)$ is the density function and $F(t)$ is the cumulative function of the failure distribution of the component. As an example, consider the normal distribution and its corresponding hazard rate function as plotted in Figure 2.

As another example, the exponential distribution has a hazard rate function that is constant, equalling the reciprocal of the mean. The nature of the hazard rate of the exponential distribution, and the hazard rates of other failure distributions, has important consequences for determining preventive maintenance policies.

Hazard rates can be of three types: decreasing, increasing, or constant. Certain time intervals of the distribution can be characterized by different types of hazard rates. If the hazard rate for a component is decreasing or constant over some time interval, there will be no advantage in performing preventive maintenance on that component because the probability of failure in the next instant is decreasing or constant. On the other hand, if the hazard rate is increasing, then there is an opportunity to apply preventive maintenance policies advantageously. For example, the exponential distribution has a strictly constant hazard rate. In this case, at any time during the life of a component, replacing the component will not change the probability of failure for the component in the next time interval. This is an interesting fact because, according to

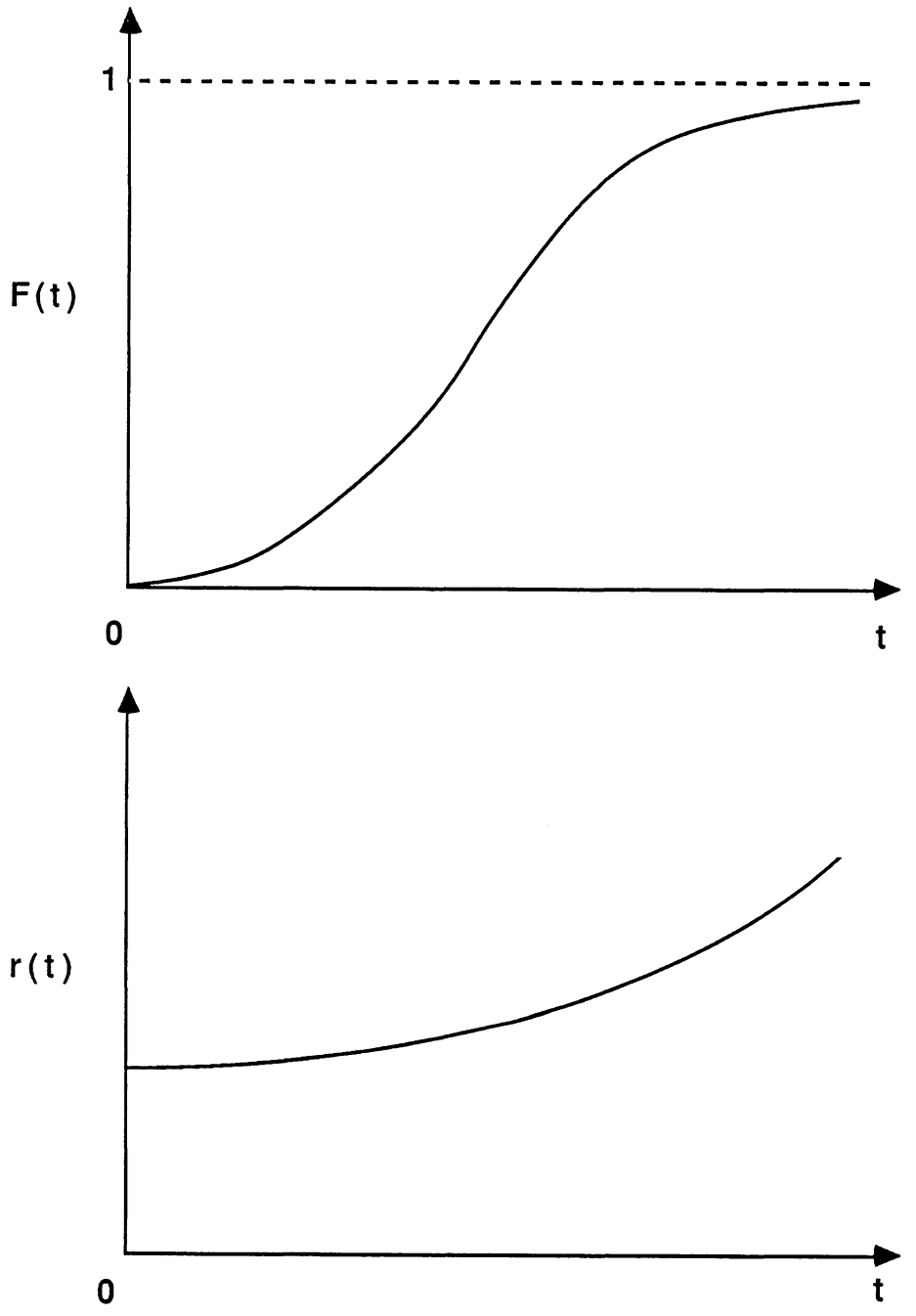

Figure 2. Normal Probability Distribution and Hazard Rate

Jorgerson, McCall and Radner [9], it has been shown that many items (as well as complex systems) exhibit exponential failure behavior. On the other hand, if failure characteristics follow a normal distribution, the hazard rate is strictly increasing and there exists an opportunity to obtain an economic benefit by performing some type of preventive maintenance.

3.1.3 Preventive Maintenance Policies

The literature in the field of preventive maintenance describes many different policies, the most common type being simple periodic replacement, whereby preventive maintenance is performed on the component, regardless of its age, at the end of some specified time interval. Another type of preventive maintenance policy, which is a modification of the simple periodic policy, is the age replacement policy. This policy operates in the following manner: if a component fails before some age T, it is replaced by a new component at the time of failure, otherwise the old component is replaced at age T, regardless of its condition. These two policies are intended for single item components and not for systems of multiple components. Policies become much more complicated when large systems are considered, but the concepts are similar.

Opportunistic policies describe a type of preventive maintenance that considers systems of components. With this type of policy, when a particular component fails or when regularly scheduled preventive maintenance is performed (no component failures), a decision must be made

whether to replace other system components or not. This policy often requires subjective evaluations of the various system components. Another more complicated type of preventive maintenance policy as described by Gertsbakh [5], which is not necessarily for systems of components, considers groups of similar components. The operation of this policy is as follows:

> There are two critical times, t and T, with t < T. If any single component fails in the interval (0,t) then replace that component. If any single component fails in the interval (t,T) then replace that component and every other operating component, or at time T perform preventive maintenance on all components. The time at which all components are replaced, t1, becomes the new starting point for the policy.

The above discussion is intended to convey some general concepts of preventive maintenance policies that are available to a logistics planner. This serves to illustrate the obvious complexity which can arise when large systems of components are considered and an attempt is made to model these systems.

3.1.4 Preventive Maintenance Models

An attempt at system modeling allows predictions to be made concerning the actual operation of the system. The modeling of a typical single component preventive maintenance policy, on the basis of a Markov chain process observing the state of the component (as suggested by Gertsbakh), is now considered. There are three states involved in this particular Markov process: (1) the component is operating normally, (2) the component is undergoing preventive maintenance (PM) and (3) the component is

in "emergency repair" (ER) following an unexpected failure. The diagram in Figure 3 shows a possible realization of this process for some sample component.

The most important thing to note in this realization is that preventive maintenance consumes less time, and therefore less cost, than an emergency repair. If this time differential does not exist, then preventive maintenance should not be performed.

As was stated above, preventive maintenance can always be considered as a "replacement" problem if any spare parts are needed. An important consequence of this fact is that the used/failed component is replaced by a "new" component, that is, the failure distribution describing the lifetime of the component regenerates back to time zero. Bringing the distribution back to time zero might not always be a feasible assumption. However, for all components for which preventive maintenance is performed, the probability of component failure in the succeeding time period must be lower than for the period in which preventive maintenance was performed. The Markov chain realization in Figure 4, illustrates the above concept; where each state, E(k), represents the age, k, of the component, and E(-1) represents failure of the component.

At each state a decision is made whether to do nothing or to perform preventive maintenance. If the state observed is failure, the component is replaced and returns to state E(0). This regeneration of states must

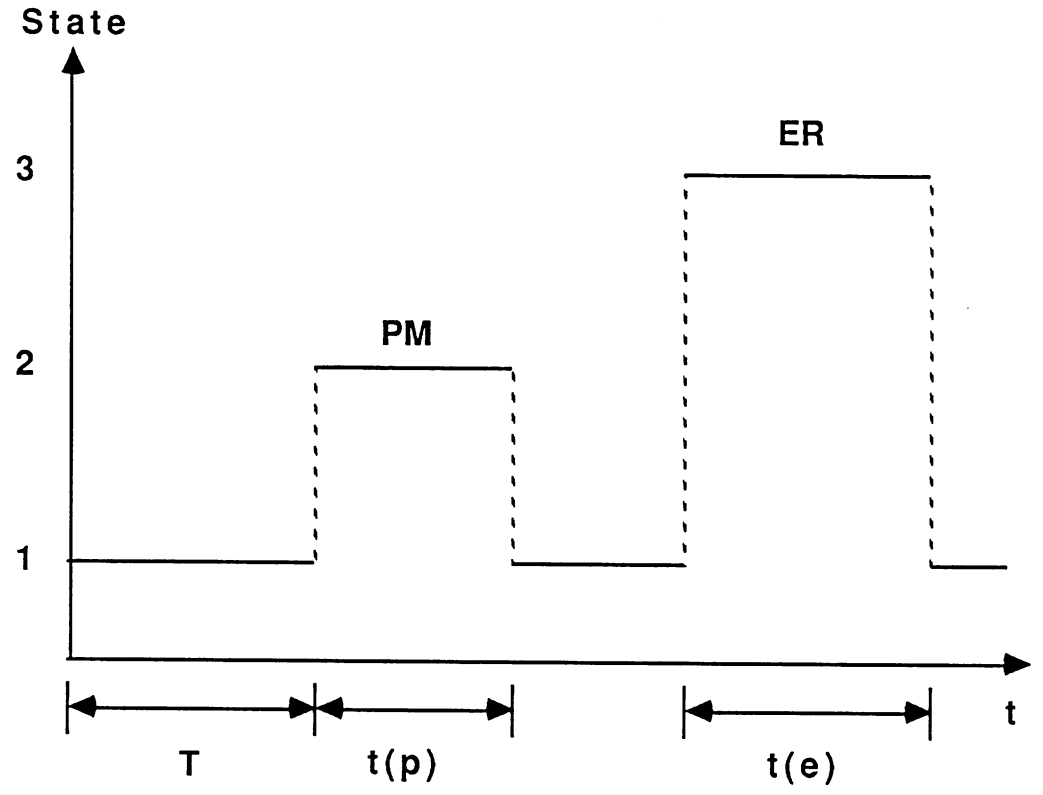

Figure 3. Realization of the Maintenance Process

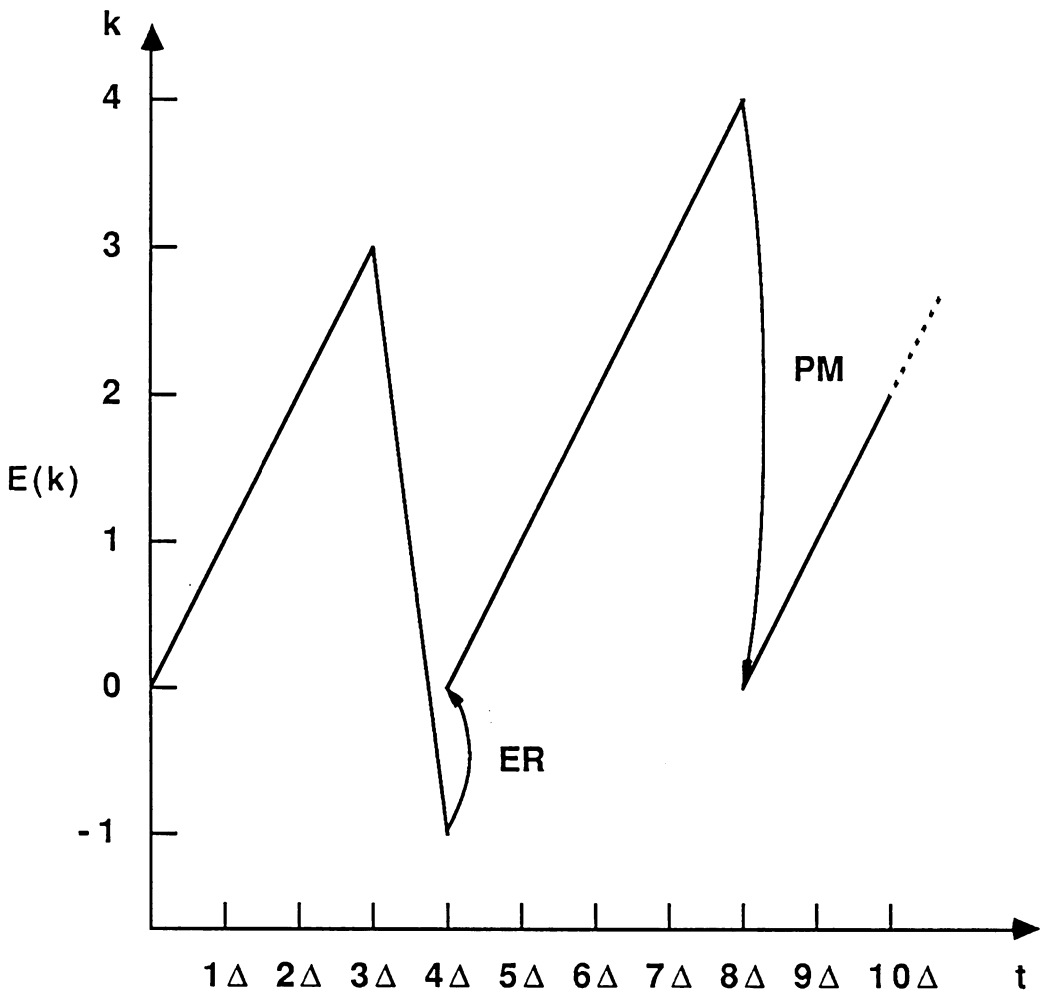

Figure 4. Regeneration in the Maintenance Process

occur in some form in a model for any preventive maintenance policy to produce beneficial results for the component under consideration.

3.2 IMPORTANT DEFINITIONS

The terminology used in the maintenance management field varies widely depending on the source of information used. Therefore, to make the material contained in the rest of this document easier to comprehend, a list of terms, definitions and abbreviations used in this thesis is presented on the following pages.

EI End Item, some entity which contains one or more critical components for which failure data is available; e.g., a fork lift truck, a computer system, an automobile.

Population A group of one or more identical end items; e.g., a fleet of fork lift trucks, a computer system, a fleet of automobiles.

LOR Level of Repair, the "location" at which a maintenance action is performed; "location" being a physical location or one class of repair facilities.

STD_LOR Standard Level of Repair, the level of repair at which maintenance actions are normally performed; usually has the lowest cost associated with it.

AUX_LOR Auxiliary Level of Repair, the level of repair at which maintenance actions are performed under extraordinary conditions; might be thought of as an off-site facility or possibly overtime work.

ER Emergency Repair, implies that an unscheduled maintenance action is required, e.g., on component failure.

PM Preventive Maintenance, implies that a scheduled maintenance action is performed.

Downtime Elapsed time between the time of failure occurrence(s) or the beginning of preventive maintenance action(s) for some end item and the time at which the end item rejoins the operating population.

MOF Mode of Fielding, the manner in which the population of end items is built up; could be a single block procurement at time zero, or a procurement schedule over time.

OPM Opportunistic Preventive Maintenance, preventive maintenance actions for some EI which are grouped together due to the fact that they occur in close proximity to each other; usually means moving some scheduled preventive maintenance actions ahead in time.

3.3 THE MAINTAIN SIMULATION MODEL

This section of the document presents the basic conceptual elements of the developed simulation model; the more detailed design elements are discussed in Chapter 4. The MAINTAIN system began with the notion that there should be a way to model some population of generic EIs that would permit the evaluation of alternative PM policies by generating unconstrained manpower and spares requirements on the basis of the use of a specified set of policies. The model evolved during the development

stages into a decision support system with more flexibility than originally conceived, mostly due to the addition of constrained maintenance LORs. The remaining portion of this chapter discusses the decisions made regarding the operation and features of the model.

3.3.1 Model Description

MAINTAIN is a next-event driven simulation model which allows the evaluation of alternative PM policies for a population of generic EIs. The model produces output results by generating several realizations of the operation and maintenance of the population of EIs, and then summarizes across all realizations. This type of statistics generation is known as the "method of replications" and requires at least thirty realizations for the output statistics to be considered valid. This requirement is a consequence of the Central Limit Theorem from probability theory. To generate each realization in a next-event fashion, the following logic is employed:

1. Generate a lifetime for each component of each EI in the population.
2. For each lifetime, determine if an ER or PM event will occur, and the time at which the event occurs.
3. Select the earliest event time in the entire set of times.
4. If the selected event time is greater than the time to stop the simulation, summarize statistics for the realization just completed and go to step 8, else proceed with the next step.

5. Process the event based on the event type, perform either ER or PM tasks (or both, if events get grouped).
6. Collect statistics on the actions performed.
7. Generate a new lifetime and an event to replace the event selected in step 3.
8. If all realizations have been generated, stop the simulation, else go to step 1.

In order for a simulation of the maintenance process to be performed, some associations must be made. Within the framework of the MAINTAIN model, the simulation entities are the components that comprise an EI, but in essence, a set of components are tied together to form an EI. That is, the components of an EI actually determine the behavior of the EI, and the behavior of the set of all EIs determines the population behavior. Although the primary focus of the model is on the population and the components that comprise the EIs, it is easiest to conceptualize the simulation system in terms of EIs.

The flow of EI entities within the simulation system is shown in Figure 5. Simplifications in the figure are necessary due to complex interactions that occur when a PM event is scheduled and the queue at the desired LOR is non-empty. In this case, the EI remains operational in the population until the time for its maintenance occurs or an ER event occurs in the EI. These interactions are discussed in more detail in the "Processing Options" section of this chapter.

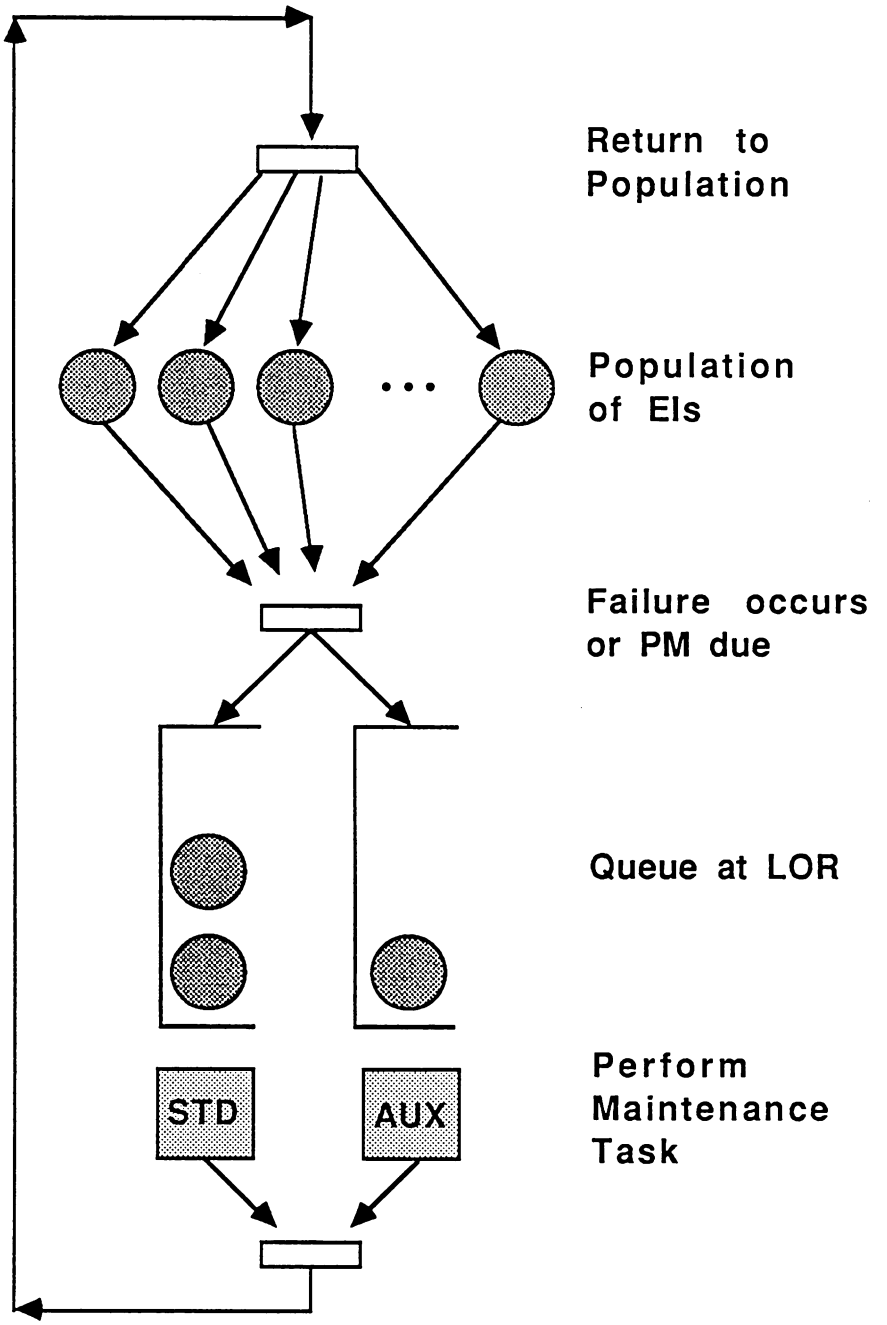

Figure 5. Flow Diagram of EI Entities

3.3.1.1 Input Requirements

The model requires a predefined set of input parameters which describe the population of EIs under consideration; in addition, several decision parameters must be provided to specify the characteristics of maintenance operations. The following outline, divided into classes, summarizes the general input requirements for any MAINTAIN model, after which it will be appropriate to elaborate on some of the input parameters and the method by which they are determined:

1. Component Data (for each component in the EI)
    a. Failure Distribution and Parameters
    b. ER and PM Repair Times
    c. ER and PM Spares Costs
    d. PM Policy Parameters
2. EI Data
    a. MOF for Population
    b. Downtime Cost
3. LOR Data (for both STD and AUX_LOR)
    a. Capacity
    b. Travel Costs
    c. Resource Usage Costs
    d. Method of Using Multiple Servers
4. Simulation Data
    a. Time Unit for Simulation
    b. Length of Simulation Run

c. Number of Realizations to Generate

d. Statistic Collection Parameters

e. Processing Option Flags

3.3.1.1.1 Component Input Data

An input analysis begins by considering the EI in question and determining the components which are critical to its operation. This breakdown of EIs into components is often done using the concept of a maintenance tree as presented by Brammer [1], which assumes vertical dependency along branches of the tree and horizontal independence between branches. Figure 6 shows an example of a maintenance tree which has three levels of nodes. The top node is the EI itself. Intermediate nodes are usually subassemblies which consist of one or more components. The highlighted nodes are end, or component level, nodes. The failure of any one of these end nodes causes the subassembly to fail, therefore causing the EI to fail. For this reason, end nodes constitute valid components for input into the model. Essentially, the tree should be developed such that the end nodes are those components or subassemblies for which failure probability distributions can be determined. Stated another way, the behavior of an EI is determined by, and modeled at, its end node components or subassemblies, with intermediate nodes being non-essential in the MAINTAIN system's analysis.

Assuming that the data are available that allow estimation of the failure probability distributions, or times between failures, for each of these

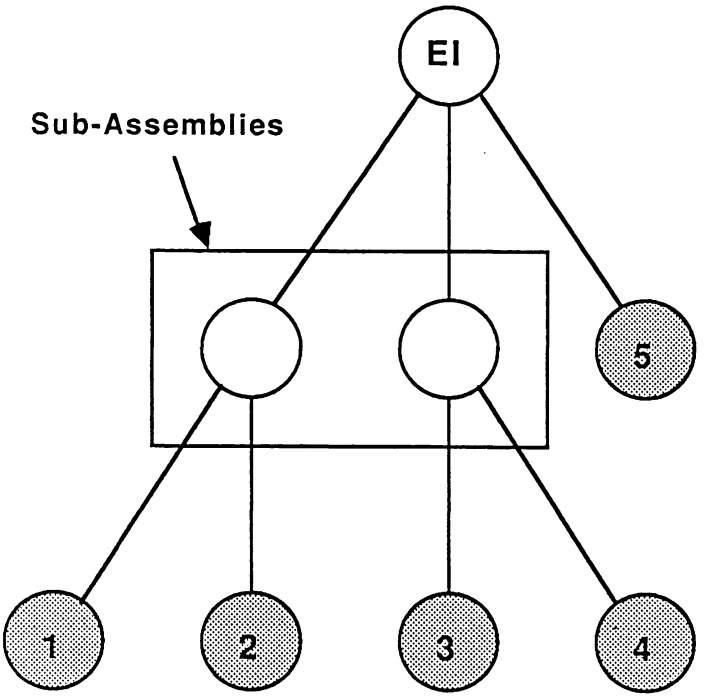

Figure 6. Sample Maintenance Tree

end node components, then that distribution is selected as input to the model. The specific probability distributions available within the MAINTAIN system include:

- Normal
- Lognormal
- Exponential
- Weibull
- Gamma
- Uniform
- Erlang-k
- Empirical

The next consideration is the associated maintenance policies and costs for each component.

There are two types of PM policies available within MAINTAIN: an age-based replacement policy or no replacement until failure. Age-based policies operate on the premise that a component is replaced when it has operated for some predetermined period of time. The consequences of this age-based policy are discussed in the "Assumptions/Limitations" section of this chapter. Two parameters are used to specify the PM policy used for any component:

1. A flag that indicates whether replacement will be performed or not.
2. The time interval, or age, at which replacement occurs.

It should be noted at this time that the MAINTAIN system provides no assistance in estimating the initial PM parameters to be specified for a component. There is also no analysis performed as to whether PM should even be performed. These analysis tasks are left to the user.

The only additional data needed to describe any component are the repair times and spares costs associated with ER and PM tasks. Spares costs refer to the costs of all spares that are required to perform an ER and a PM. Repair times are deterministic and assumed to consume one unit of capacity of the LOR to which the action is assigned. Repair time parameters are specified for both ER and PM for two reasons:

1. If a failure occurs, it might be necessary to perform a diagnostic survey to determine which component failed. Therefore, the average repair times required for some ER tasks increase.
2. There might be different sets of spares used for ER and PM, causing a variation for their respective repair times (of course, this would probably result in a contradiction of the assumption listed at the end of this chapter, which specifies that components regenerate to age zero after any maintenance event).

3.3.1.1.2 EI Input Data

The specification of the population of EIs requires two types of input data. The first is the MOF that builds the population. There are two methods by which the population of EIs can be built:

1. Only an initial block procurement at time zero.
2. An initial block at time zero, plus a time-phased procurement schedule.

The schedule method can represent any predetermined MOF: uniform, variable, geometric, gradient, and others. The second data required concerns the cost of EI downtime. This value is specified in dollars per time unit downtime, and can include any measure of downtime cost that is desired, subjective and/or objective.

3.3.1.1.3 LOR Input Data

The next set of required input data concerns the description of each LOR, both the STD_LOR and AUX_LOR. The capacity of an LOR is the number of servers, or units of resource, available to perform maintenance tasks. Each server or unit can be assigned to one ER or PM task at a time. In the event that there is some distance between the EI location and the LOR, the input requirement specifies a one- way travel time to the LOR; a value of zero is appropriate most of the time, with either the LOR resource coming to the EI or the time being insignificant. This travel time contributes to the total downtime experienced during an EI maintenance event.

Whether or not there is a travel time associated with maintenance tasks performed at a LOR, there might be a fixed cost incurred each time there is interaction between a LOR and an EI. This "travel cost" is also a part

of the input specification describing each LOR. The value is assumed to be input as a one-way cost.

Each LOR has an input data requirement that specifies one of the two methods in which the servers can process an EI which has a group, or set, of components which require maintenance. These methods are referred to as "gang" and "sequential" repair. Gang repair lets more than one server work on the group of tasks, assigning the earliest available server to each component which requires maintenance. On the other hand, sequential repair assigns the first available server to perform all tasks in the group. The server works on the first task, then the second, and so on, until all tasks are complete. If the capacity of the LOR is greater than one, and grouped tasks occur during the simulation, then gang repair produces shorter net downtimes. Each LOR can have a different, or the same, method associated with it.

3.3.1.1.4 Simulation Input Data

Lastly, the parameters that control the execution of a simulation model must be specified. The first parameter required is the time unit in which the simulation clock increments during execution. This unit specifies the reference time unit associated with all other input parameters and also controls text associated with certain output data. The time units allowed in the MAINTAIN system are the following:

- Hours

- Days
- Weeks
- Months
- Years

The second simulation parameter specified is the time frame, or planning horizon, of the simulation. The time frame begins with time zero and ends at the time specified.

As was discussed previously, some number of realizations must be generated in order to collect valid statistics. This number is an input requirement associated with the simulation and can take on any value desired, recalling that the higher the number, the better the statistics. Since the statistics collected in the MAINTAIN model are time-series oriented, a time interval must be specified that sets the length of each time-series period. The time interval specified should be a meaningful multiple, greater than one, of the simulation time unit. The planning horizon specified must also be taken into consideration, as it drives the total number of periods in the resulting time-series. Cautions relating to interactions among these simulation parameters can be found in "Appendix A. MAINTAIN User's Guide."

The last piece of input data required as a simulation parameter concerns the model's processing options. These input data are in the form of selection flags that are either turned on or off. The flag values can be

mixed to provide multiple combinations of the available options. These options are discussed in the next section of this thesis.

3.3.1.2 Processing Options

There are two processing options which can be specified to control the way in which the model executes. The first option allows usage of the concept of OPM, and the second allows the AUX_LOR to be considered when necessary.

3.3.1.2.1 OPM Option

When the OPM option is used, a segment of the MAINTAIN simulation system is activated and attempts to find potential OPM events. OPM events are determined by looking at all other component PM events associated with the EI of the current component event. The current event time marks the beginning of a scan, and the end of the scan is determined by the "look-ahead" time, an input simulation parameter, which is added to the starting scan time to produce an absolute time after which no PM events will be investigated. Any PM events for the EI which fall within the scanning range are grouped together and sent to the desired LOR at the same time.

There is one other type of OPM scanning that is performed for an EI regardless of the OPM option setting. This "automatic" OPM will occur when the current event is PM and there is a wait at the LOR to which the event is assigned. In this case, the event is assigned a service appointment

at the LOR and the EI remains in operation until the appointment time occurs. If the current component or any other component in the EI fails (an ER event occurs or a PM event turns into a failure) during the wait time, then the EI becomes non-operational and downtime begins. Any component events that come due during the wait or before downtime begins are performed as a group at the appointment time. Event attributes for each component in the group reflect the state of that component at the time downtime begins. This automatic OPM reflects the logical outcome of event interactions of the preceding type. As an example, suppose an auto repair shop gives John Doe an appointment to bring his car in for a scheduled fuel pump replacement. In the meantime, he realizes that the engine's oil needs changing and, in addition, the front brakes fail. The brake failure causes the car to be non-operational, and more than likely, the repair shop cannot take the car until the appointment date. When John finally brings the car in he would have all necessary services and repairs performed.

3.3.1.2.2 AUX_LOR Option

If the AUX_LOR option is used, the simulation system will investigate whether to perform the current maintenance task(s) at the STD or AUX_LOR on the basis of some trade-off criteria. This investigation occurs only if there is a non-zero queue length at the STD_LOR and the event in question is an ER, since the STD_LOR is assumed to be the preferred LOR, with lower usage and travel costs, and a PM event does not cause failure of the EI. If a PM event becomes an ER event during a wait, then the

AUX_LOR is addressed at that time. When the AUX_LOR is to be considered, the simulation pseudo-schedules the maintenance action(s) at each LOR and determines which LOR results in a lower cost, then actually schedules the event(s) at the selected LOR. The total cost of a maintenance action is determined by the following equation:

Total Cost = EI Downtime Cost + LOR Travel Cost +
LOR Usage Cost + Spares Cost

Although the AUX_LOR has a higher usage cost and might have a higher travel cost, the downtime incurred is usually less, in spite of any additional travel time to the AUX_LOR being included in the downtime. This aspect of the comparison causes the supplied cost input data to be very important and notes to this affect are summarized below:

1. EI downtime cost per time unit is probably the major cost driver for the population. For this reason, all possible costs should be considered when determining the downtime cost rate.
2. LOR usage cost rates can be another large cost driver and their values should also be carefully considered.
3. Capacities of each LOR are important. If the queues grow large at the STD_LOR then the AUX_LOR will become feasible more often as waiting increases at the STD_LOR. Then, as queues lengthen at the AUX_LOR, the scheduling preference will return to the lower cost STD_LOR.

3.3.1.3 Model Output

As was introduced earlier, all statistics collected during a model's simulation take the form of a time-series, with each period being composed of a pre-defined number of simulation time units. Data overlaps between periods are reflected in the statistic; time durations that span periods are broken up and placed in the period in which they occur. A set of "raw" statistics is collected while the simulation executes each realization, with the time-series for each statistic saved in its entirety. Thus, the output from a simulation consists of a set of time-series point statistics for each realization generated.

When the analysis phase of MAINTAIN is started, a single set of statistics is generated from the set of all realizations. The statistics for each period in the time-series are determined with a summation across realizations. The values for the mean, maximum, minimum, and standard deviation are computed for most of these statistics. The set of time-series statistics maintained as raw statistics are summarized in the following list:

- For the entire population of EIs.
    1. Downtime incurred.
    2. Number of downtime instances to each LOR.
    3. Service time used at each LOR (no max, min, or std dev).
- For each EI component type.
    1. Number of ER events to each LOR.

2. Number of PM events to each LOR.

This set of raw statistics is used in combination with the original input data of the model to create "built" statistics. The built statistics are formed from the raw time-series statistics in several different ways. These include adding time-series data together, multiplying and dividing by model input data to perform conversions on the raw data, and various combinations of these operations. Only the mean value of a "built" statistic is available in the MAINTAIN analysis system, although it would have been possible to provide maximums, minimums and standard deviations, if the original simulation output were re-read for each new statistic desired. It was felt that the mean value provided for more timely use of the Analyze system, and if more extensive data is required, it can be generated externally. The built statistics available in the Analyze system include:

- For the entire population of EIs.
  1. Availability, expressed as the percent of time the EIs in the population were not experiencing downtime.
  2. Total downtime.
  3. Cost of downtime.
  4. Cost of spare parts.
  5. Cost of LOR resource usage.
  6. Total cost of maintenance (including travel costs).
- For each LOR.

1. Utilization, expressed as the percent of time the entire capacity of the LOR was busy.

- For each EI component type.
  1. Number of ER events that occurred.
  2. Number of PM events that occurred.

These built statistics are available for viewing as graphic charts or as printouts to paper or file. The raw statistics, which contain a larger quantity of data, are available only as printouts to paper or file. Since the MAINTAIN system is intended to be a decision support system, the capability to extend certain analysis features has been provided by allowing access to, or the creation of, any files necessary to perform additional analyses.

3.3.2 Model Assumptions/Limitations

No modeling effort of the size of MAINTAIN is ever completed in a reasonable amount of time without relying on some set of assumptions, simplifying or otherwise. Some of the assumptions discussed in this section may be considered limiting, others may seem completely logical, but most were decided on while keeping in mind the original reason for the development of this model - analysis of alternative PM policies for critical components in a homogenous population of identical EIs. The following list presents the assumptions made in the formulation of the MAINTAIN modeling system and briefly expands on any reasons and limitations:

1. Components regenerate to age zero after an ER or PM action has been performed - This is probably the most limiting assumption made in the model formulation. This is basically a simplifying assumption that does not force the requirement of some function(s) that would relate the age of the component with its maintenance times, policies, ages, regeneration points, and new failure distributions.
2. Components that are not a contributory part of EI downtime do not age during downtime - This assumption caused the most problems in the model development stages; letting all components age during downtime would have been implicit, but it was felt that in most applications this assumption would be realistic. The additional development time to allow both methods for any component would not have been worth the extra bit of realism that would have been provided in the model.
3. Fatal failures and retirement are not considered for EIs - This assumption might be construed as a severe model limitation, but the intent of the model was to analyze PM for components and the performance of the population on the basis of those PM policies. It was assumed that the probability of an EI fatal failure was negligible in most applications. EI retirement was not considered due to the assumption that components regenerate back to age zero after maintenance. Since the EI is considered to be composed entirely of its components, and the components are continuously regenerating, it would not make sense to say that the EI has reached a retirement age. The only type of retirement that might be logical within the scope of this model, would be in the case of debilitating technological

changes, and that would require the retirement of the entire population, ending the analysis.

4. Rate of resource usage at each LOR is one per unit of repair time - This assumption became a necessary part of the model due to the method of modeling the repair facilities. There might be cases when more that one server is required for a maintenance task, but an average of more than one server for the entire duration of the task is probably uncommon. The only other way to handle resource usage requirements of more than one at a time would have been to require the number specified for the entire duration of the repair. As stated above, this method seemed more illogical than the assumption which was made.
5. LOR queues are treated using the First In, First Out (FIFO) queueing discipline - This assumption was made in order to simplify modeling the LOR, but it is not overly limiting. Often, in the actual performance of maintenance tasks, whether it be an ER or PM, priorities are often assigned to certain EIs. This type of policy results in EIs being relocated within a queue. Applications modeled with the MAINTAIN system are assumed to have identical EIs, each having equal priorities. Thus, the extra priority that might have been assigned to an EI would not be useful within the context of this system. On the other hand, it might be useful to expedite the maintenance of an EI which experiences an ER event, over an EI which is waiting for PM, which would reduce the overall population downtime. If this is the case, then the MAINTAIN system could be used to conceptually model this type of maintenance system. The user's model input data would determine the feasibility of changing EI "priorities" while a simu-

lation processes. By this statement, it is meant that the AUX_LOR could be viewed as an overflow facility which is used for priority repairs. If the input data is such that EI downtime cost is high, then an ER event might be shuttled off to the AUX_LOR for a more immediate repair (provided the total cost at the STD_LOR is greater than the total cost at the AUX_LOR, with waiting included). But this outcome does depend on the input data; if the model data is such that this does not occur, then it is probably more feasible to just let the EI wait its turn for repair at the STD_LOR.

6. Service times are deterministic - This assumption is limiting and could have been avoided by requiring extra input data for each component. The data requirements are already difficult to obtain and it was felt that since the maintenance tasks associated with each component do not vary, that the variation of the service time would be small, thereby approaching a constant value.
7. Only a "start-up" population can be used - This assumption is a simplifying assumption which limits the amount of input data required to specify a model. The modeling system is already capable of handling existing populations, but age-based input data would be required for each component of each EI in the population, and that was deemed excessive within the scope of this thesis.
8. Age-based PM is the only replacement policy other than no replacement until failure - Most other types of PM policies operate on some type of population-wide (or subset of) block replacement of components. These types of policies are less expensive to administer and plan, but go against the basic concept of finding the appropriate time at

which to perform maintenance, since age drives the probability that the component will fail. The age-based policies result in the most cost-effective PM policy with respect to spares and downtime. Thus, the most cost effective PM policy is the only logical policy to include in this type of modeling effort.

9. EI components fail independently of each other - This is both a simplifying and limiting assumption which allowed the modeling effort to be performed. Incorporation of dependent components was beyond the scope of the model. In addition, the input data to support the incorporation of dependent components would probably be very difficult, or impossible, to obtain.

3.3.3 Model Uses

Once an EI has been described in terms of its components, the MAINTAIN system offers many opportunities for the analysis of alternative operating scenarios. If it is used for modeling a population of EIs which have pre-specified maintenance requirements and facilities, one particular set of input data which most closely reflects the actual requirements will be all that is needed. Using the MAINTAIN system in this way would occur during estimating or budgeting tasks. On the other hand, if the goal for using MAINTAIN is to improve an existing or proposed maintenance system, there are many variables which can be manipulated.

A large part of the input data set is relatively static: downtime costs, repair times, spares costs, and probability distributions. Certain sets

of other variables might be manipulated to investigate the effects of particular changes, such as:

1. Change the PM interval of components to evaluate alternate PM policies, or turn PM on/off for certain components.
2. If using OPM, vary the look-ahead time interval.
3. If flexibility exists in building the population from zero EIs, vary the MOF schedule to evaluate start-up costs
4. Evaluate changes in the capacities the LORs.
5. Vary combinations of the above parameters.

The model, as a generic maintenance planning tool, would be useful in many evaluation tasks. Some of these tasks are large in scope and the focus for MAINTAIN would be in its use as a supporting tool, but these tasks could include:

- Life-cycle maintenance costing
- Inventory planning
- Manpower planning
- Budgeting/bidding
- General maintenance planning

An interesting by-product of the way in which the modeling system operates is that a valid population can be as small as one EI, which in turn can be composed of as few as one component. With this concept in mind, the use of a modeling system like MAINTAIN becomes feasible for use with

single item populations that exist in almost every organization, provided the necessary input data can be obtained. The LOR features of MAINTAIN would probably not be useful in this context, but alternative PM strategies can be evaluated for the single EI, and the availability and costs of the maintenance program analyzed.

4.0 CHAPTER 4. MODEL DESIGN

This chapter presents the technical constructs used in the design of the MAINTAIN simulation system. This includes the overall program structure of the system, a description of the system files and data structures, and a discussion of underlying algorithms that are critical to the operation of the system. However, this chapter contains no reference to actual use of the MAINTAIN system. For further information on using the system, the reader is referred to "Appendix A. MAINTAIN User's Guide".

The modeling system was implemented for an IBM PC (or compatible) with a color graphics card and monitor and a minimum of 512k memory. The system is user-interactive and once the modeling environment is entered, all steps necessary for performing a simulation and analysis session can be carried out without leaving the system, with the exception of a few file maintenance tasks which would not occur in normal use.

4.1 IMPLEMENTATION LANGUAGE

Once the general model concepts were decided upon, the computer language with which to implement the system was chosen. There were essentially two classes of languages to choose from:

1. A commercial simulation programming language (SPL).
2. A lower level, structured general purpose programming language (GPL).

Both classes had their respective benefits: development time would be minimized by using a commercial simulation language, while flexibility would be maximized if a low-level programming language were used. There were two characteristics of low-level languages that tended to bias the decision in that direction. Simulation languages are relatively large in size, consuming a large amount of memory, and usually having slow execution times, while low-level languages provided better memory management opportunities with much faster execution. In addition, many simulation languages do not produce executable files (meaning that the language used must be available to an end user). If a low-level language were to be used, the model would be distributable to any user in an executable form, requiring only the proper hardware.

This issue seemingly decided, a low-level language was chosen. The "C" language seemed to have gained a large following at the present time, mostly due to its portability and increasing support by microcomputer software developers. The "C" language supported a wide variety of data structures and contained many other features, in addition to having execution speeds rivaling assembly language programming. There are also many pre-programmed software libraries supporting the "C" language. These libraries would speed up the development process and allow more time to be spent on the design of the system. Further support for the use of "C" came from Grant and MacFarland [6] (of Pritsker and Associates, Inc., supporters of the simulation language SLAM), who concluded that "C" is very well suited to simulation modeling, and hinted that a future simulation language might be written in the "C" language. It was surmised

that the vast flexibility of this language might help reduce the impact of obstacles that would be encountered during the system development stages. For these reasons, the "C" programming language was chosen as the development language for this research.

4.2 PROGRAM STRUCTURE

There are three separate programs and several Input/Output (I/O) files that comprise the MAINTAIN simulation system. Figure 7 shows the relationships among these system elements. The first program, called Maintain, is what might be referred to as a "driver" program, and needs little discussion. Its sole purpose is to allow selection of one of the two main sub-systems of the MAINTAIN system. These two sub-systems are the Analyze and Simulate programs. The Simulate program reads an input data file which describes the population under consideration, executes a simulation using that data, and produces an output file containing the simulation results. The Analyze program reads and summarizes the output files created by Simulate. Analyze produces printed or file output of all its available data.

4.2.1 Main Sub-Systems

The Simulate program performs the model data entry and simulation functions. Model data is created in a user interactive process, and is saved to a specified file when complete. Once entered, the data can be edited and re-saved in the same or new file, or temporarily edited for the pur-

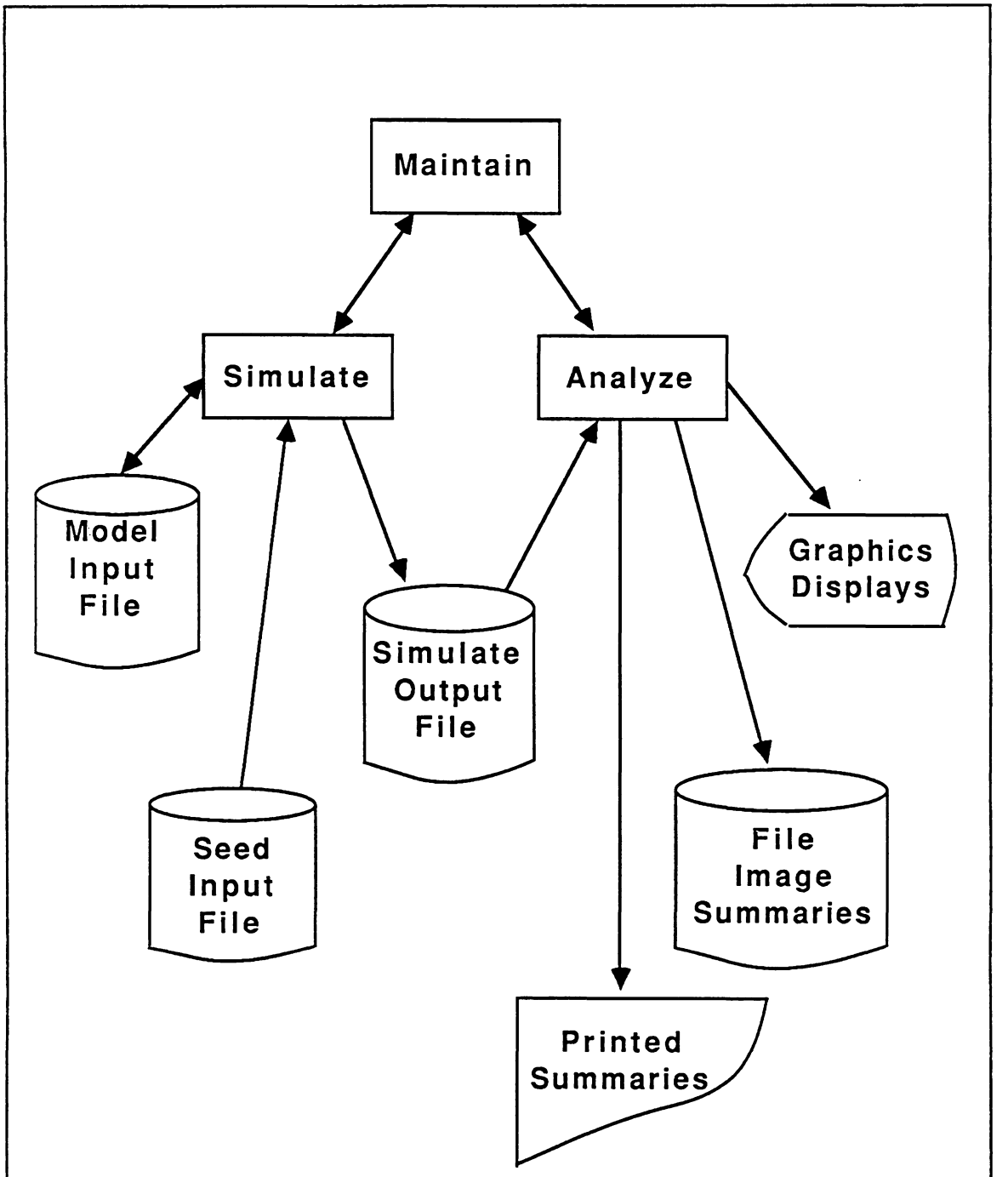

Figure 7. Structure of the MAINTAIN Simulation System

pose of executing the new model data to create an output file for analysis. When the simulation is executed for a set of model data, several file specifications must be determined: the random number seed file and output file must be validly specified before the simulation begins. These files are discussed below.

The Analyze program performs analyses of the output file(s) created in the Simulate program. Once an output file is created, the Analyze program is executed and that output file is specified as input into Analyze. The statistics contained in the output file are read and summarized into internal lists of time-series statistics. These statistics are in turn translated into the built statistics which are available for viewing within Analyze. The original model input data, the intermediate statistics and the built statistics are all available as printed output. In addition, the built statistics are available as graphic output in the form of scrollable line graphs which present the mean statistic value for each period in the time series.

4.2.2 Input/Output Files for Simulate

The multiple I/O files generated by the MAINTAIN system will be described in this section. This information would be useful to anyone desiring to expand upon or modify the features available within MAINTAIN, and also helps to clarify the operation of the system.

4.2.2.1 Model Data Input File

The model input file is a simple ASCII text file created by using either the built-in routines of MAINTAIN or with a text editor. The layout of the input file is predefined but varies slightly based on the type of data entered. Figure 8 shows the basic layout of the input file.
As shown in the figure, the data in the file is ordered as follows:

1. Population Data
2. Simulation Data
3. LOR Data
4. Data for each component in the EI

A single data value is located in each record of the file, representing a single field of input data, but there are two exceptions to this rule. An empirical probability distribution specification for a component or an MOF schedule for the population each requires a two element list of data for their description. This list is expected if either of these two lists are specified in a prior field. These lists can have any number of records, but the Simulate program expects a single value of "-1" in the record following the last record of the list.

4.2.2.2 Seed Input File

Each separate realization for a set of model data to be generated by the MAINTAIN system should have a different value with which to "seed" or

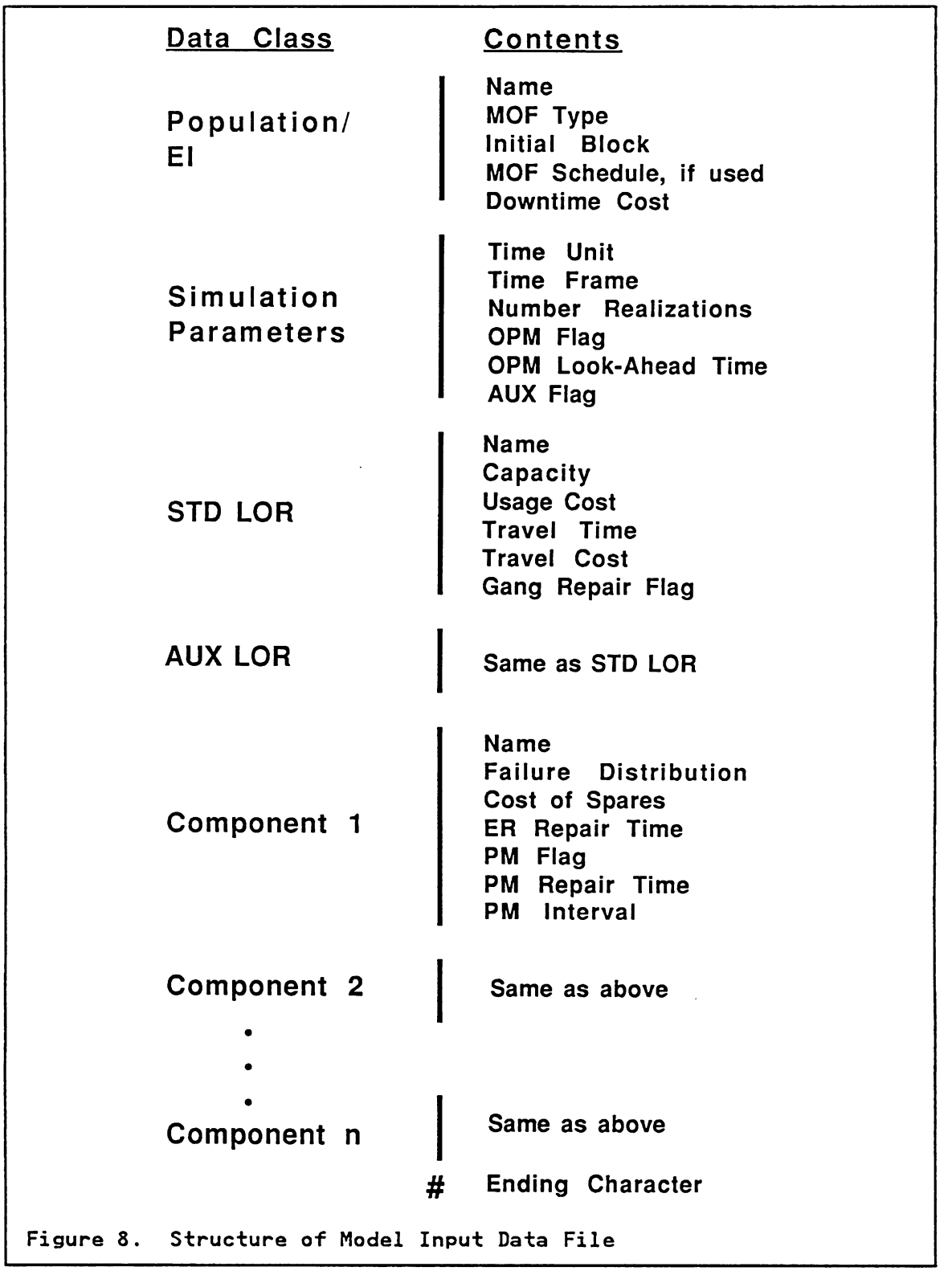

Figure 8. Structure of Model Input Data File

"start up" the random number generator that creates samples of component lifetimes. Otherwise, if the same seed is used, the same results will be obtained for each realization. If a new seed is not used, the random number generator might cycle and begin reproducing the same stream of random numbers. Both of these cases are undesirable. It is also desirable to use the same set of seed values when evaluating alternative maintenance scenarios for some application. For this reason, the MAINTAIN system expects to be provided with a simple input file which contains at least one seed value for each realization to be generated. The seed input file is a ASCII text file with one seed value per record and can be created using any conventional text editor. The file may contain as many seeds as desired; the system will only use the number it needs.

4.2.2.3 Simulate Output File

The output file produced by the Simulate system contains a large quantity of data. The first set of data contained in this output file is an echo of the original model data that drove the simulation. This redundant data is saved because a user might change certain parameters for a maintenance scenario and possibly not save this data, yet that data must be known in order for the Analyze system to produce its results. A special character, "#", follows the echoed data to indicate the beginning of the statistics for the first realization.

The "#" record is followed by a realization header record, beginning with a "*", which separates each set of realization statistics and identifies

the realization number and the random number seed which applied to that realization. Each realization set is composed of the time-series data collected from the simulation. These data are collected in two logical steps throughout the execution of a simulation model. The first step collects two classes of time-series data that relate to the behavior of the particular realization being generated. One class of data pertains to those period statistics which concern the entire population of EIs, and include for each period of each realization (followed by its abbreviation), the:

- Downtime incurred (Down)
- Number of trips to the STD_LOR (Down-S)
- Number of trips to the AUX_LOR (Down-A)
- Average number of EIs (Avg Num EIs)

This output record is labeled as a "D" record and contains the period number (Per) and upper limit, or breakpoint (Brkpt), of the absolute simulation time associated with that period, in addition to the above statistics.

It should be noted that the "D" and "L" records, and the time periods associated with each record, are in no particular order within the output file. The time periods are approximately in increasing order within each realization set, but the flag value that specifies when the data are written out is driven by the current event set being handled within the simulation. Since this event set can include times in the future, there

is no guarantee on the ordering of the output records. This fact caused some problems in reading and summarizing the output file for use in the Analyze system.

The other class of data collected for a realization is associated with the set of components defined for an EI. These data are labeled as "L" records, since the statistic is also divided according to the LOR it concerns. These data are collected for the set of all components in the population during a given period. The particular statistics collected for each component, in addition to the period and breakpoint values, are the:

- Number of times ER was performed at the STD_LOR (ER-S)
- Number of times PM was performed at the STD_LOR (PM-S)
- Number of times ER was performed at the AUX_LOR (ER-A)
- Number of times PM was performed at the AUX_LOR (PM-A)
- Resource time used at the STD_LOR (Serv-S)
- Resource time used at the AUX_LOR (Serv-A)

The last step in the data collection process occurs after the simulation is complete. This step creates a third class of output data which are the weighted averages of the number of EIs in the population during each time period. Since these values do not change between realizations, they are collected only once, and their records are located at the end of the file. If no MOF schedule is used then there is only one output record, an "N" record, that specifies the number of EIs fielded in every period.

If an MOF schedule is used, then there are as many output records as there are intervals in the simulation, each record identified by an "M" record type, and specifying the weighted average of the number of EIs in the population during that time period. Figure 9 summarizes the logical structure of the output file and presents the layout of each record type.

4.3 MAINTAIN DATA STRUCTURES

Since the MAINTAIN simulation system was not implemented using a commercial simulation language, control was available over the design of the structures to used. The "C" language, as discussed earlier, permits the dynamic allocation of memory. This scheme allows the construction of data structures, and a final system, that more efficiently operates within the constraints of the machine on which it runs. For this reason, there are several data structures and techniques which are unique to this particular simulation application. The following sections describe these data structures and their dynamics.

4.3.1 Simulation Clock

The simulation clock is not actually a data structure, but its discussion is important because several interesting notes can be made about its development. The simulation clock describes the time unit on which the simulation is based, and is usually represented by the current event time. Comparisons are made against this clock to determine the processing of current and subsequent events.

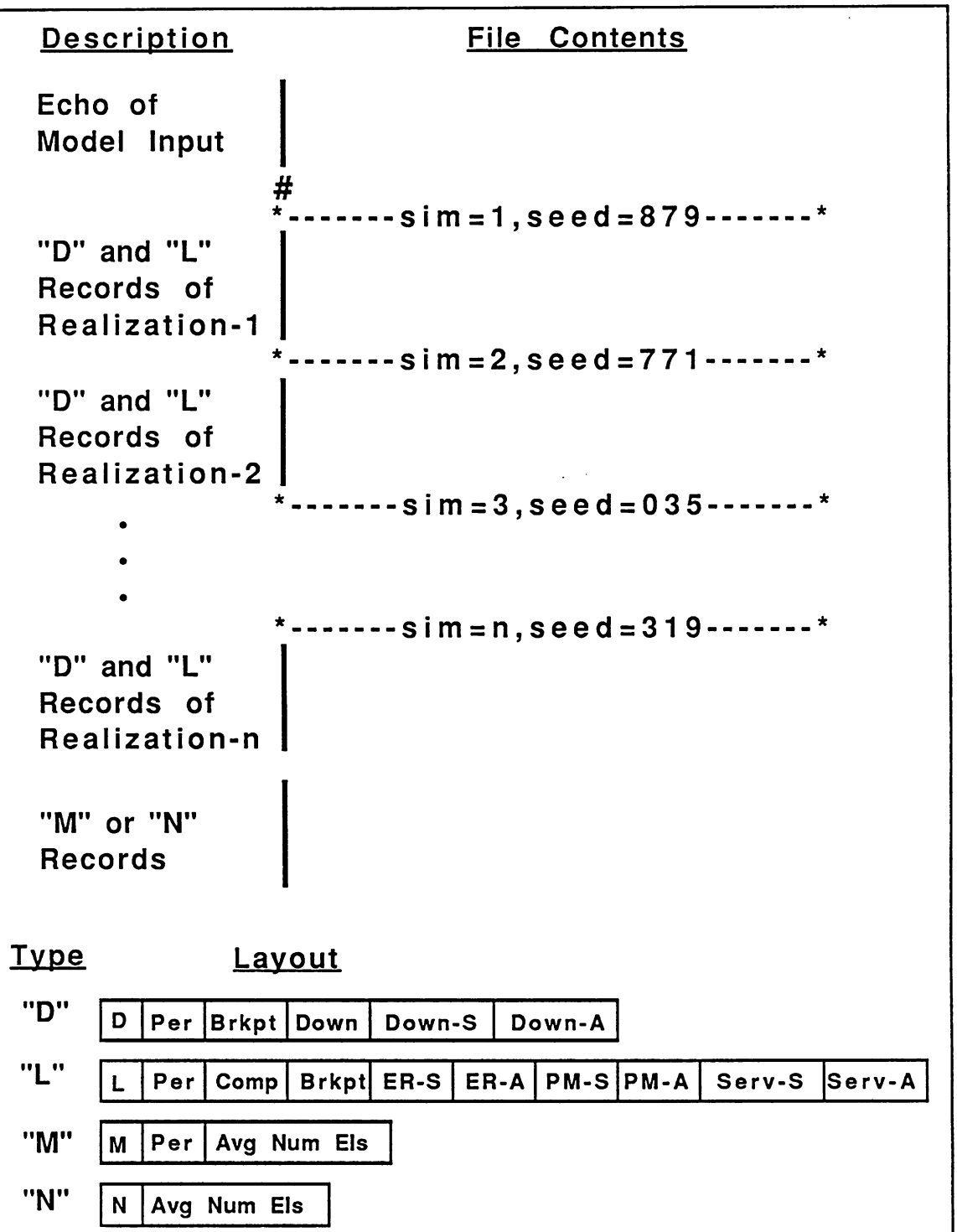

Figure 9. Structure of the Simulate Output File

In the first attempt at creating the modeling system, an integer clock was used. This meant that other event attributes were generated as integer or real values, and then truncated or rounded if necessary. The execution speed of the simulation was very good using this type of clock, but the conversion and maintenance of output data to a form more in line with that specified by a user, proved to be quite cumbersome. For this reason, the clock was converted to a real number clock, actually an 8 byte, double precision variable. This conversion caused the development process to proceed much more smoothly and resulted in only a slight degradation in execution speed. The resulting program was much easier to follow and intermediate results were more meaningful.

4.3.2 Event Calendar

The development of the event calendar structure used in MAINTAIN proved to be one of the more interesting stages of the research. An event calendar in a next-event simulation is a data structure which contains references to each future event, ordered according to their occurrence times.

4.3.2.1 Calendar Structure

This calendar is usually a singly-linked list of event nodes, with each node having an attribute, or pointer, which points to the next node in the list. The first node in the list, or head node, is selected to be the current event once processing has been completed on the last current event, and the event pointed in by the head node becomes the new head

node. When a event is generated, it is inserted into a list location based on its event time.

It was this preceding calendar structure that was used in the early stages of the simulation system. Figure 10 gives an example event calendar using this type of structure, where each node represents a component belonging to one of the EIs in the population. The first node number is the EI identification, the second number is the component identifier and the third number in the node is the event time. The number of nodes in the list is equal to the product of the number of components and the number of EIs, all ordered by their respective event times. As the modeling effort progressed, and non-aging of unaffected components and OPM event scanning (as discussed in Chapter 3) was added to the system, it became necessary to scan through the list, searching for other component events that belonged to the EI of the current component event. This scan requirement quickly revealed that the singly-linked list was an inefficient calendar structure for this type of event processing.

An alternative calendar structure was investigated in an attempt to improve the efficiency of event processing. The resulting structure is shown in Figure 11. This new structure is composed of two sets of singly-linked lists. Each vertical list node represents an EI in the population and has an attribute that points to another list. This second list contains the ordered component event nodes for that EI. The EI nodes are then ordered according to the head node time of its respective component list. This means that the next current component event is selected

by obtaining the head node of the component list pointed to by the head node of the EI list.

When a current component event is processed, unaffected component nodes have their event times incremented by the downtime incurred by the EI, or if an OPM scan for components associated with that EI is needed, the entire event calendar need not be searched. In this case, only the shorter component list attached to the current EI needs to be searched.

Assuming that a population consists of m EIs each with n components, a worst case complexity analysis was performed for these two structures. The outcome of the analysis was that (m x n) comparison operations are needed for the singly-linked calendar while only (m + n) operations are required for the "rectangular" calendar. A time study revealed approximately a fifty percent reduction in processing time when using the rectangular structure. More memory storage was required for this rectangular event calendar, and the maintenance of the calendar was more complicated, but the improvement in processing time was worth the extra effort.

Each of the two types of list nodes used in the calendar contain different attributes. The EI list nodes are required to carry very little information. For this reason, the only event attributes carried in the EI nodes are the:

- EI Identification Number
- Pointer to Next EI Node

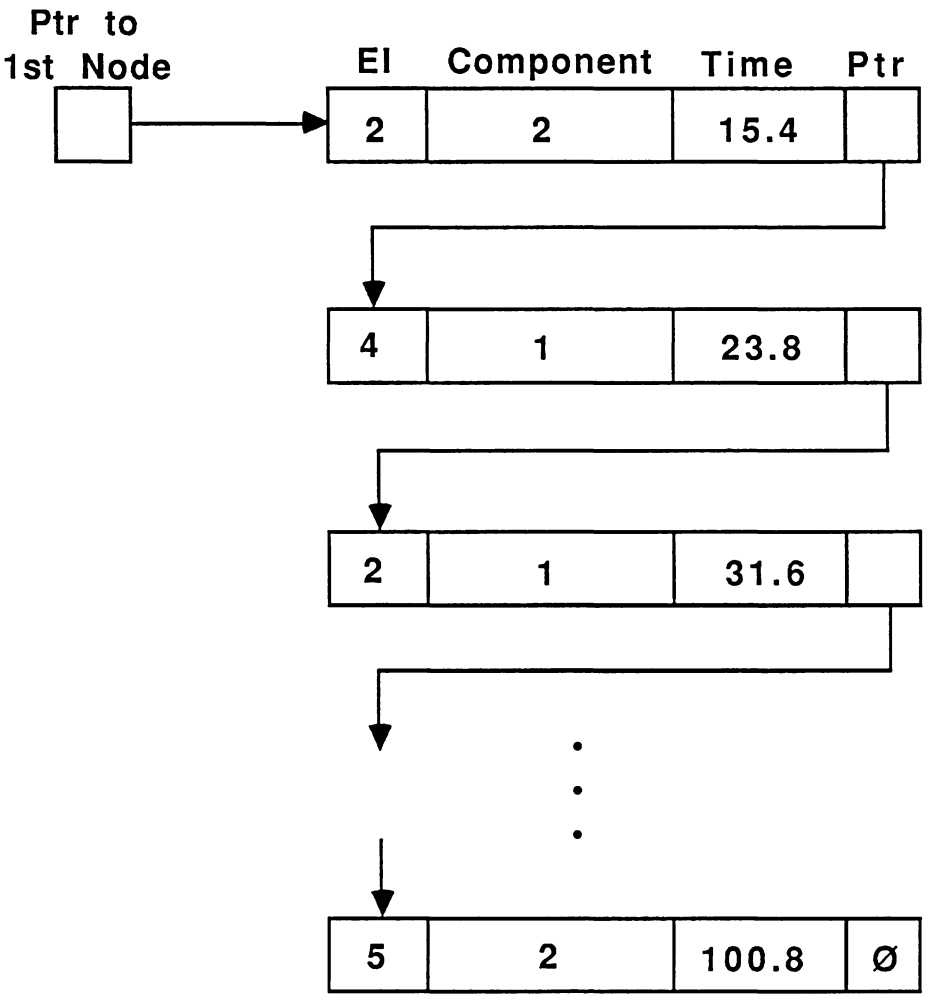

Figure 10. Singly Linked Event Calendar

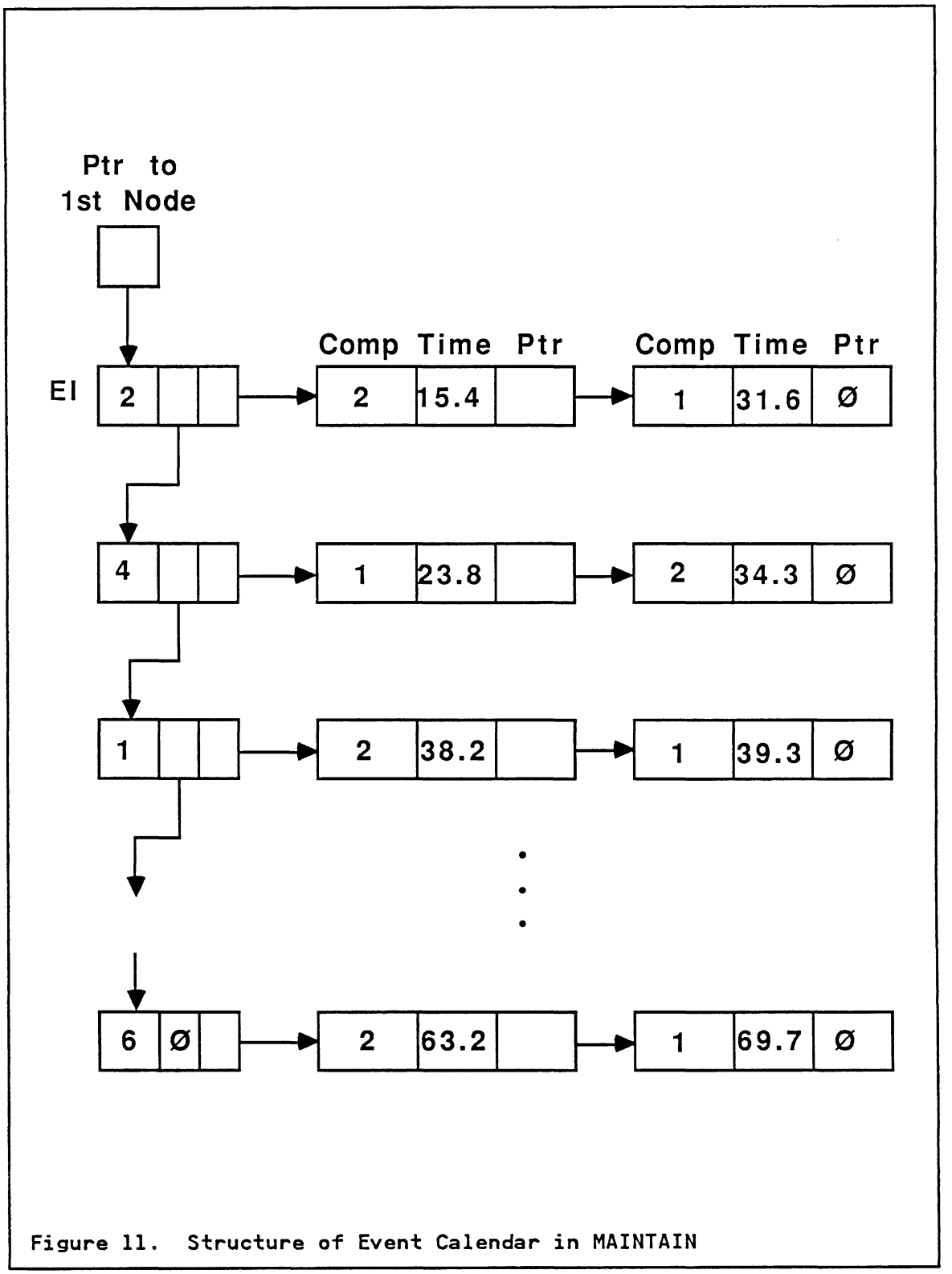

Figure 11. Structure of Event Calendar in MAINTAIN

The EI identification numbers begin with zero for the first EI and continue to (Number of EIs in Population - 1) for the last EI. This convention is used so that the loop counters in the program code are more understandable.

The component list nodes associated with a particular EI node carry the majority of the information needed during event processing. The attributes of the component list nodes are the:

- Event Time
- Event Type
- Component Identification Number
- Sampled Lifetime
- Pointer to Next Component Node

Component identification numbers follow the same convention as the EI nodes, but end at (Number of Components in an EI - 1). The event time is the time at which the event is scheduled to occur, but is subject to change due to the OPM processing option, which scans future events and possibly groups some events together to be performed earlier than scheduled. It is for this reason that the lifetime attribute is necessary. The sampled lifetime is the random variant that was generated for the age of that component at failure. This lifetime determines the original event type, but is also subject to change. If PM is specified for that component and the lifetime is longer than the PM interval, then PM will be performed for that component, with the event type being set to "P". On

the other hand, if the lifetime is shorter than the PM interval or no PM is specified, then the component will fail, causing an ER to occur, and the event type will be set to "E".

It is possible for an event to be delayed or advanced. A delay occurs when the current event is PM, or "P", and a wait is required for the selected LOR. Since an EI remains in the population and continues to operate while waiting for the appointment time at which a PM event will be performed, the current component could fail during the wait and become an ER. The original sampled lifetime for the component is used to make this determination. If this type of event occurs, the event time is changed to the failure time and the event type is changed to "e", signifying that it was changed from a "P" event. The EI then leaves the population and proceeds to the LOR to await its appointment.

When OPM is being utilized for the population, a scan for eligible PM events occurring within the scan frame specified is performed for each current event. An eligible component event might originally be scheduled in the calendar as an "E" event, but if a PM policy is specified for that component and the PM interval is such that the (Current Time + PM interval) is less than before the original event time, then the PM will be performed before the failure would have occurred, and the event type is changed to "p", signifying the change from an ER to a PM.

4.3.2.2 Calendar Dynamics

The processing and maintenance of the event calendar in MAINTAIN is an interesting operation. The maximum number of nodes in the calendar is determined at the beginning of the simulation. It is limited by the maximum size the population will attain, and by the number of components in an EI. If an MOF schedule is specified in addition to an initial block of EIs, events are generated for these EIs at the initialization stage of the simulation, but their event times reflect the time at which they are scheduled to enter the population. After the initial process of allocating space for each EI and component node, and sampling and determining the starting attributes for each node, the simulation is started by obtaining the first event from the calendar, which is removed from the calendar using a "delete" routine. The delete routine removes a node from a linked list by resetting the pointers of nodes adjacent to the node that is removed. When a replacement component node is added back to the list (with the same memory location reused), a set of "insert" routines place the node in the correct location of the list by locating the node after which the event to be inserted belongs, resetting adjacent node pointers, and then adjusting the ordering of the EI list so that the EI with the earliest component event is at the head of the EI list.

The following outline summarizes the operations that take place with the event nodes of the calendar during one iteration with a current event:

1. Delete the current event from the head EI list and add it to a temporary storage array.
2. If current event time is beyond the simulation time frame, stop simulation of current realization.
3. If specified, perform OPM scanning, deleting each grouped event from the calendar and adding it to a temporary storage array.
4. Process the array of current events (single or group).
5. Generate events to replace those just processed, storing the information in the temporary array.
6. Insert array events into the component list of the EI, resulting in a complete, ordered set of component events.
7. Reorder the EI list, if necessary, using insert and delete.
8. Go to step 1 for next event.

4.3.3 LOR Arrays

When a EI entity visits an LOR, it either joins a queue and waits if a component has failed, gets an appointment if a PM event is current or is serviced immediately if there is a zero queue length. In order to determine the outcome of an event when it requires service, a method of modeling the LOR was established. In the context of a next event simulation, this modeling usually takes the form of a new event added to the calendar. This event type has a time and certain attributes which specify the time and mode of service to be performed. When the simulation executive processes a next event and finds it to be a service event, it per-

forms the service activity. This is not the way the LORs were modeled in the MAINTAIN system.

In the MAINTAIN system, the current event(s) is processed entirely through the LOR while the current event is active. This type of queue modeling prohibits queue disciplines other than first in, first out (FIFO), but is adequate, and more efficient, for this type of simulation. This method is implemented by using an array that represents each resource unit available at a LOR, and is referred to as the "next-free" array. An array of this type is used for each LOR. Figure 12 shows the structure of these arrays.

Each unit of capacity available in each LOR has an array element, each initialized to zero when the simulation of each realization begins. The number in each array element specifies the time when that resource will be available. The first element of the array represents server one, the second, server two, and so on. These arrays are dynamically allocated at the start of the execution phase of the Simulate system, so as to not require a fixed amount of storage and further increase program efficiency. These structures are not really arrays, but are in actuality, contiguous blocks of memory. A pointer to the beginning of each memory block is used as the array name, and is referenced with an offset to address a particular array element.

When a current event requires service at either LOR, the next-free array is scanned to determine the earliest available server. If the next-free time is earlier than the current event time (taking into account any

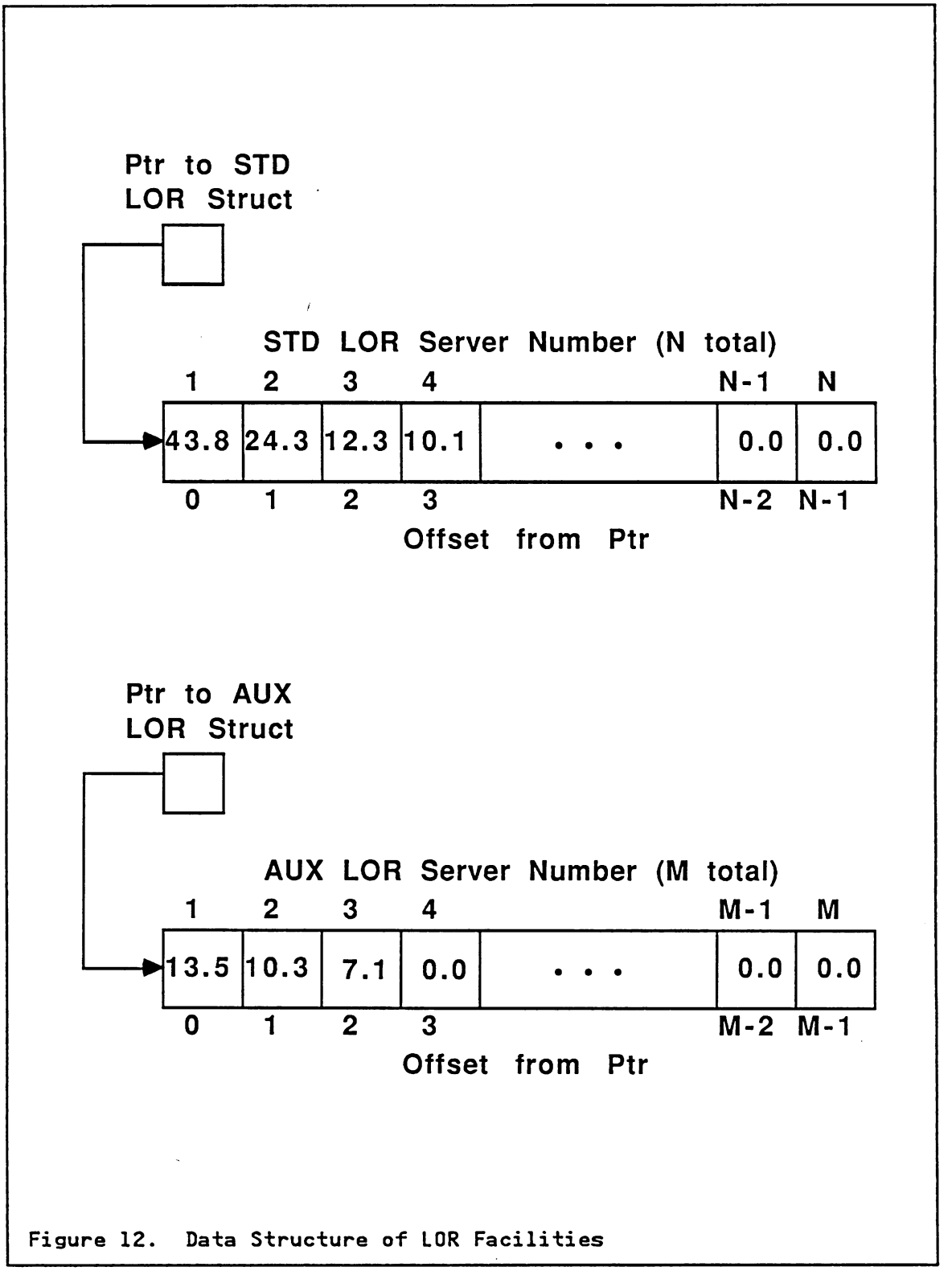

Figure 12. Data Structure of LOR Facilities

travel time), the next-free time is changed to the current time, and the maintenance event is processed. The event completion time is determined by adding the repair time to the current time and the next-free time is then updated to reflect that completion time. This process is used whenever a server is required for a maintenance event, with a single exception noted below.

If a LOR has been flagged for "sequential service", and a group of more than one event requires service, then the first event in the group "grabs" the server with the earliest next-free time, and that server is used for every other component event in the group. The next-free time for that server is then updated to reflect the completion time of the group of events.

4.3.4 Statistics Collection Lists

The MAINTAIN system, as previously discussed, collects time-series statistics to be used in the analysis phase of the modeling task. The collection of these time-series statistics appeared to be a difficult task to perform efficiently. Most collected data had starting and ending times which would often span one or more of the time-series periods, especially if repair queues grew long and led to excessive downtimes. In order to portray the performance of the system as accurately as possible, it was determined to collect these data based on their endpoints, and therefore divide the total duration into portions of time allocated to the periods

in which they occurred. With this decision made, a method and data structure that would implement the collection were developed.

The easiest way to perform the data collection would have been to place a limit on the maximum number of periods in the time-series, and to set up fixed arrays to collect data for each period, to write the data to a file and to clear the array after each realization. This approach seemed too constraining and would not have been very memory efficient. An alternative approach was investigated and ultimately proved to be the desirable method.

By setting up a sequence of periods as a linked list, and creating new collection nodes as necessary, system memory could be dynamically allocated as more periods were needed. This was a good method, except that (1 + Number of Component types) lists were needed for all the statistics necessary, consuming a large block of memory; a single list for population based statistics, and a list for each component type. A modification to the linked list method was developed to solve this problem, and is what might be referred to as "floating" lists. Figure 13 displays the overall structure of these lists.

Whenever a statistic is passed to the collection routines, an update time is passed as well. This update time is an absolute value which specifies the lower limit on the period number that will be needed in future collection calls. This update time is most often the current event time, but varies slightly with the type of statistic collected. The collection

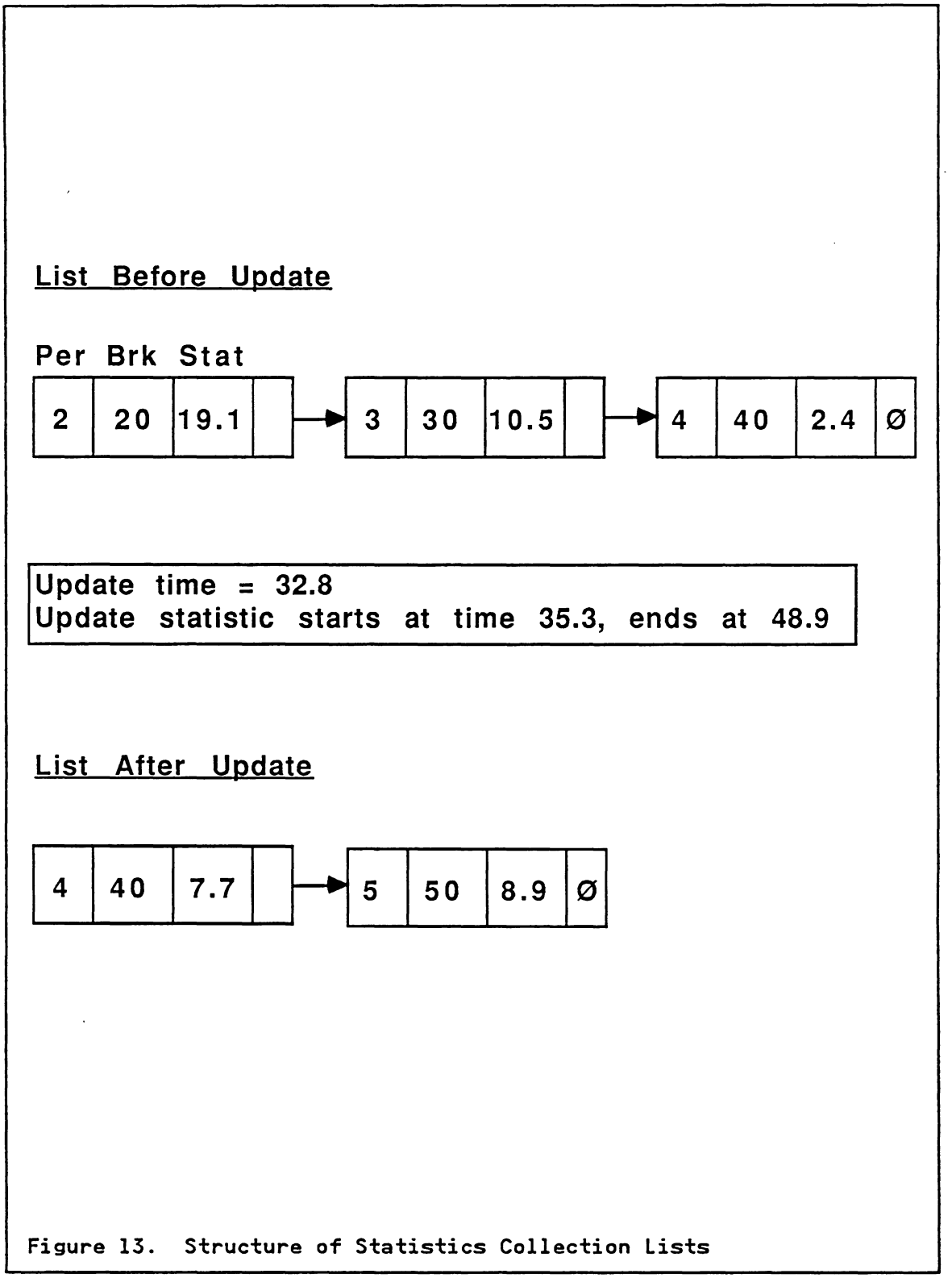

Figure 13. Structure of Statistics Collection Lists

routines react to the update time by writing to the output file, any nodes in the related list which are periods occurring before the period of the update time. Once this data is output, it is then deleted from the beginning of the list. This structure and method results in increased disk activity, but requires a minimum number of active nodes in the lists at one time, usually no more than two or three. If some statistic duration consistently increases during the simulation, possibly because repair queues continue to grow and service never catches up (which indicates an extremely poor maintenance system and is not likely to occur for a real system), these lists grow along with the durations, but still result in more efficient processing than required for maintaining all periods in memory at once.

4.3.5 Statistic "Arrays" in Analyze

The data structures and conventions used in the Analyze system were selected not to be efficient in the use of memory, but to minimize the processing time required to prepare a selected statistic for graphing or printing. This approach consists of performing almost all processing work at the time a Simulate output file is read. This method alleviates user frustration due to excessive waiting, but limits the type of statistic that can be presented. That is, a new file read is required if more than just the mean of each built statistic is to be presented. It was felt that the statistic mean is sufficient in order to obtain a drastically improved processing time. In addition, the Simulate output file is in a

form which is easily manipulated by many commercial software products, allowing a user to create new output analyses.

When a Simulate output file is specified as the input file for the Analyze system, all raw time-series statistics used by Analyze are computed during the file read and then stored in memory. Once a built statistic is selected for processing, the necessary raw statistics are manipulated into the proper form, the results are placed in a data structure dedicated to the current built statistic, and the output function is performed, either graphic display or printing to some device.

Figure 14 shows the data structure used to hold the raw time-series data describing the population of EIs. Each structure is dynamically allocated in a size that is determined by the simulation time frame and interval size model data that is read from the header section of the Simulate output file. A contiguous block of memory, containing a pointer for each time period in the series, is allocated first. Each pointer of the time series structure points to another data structure which contains a field for each type of statistic defined for that time series structure. If this field is a pointer, it points to a block of memory that holds the values for the mean, maximum, minimum, and standard deviation (MMMS blocks) of that period statistic. Otherwise, the field represents the value for that statistic.

There is one time series structure of this type for the population of EIs, which uses pointers (to MMMS blocks) for fields representing downtimes,

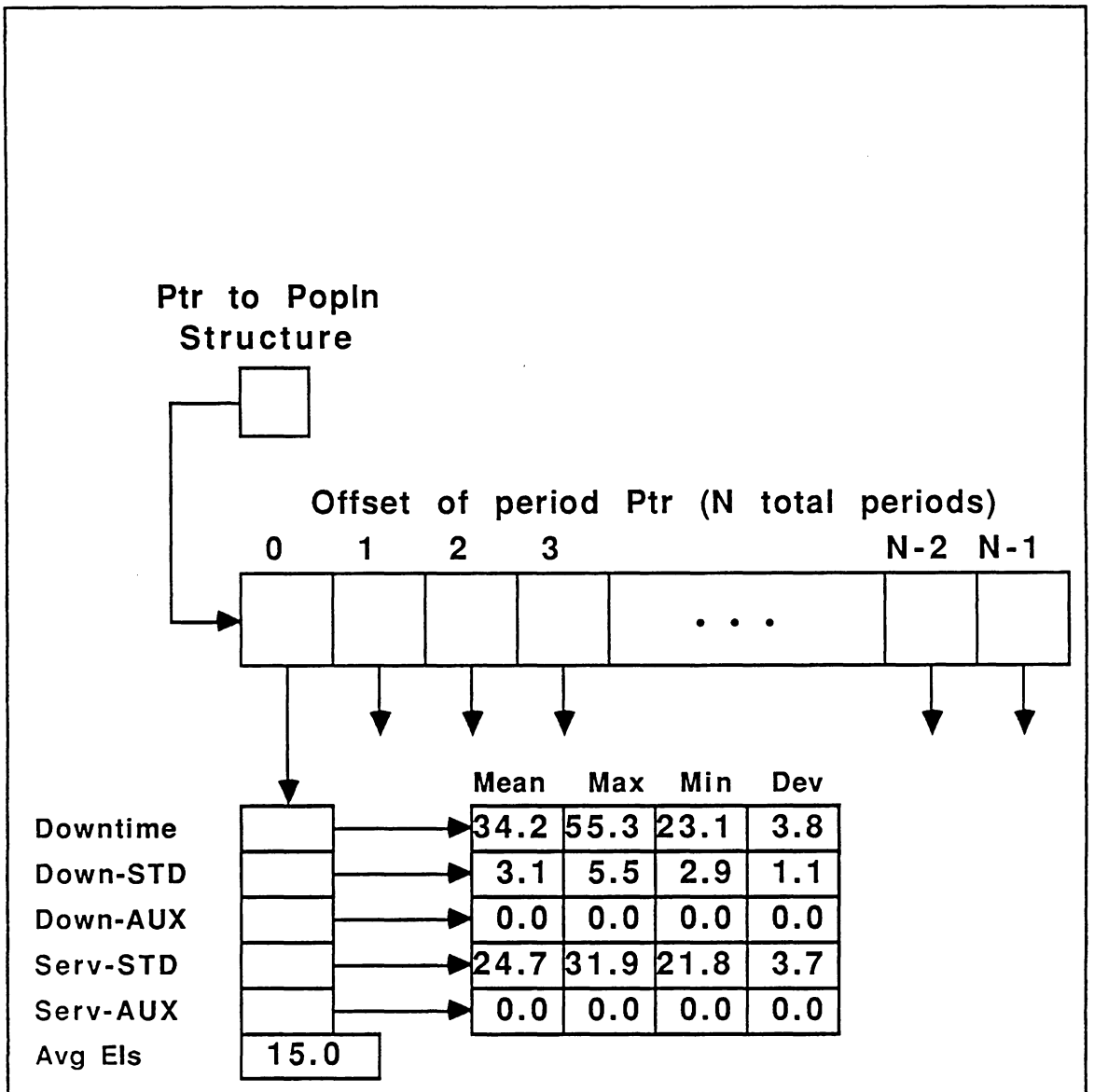

Figure 14. Structure of Internal Analyze Statistics

service times, and downtime occurrences for each LOR, and a non-pointer field that represents the average numbers of EIs in the population. There is also a slightly different structure, containing fewer data fields, for each component. Each component structure uses pointers to MMMS blocks for fields which represent data on a number count of PM and ER events to each LOR during the period.

As mentioned above, a data structure is also dedicated to contain the current built statistic. This structure looks similar to the raw statistic structure, but each period pointer in the time series structure points to a MMMS block instead of an intermediate structure, since only one built statistic is stored in the "plotting" structure. The MMMS block is where the computed values for the built statistic are stored.

5.0 CHAPTER 5. RECOMMENDATIONS AND CONCLUSIONS

The research activities that supported this thesis encountered a wide variety of specialty areas - logistics engineering, computer science, operations research, statistics, and many others. These activities were focused on developing a system that would extend the current state-of-the-art in maintenance modeling, which appeared to be application specific, to a more general level. In the author's opinion, this objective was definitely accomplished. MAINTAIN is a generic maintenance modeling system, with applications in many areas, and for many types of equipment and systems.

Another goal of the research was to develop a tool which could be made available to those companies which do not have the resources to thoroughly analyze their maintenance activities, but would greatly benefit by performing an analysis of some type. This is one reason that the system was implemented on a microcomputer.

5.1 FUTURE RESEARCH

Throughout the development of the MAINTAIN simulation system, some assumptions were made in the interest of completing the research in a reasonable amount of time. If someone were interested in taking the time to add extensions or modifications to MAINTAIN, there are several areas in which work could be done. Unfortunately, the extensions mentioned

below would each require an additional set of input data to describe the application being modeled, and the system, as currently defined, already stretches the assumption that relevant data are available. Nonetheless, the following list summarizes some of these areas of future research possibilities:

1. Time-Dependent Failure Distributions - Research in this area could eliminate the assumption that repaired components regenerate back to age zero.
2. Stochastic Service Times - This characteristic was not implemented in order to limit the model input requirements. This would be a logical extension of the MAINTAIN system.
3. Inventory Constraints - Model the inventory required to support ER and PM events as a resource. This would provide a more accurate representation of spare parts availability. If the necessary spares are not available for an ER then the EI will experience downtime while waiting.
4. Expand the Representation of LOR Facilities - Allowing a more comprehensive specification of the repair task necessary. For example, incorporating multiple service trades and skill levels at each LOR.
5. Allow for Redundant EI Components - Parallel components are found in many EIs and systems, handling this type of component specification would further expand the usefulness and accuracy of the maintenance system's representation.
6. Allow Dependencies between EI Component Failures - In the MAINTAIN system, component failures are assumed to be totally independent of

each other. The incorporation of dependent component specifications would allow modeling component interactions of the type such that when a component failure occurs, it is the result of, or contributes to, some other component failure.

7. Incorporate Measures of "Operational Readiness" - Some EIs continue operating to some degree even when certain components have failed. The addition of input parameters to describe, and functions to compute, the readiness would be of prime interest to the defense industries, but could be useful in commercial applications as well.
8. Provide Time Unit Conversions - At the present time, MAINTAIN requires that all input parameters have the same time unit. It would be convenient to have a preprocessor module that could convert all time units to the time unit selected for the simulation time frame.
9. Provide Input Analysis/Feedback - More thorough error checking would be useful, especially if it were in the form of an expert system module that could analyze input parameters and flagging potential user oversights (such as specifying a PM policy for a component with an exponential failure distribution) or suggesting possible input data based on that data already provided.

5.2 CONCLUSION

The need for a model like MAINTAIN has evolved from the increasing necessity of businesses and governments to get a better handle on planning for maintenance requirements. As far as the MAINTAIN model is concerned, populations of end items are considered (although a population could be

a single end item). This simulation model could be a tool which is useful at almost any level in an organization, from the strategic planning level to use by line level support managers, depending on the resolution in the specification of an end item's constituent components. This generality gives MAINTAIN a large degree of flexibility. All in all, the learning experience derived from such a large scale development was worth the effort, and the benefit derived from the model should assist both commercial and government industries in determining management policies concerning the maintenance of populations of end items.

BIBLIOGRAPHY

1. Brammer, K.W. "A Transient State Maintenance Requirements Planning Model," M.S. Thesis, Virginia Polytechnic Institute and State University, (1985).

2. Czajkiewicz, Zbigniew "Simulation Modeling for Heavy Industrial Equipment Maintenance," Computers and Industrial Engineering. Vol. 4, 185-191, (1980).

3. Dougherty J.J., R.G. Hazlett "Maintenance Activities and Resources Simulation (MARS) Model," Fourth Conference on Applications of Simulation. New York, December 9-11, 106-110, (1970).

4. Frisch, Franz. "Mortality and Spareparts: A Conceptual Analysis," Proceedings of the 1983 Federal Acquisition Research Symposium, 467-480, (1983).

5. Gertsbakh, I.B. Models of Preventive Maintenance. New York: Elsevier/North-Holland, Inc., (1977).

6. Grant, Floyd H. and Douglas G. MacFarland. "Simulation with C," Proceedings of the 1984 Winter Simulation Conference. Dallas, Texas, November 28-30, 491-496, (1984).

7. Hixson, Harold G. "Equipment Maintenance Studies Using a Combination of Discrete Event and Continuous System Simulation," Third Conference on Applications of Simulation. Los Angeles, CA, December 8-10, 76-85, (1969).

8. Jardine, A.K.S. Maintenance, Replacement, and Reliability. New York: John Wiley and Sons, Inc., (1973).

9. Jorgerson, D.W., J.J. McCall, and R. Radner. Optimal Equipment Policy. New York: American Elsevier Publishing Co., Inc., (1974).

10. Keeney, J.H. "Detailed Simulation of Military Aircraft Operations and Logistics," Fourth Conference on Applications of Simulation. New York, December 9-11, 32-38, (1970).

11. Kelly, A. and M.J. Harris. "Simulation - An Aid to Maintenance Decisions," Plant Engineer. Vol. 15, No. 11, 43, (1971).

12. Luxhoj, J.T. "A Taxonomy of Economic Replacement/Maintenance Models," Virginia Polytechnic Institute and State University, (1985).

13. Maze, T.H., Utpal Dutta, Mehmet D. Kutsal. "Role of Quantitive Analysis in Bus Maintenance Planning," Transportation Research Record 915. Transportation Research Board, Washington D.C., 39-48, (1983).

14. Maze, T.H., George C. Jackson, Utpal Dutta. "Bus Maintenance Planning with Computer Simulation," Journal of Transportation Engineering. Vol. 109, No. 3, 389-401, (May 1983).

15. McGrath, Michael F., Matthew G. Henry. "Logistic Simulation - A Credible Tool for Decision Makers?," IEEE Transactions on Reliability. Vol. R-30, No. 3, 258-264, (August 1981).

16. Pierskalla, William P., John A. Voelker. "A Survey of Maintenance Models: The Control and Surveillance of Deteriorating Systems," Naval Research Logistics Quarterly. Vol. 23, No. 3, 353-375, (September 1976).

17. Sherif, Y.S. and M.L. Smith. "Optimal Maintenance Models for Systems Subject to Failure - A Review," Naval Research Logistics Quarterly. Vol 28, No 1, 47-74, (March 1981).

18. Widawsky, William H. "Reliability and Maintainability Parameters Evaluated with Simulation," IEEE Transactions on Reliability. Vol. R-20, No. 3, 158-164, (August 1971)

19. Williams, William W. "Manpower Requirements Planning in a Chemical Maintenance Facility: A Time-Dependent Interactive Simulation," Proceedings of 1978 Winter Simulation Conference, 873-877, (1978).

APPENDIX A. MAINTAIN USER'S GUIDE

A.1 INTRODUCTION

This document is the User's Guide to the MAINTAIN simulation system, which was developed to provide decision support for the maintenance logistics of populations of identical end items, or for a single end item. It is assumed that the user has a basic understanding of the maintenance terminology specific to this system. This information can be acquired by reading the supporting thesis document. A good understanding of MS-DOS computers is also assumed throughout this User's Guide.

A.1.1 Hardware Requirements

1. An IBM PC compatible, MS-DOS machine with at least 512k memory. A hard disk drive is recommended, as the output files generated by the system can grow quite large.
2. A CGA compatible color graphics adapter and monitor (only necessary to view graphics, otherwise the entire system is available in monochrome).
3. A graphics capable printer (again, only for using graphics output).

A.1.2 Menu Processing

1. Selecting an Option from a Menu - Press the highlighted letter (using lower or upper case) next to the desired option to select that option from a menu. This will shift control to the next menu or process selected.
2. Moving Between Menus - When finished at a menu, exit by using the "Quit" option, pressing "q" or "Q" will present the previous menu.
3. Verify Quit at Critical Places - Several menu levels have a "verify" feature which requires responding with either a yes (y or Y) indicating a mistake was made, or no (n or N) to proceed with the quit. Verify occurs when leaving the "Maintain", "Simulate", and "Analyze" main menus.
4. Active Data File - One area of the menu is devoted to displaying the name of the currently defined data file to use for processing. The message "None Defined" is displayed in the filename area until a file is retrieved or created. Menu options which require active files will respond with a "beep" until that time.
5. File Names - The user is often prompted to provide names for input/output files. These names must conform to DOS conventions, i.e. d:XXXXXXXX.XXX, where "d:" is the drive designator and "XXXXXXXX.XXX" are the file name and extension (limited to eight and three characters, respectively, separated by a "."). The extension is optional but recommended (e.g. use ".DAT" for input data files, ".IN" for seed input files, and ".OUT" for output statistics)

6. Unintended File Prompt - Press <F5> to clear field then <return>. A "beep" will be heard, then control will be returned to the menu level.

A.1.3 Data Entry Environment

A commercial software library package, "Windows for Data" from Vermont Creative Software, was utilized in the development of the data entry environment. This approach streamlined the system and provided a more consistent overall system appearance. When in the data entry mode, several key-invoked functions are provided for user control. These keys respond with a "beep" in other areas of the MAINTAIN system. The keys used and their functions are summarized in Figure 15.

Key	Action
<F3>	Restores data field to initially displayed value
<F5>	Clears data field from cursor to end of field
<F6>	Clears entire data field
<F10>	Move to verify exit position (except for entry of file names, for which verify is inactive and the entry is automatically processed)
<Left Arrow>	Moves cursor left within data field
<Right Arrow>	Moves cursor right within data field
<PgUp>	Move to previous data field
<PgDn>	Move to next data field
<Return>	Move to next data field
<Home>	Move to first data field
<End>	Move to last data field
<Ins>	Activates insert mode, typed characters appear to insert left of cursor
<Del>	Deletes character at cursor
<Backspace>	Deletes character to left of cursor
<Esc>	Exits data entry screen without saving any changes to fields

Figure 15. Data Field Editing Keys and Functions

Some quantitative entry validation is performed by the library routines that monitor data entry. If a numeric value is required, non-numeric input will result in a "beep". Decimal numbers can be entered in scientific notation by using the standard convention of (+/-)xxxx.xxE(+/-)xx, where the +/- signs are optional. The "E" is the only character allowed in decimal fields. Anytime a validation error occurs, a "beep" will be heard and an error message will be displayed on the last line of the display. Correct the error and continue. Some notes of caution regarding values input for data fields are presented below:

1. Even though there is a time unit field for many input values, all values must be input in accordance with the time value specified for the simulation time frame.
2. All fields must be filled with some value or character(s).
3. Names requested in all data forms must be filled and be composed of strings that contain no embedded spaces. Use an underscore or hyphen to separate multiple words used as names.

A.1.4 Starting MAINTAIN

Three executable programs comprise the MAINTAIN simulation system, they are:

- MAINTAIN.EXE

- SIMULATE.EXE
- ANALYZE.EXE

MAINTAIN.EXE is a driver program (with menu) which loads a user's choice of the SIMULATE.EXE or ANALYZE.EXE programs. SIMULATE.EXE performs the simulation and creates output statistics files for a given set of input data; and ANALYZE.EXE uses the statistics files to develop graphic and printed output to assist in the analysis of the maintenance system under consideration. Usage of the system may vary - a user may simulate many scenarios, creating output files for each, then enter the Analyze system and process each output file (although each output file is considered separately), or the user may choose to run a simulation, analyze it, run a modified simulation, analyze it, etc. To initiate a MAINTAIN session, perform the following steps:

1. Run GRAPHICS.COM program from DOS (if graph printouts are desired)
2. Run MAINTAIN.EXE
   - from the default disk directory
   - keep the disk in the default drive
3. Make sure that sufficient disk space is available for input and output data files
   - it is a good idea to keep a list of previously defined files handy, since directories are not supplied within MAINTAIN
4. Select the "Simulate" or "Analyze" option from the Main Menu

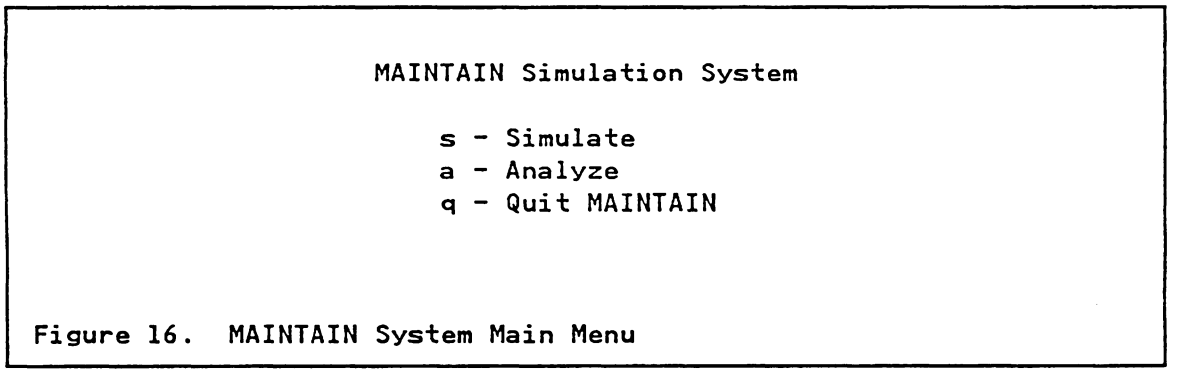

Figure 16. MAINTAIN System Main Menu

The Main Menu appears as shown in Figure 16. Note that SIMULATE.EXE and ANALYZE.EXE can be executed individually without using the MAINTAIN main menu by running the desired program from DOS.

A.2 THE SIMULATE SYSTEM

The Simulate system is central to the MAINTAIN system. It is in this program that input data files are retrieved or created, and edited. Once a file is defined, the simulation is executed by selecting the appropriate option from the Simulate menu, creating the output statistics file that is retrieved and processed by the Analyze system.

A.2.1 Simulate Menu

After the "Simulate" option is selected from the MAINTAIN Main Menu, the program SIMULATE.EXE loads and executes, displaying the Simulate main menu as shown in Figure 17 on page 90. Initially, since there is no active

data file, the Execute option will not work, therefore, "Model Data" must first be selected to allow the definition of a data file.

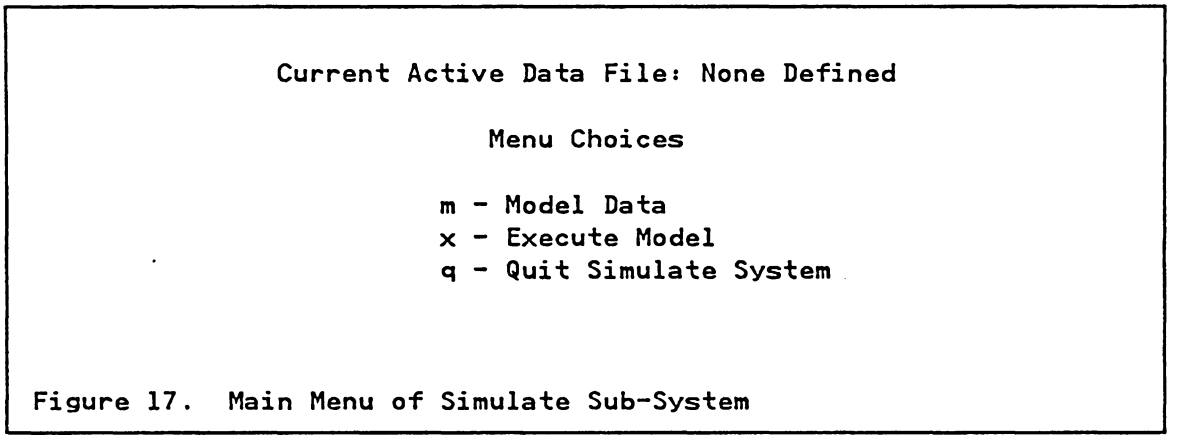

Figure 17. Main Menu of Simulate Sub-System

A.2.2 Model Data Menu

The Model Data option (option "m" from Simulate Menu) allows the retrieval and editing of existing input data files, or the creation of new files (which can be then be edited, saved, etc.). The Model Data menu appears as shown in Figure 18.

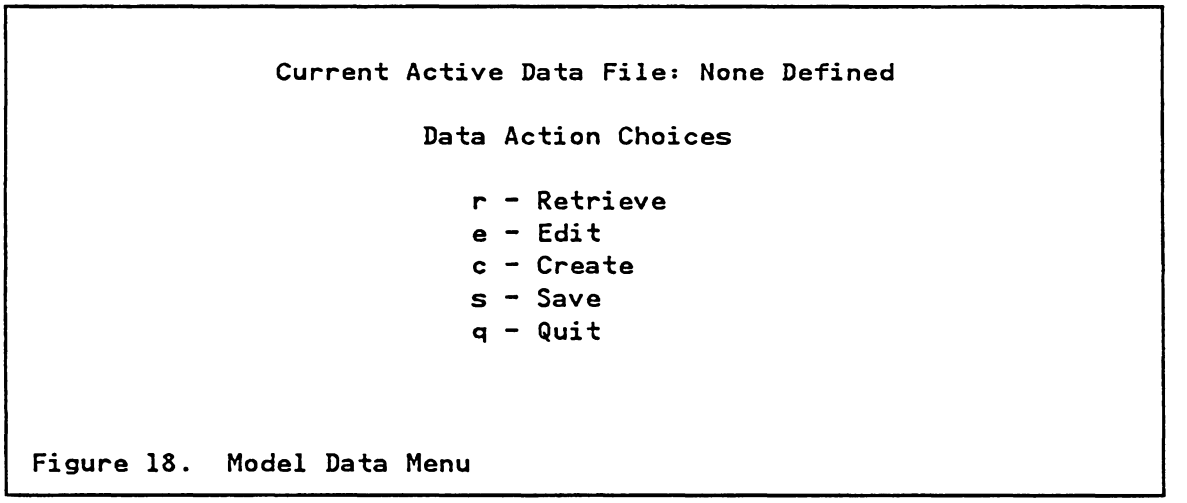

Figure 18. Model Data Menu

A.2.2.1 Retrieving Data Files

If it is desired to work with a previously defined file, then select the "Retrieve" option from the Model Data Menu. A file name prompt will appear at the bottom of the menu, with a default input file name already entered. If a new name is desired, clear the data field by pressing <F5>, type the file name to retrieve and press <return>, otherwise press <return> to accept the default. If the file is found, retrieval will occur, otherwise the system will respond with a "beep". The file name may include a drive designator if not on the default drive. Once a successful retrieve has been performed, the data can be edited or a simulation can be performed.

A.2.2.2 Creating Data Files

Building a data file from scratch has been simplified through the use of a file creation facility. Selection of the "Create" option from the Model Data Menu will result in a prompt at the bottom of the menu. Enter the file name to create. If the system is unable to open the file it will respond with a "beep", otherwise a new file will be created with the entered name, replacing any previous file on the designated drive which might have had the same name. Thus, use caution when specifying file names.

Once a new file has been opened, the data entry forms are presented to the user. Using the key functions described earlier, the user provides

input for each data field (all fields are initialized to default values) and respond with a "y" when prompted to verify exit from the current form. A "yes" verification initiates a data form read and exits from the form. Responding "no" returns the user to the data form where further editing or an abort with the <esc> key can be performed. The data forms are presented in the following order with important user actions outlined. Care should be taken to ensure that all input data, including failure distribution parameters, are specified with respect to the time unit selected for the simulation time frame.

1. Enter Component Data (2 or more forms for each component).
   a. One major data entry form for each component.
   b. Failure distribution data is entered following the major form. If a empirical distribution code is specified in the component form, a form representing each time interval will be displayed and must be filled out. Pressing <esc> after exiting the last interval's form, will terminate the distribution data entry. Otherwise, if a parametric distribution code was specified, a single form requesting the 1 or 2 failure parameters needed to describe the distribution will be presented (data field descriptions can be found in a later section).
   c. Press <esc> on a fresh component form once all components have been entered (proceeds to next form).
2. Enter End Item Data Fields (1 form).
3. Enter Mode of Fielding Data Fields (1 form).

a. If an End Item implementation schedule is desired then the schedule's data entry will follow the major form, in the same way as for an empirical distribution.

4. Enter Simulation Parameters and Flags (1 form).

5. Data is then automatically saved to the file name specified.

A.2.2.3 Editing Data Files

Once a data file has been retrieved or created, any one of the data forms can be displayed and will contain the current data. The user can then change data values by typing new values over the existing values. Use the special keys to move around. To abort the form without reading any changes, press <esc>. To leave the form with changes intact, be sure to exit the form by verifying an exit.

There are two types of data that cannot be edited in this fashion: any empirical distributions specified and any MOF schedule that may have been defined. There are other, not so convenient, ways to change this data and a short description appears later in this document.

Selecting "Edit" from the Model Data Menu brings up the menu shown in Figure 19 on page 94. Now the user can select one of the four types of data defined for an input file: components, end item, level of repair, or simulation.

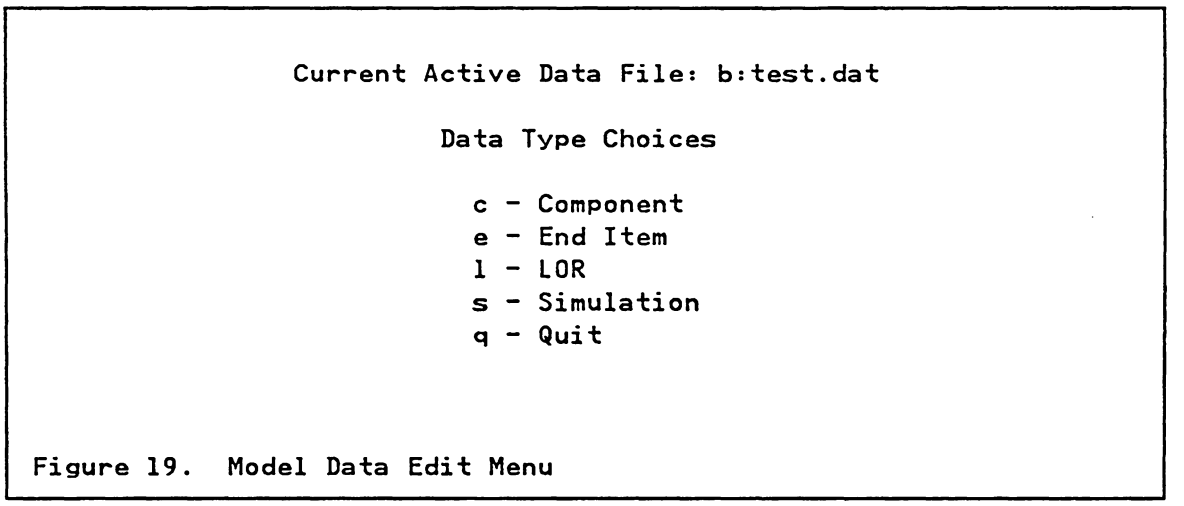

Figure 19. Model Data Edit Menu

Selecting "Component" presents each successive component's data. At present there's no way to move easily among the components; to bypass some component, just use the <esc> key to abort from that component's form and continue to the next form. If an empirical distribution appears in a screen window, the user can look at all of the data (including that lying off the screen) by using the cursor control keys to scroll within the window. The same method applies to viewing a MOF schedule (if defined). The MOF schedule would follow a user exit of an "End Item" form.

The other two selections from the menu, "LOR" and "Simulation Parameters", bring up a single form, which is edited as the others.

A.2.2.4 Saving Changes

After editing an input data file, it might be desired to save this new scenario as a new data file, or as an update to the current file. To

perform this process, select the "Save" option from the Model Data Menu. The user is prompted for the file name to save the data to. Enter the desired name, either a new name or the current name, and the save operation is performed.

If the "Save" option is accidentally selected, press the <F5> key to clear the name field, then press <return> to enter a blank field. This will result in an error which causes a "beep", but control is returned to the Model Data Menu.

A.2.3 Executing the Simulation

Once an active data file has been defined, simulation of the maintenance system can be performed. Selecting the "Execute" option from the Simulate Menu by pressing "x" prompts the user for two file names. The first file requested corresponds to the file that contains seeds for the random number generator. A new seed value is used for each iteration of the simulation model. As many seeds are needed as simulations specified for "Number of Simulations" in the "Simulation Parameters" data form. The user does not specify the number of seeds to use, but instead, the Simulate system reads a new seed whenever it requires one. Running out of seeds could produce unpredictable results. For the user's convenience, there is a file provided with the system, SEEDS.IN, which contains fifty (50) seed values, this should be more that adequate for most users. If more than fifty simulations will be run, then it will be necessary to edit the seed file and add more seed values. The file is a flat ASCII text

file, with one seed value per line. Thus, any text editor can be used to add to or modify the existing set of seeds. In order to provide a consistent comparison base, the same set of seeds should be used for each simulated scenario.

The second file that must be specified is the output statistics file. This file can get quite large (greater than 100k) for a simulation of even modest size. There is a collection of statistics written to the file throughout each iteration of the simulation model, therefore, DO NOT remove a data diskette during simulation execution. Make sure that the disk drive specified for this file can accomodate the output file sizes generated by the model's set of input data. Due to the variety of parameters which affect the size of the output file, the user must learn this aspect of the system by experimentation, (see the "Memory/Size Considerations" section for more information on parameter effects). One suggested method might proceed as follows:

1. Execute a test model that runs only one (1) simulation. Use the actual set of model data for the application under consideration, but specify in the "Simulation Parameters" data form that only 1 simulation is to be performed.
2. Exit the MAINTAIN environment and return to DOS.
3. Use the DOS "dir" command to determine the sizes (in bytes) of the created test output file and the original model data input file.
4. Subtract the input file size from the test output file size. The resulting number is an estimate of the contribution of each simulation

iteration to the total output file size (since the original model data is echoed to the beginning of the output file).

5. Multiply the variable size found in step 4 by the total number of simulations required in the full model (as discussed previously, this number should be at least 30) and then add in the size of the model input file. The resulting number is an estimate of the amount of disk space required for model's output file.

Another approach might be to study the example model and the output file sizes presented in a later section of this user's guide. This discussion might give users some feeling towards the output size requirements of their model.

While the simulation executes, a window is displayed on the screen. This status window gives the number of the current simulation iteration executing. When the simulation is complete, the status window clears and control returns to the Simulate Menu. At this point, the user can modify the input file and perform more simulations, creating a new output file for each; or exit the Simulate system by selecting "Quit" and then verifying, and continue to the "Analyze" system.

A.3 THE ANALYZE SYSTEM

After a statistics output file is created with the Simulate system, the Analyze system is used to present the output in a more usable form. More specifically, plots of various statistics are presented graphically, and

printer or file dumps of the original input data, statistics output file, and computed statistics, are available.

A.3.1 Analyze Menu

Selecting "Analyze" from the MAINTAIN Main Menu loads the Analyze system and displays the Analyze Menu as shown in Figure 20. From this menu, access is available to the "Graph" and "Print" options, but first, as with the Simulate system, a data file must be retrieved (noted by the menu section that specifies the "Current Active Data File" as "None Defined"). However, in this case, the data file retrieved must be an output file generated by the Simulate system.

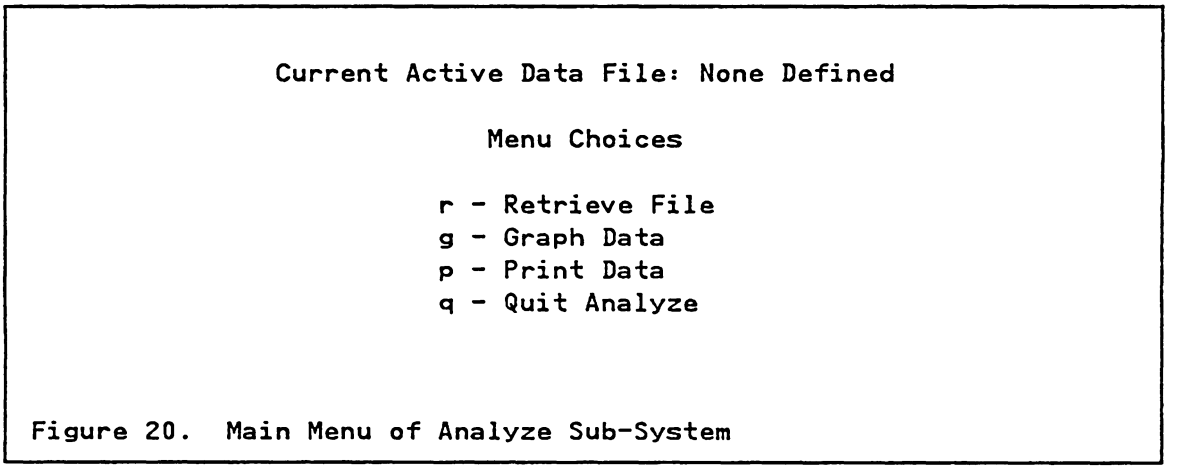

Figure 20. Main Menu of Analyze Sub-System

A.3.1.1 Retrieve

When "Retrieve" is selected from the Analyze Menu, the user is prompted for the statistics file to load and process. The user should enter the name (including drive designator if not on the default drive) for a sta-

tistics file created by the Simulation system. Once a valid file is specified, it is read and processed. The processing of this file can take quite a while, so be patient. A message is displayed during the operation to remind the user of this fact.

A.3.1.2 Graphic Output

Selecting the "Graph" option from the Analyze Menu will display the Graph Menu shown in Figure 21 (provided an active data file exists). A category of graphs can now be selected for viewing. There are three categories of data that can be graphed:

- Components
- Levels of Repair
- Population

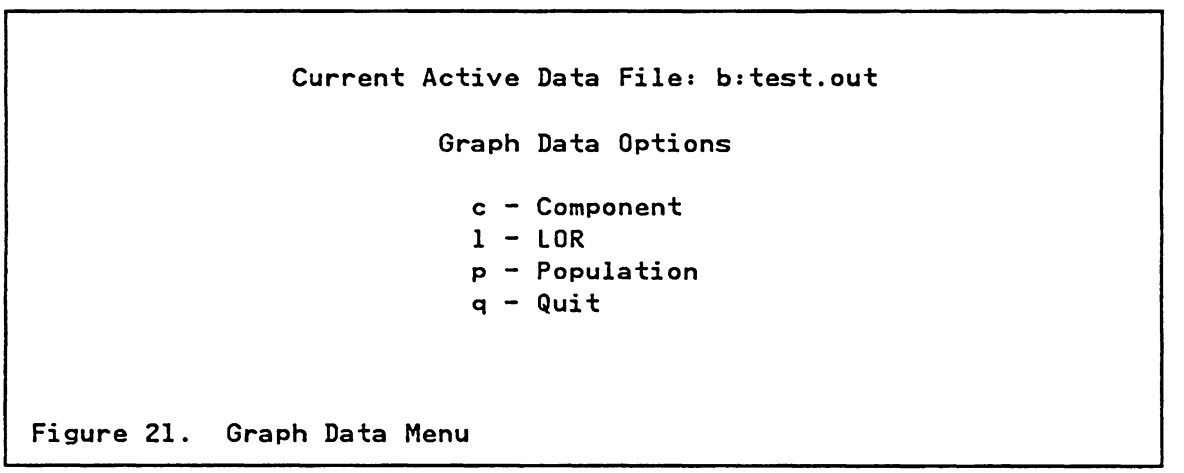

Figure 21. Graph Data Menu

The population graphs present several different statistics about the group of end items as a whole and include:

- Availability
- Downtime Instances
- Downtime Cost
- Service Cost
- Spares Cost
- Total Program Cost

One of these graph types is selected from a secondary menu after choosing the "Population" option from the Graph Menu. The selected graph is displayed once a type is selected. While the graph is active, there are several options available to the user. These options are summarized at the end of this section, but include for the Population graphs:

- Scrolling through the time periods
- Drawing grid lines on the graph
- Changing the graph statistic scales
- Printing the graph

The user selects to quit viewing the current graph by pressing "q" or <esc>. User control is returned to the menu from which the graph type was selected.

When the "LOR" graph category is selected from the Graph Menu, there is only one graph type available - server utilization. The graph of the standard LOR is displayed first. If the auxiliary LOR was used in the simulation, then that utilization statistic is also available for view-

ing. The user switches between the standard, auxiliary, and a weighted total of both LORs by pressing the <PgUp> and <PgDn> keys to scroll to the previous and next type of statistic. The other viewing options mentioned previously are also active at this time.

The "Component" graph category gives PM and ER instances for each component in an end item. The number of instances is the total across all end items in the population during the time period shown. The <PgUp> and <PgDn> are used to switch between PM, ER, and total instances for a particular component. To scroll through the various components in an end item, use the <up arrow> and <down arrow> keys.

The following notes summarize the functions available while viewing a graph on the screen.

1. Use the <left arrow> and <right arrow> keys to scroll through six periods per key press. A "beep" occurs at either end of the time frame.
2. Use the <PgDn> and <PgUp> keys to switch statistic being viewed (only for "LOR" and "Component" graphs).
3. Use the <up arrow> and <down arrow> keys to switch graph to previous and next component in the end item (only for "Component" graphs).
4. The "Scales" option allows rescaling the statistic axis. The values entered must be within the values for the min/max of the statistic displayed. All graphs are initially displayed with the statistic

scale set to the min/max values, except for statistics displayed as a percentage, which initially have the scales set for 0 to 100.

5. The "Grid" option draws horizontal lines corresponding to each mark on the statistic axis. Choosing the "Grid" option while the grid is being displayed, turns off the grid.
6. The "Print" option removes the bottom prompt line, displays the current file name on the screen, and performs a screen dump to the printer.
7. The "Quit" option returns control to the previous menu.

A.3.1.3 Printed Output

If the "Print" option is selected from the Analyze Menu, the Print Menu is displayed, as shown in Figure 22 on page 103. From this menu, the user has the option to print three types of data:

- an echo of the original input data
- a printed summary of the statistics contained in the output file produced by the Simulate system
- any selected statistic available in graph form

Another type of printed output is also possible - the statistics output file from the Simulate system, but that must be printed from DOS. The physical layout and description of the statistics file is described elsewhere in the thesis.

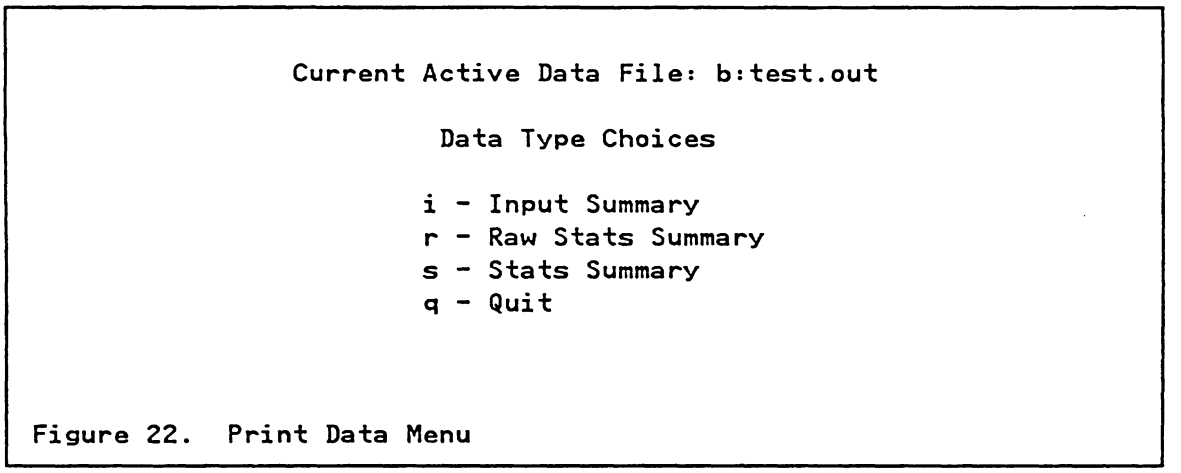

Figure 22. Print Data Menu

The user has the choice of routing output to a printer or to a disk file for later printing. When a particular print option is selected, the prompt "Printer or File? (p/f)" appears at the bottom of the menu. Press the letter corresponding to the desired option. If "File" is selected, the user is prompted for a file name in the manner previously described, and if the file can be opened, the selected output is printed to that file (a new file is created or an old file is written over if the name is already in use). If the "Printer" option is selected, then the selected data is routed to the system printer. If the printer is turned off or off-line, then it is considered "unopened" and an error "beep" will result, otherwise printing will occur.

Selecting the "i" or "r" options produces output requiring no user intervention once the output device is selected. The "s" option presents a series of menus identical to those in the graphing system. The user moves through these menus selecting the statistic to print, just as with selecting a graph to view. When a statistic type is selected, the data

are appended to the output device selected. The only difference from the graph options is that there is no statistic scrolling using the cursor control keys. For example, when the LOR data is printed, all available data for each LOR are printed (including the total), and if the Component stats are selected, the statistic is printed for each component in the end item. Quitting back to the main print menu closes the output device. One note of caution is necessary: if using a disk file as the output device, use different file names for each print option because each time the same name is used, any data contained in that file is replaced by the new data.

A.4 ADDITIONAL TIPS/WARNINGS

The MAINTAIN system is by no means completely flawless. For that reason the following warnings and tips are presented.

A.4.1 Error Checking

The only error checking performed is for runtime memory overflow (abort) and if specified file not found ("beep" after entry). There is no error checking in cases such as:

- Invalid lifetime distribution specification
- Integrity of the model input file
- Unusual parameter combinations

These types of errors cause unpredictable results, but might be recognizable due to the inability to execute a simulation or by resulting "garbage" output.

A.4.2 Changing Lifetime Distributions

Once a particular lifetime distribution has been specified for a component, it might be desirable to change the type for different scenarios. There is no feature built into MAINTAIN to explicitly perform this type of edit, but there are procedures that can be followed to accomplish distribution changes. These procedures are explained in the sections below.

A.4.2.1 Any Distribution to a Parametric Distribution

Since the different distributions have varying numbers of parameters, and only the parameters necessary have fields on the component data form, the editing method is divided into two editing steps. From the Edit Menu:

1. Select Edit Component Data.
    a. Locate the desired component for change by stepping through the components.
    b. Change its distribution code to the desired distribution.
    c. Exit the form by verifying.
    d. Step through the rest of the components to return to the Edit Menu.

2. Reselect Edit Component Data.
    a. Locate the desired component.
    b. The correct number of parameter fields for the desired distribution is now displayed. Then, fill the one or two parameter fields with desired values.
    c. Exit the form by verifying.
    d. Step through the remaining components to return to the Edit Menu.
3. Continue by saving the change, or execute a simulation.

A.4.2.2 Parametric to Empirical

Changing to, or editing, an existing empirical distribution requires a little more work from the user. An ASCII text editor or word processor must be used from outside the MAINTAIN system to physically edit the data file containing the distribution that needs changing. Each input value in the data file occupies one line, except for empirical distributions and MOF schedules, which have two values on each data file line. The user makes a change to an existing empirical distribution by doing the following:

1. Locate the distribution that needs changing (the component name is just above the distribution data).
2. Change the values desired, or insert new lines containing appropriate values (the end of a distribution or schedule is located by including a "-1" value that follows on the next line).
3. "Save" the change.

4. Run MAINTAIN and "Retrieve" the modified data file.

To change to an empirical distribution, the user must change the distribution code, delete the old distribution parameters, and add the new distribution data and the terminating "-1" value. This step is a little tricky and probably will not be needed by most users, therefore it will not be discussed in detail. If this type of change is really needed, either experiment with editing the data file, or create a new file using the "Create" option of MAINTAIN.

A.4.3 Changing MOF type

Using the techniques discussed above, the Mode of Fielding parameters can also be edited. The field for the initial block of end items in the population is always shown, but to change/add a schedule or remove a MOF schedule, the user must follow the above instructions but locate the area of the data file which contains the MOF information.

A.4.4 Memory/Time Considerations

The MAINTAIN system can currently handle a maximum of twenty-five (25) components defined for the EI. The number of components associated with an EI affects the following MAINTAIN data structures with an approximate 1:1 increase in the:

- Size of the event calendar

- Number of Simulate system statistic collection lists needed
- Size of all Analyze system structures
- Execution time for each simulation iteration

The number of end items specified in the population affects only the size of the event calendar and therefore the execution time of the simulation.

The statistics collection interval size with respect to the planning horizon affects several areas. If the planning horizon divided by the interval size) increases, then there will be an almost proportional increase in the:

- Number active nodes in stat-collect lists
- Size of Simulate system output file
- Size of all Analyze system data structures

The mean value of component lifetimes with respect to the size of the planning horizon can also affect several system areas. The planning horizon divided by a component mean gives an estimate of the average number of failure events that must be processed during the planning horizon. This is a conservative estimate, because if PM is in effect, then the number of maintenance events encountered is likely to be large than the average number of failures. An increase in the average number of events to be processed causes an increase in the:

- Execution time for each simulation iteration (1:1 increase)

- Size of the Simulate system statistics output file (the amount of the increase is hard to estimate in this case, it depends mostly on how many more events are in queue at a LOR)

If the MAINTAIN system is unable to allocate memory for any data structure during processing, an abort() call is executed, resulting in an immediate return of control to the MAINTAIN Menu or to DOS (depending on how MAINTAIN was started). This error should be easily recognized, as this type of shift of control does not normally occur anywhere else in the system.

A.5 USING OUTPUT FILES

Users might wish to generate their own statistics in addition to those provided by the Analyze system. The files created by the "Print" option from the Analyze Menu, combined with the statistics output file from the Simulate system, can be used for this purpose. A good understanding of the file structure of the Simulate output file and the original input data is necessary, as that data is essential for most values that can be computed. The following notes can be used to get started:

1. Files are output as plain ASCII text files. Thus, they can be read by many commercially available software packages.
2. Transfer the files to some spreadsheet program, or to a statistical package with good data manipulation capabilities.
3. Perform analysis by defining, moving, sorting, summarizing and graphing data.

A.6 SUMMARY OF INPUT PARAMETERS

Many of the input fields associated with the various data forms are self-explanatory: names, dollar amounts, time values (except for the time unit field). These particular fields are not outlined here but information about them can be found in the supporting thesis document. The following descriptions relate to the non-obvious aspects of some input fields.

A.6.1 Lifetime Distribution Specification

The probability distributions used to describe a component's lifetime (or time between failure) are specified by a single character code. Parameters for each distribution consist of one or two values. The display of these parameters are based on the particular parametric distribution selected. If an empirical distribution is specifed, a list of user input is required. These distributions and their parameters are summarized in Figure 23 on page 111.

Distribution	Code	Fail Parm 1	Fail Parm 2
Normal	n	mean	std. deviation
Exponential	e	mean	N/A
Gamma	g	scale	shape
Uniform	u	min value	max value
Lognormal	l	mean	std. deviation
Weibull	w	scale	shape
Erlang-k	r	k	mean
Empirical	m	N/A	N/A

Figure 23. Lifetime Distribution Field Descriptions

A.6.2 MOF types

The mode of fielding type is specified as one of the following:

- "n" for no MOF schedule
- "s" for schedule to be used

A.6.3 PM types

The type of PM really only specifies whether PM is performed for that component or not. The following convention is used:

- Use "0" for no PM
- Use "1" to indicate PM is performed (be sure to specify the time interval or age when PM is performed)

A.6.4 Time Unit Specification

On almost every field entry that can have a unit of time associated with it, there is a corresponding field that will contain the unit of time for that entry. The following code is used to attach a time unit to some value:

- "h" - hours
- "d" - days
- "w" - weeks
- "m" - months
- "y" - years

The time unit specified for the simulation run is the most critical. This field specifies the unit of time for each unit value in the simulation time frame. For example, a time frame of 0 to 200 with a unit of "d", specifies running the simulation for 200 days. The collection interval also uses this same time unit. These time units are not currently used in MAINTAIN except to specify the units for the simulation. It is assumed that all input values reflect the unit of time input for the simulation time frame. A future version of MAINTAIN might incorporate a pre-processor which will convert varying specifications to that of the overall time frame.

A.6.5 Flag Parameters

Several maintenance processing options are available and are referred to as "flags". A flag value of "0" indicates that the option corresponding to the flag is inactive and a "1" indicates the option is active. The following flags occur in the location and for the option indicated:

1. OPM Flag
    - Located in the Simulation Parameter data form
    - A value of "0" disables opportunistic maintenance scanning; else a value of "1" activates OPM and a scan-ahead time must be entered and have a value greater than zero
2. Aux Flag
    - Located in the Simulation Parameter data form
    - A value of "0" disables investigation of the use of the auxiliary LOR; else a value of "1" activates the use of the AUX LOR
3. Gang Flag
    - Located in the LOR data form of each LOR
    - A value of "0" disables use of first available server for each separate grouped repair event, and forces the server assigned to first event to perform each repair in the group; else a value of "1" activates gang repair.

A.7 EXAMPLE PROBLEM

This section of the User's Guide provides a simple example of using the MAINTAIN system. The example population consists of Side Loadable Warping Tugs (SLWTs) that are used by the U.S. Navy. The SLWTs are large, flat ships which are used in support operations. It is assumed this set of SLWTs will be used in a combat environment.

It is proposed to field twenty of the SLWTs by a single block acquisition. A set of five components have been determined as critical to the operation of a SLWT. Of these five components, three have predetermined PM policies while two will have no preventive replacement performed (these two have exponential failure distributions). The STD (Standard) and AUX LOR (Overtime) will initially have repair capacities of three and one SLWTs, respectively. The AUX LOR will not be utilized in the initial analysis. The entire set of input data describing this population and maintenance system is summarized below, including any required costs and times.

1. End Item Name: SLWT-Combat
    a. Initial Block to Acquire: 20
    b. Cost of Downtime: $200 per day
2. STD LOR Name: Standard
    a. Capacity: 2
    b. Cost of Usage: $110 per day
    c. Time to Transport: 0
    d. Cost to Transport: 0

e. Use Gang Repair if possible

3. AUX LOR Name: Overtime

   a. LOR Capacity: 1

   b. Cost of Usage: $220 per day

   c. Time to Transport: 0

   d. Cost to Transport: 0

   e. Use Gang Repair if possible

4. Component 1 Name: Engine_Oil_Cooler

   a. Failure Distribution: lognormal

      1) Mean: 275 days

      2) Standard Deviation: 40 days

   b. Cost of PM Spare: $400

   c. Cost of ER Spare: $400

   d. Repair Time for ER: 3.5 days

   e. Repair Time for PM: 3.5 days

   f. Interval Between PMs: 270 days

5. Component 2 Name: Sea_Wtr_Pump

   a. Failure Distribution: normal

      1) Mean: 200 days

      2) Standard Deviation: 25 days

   b. Cost of PM Spare: $300

   c. Cost of ER Spare: $300

   d. Repair Time for ER: 5 days

   e. Repair Time for PM: 5 days

   f. Interval Between PMs: 180 days

6. Component 3 Name: Manifold

a. Failure Distribution: exponential

1) Mean: 350 days

b. Cost of ER Spare: $800

c. Repair Time for ER: 15 days

d. No PM

7. Component 4 Name: Wtr_Jet_Pump

a. Failure Distribution: exponential

1) Mean: 300 days

b. Cost of ER Spare: $350

c. Repair Time for ER: 12 days

d. No PM

8. Component 5 Name: Auto_Lube

a. Failure Distribution: lognormal

1) Mean: 300 days

2) Standard Deviation: 25 days

b. Cost of PM Spare: $125

c. Cost of ER Spare: $125

d. Repair Time for ER: 4 days

e. Repair Time for PM: 4 days

f. Interval Between PMs: 180 days

The model will be executed for a time frame of 2500 days and statistics collected for intervals of 250 days. In this abbreviated example, only five realizations will be generated for each scenario executed. The list below summarizes the input data that controls the execution of the simu-

lation, where initially, no OPM will be performed and the AUX LOR will not be used.

1. Simulation Time Unit: d
2. Simulation Time Length: 2500
3. Number of Simulations to Run: 5
4. Statistics Collection Interval: 250
5. OPM Flag: 0
6. OPM Look-Ahead Time: 0
7. Auxiliary LOR Flag: 0

A.7.1 Creating the Example Model

The model data for the example already exists on the distribution diskette in a file named SLWT.DAT. This file may be retrieved, or recreated if desired. In either case the operation is performed from the "Model Data" menu within the Simulate program.

If "Create" is selected by pressing "c", then a name for the file must be given. Enter any valid DOS filename (including a drive designator, if desired) by typing the name and then pressing <return>. Once this step is performed the data entry screens are presented.

The first set of screens presented are the "Component Data" forms. A blank data form (with default input already provided) is displayed. Enter the data requested using the editing keys previously discussed. When

satisfied with the data entered, press the <F10> key to activate the screen bottom prompt "Do you want to exit this form [y/n]". Answer "yes" by pressing the "y" key. For component 1, the form should appear as in Figure 24.

Once this form is exited, the "Fail Data Entry" form, which requests the failure distribution parameters, is presented. Use the information in Figure 15 on page 86 to determine what values are required for the distribution that was specified. Since component 1 has a normal failure distribution, there are two parameters required. Fail Parameter #1 represents the mean and Fail Parameter #2 represents the standard deviation. Figure 25 shows what this form should look like when filled out properly. Proceed with exiting the form using the procedure discussed above.

Once the failure parameter data form has been exited, a new blank component form will be presented. Proceed with entering the rest of the component data as above, referring to the User's Guide when needed. When a blank form is presented and there are no more components to enter, press the <esc> key to complete component data entry. The "End Item Data" form will then be presented.

Fill out this form using the same methods. The proper layout of this form is shown in Figure 26. Verifying exit brings up the "LOR Data" form.

Once the LOR Data form is complete, verify an exit as with the other forms. This action brings up the last data form - "Simulation Parame-

```
Component Name:  Engine_Oil_Cooler
Failure Distribution:  L
Time Unit of Mean:  D
Note: Failure Distribution Data
      Entry Follows This Screen

Cost of ER Spares:  400
Cost of PM Spares:  400
ER Duration:  3.5
PM Duration:  3.5
Time Unit for Durations:  D
PM Policy:  1
PM Interval:  270
Time Unit for PM Interval:  D
```

Figure 24. Data Form for Component 1

ters". This form appears, and should be filled out, just as outlined in the previous discussion of the simulation parameters for the example model. Verifying an exit from this form results in saving the data to disk and returns control to the "Model Data" menu. The size of this data file is approximately 700 bytes.

Since the model data already exists on the distribution diskette, it can be retrieved without entering the data using the "Create" step. From the "Model Data" menu select "Retrieve" by pressing "r". When the file name prompt appears enter the name d:SLWT.DAT, where d: represents the id of the drive containing the file. The model data is then loaded into the MAINTAIN system.

```
        Enter Failure Distribution Data

        Fail Parameter #1:    275
        Fail Parameter #2:     40
```

Figure 25. Data Form for Component 1 Failure Parameters

A.7.2 Executing the Example Model

Once the SLWT data has been created or retrieved, the simulation can then be performed. Return to the Main Simulate menu by pressing the "q" key until the menu appears. Select the "Execute" option by pressing "x". User prompts for the seed and output file appear. The file name provided for the seed file (SEEDS.IN) matches a file on the distribution diskette. Make sure that the distribution disk is in the default drive. Press <return> to accept the seed file name as given. The output file has a default name provided. To change the name (e.g., SLWT1.OUT), type over the name already present and press <return>. Model execution then proceeds. Be patient, each iteration of the simulation takes about one minute. When the simulation is complete, control returns to the Main Simulate menu. The size of the output file created is approximately 14500 bytes.

At this point, create and execute some alternative scenarios by modifying the model data and re-executing the simulation, giving a different name to each output file. Perform the following seven scenarios, changing the

```
Standard Repair Facility
   Name:  Standard
   Capacity:  2
   Usage Cost:  110
   Unit for Cost:  D
   One-Way Transport Time:  0
   One-Way Transport Cost:  0
   Gang Repair Flag:  1

Auxiliary Repair Facility
   Name:  Overtime
   Capacity:  1
   Usage Cost:  220
   Unit for Cost:  D
   One-Way Transport Time:  0
   One-Way Transport Cost:  0
   Gang Repair Flag:  1
```

Figure 26. Data Form for LOR Data

data relative to the original input data and using the indicated output file name at execution time. Select the "Model Data" menu option from the Main Simulate menu, then select "Edit" from the Model Data menu. Locate the parameter to change from the areas presented in the Edit menu. Refer to the User's Guide for the location of each of the parameters.

1. Use a capacity of four at the STD LOR (SLWT2.OUT).
2. Use a capacity of three at the STD LOR, plus set the AUX LOR Flag to "1" (SLWT3.OUT).
3. Use a capacity of four at the STD LOR, plus set the AUX LOR Flag to "1" (SLWT4.OUT).

4. Use a capacity of three at the STD LOR, plus set the OPM Flag to "1" with a look-ahead time of 50 (SLWT5.OUT).
5. Use a capacity of four at the STD LOR, plus set the OPM Flag to "1" with a look-ahead of 50 (SLWT6.OUT).
6. Use a capacity of three at the STD LOR, plus set the AUX LOR Flag to "1", plus set the OPM Flag to "1" with a look-ahead time of 50 (SLWT7.OUT).
7. Use a capacity of four at the STD LOR, plus set the AUX LOR Flag to "1", plus set the OPM Flag to "1" with a look-ahead time of 50 (SLWT8.OUT).

Each of the above scenarios have been performed using the SLWT data and reside on the distribution disk in the files indicated. Each can be retrieved into the Analyze system for graph/print operations.

A.7.3 Analyzing the Example Output

Once the various output files have been created, they can then be loaded into the Analyze system, although, only one file can be loaded and processed at a time. This restriction limits the use of the graphs for scenario comparison, unless each is printed out and then superimposed on another. Thus, for the purposes of this example, only printed output will be considered. In particular, the following four statistics will be compared:

- Total Program Cost

- Population Availability
- Downtime Cost
- Server Utilization

There are many other statistics that could also have been investigated, but since no component parameters were changed in these scenarios, there is no reason to consider statistics specific to component ER and PM events. The following pages summarize the output from each scenario with respect to the above statistics. For the Program Cost statistic, the equivalent interval worths and the present worths at time zero of the cash flows (assuming that flows occur at the end of each interval) are presented in a table following the output listings. These computations are important because the time value of money should be considered when working with time-series cost data. An interest rate of 8 percent per 250 day interval was used for these calculations.

SLWT1 - 3 Servers

Program Cost, Total Population

Interval	Mean
1	$256735.46
2	296245.50
3	274042.26
4	257015.42
5	266646.68
6	274038.74
7	265198.54
8	279671.72
9	257958.86
10	269307.48

Availability, Total Population

Interval	Mean
1	82.21
2	80.57
3	82.89
4	84.11
5	83.47
6	82.89
7	83.41
8	82.54
9	84.02
10	83.14

Downtime Cost, Total Population

Interval	Mean
1	$177886.80
2	194264.00
3	171070.00
4	158923.20
5	165330.40
6	171089.60
7	165873.60
8	174598.80
9	159766.00
10	168572.00

Server Utilization - Standard

Interval	Mean
1	67.51
2	85.69
3	85.94
4	82.03
5	84.09
6	85.36
7	82.86
8	87.61

SLWT2 - 4 Servers

Program Cost, Total Population

Interval	Mean
1	$225799.14
2	245447.26
3	246582.58
4	252910.40
5	250162.90
6	238004.44
7	237025.84
8	217063.02
9	259841.70
10	280138.22

Availability, Total Population

Interval	Mean
1	86.52
2	86.01
3	85.98
4	85.39
5	85.60
6	86.43
7	86.47
8	87.72
9	84.99
10	83.77

Downtime Cost, Total Population

Interval	Mean
1	$134762.40
2	139934.40
3	140202.40
4	146130.40
5	143957.20
6	135736.40
7	135321.20
8	122769.60
9	150059.60
10	162277.20

Server Utilization - Standard

Interval	Mean
1	57.98
2	65.25
3	66.01
4	66.40
5	66.74
6	64.07
7	63.72
8	58.41

9	81.67	9	68.71
10	83.58	10	73.93

SLWT3 - 3 Servers and AUX LOR

Program Cost, Total Population

Interval	Mean
1	$235474.90
2	247316.54
3	241062.26
4	248678.64
5	275633.96
6	242001.36
7	244731.32
8	253191.84
9	246988.42
10	271231.88

Availability, Total Population

Interval	Mean
1	86.13
2	86.03
3	86.51
4	85.85
5	84.21
6	86.38
7	86.01
8	85.59
9	86.08
10	84.52

Downtime Cost, Total Population

Interval	Mean
1	$138684.40
2	139677.20
3	134890.40
4	141452.00
5	157891.60
6	136184.40
7	139876.80
8	144064.80
9	139212.80
10	154848.80

Server Utilization - Standard

Interval	Mean
1	66.51
2	76.73
3	76.56
4	78.91
5	82.92
6	76.82
7	78.17
8	78.06

SLWT4 - 4 Servers and AUX LOR

Program Cost, Total Population

Interval	Mean
1	$227443.00
2	222562.64
3	240095.42
4	221371.78
5	258678.32
6	245285.46
7	228625.60
8	244808.68
9	244838.58
10	246706.06

Availability, Total Population

Interval	Mean
1	87.15
2	87.57
3	86.58
4	87.76
5	85.64
6	86.29
7	87.22
8	86.29
9	86.28
10	86.14

Downtime Cost, Total Population

Interval	Mean
1	$128537.20
2	124312.00
3	134188.40
4	122357.60
5	143568.00
6	137086.00
7	127848.80
8	137075.60
9	137186.00
10	138554.00

Server Utilization - Standard

Interval	Mean
1	56.19
2	59.11
3	63.14
4	59.30
5	67.76
6	64.54
7	61.21
8	63.80

9	78.30	9	64.86
10	84.43	10	64.68

Server Utilization - Overtime

Interval	Mean
1	28.54
2	23.30
3	19.64
4	18.81
5	26.92
6	19.23
7	18.28
8	21.82
9	20.58
10	22.76

Server Utilization - Overtime

Interval	Mean
1	15.48
2	4.84
3	6.80
4	5.00
5	7.56
6	6.44
7	4.36
8	8.22
9	4.98
10	6.68

Server Utilization - Both LORs

Interval	Mean
1	57.02
2	63.37
3	62.33
4	63.89
5	68.92
6	62.43
7	63.19
8	64.00
9	63.87
10	69.01

Server Utilization - Both LORs

Interval	Mean
1	48.05
2	48.26
3	51.87
4	48.44
5	55.72
6	52.92
7	49.84
8	52.68
9	52.88
10	53.08

SLWT5 - 3 Servers and OPM 50

Program Cost, Total Population

Interval	Mean
1	$253930.18
2	259988.92
3	276020.48
4	271340.26
5	274423.44
6	271002.22
7	258490.04
8	256647.96
9	244775.54
10	278432.60

Availability, Total Population

Interval	Mean
1	82.92
2	83.79
3	82.55
4	83.34
5	82.93
6	83.29
7	84.11
8	84.06
9	85.30
10	82.61

Downtime Cost, Total Population

Interval	Mean
1	$170838.80
2	162070.00
3	174523.60
4	166584.40
5	170710.80
6	167069.20
7	158938.80
8	159389.20
9	147013.20
10	173948.40

Server Utilization - Standard

Interval	Mean
1	71.07
2	82.02
3	86.15
4	88.78
5	87.60
6	87.73
7	84.70
8	81.96

SLWT6 - 4 Servers and OPM 50

Program Cost, Total Population

Interval	Mean
1	$223452.68
2	226603.70
3	255603.38
4	233685.88
5	222733.54
6	217750.46
7	230903.08
8	227623.42
9	233138.98
10	260315.60

Availability, Total Population

Interval	Mean
1	86.64
2	87.64
3	86.04
4	87.26
5	87.94
6	88.37
7	87.52
8	87.88
9	87.51
10	85.65

Downtime Cost, Total Population

Interval	Mean
1	$133573.20
2	123643.60
3	139580.80
4	127425.60
5	120571.20
6	116266.40
7	124828.00
8	121193.20
9	124893.60
10	143540.40

Server Utilization - Standard

Interval	Mean
1	57.48
2	65.04
3	73.20
4	67.25
5	64.63
6	63.49
7	66.87
8	66.80

9	81.97	9	68.80
10	88.36	10	73.58

SLWT7 - 3 Serv., AUX LOR, OPM 50

Program Cost, Total Population

Interval	Mean
1	$241763.34
2	238777.12
3	238783.76
4	243569.64
5	240082.86
6	245378.40
7	246681.24
8	261702.28
9	255260.74
10	272682.10

Availability, Total Population

Interval	Mean
1	85.63
2	86.98
3	87.13
4	86.71
5	87.02
6	86.70
7	86.40
8	85.47
9	86.11
10	84.87

Downtime Cost, Total Population

Interval	Mean
1	$143718.80
2	130178.40
3	128650.80
4	132937.20
5	129832.4a
6	132954.40
7	135996.80
8	145285.60
9	138904.00
10	151312.40

Server Utilization - Standard

Interval	Mean
1	67.02
2	79.81
3	80.66
4	79.33
5	79.21
6	79.46
7	78.35
8	82.27

SLWT8 - 4 Serv., AUX LOR, OPM 50

Program Cost, Total Population

Interval	Mean
1	$216656.42
2	215964.50
3	242273.92
4	220063.22
5	254254.72
6	227836.36
7	227928.48
8	219656.00
9	224198.88
10	243884.04

Availability, Total Population

Interval	Mean
1	88.08
2	88.68
3	87.27
4	88.70
5	86.44
6	88.06
7	88.26
8	88.53
9	88.34
10	87.06

Downtime Cost, Total Population

Interval	Mean
1	$119220.00
2	113200.00
3	127293.20
4	113035.20
5	135634.00
6	119361.60
7	117443.20
8	114747.20
9	116602.80
10	129382.80

Server Utilization - Standard

Interval	Mean
1	56.45
2	62.80
3	69.55
4	63.73
5	70.73
6	64.72
7	66.33
8	64.01

9	83.99	9	64.61
10	85.22	10	69.32

Server Utilization - Overtime

Interval	Mean
1	29.73
2	20.15
3	20.79
4	23.75
5	23.42
6	25.99
7	25.85
8	27.75
9	25.39
10	29.53

Server Utilization - Overtime

Interval	Mean
1	13.08
2	5.59
3	7.21
4	8.44
5	10.36
6	8.13
7	6.75
8	4.29
9	8.01
10	7.72

Server Utilization - Both LORs

Interval	Mean
1	57.70
2	64.90
3	65.70
4	65.43
5	65.26
6	66.09
7	65.23
8	68.64
9	69.34
10	71.29

Server Utilization - Both LORs

Interval	Mean
1	47.78
2	51.36
3	57.08
4	52.67
5	58.65
6	53.40
7	54.41
8	52.07
9	53.29
10	57.00

Comparison of Measures of Performance for the Total Program Cost

Scenario	Equivalent Interval Worth	Present Worth
SLWT1	$270,034	$1,811,948
SLWT2	243,917	1,636,704
SLWT3	249,339	1,673,087
SLWT4	236,623	1,587,757
SLWT5	264,531	1,775,023
SLWT6	232,422	1,559,569
SLWT7	246,570	1,654,503
SLWT8	228,499	1,533,245

One note about differences between scenarios in which the STD LOR capacity is changed from three to four. The Program Cost of each policy includes only those LOR resource usage costs that occur when a unit of capacity is being utilized (not idle). There is certainly some other incremental difference in costs incurred when the capacity of a LOR is increased. That is, when a server is not being utilized, there should be a cost associated with having an extra unit of capacity idle. This difference should be taken into consideration when performing a trade-off analysis between alternative scenarios with varying LOR capacities. In this example problem it is assumed that when LOR servers are idle, they are used in other productive activities, and the idle cost is absorbed somewhere else.

From an analysis of the output data, it is hard to select the best overall scenario. Using the equivalent worth of the Program Cost as the main performance indicator, then policy SLWT8 would be the preferred choice. Though note that only a few of many possible scenarios were investigated. There were many other combinations of changes to the input data that could have been performed: PM policies could be changed, different look-ahead times could be used, service capacities could be further increased, and so on. Thus, it cannot be said that policy SLWT8 results in the optimum maintenance system, more investigation would be necessary before a statement is made. What can be said is that policy SLWT8 provides a better maintenance system than specified in the original model data (STD LOR capacity of three, no AUX LOR, no OPM). Users should find that the

MAINTAIN system can provide useful results for program scenarios that might go uninvestigated using manual methods.

APPENDIX B. MAINTAIN PROGRAM LISTING

This section presents listings of the "C" source code for the individual modules used in the MAINTAIN system. The program was compiled using the large model option (up to one megabyte of program and data) of the Mark Williams C Programming System, Version 3.0. Two additional add-on utility libraries, whose source code is not listed, were also utilized in the development of MAINTAIN:

1. "Windows for Data" from Vermont Creative Software, was used in the development of the data entry environment.
2. "C Utility Library" from Essential Software, was used in the development of the graphics routines that produce plots of the time-series statistics.

There was a problem encountered using the "C Utility Library", in that one critical routine did not function properly. The "grline" routine, which was an assembly language program that draws a line from point (x1,y1) to point (x2,y2), did not function and had to be recreated using modified versions of other library functions. Thus, line draw times are longer than they might have been otherwise.

The MAINTAIN system consists of three executable programs, MAINTAIN, SIMULATE, and ANALYZE, whose respective source code and library modules are listed below:

1. MAINTAIN.EXE

    a. maintain.c

    b. Windows for Data Library

2. SIMULATE.EXE

    a. sm_main.h (header definition file)

    b. main.c

    c. mainmenu.c

    d. loadcal.c

    e. lor.c

    f. opm.c

    g. update.c

    h. stats.c

    i. randsamp.c

    j. showdata.c

    k. create.c

    l. readdata.c

    m. Windows for Data Library

3. ANALYZE.EXE

    a. gr_main.h (header definition file)

    b. analyze.c

    c. readdata.c

    d. readstat.c

    e. graphmnu.c

    f. graphdrv.c

    g. printdrv.c

    h. prepare.c

i. graph.c

j. Windows for Data Library

k. C Utility Library

In addition to compiling and linking the source code files to form an executable program, the library files specified above must also be included for the linking session.

The following pages contain listings of each of the above source code modules, presented in approximately the same order.

```
/*
 * File Name:  MAINTAIN.C
 */
#include <stdio.h>
#include <bios.h>
#include <window.h>

/*
 * The main menu module for "Maintain". Displays Title/Copywrite and
 * displays menu.  Loads a user's choice of "Simulate" or "Analyze"
 * programs.  Each program returns to this menu.
 */
main()
{
   int choice;
   int verify;
   void main_menu();
   WINDOW wn;
   WINDOW wait_wn;
   WINDOW title;

   init_wfc();

   defs_wn(&wait_wn,10,21,3,37,BDR_DLNP);     /* define status window */
   wait_wn.att = LHIGHLITE;
   sw_csadv(OFF,&wait_wn);
   sw_cleor(OFF,&wait_wn);

   defs_wn(&title,1,6,22,66,BDR_DLNP);            /* define title window */
   cls();
   csr_hide();
   set_wn(&title);
   title.att = LHIGHLITE;
   v_plst(1,CENTER_TXT,"MAINTAIN: A Simulation-Based DSS",&title);
   v_plst(2,CENTER_TXT,"for Maintenance Logistics Planning",&title);
   title.att = LNORMAL;
   v_plst(4,CENTER_TXT,"by ",&title);
   v_plst(6,CENTER_TXT,"Dr. Marvin H. Agee",&title);
   v_plst(8,CENTER_TXT,"and",&title);
   v_plst(10,CENTER_TXT,"Mark S. Gallion",&title);
   v_plst(12,1,"Department of Industrial Engineering ",&title);
   v_st("and Operations Research",&title);
   v_plst(13,CENTER_TXT,"Virginia Polytechnic Institute and State University",
          &title);
   v_plst(14,CENTER_TXT,"Blacksburg, Virginia",&title);
   v_plst(16,CENTER_TXT,"(C) Copyright December 1986, All Rights Reserved.",
          &title);
   title.att = LHIGHLITE;
   v_plst(19,CENTER_TXT,"<Press Any Key to Continue>",&title);
   ki();
   unset_wn(&title);

   defs_wn(&wn,7,16,10,47,BDR_DLNP);          /* define menu window */
   sw_name("Main Menu", &wn);
   sw_plcsr(ON, &wn);
   main_menu(&wn);         /* define and set menu on screen for first time */

   for (;;)
   {
      choice = ki();      /* get keystroke from user */
      switch ( choice )   /* branch on keystroke */
      {
         case 115:
         case  83:
                  unset_wn(&wn);
                  set_wn(&wait_wn);
                  v_plst(0,0,"Wait...Loading Simulation Program",&wait_wn);
```

```
                execall("simulate.exe","");
                unset_wn(&wait_wn);
                main_menu(&wn);
                break;

        case 97:
        case 65:
                unset_wn(&wn);
                set_wn(&wait_wn);
                v_plst(0,1,"Wait...Loading Analyze Program",&wait_wn);
                execall("analyze.exe","");
                unset_wn(&wait_wn);
                main_menu(&wn);
                break;

        case 113:
        case  81:
                    if ( (verify = verify_exit(15,18)) == 1)
                    {
                       unset_wn(&wn);
                       csr_show();
                       exit();
                    }
                    else
                       csr_hide();
                    break;
        default:
                bell();
      }
   }
}

void main_menu(wn)
WINDOW *wn;
{
   cls();
   set_wn(wn);     /* place window on screen */

   v_plst(1,CENTER_TXT,"Maintenance Simulation System",wn);
   wn->att = LHIGHLITE;
   v_plst(3,13,"s",wn);
   v_plst(4,13,"a",wn);
   v_plst(5,13,"q",wn);
   wn->att = LNORMAL;

   v_plst(3,15,"- Simulate",wn);
   v_plst(4,15,"- Analyze",wn);
   v_plst(5,15,"- Quit Program",wn);
   csr_hide();

   return;
}


verify_exit(row, col)
int row;
int col;
{
   int choice;
   WINDOW exit_wn;

   defs_wn(&exit_wn,row,col,1,37,BDR_OP);     /* define status window */
   exit_wn.att = LHIGHLITE;
   sw_popup(ON,&exit_wn);
   sw_plcsr(ON, &exit_wn);
   set_wn(&exit_wn);
   v_plst(0,0,"Do you really want to exit? (y/n) ",&exit_wn);
```

```
   for (;;)
   {
      choice = ki();
      switch ( choice )
      {
         case 121:            /* yes */
         case  89:
                  unset_wn(&exit_wn);
                  return(1);
         case 110:
         case  78:            /* no */
                  unset_wn(&exit_wn);
                  return(0);
         default:
                  bell();
      }
   }
}
```

```
/*
 * File Name:  SM_MAIN.H
 *
 * This header file is included in the main module of the Simulate
 * system (MAIN.C).  A duplicate of this file, with the global
 * declarations following the last "typedef" changed to "external",
 * is included in each module of Simulate.
 */

#define MAX_COMP 25                 /* maximum number of components allowed */
                                    /* in an EI */
#define TRUE  1
#define YES 1
#define FALSE 0
#define NO 0

#define STD_LOR   0                 /* next_serv array element for std lor */
#define AUX_LOR   1                 /* next_serv array element for aux lor */

typedef struct emp_data             /* list node for empirical distribution */
        {
        double probab;              /* probability interval */
        double response;            /* response value */
        struct emp_data *next_intv;    /* ptr to next interval node */
        }
        EMP_DATA;

typedef struct component
        {
        char name[20];              /* name of component */
        char fail_dist;             /* char that specifies fail dist */
        char u_fail;                /* time unit for mean of dist */
        double parm1,parm2;         /* parameters for failure dist */
        struct emp_data *emp_head; /* first node in list of emp data*/
        int emp_nodes;              /* number nodes read into list */
        double c_pmspare;           /* cost of spares reqd for pm */
        double c_erspare;           /* cost of spares reqd for er */
        char u_durn;                /* time unit for service durations */
        double er_durn;             /* service time for ER event */
        char pm_polcy;              /* flag enabling PM policy */
        double pm_durn;             /* service time for PM event */
        double pmintv;              /* interval after which pm is performed */
        char u_pmintv;              /* unit of interval specification */
        }
        COMPONENT;

typedef struct mof_sched            /* list node for MOF schedule */
        {
        int s_time;                 /* time for current MOF event */
        char u_time;                /* time unit of acquistion times */
        int s_block;                /* block size for current MOF event */
        struct mof_sched *next;     /* ptr to next scheduled MOF event */
        }
        MOF_SCHED;

typedef struct popln
        {
        char name[20];              /* EI name */
        char data_file[16];         /* file where data read from */
        char mof;                   /* type of mof */
          int mof_init;             /* initial number of end items */
          struct mof_sched *sched; /* head of "schedule" linked-list */
          int mof_nodes;            /* number nodes read into schedule */
        double c_eidt;              /* end item downtime cost/unit */
        char u_eidt;                /* time unit for downtime cost */
        char u_sim;                 /* time unit used in simulation */
        double sim_lgth;            /* time frame for simul, 0 -> what? */
```

```
        int sim_num;                /* number of simuls to run */
        double intv_size;           /* size of stats collection interval */
        int opm_flag;               /* flag to enable OPM scanning */
        double opm_look;            /* how far to look ahead if OPM "on" */
        int aux_flag;               /* flag to enable AUX LOR checking */
        }
        POPLN;

typedef struct lor
        {
        char name[20];              /* name of lor */
        int capacity;               /* capacity of lor (man-units/unit time) */
        double c_usage;             /* cost of resource usage/unit time */
        char u_usage;               /* time unit for resource usage cost */
        double t_trans;             /* one-way time for transport to lor */
        double c_trans;             /* one-way cost for transport to lor */
        int gang_flag;              /* flag to indicate gang repair enabled */
        }
        LOR;

typedef struct comp_event        /* calendar node for component event lists */
        {
        char type;                  /* type of event, ER or PM */
        double time;                /* time that event occurs */
        double lifetime;            /* sampled lifetime for component */
        int comp_id;                /* integer id of component */
        struct comp_event *next;    /* ptr to next component in list */
        }
        COMP_EVENT;

typedef struct ei_event             /* calendar node for EI lists */
        {
        int ei_id;                  /* integer id of EI */
        struct COMP_EVENT *comp_head; /* ptr to head of component list */
        struct ei_event *next;      /* ptr to next EI list */
        }
        EI_EVENT;


typedef struct dt_intv             /* node of "dt" stats list */
        {
        int id;                     /* interval number */
        double brkpt;               /* upper limit time for interval */
        double downtime;            /* cumulative downtime */
        int dt_count[2];            /* number of dt instances at AUX/STD lor */
        struct dt_intv *next;       /* ptr to next interval node */
        }
        DT_INTV;

typedef struct sp_intv             /* node of "spares" stats list */
        {
        int id;                     /* interval number */
        double brkpt;               /* upper limit time for interval */
        int er_count[2];            /* counter for ERs at each LOR */
        int pm_count[2];            /* counter for PMs at each LOR */
        double rep_time[2];         /* counter for service time at each LOR */
        struct sp_intv *next;       /* ptr to next interval node */
        }
        SP_INTV;

COMPONENT *comp_data[MAX_COMP];  /* array of pointers to structures */
                                 /* to describe components attached */
                                 /* to an end item */

POPLN popln_data;                /* structure to hold  pertinent */
                                 /* simulation and end item data */
```

```
LOR lor_data[2];                    /* structure to hold data about */
                                    /* each LOR available to the popln */
                                    /* only 3 levels are defined */

EI_EVENT *ei_head;                  /* pointer to the highest priority */
                                    /* EI in event "block" (calendar) */

COMP_EVENT *opm_group[MAX_COMP];    /* array of ptrs to hold grouped */
                                    /* component events for current EI */

double *next_serv[2];               /* ptrs to time servers are avail */
                                    /* for each LOR */

DT_INTV *dt_head;                   /* ptr to head of downtime accum list */


SP_INTV *sp_head[MAX_COMP];         /* array of ptrs to component lists that */
                                    /* accumulate ER/PM spares demands */

int _num_comp;                      /* global number components defined */
int _num_items;                     /* total number EI's in system at */
                                    /* any time */
int _sim_err;                       /* set to defined error number when */
                                    /* error occurs anywhere in simul */
int _sim_num;                       /* current simulation number being executed */
double _sim_time;                   /* current event time */
```

```
/*
 * File Name:  MAIN.C
 */
#include <stdio.h>
#include <wfd.h>
#define WFD_FLOAT
#include <wfd_glob.h>
#include "sm_main.h"

/*
 * Function: Main module for the Simulate program
 */
main( )
{
   init_wfd( );
   main_menu( );
}
```

```
/*
 * File Name:  MAINMENU.C
 */
#include <stdio.h>
#include <wfd.h>
#include "sm_struc.h"

/*
 * main_menu( )
 *
 * Usage: Display and process the main system menu.
 * Returns: None
 */
void main_menu( )
{
   char choice;
   char seed_file[16] = "seeds.in       ";     /* default seed file name */
   char stat_file[16] = "stats.out      ";     /* default stat file name */
   int verify;
   FILE *fseed;
   FILE *fstat;
   static int called = FALSE;
   void edit_menu( );
   void def_main( );
   char *strcpy( );
   WINDOW wn;

   if (called == FALSE)
   {
      strcpy(popln_data.data_file,"None Defined   ");
      called == TRUE;
   }

   sw_att(LDOS,&help_wn);
   defs_wn(&wn,4,16,12,47,BDR_DLNP);      /* define menu window */
   sw_name("Simulation Menu", &wn);
   sw_plcsr(ON, &wn);
   cls( );
   def_main(&wn);         /* define and set menu on screen for first time */

   for (;;)
   {
      choice = ki( );      /* get keystroke from user */
      switch ( choice )   /* branch on keystroke */
      {
         case 109:                        /* "m" */
         case 77 :                        /* "M" */
                  unset_wn(&wn);
                  edit_menu( );           /* retreive/edit model data */
                  def_main(&wn);
                  break;
         case 120:                        /* "x" */
         case 88 :                        /* "X" */
                  if (strncmp(popln_data.data_file,"None Defined",12) != 0)
                  {
                     get_data(14,19,"RN Seed Input File:", def_pic('X',15),
                              F_STRING, seed_file);
                     if ( (fseed = fopen(seed_file,"r")) != NULL )
                     {
                        get_data(14,19,"Statistics Output File:",
                                 def_pic('X',15),F_STRING, stat_file);
                        if ( (fstat = fopen(stat_file,"w")) != NULL )
                        {
                           csr_hide( );
                           unset_wn(&wn);
                           exe_simul( fseed, fstat );
                           def_main(&wn);
```

```
                        fclose( fseed );
                        fclose( fstat );
                     }
                     else
                     {
                        csr_hide( );
                        bell( );
                        fclose( fseed );
                        goto bad_name;
                     }
                  }
                  else
                  {
                     csr_hide( );
                     bell( );
                     goto bad_name;
                  }
               }
               else
                  bell( );

               bad_name:
                  break;
         case 113:                        /* "q" */
         case 81 :                        /* "Q" */
               if ( (verify = verify_exit(14,18)) == 1)
               {
                  unset_wn(&wn);
                  return;
               }
               csr_hide( );
               break;
         default :
               bell( );
      }
   }
}


void def_main(wn)
WINDOW *wn;
{
   set_wn(wn);     /* place window on screen */

   v_plst(1,1,"Current Active Data File: ",wn);

   wn->att = LHIGHLITE;
   v_st(popln_data.data_file, wn);
   v_plst(5,12,"m",wn);
   v_plst(6,12,"x",wn);
   v_plst(7,12,"q",wn);
   wn->att = LNORMAL;

   v_plst(3,CENTER_TXT,"Menu Choices",wn);
   v_plst(5,14,"- Model Data",wn);
   v_plst(6,14,"- Execute Model",wn);
   v_plst(7,14,"- Quit Program",wn);
   csr_hide( );

   return;
}

verify_exit(row, col)
int row;
int col;
{
   int choice;
```

```
   WINDOW exit_wn;

   defs_wn(&exit_wn,row,col,1,37,BDR_OP);     /* define status window */
   exit_wn.att = LHIGHLITE;
   sw_popup(ON,&exit_wn);
   sw_plcsr(ON, &exit_wn);
   set_wn(&exit_wn);
   v_plst(0,0,"Do you really want to exit? (y/n) ",&exit_wn);
   for (;;)
   {
      choice = ki();
      switch ( choice )
      {
         case 121:            /* yes */
         case  89:
                  unset_wn(&exit_wn);
                  return(1);
         case 110:
         case  78:            /* no */
                  unset_wn(&exit_wn);
                  return(0);
         default:
                  bell();
      }
   }
}
```

```
/*
 * File Name:  LOADCAL.C
 */
#include <stdio.h>
#include "sm_struc.h"

/*
 * void load_cal(iseed)
 *
 * Usage: loads the simulation event calendar with initial
 *        events as dictated by component failure distributions
 *        and MOF considerations. Allocates and initializes
 *        the array of ptrs (next_serv) that keep track of
 *        all server's next available time in each LOR.
 *        Clears event calendar that might have existed from
 *        a previous load.
 *        Initializes LOR server avail array and sets global
 *        simulation time to zero (_sim_time).
 * Returns: None
 */
void load_cal( iseed )
int iseed;                          /* new value for r.n. seed */
{
   EI_EVENT *now_ei;
   COMP_EVENT *now_comp;
   MOF_SCHED *ptr;

   register int i, j;
   double rand_samp( );
   double lifetime;
   char *calloc( );
   void push_ei( ), push_comp( );

   srand(iseed);                    /* seed r.n. generator */

   while (ei_head)     /* deallocate any previously allocated event calendar */
   {
      now_ei = ei_head;
      ei_head = ei_head->next;
      while (now_ei->comp_head)
      {
         now_comp = now_ei->comp_head;

         now_ei->comp_head = now_comp->next;
         free( (char *) now_comp );
      }
      free( (char *) now_ei );
   }

   /*
    * initialize extern head ptrs for statistic collection lists
    * (either for first simulation or after any previous list
    * has been removed during list flushing at end of sim_lgth
    */
   dt_head = (DT_INTV *) calloc( 1, sizeof(DT_INTV) );
   dt_head->downtime = 0;
   dt_head->id = 1;
   dt_head->brkpt = popln_data.intv_size;
   dt_head->dt_count[0] = dt_head->dt_count[1] = 0;
   dt_head->next = NULL;

   for (i = 0; i < _num_comp; ++i)
   {
      sp_head[i] = (SP_INTV *) calloc( 1, sizeof(SP_INTV) );
      sp_head[i]->id = 1;
      sp_head[i]->brkpt = popln_data.intv_size;
      sp_head[i]->er_count[0] = sp_head[i]->er_count[1] = 0;
```

```
   sp_head[i]->pm_count[0] = sp_head[i]->pm_count[1] = 0;
   sp_head[i]->rep_time[0] = sp_head[i]->rep_time[1] = 0.0;
   sp_head[i]->next = NULL;
}

/* loop thru all end items in "to be fielded list", sampling
 * failure times for each component by passing component
 * address to sampling routine which gets the necessary
 * parameters from the component structure passed
 */
for (i=0; i < popln_data.mof_init; ++i)
{
   now_ei = (EI_EVENT *) calloc(1,sizeof(EI_EVENT));
   now_ei->ei_id = i;

   for (j=0; j < _num_comp; ++j)
   {
      now_comp = (COMP_EVENT *) calloc(1, sizeof(COMP_EVENT));
      lifetime = rand_samp(j);        /* sample lifetime of jth component */
      now_comp->lifetime = lifetime;      /* set lifetime attrib for comp */
      /*
       * check and see if pm or er occurs for the just serviced
       * component and compute event times accordingly
       */
      if ( comp_data[j]->pm_polcy != '0' &&
           lifetime >= comp_data[j]->pmintv)        /* pm will take place */
      {
         now_comp->time = comp_data[j]->pmintv;
         now_comp->type = 'P';
      }
      else                                          /* er will take place */
      {
         now_comp->time = lifetime;
         now_comp->type = 'E';
      }
      now_comp->comp_id = j;        /* assign identification to event */

      push_comp(now_ei, now_comp);     /* place comp event into comp list */
   }
   push_ei(now_ei);                       /* place comp list into ei list */
}

_num_items = popln_data.mof_init;    /* set number EIs initially in popln */

/*
 * Following code generates first set of events for EIs that are
 * fielded according to a known schedule.  Initial information
 * is relative to the time that the groups are fielded.
 */
if (popln_data.mof == 's' || popln_data.mof == 'S')
{
   for (ptr = popln_data.sched; ptr; ptr = ptr->next)
   {
      for (i=0; i < ptr->s_block; ++i, _num_items++)
      {
         now_ei = (EI_EVENT *) calloc(1,sizeof(EI_EVENT));
         now_ei->ei_id = _num_items;

         for (j=0; j < _num_comp; ++j)
         {
            now_comp = (COMP_EVENT *) calloc(1, sizeof(COMP_EVENT));
            lifetime = rand_samp(j);  /* sample lifetime of jth component */
            now_comp->lifetime = lifetime;/* set lifetime attrib for comp */
            /*
             * check and see if pm or er occurs for the just serviced
             * component and compute event times accordingly
             */
```

```
            if ( comp_data[j]->pm_policy != 'O' &&
                 lifetime >= comp_data[j]->pmintv)  /* pm will take place */
            {
               now_comp->time = (double) ptr->s_time +
                                comp_data[j]->pmintv;
               now_comp->type = 'P';
            }
            else                                    /* er will take place */
            {
               now_comp->time = (double) ptr->s_time + lifetime;
               now_comp->type = 'E';
            }
            now_comp->comp_id = j;  /* assign identification to event */
            push_comp(now_ei, now_comp);    /* place comp event into list */
         }
         push_ei(now_ei);                /* place comp list into ei list */
      }
   }
   /*
    * following code deallocates, allocates, then initializes
    * servers' next avail time for each server of each LOR to
    * beginning of simulation. Also initializes global value of
    * _sim_time to zero.
    */
   for (i=0; i < 2; ++i)
   {
      free( (char *) next_serv[i] );
      next_serv[i] = (double *) calloc( lor_data[i].capacity, sizeof(double) );
      for (j=0; j < lor_data[i].capacity; ++j)
         *(next_serv[i] + j) = 0.0;
   }
   _sim_time = 0.0;                    /* initialize global simulation time */
   return;
}


/*
 * void push_comp(EI_EVENT *ei_ptr, COMP_EVENT *comp_ptr)
 *
 * Usage: Adds component event to the list maintained for
 *        the end item to which that component belongs.
 *        If new_event is first event in cal then head ptr
 *        is initialized.
 * Returns: None
 */
void push_comp(ei_ptr, comp_ptr)
EI_EVENT *ei_ptr;
COMP_EVENT *comp_ptr;
{
   register COMP_EVENT *prev;

   if (ei_ptr->comp_head == NULL)              /* if component list is empty */
   {
      comp_ptr->next = NULL;
      ei_ptr->comp_head = comp_ptr;
      return;
   }
   if (comp_ptr->time <= ei_ptr->comp_head->time)     /* new is head of list */
   {
      comp_ptr->next = ei_ptr->comp_head;
      ei_ptr->comp_head = comp_ptr;
      return;
   }

   prev = ei_ptr->comp_head;                          /* init temporary ptr */
```

```
   while (prev->next != NULL && prev->next->time <= comp_ptr->time)
      prev = prev->next;

   comp_ptr->next = prev->next;
   prev->next = comp_ptr;
   return;
}

/*
 * void push_ei(EI_EVENT *ei_ptr)
 *
 * Usage: Locates and pushes an ei event ptr into the list
 *        of prioritized eis, based on the event time
 *        of the first component in the ei's component list.
 * Returns: None
 */

void push_ei(ei_ptr)
EI_EVENT *ei_ptr;
{
   register EI_EVENT *save;

   if (ei_head == NULL)                              /* check if list empty */
   {
      ei_head = ei_ptr;
      ei_ptr->next = NULL;
      return;
   }
   if (ei_ptr->comp_head->time <= ei_head->comp_head->time)   /* new is head */
   {
      ei_ptr->next = ei_head;
      ei_head = ei_ptr;
      return;
   }

   save = ei_head;                                    /* init temporary ptr */

   while (save->next != NULL &&
          save->next->comp_head->time <= ei_ptr->comp_head->time)
      save = save->next;

   ei_ptr->next = save->next;
   save->next = ei_ptr;
   return;
}

/*
 * pop_comp( ei_ptr, comp_ptr )
 *
 * Function: Removes a component node from the event calendar
 * Returns: None
 */
void pop_comp( ei_ptr, comp_ptr )
EI_EVENT *ei_ptr;
COMP_EVENT *comp_ptr;
{
   register COMP_EVENT *temp, *pred;

   for (temp = ei_ptr->comp_head; temp; temp = temp->next)
   {
      if (temp == comp_ptr)                        /* found the node to pop */
      {
         if (comp_ptr == ei_ptr->comp_head)         /* popping head of list */
            ei_ptr->comp_head = comp_ptr->next;
         else
            pred->next = comp_ptr->next;           /* popping inside of list */
         return;
```

```
      }
      pred = temp;
   }
   _sim_err = 1;     /* set global error flag to indicate pop node not found */
   return;
}

/*
 * pop_ei( ei_ptr )
 *
 * Function: Removes an EI node from the event calendar
 * Returns: None
 */
void pop_ei( ei_ptr )
EI_EVENT *ei_ptr;
{
   register EI_EVENT *temp, *pred;

   for (temp = ei_head; temp; temp = temp->next)
   {
      if (temp == ei_ptr)                          /* found the node to pop */
      {
         if (ei_ptr == ei_head)                     /* popping head of list */
            ei_head = ei_ptr->next;
         else
            pred->next = ei_ptr->next;             /* popping inside of list */
         return;
      }
      pred = temp;
   }
   _sim_err = 1;     /* set global error flag to indicate pop node not found */
   return;
}
```

```
/*
 * File Name:  LOR.C
 */
#include <stdio.h>
#include <wfd.h>
#include "sm_struc.h"

/*
 * exe_simul( seed_file, stat_file )
 *
 * Usage: Routine to grab PM and ER events off of event list and
 *        act upon them.
 *        String passed is file name for output data file.
 *        Routine operates until next event time is beyond
 *        specified simulation duration (popln_data.sim_lgth).
 *        Calls the necessary routines to:
 *        - read and reset queuing for LOR repair channels
 *        - determine historical times for output record records
 *        - write the output record
 *        - generate new event for "repaired" components
 *        - update the event calendar with new event
 *
 */
exe_simul( fseed, fstat )
FILE *fseed;              /* input file containing r.n. seeds */
FILE *fstat;              /* output file for simulation statistics */
{
   int iseed;             /* var to hold the currently read seed */
   double avl_time;       /* absolute time that first server is available */
   double cmp_time;       /* completion time of service activity */
   double rep_time;       /* time required in service facility for cur event */
   double need_time;
   double wait_time;
   double std_cost;
   double aux_cost;
   int wait_flag;
   int fail_flag;
   int fail_indx;
   int serv_id;           /* id of server assigned to service event */
   int comp_id;           /* counter for flush spares list loop */
   int num_grp;           /* number of comps in opm grouping */
   int best_lor;          /* return int for cost compare of STD and AUX lor */
   SP_INTV *sp_ptr;
   register COMP_EVENT *cur_comp;
   EI_EVENT *cur_ei;
   DT_INTV *dt_ptr;
   void updt_age( ), gnrt_comp( ), load_cal( ), lor_updt( );
   void sp_collect( ), dt_collect( );
   WINDOW wn;

   save_data(fstat);

   defs_wn(&wn,10,20,3,36,BDR_DLNP);     /* define status window */
   sw_csadv(OFF,&wn);
   sw_cleor(OFF,&wn);
   csr_hide( );
   set_wn(&wn);                          /* place window on screen */
   mv_cs(0,0,&wn);
   v_st("Executing Simulation Number:",&wn);
   wn.att = LHIGHLITE;

   for (_sim_num = 1; _sim_num <= popln_data.sim_num; ++_sim_num)
   {
      mv_cs(0,29,&wn);
      v_printf(&wn,"%3d", _sim_num);

      fscanf(fseed,"%d", &iseed);      /* read next seed value */
```

```
fprintf(fstat,"*---------- sim = %3d, seed = %5d ----------*\n",
        _sim_num, iseed);
load_cal( iseed );              /* load initial event calendar */

while ( ei_head->comp_head->time <= popln_data.sim_lgth )
{
   cur_ei = ei_head;                /* set working names for head */
   cur_comp = cur_ei->comp_head;    /* ei and component */

   _sim_time = cur_comp->time;      /* set global simulation time */

   if (cur_comp->type == 'E')       /* determine repair time to use */
      rep_time = comp_data[cur_comp->comp_id]->er_durn;
   else
   if (cur_comp->type == 'P')
      rep_time = comp_data[cur_comp->comp_id]->pm_durn;

   serv_id = next_free( next_serv[STD_LOR], STD_LOR, &avl_time );
   /* get time and id of first */
   /* avail server at std lor */
   num_grp = 0;
   fail_flag = 0;

   wait_time = avl_time - (_sim_time + lor_data[STD_LOR].t_trans);
   if (wait_time > 0)
      wait_flag = 1;
   else
      wait_flag = 0;

   /*
    * now for the meat of the program
    */
   if ( (popln_data.opm_flag) || (cur_comp->type == 'P' && (wait_flag)) )
   {
      if ( (wait_flag) && (cur_comp->type == 'P') )
      {
         num_grp = pm_wait( cur_ei, avl_time, &fail_flag, &fail_indx);

         if ( (num_grp < MAX_COMP) && (popln_data.opm_flag) )
         {
            if (fail_flag)    /* failure during PM wait (and opm) */
            {
               num_grp += opm_scan(cur_ei, (opm_group[fail_indx]->time +
                                popln_data.opm_look), num_grp);
               need_time = opm_group[fail_indx]->time;
            }
            else     /* no failure during PM wait (and opm) */
            {
               num_grp += opm_scan(cur_ei, (avl_time +
                                popln_data.opm_look), num_grp);
               need_time = avl_time;
            }
         }
         else     /* PM and wait with no_opm (or ALL comps grouped) */
         {
            if (fail_flag)
               need_time = opm_group[fail_indx]->time;
            else
               need_time = avl_time;
         }
      }
      else              /* ER wait/no_wait or PM no_wait */
      {
         num_grp = opm_scan(cur_ei, (_sim_time +
                           popln_data.opm_look), 0);
         need_time = cur_comp->time;
         if (cur_comp->type == 'E')
```

```
            fail_flag = 1;
         }

         if ( (popln_data.aux_flag) && (wait_flag) && (fail_flag) )
            best_lor = lor_compare(num_grp, need_time, &std_cost, &aux_cost);
         else
            best_lor = STD_LOR;

         lor_updt(best_lor, num_grp, need_time, cur_ei, fstat);
      }
      else     /* no opm or no wait */
      {
         if ( (popln_data.aux_flag == 0) || (!wait_flag) )
         {
            if (!wait_flag)
            {
               cmp_time = _sim_time + (2 * lor_data[STD_LOR].t_trans) +
                          rep_time;
               sp_collect(_sim_time, cur_comp->comp_id, cur_comp->type,
                          STD_LOR, _sim_time + lor_data[STD_LOR].t_trans,
                          cmp_time - lor_data[STD_LOR].t_trans, fstat);
               *(next_serv[STD_LOR] + serv_id) = _sim_time + rep_time +
                                                 lor_data[STD_LOR].t_trans;
            }
            else   /* ER, no aux, and wait after transport to lor */
            {
               cmp_time = avl_time + rep_time + lor_data[STD_LOR].t_trans;
               sp_collect(_sim_time, cur_comp->comp_id, cur_comp->type,
                          STD_LOR, avl_time, cmp_time -
                          lor_data[STD_LOR].t_trans, fstat);
               *(next_serv[STD_LOR] + serv_id) += rep_time;
            }
            dt_collect(_sim_time, STD_LOR, _sim_time, cmp_time, fstat);
            pop_comp(cur_ei, cur_comp);
            updt_age(cur_ei, (cmp_time - _sim_time));
            gnrt_comp(cur_ei, cur_comp, cmp_time);
         }
         else     /* ER, with aux, no opm, and wait */
         {
            opm_group[0] = cur_comp;
            pop_comp(cur_ei, cur_comp);
            best_lor = lor_compare(1, _sim_time, &std_cost, &aux_cost);
            lor_updt(best_lor, 1, _sim_time, cur_ei, fstat);
         }
      }
   }        /* end of individual simulation run */
   /*
    * flush statistic collector lists to stat_file, freeing
    * space that was allocated to list
    */
   while (dt_head)
   {
      fprintf(fstat,"D %2d %7.2f %7.2f %2d %2d\n",
              dt_head->id, dt_head->brkpt, dt_head->downtime,
              dt_head->dt_count[0], dt_head->dt_count[1]);
      dt_ptr = dt_head;
      dt_head = dt_head->next;
      free( dt_ptr );
   }
   for (comp_id = 0; comp_id < _num_comp; ++comp_id)
   {
      while (sp_head[comp_id])
      {
         fprintf(fstat,"L %2d %2d %7.2f %2d %2d %2d %2d ",
              sp_head[comp_id]->id, comp_id, sp_head[comp_id]->brkpt,
              sp_head[comp_id]->er_count[0], sp_head[comp_id]->er_count[1],
              sp_head[comp_id]->pm_count[0], sp_head[comp_id]->pm_count[1]);
```

```
            fprintf(fstat,"%7.2f %7.2f\n",
                 sp_head[comp_id]->rep_time[0], sp_head[comp_id]->rep_time[1]);
            sp_ptr = sp_head[comp_id];
            sp_head[comp_id] = sp_head[comp_id]->next;
            free( sp_ptr );
         }
      }
   }
   avg_ei(fstat);          /* compute avg num EIs (by intveral) in popln */

   fclose(fseed);          /* close input/output files */
   fclose(fstat);

   unset_wn(&wn);          /* remove status window */
   csr_show();
   return;
}
```

```
/*
 * File Name:  OPM.C
 */
#include <stdio.h>
#include "sm_struc.h"

/*
 * pm_wait(cur_ei, avl_time, fail_flag, fail_indx)
 *
 * Function: Performs the opportunistic "packing" function of determining
 *           and scheduling events which are grouped together when PM
 *           event must wait for service, all other related PM's during
 *           the wait are grouped and scheduled; if a related ER or
 *           failure of one of the PM events during the wait occurs then
 *           the er is scheduled instead of the pm, and downtime begins
 *           at that time.
 * Usage: Pass to function the current 1st server and avl_time (used for
 *        all services if gang_flag disabled or for first service event
 *        grouped)
 * Returns: number of components placed into opm_group[]
 */
int pm_wait(cur_ei, avl_time, fail_flag, fail_indx)
EI_EVENT *cur_ei;              /* should be ei_head */
double avl_time;
int *fail_flag;                /* ptr to flag in caller that id's failure */
int *fail_indx;                /* ptr specifying offset of failing comp */
{
   register COMP_EVENT *comp_ptr;
   int num_grp;                      /* indx of current comp_ptr in omp_event */
                                     /* and later, tot num of comps in group */
   int er_flag = 0;
   double pm_fail;
   double fail_time;
   void pop_comp( );

   *fail_flag = 0;
   *fail_indx = 0;

   for (comp_ptr = cur_ei->comp_head, fail_time = avl_time, num_grp = 0;
        ( (comp_ptr != NULL) && (comp_ptr->time <= avl_time) );
        comp_ptr = comp_ptr->next)
   {
      if (comp_ptr->type == 'P')
      {
         pm_fail = comp_ptr->time + comp_ptr->lifetime -
                   comp_data[comp_ptr->comp_id]->pmintv;
         opm_group[num_grp] = comp_ptr; /* save PM event in grouping array */
         pop_comp(cur_ei, comp_ptr);     /* remove PM event from calendar */
         if (!er_flag)                   /* test all PM events up to 1st */
            if (pm_fail < fail_time)        /* ER failure that occurs */
            {
               fail_time = pm_fail;         /* set new failure comparison time */
               *fail_indx = num_grp;     /* set index of current failing comp */
               *fail_flag = 1;       /* flag that a failure occurs during wait */
            }
         ++num_grp;                     /* increment num events grouped */
      }
      else
      if ( (!er_flag) && (comp_ptr->type == 'E') &&
           (comp_ptr->time < fail_time) )
      {
         opm_group[num_grp] = comp_ptr; /* save ER event in grouping array */
         pop_comp(cur_ei, comp_ptr);    /* remove ER event from calendar */
         fail_time = comp_ptr->time;    /* set fail time of the EI */
         *fail_indx = num_grp;       /* set index of ER comp that fails */
         *fail_flag = 1;             /* flag that a failure occurs during wait */
         er_flag = 1;                /* flag that ER event causes EI failure */
```

```
            ++num_grp;                  /* increment num events grouped */
         }
         else
         if ( (er_flag) && (comp_ptr->type == 'E') &&
              (comp_data[comp_ptr->comp_id]->pm_polcy != '0') )
         {
            if ( (comp_ptr->time + comp_data[comp_ptr->comp_id]->pmintv -
                  comp_ptr->lifetime) <= avl_time )
            {
               opm_group[num_grp] = comp_ptr;
               opm_group[num_grp]->type = 'p';
               opm_group[num_grp]->time += (comp_data[comp_ptr->comp_id]->pmintv -
                                            comp_ptr->lifetime);
               pop_comp(cur_ei, comp_ptr);
               ++num_grp;
            }
         }
      }
      /*
       * reset event type for PM that failed, if necessary
       */
      if ( (*fail_flag == 1) && (er_flag != 1) )
      {
         opm_group[*fail_indx]->type = 'e';
         opm_group[*fail_indx]->time = fail_time;
      }
      return(num_grp);
}


/*
 * lor_updt( lor, num_grp, need_time, cur_ei, fstat )
 *
 * Function: Performs the updating of the next_serv structures
 *           based on the selected LOR at which to perform
 *           the repair. Collects stats and generates new events
 * Returns: None
 */
void lor_updt(lor, num_grp, need_time, cur_ei, fstat)
int lor;
int num_grp;
double need_time;
EI_EVENT *cur_ei;
FILE *fstat;
{
   int serv_id;
   double avl_time;
   double dt_start;
   double dt_end;
   double cmp_time;
   double rep_time;
   register int i;

   /*
    * Get servers and update next_serv times based on gang_flag in lor
    * structure. Determine repair time for each comp event before scheduling
    * service. Determine downtime incurred in the group repair. Begins
    * with 1st failure or avl_time (if no failure during wait) and
    * ends when all comps complete repair. Perform by looping thru opm_group
    * array until "num_grp" events have been serviced.
    */

   dt_start = need_time;
   serv_id = next_free( next_serv[lor], lor, &avl_time);
   need_time += lor_data[lor].t_trans;
```

```
   for (dt_end = 0.0, i=0; i < num_grp; ++i)
   {
      if (opm_group[i]->type == 'P' || opm_group[i]->type == 'p')
         rep_time = comp_data[opm_group[i]->comp_id]->pm_durn;
      else
         rep_time = comp_data[opm_group[i]->comp_id]->er_durn;

      /*
       * if gang flag enabled and not first opm_group, perform gang
       * repair by grabbing the earliest avail server from STD_LOR
       * (first opm_group already has un-updated service appoint time)
       * otherwise, use the "gotten" server for the first event if gang
       * and for all opm_groups if sequential service
       */
      if ( (lor_data[lor].gang_flag == 1) && (i != 0) )
         serv_id = next_free( next_serv[lor], lor, &avl_time);

      if (avl_time < need_time)
         avl_time = need_time;
      /*
       * determine cmp_time from obtained serv_id and rep_time
       * for current event in opm_group, updates the avl_time
       * for that server using a double assignment program statement
       */
      cmp_time = *(next_serv[lor] + serv_id) = avl_time + rep_time;

      if (dt_end < cmp_time)            /* track latest completion time */
         dt_end = cmp_time;

      sp_collect(_sim_time, opm_group[i]->comp_id, opm_group[i]->type,
                 lor, (cmp_time - rep_time), cmp_time, fstat);
      avl_time += rep_time;
   }

   dt_end += lor_data[lor].t_trans;
   dt_collect(_sim_time, lor, dt_start, dt_end, fstat);
   updt_age(cur_ei, (dt_end - dt_start));

   for (i = 0; i < num_grp; ++i)
      gnrt_comp(cur_ei, opm_group[i], dt_end);
   return;
}

/*
 * lor_compare( num_grp, need_time, std_cost, aux_cost )
 *
 * Functions: Performs the comparison of LOR costs for std and aux.
 *            Sets "std_cost" and "aux_cost" in the calling program
 * Returns: 0, std_lor is preferred
 *          1, aux_lor is preferred
 */
int lor_compare(num_grp, need_time, std_cost, aux_cost)
int num_grp;
double need_time;
double *std_cost;
double *aux_cost;
{
   static int called = 0;
   static double *next[2];  /* ptrs to hold copy of extern next_serv[] struc */
   int serv_id;
   int lor;
   double avl_time;
   double cmp_time;
   double dt_start;
   double dt_end;
   double rep_time;
   double serv_time;
```

```
double use_time;
register int i;
char *calloc( );

if (!called)    /* allocate space for temp next_serv structures */
{
   next[0] = (double *) calloc(lor_data[0].capacity, sizeof(double));
   next[1] = (double *) calloc(lor_data[1].capacity, sizeof(double));
   called = 1;
}
/*
 * create working copies of next_serv[ ] structure
 */
for (i = 0; i < lor_data[0].capacity; ++i)
   *(next[0] + i) = *(next_serv[0] +i );
for (i = 0; i < lor_data[1].capacity; ++i)
   *(next[1] + i) = *(next_serv[1] + i);

/*
 * Get servers and update next_serv times based on gang_flag in lor
 * structure. Determine repair time for each comp event before scheduling
 * service. Determine downtime incurred in the group repair. Begins
 * with 1st failure or avl_time (if no failure during wait) and
 * ends when all comps complete repair. Perform by looping thru opm_group
 * array until "num_grp" events have been serviced.
 */

for (lor = 0; lor < 2; ++lor)
{
   if (lor == STD_LOR)
      serv_id = next_free( next[STD_LOR], STD_LOR, &avl_time );
   else
      serv_id = next_free( next[AUX_LOR], AUX_LOR, &avl_time );

   dt_start = need_time;
   use_time = need_time + lor_data[lor].t_trans;

   for (dt_end = 0.0, serv_time = 0.0, i=0; i < num_grp; ++i)
   {
      if (opm_group[i]->type == 'P' || opm_group[i]->type == 'p')
         rep_time = comp_data[opm_group[i]->comp_id]->pm_durn;
      else
         rep_time = comp_data[opm_group[i]->comp_id]->er_durn;
      serv_time += rep_time;

      /*
       * if gang flag enabled and not first opm_group, perform gang
       * repair by grabbing the earliest avail server from STD_LOR
       * (first opm_group already has un-updated service appoint time)
       * otherwise, use the "gotten" server for the first event if gang
       * and for all opm_groups if sequential service
       */
      if ( (lor_data[lor].gang_flag == 1) && (i != 0) )
         serv_id = next_free( next[lor], lor, &avl_time );

      if (avl_time < use_time)
         avl_time = use_time;
      /*
       * determine cmp_time from obtained serv_id and rep_time
       * for current event in opm_group, updates the avl_time
       * for that server using a double assignment program statement
       */
      cmp_time = *(next[lor] + serv_id) = avl_time + rep_time;
      avl_time += rep_time;              /* in case not gang_flag */

      if (dt_end < cmp_time)             /* track latest completion time */
         dt_end = cmp_time;
```

```
      }
      dt_end += lor_data[lor].t_trans;
      if (lor == STD_LOR)
         *std_cost = ((dt_end - dt_start) * popln_data.c_eidt) +
                     (serv_time * lor_data[STD_LOR].c_usage) +
                     (2 * lor_data[STD_LOR].c_trans);
      else
         *aux_cost = ((dt_end - dt_start) * popln_data.c_eidt) +
                     (serv_time * lor_data[AUX_LOR].c_usage) +
                     (2 * lor_data[AUX_LOR].c_trans);
   }
   if (*std_cost <= *aux_cost)
      return(STD_LOR);
   else
      return(AUX_LOR);
}

/*
 * opm_scan( cur_ei, scan_time, offset )
 *
 * Function: Performs the OPM scan ahead, selecting components
 *           for maintenance which are scheduled within the look-ahead
 *           time
 * Returns: Number components added to group
 */
int opm_scan( cur_ei, scan_time, offset )
EI_EVENT *cur_ei;       /* ptr to current end item in event calendar */
double scan_time;       /* absolute time to stop scanning for OPM events */
int offset;             /* event placement begins at opm_group[offset] */
{
   register COMP_EVENT *comp_ptr;
   int num_grp = 0;

   for (comp_ptr = cur_ei->comp_head;
        comp_ptr && (comp_ptr->time <= scan_time);
        comp_ptr = comp_ptr->next)
   {
      if (comp_ptr->type == 'P' || (comp_ptr->type == 'E' && offset == 0))
      {
         opm_group[offset] = comp_ptr;
         pop_comp(cur_ei, comp_ptr);
         ++offset;
         ++num_grp;
      }
      else
      if (comp_ptr->type == 'E' &&
          comp_data[comp_ptr->comp_id]->pm_polcy != '0')
      {
         if ( (comp_ptr->time + comp_data[comp_ptr->comp_id]->pmintv -
               comp_ptr->lifetime) <= scan_time )
         {
            opm_group[offset] = comp_ptr;
            opm_group[offset]->type = 'p';
            opm_group[offset]->time += (comp_data[comp_ptr->comp_id]->pmintv -
                                        comp_ptr->lifetime);
            pop_comp(cur_ei, comp_ptr);
            ++offset;
            ++num_grp;
         }
      }
   }
   return(num_grp);
}
```

```
/*
 * File Name:  UPDATE.C
 */
#include <stdio.h>
#include "sm_struc.h"

/*
 * next_free( int lor_index, double *avl_time)
 *
 * Function:  for the passed LOR, sets the earliest time that
 *            a server is avail (in "avl_time"), and returns
 *            the index of that server.
 * Usage:  user must perform the update of that servers avail time
 *         once the resource usage time is determined, use the serv_id
 *         and the LOR specified to directly perform the update
 *         (i.e. *(next_serv[LOR] + serv_id) += usage_time )
 */
next_free( next, lor_index, avl_time)
double *next;                   /* ptr to allocated mem holding server data */
int lor_index;                  /* index of LOR (0 or 1, i.e. std or aux) */
double *avl_time;               /* ptr to allow return of next avl time */
{
   register int i;
   int serv_id;

   *avl_time = *next;           /* init min avail time to 1st server */

   /*
    * loop and find the minimum avail time from the set of
    * servers at the specified LOR
    */
   for ( i=0, serv_id=0; i < lor_data[lor_index].capacity; ++i )
   {
      if ( *(next + i) < *avl_time )
      {
         *avl_time = *(next + i);   /* set new minimum and */
         serv_id = i;               /* id of server */
      }
   }
   return(serv_id);
}

/*
 * void updt_age(cur_ei, downtime)
 *
 * Function: Performs the "non-aging" update of comp events remaining
 *           in an EI's component list (after current comp event or
 *           group of comps is popped from the list)
 */
void updt_age(cur_ei, downtime)
EI_EVENT *cur_ei;
double downtime;
{
   register COMP_EVENT *comp_ptr;

   for (comp_ptr=cur_ei->comp_head; comp_ptr; comp_ptr = comp_ptr->next)
         comp_ptr->time += downtime;
}

/*
 * void gnrt_comp(cur_ei, cur_comp, cmp_time)
 *
 * Function: Generates new event for a component just serviced.
 *           Reuses memory allocated to the old event, samples
 *           a lifetime for that comp, determines whether ER or PM
 *           is the outcome, changes the comp event structure
 *           accordingly, and pushes the new event into the comp
```

```
 *           list for the EI identified in the call (cmp_time is
 *           service completion time for the EI in which the cur_comp
 *           belongs).
 */
void gnrt_comp(cur_ei, cur_comp, cmp_time)
EI_EVENT *cur_ei;
COMP_EVENT *cur_comp;
double cmp_time;
{
   double lifetime, rand_samp();

   /*
    * generate new event data, reusing cur_comp and data from event
    * just performed (for the same ei and comp), only need a
    * new sample lifetime for component, and set cur_comp to
    * next event in calendar
    */

   lifetime = rand_samp(cur_comp->comp_id);      /* generate new lifetime */
                                                 /* for comp just serviced */
   cur_comp->lifetime = lifetime;     /* get new lifetime attrib for comp */

   /*
    * check and see if PM or ER occurs for the just serviced
    * component and compute event times accordingly
    */
   if ( comp_data[cur_comp->comp_id]->pm_policy != '0' &&
        lifetime >= comp_data[cur_comp->comp_id]->pmintv )
   {
      cur_comp->time = cmp_time +
                       comp_data[cur_comp->comp_id]->pmintv;
      cur_comp->type = 'P';
   }
   else                                          /* ER will take place */
   {
      cur_comp->time = cmp_time + lifetime;
      cur_comp->type = 'E';
   }
   push_comp( cur_ei, cur_comp );     /* reinsert comp evnt with new data */

   if ( cur_ei->comp_head->time  >  (cur_ei->next)->comp_head->time )
   {
      pop_ei( cur_ei );           /* remove current ei event from ei list */
      push_ei( cur_ei );             /* reinsert current ei with new data */
   }
   return;
}
```

```
/*
 * File Name:  STATS.C
 */
#include <stdio.h>
#include "sm_struc.h"

/*
 * dt_collect(tar_time, dt_start, dt_end, fstat)
 *
 * Function: Collects population downtime according to intervals
 *           specified in "popln_data" structure.  Active collection
 *           nodes are kept in a linked list whose head is the earliest
 *           interval.
 *           Writes history record of downtime for a specified interval
 *           and deletes interval collection nodes that are no longer needed.
 *           User will want to flush and output any nodes remaining in the
 *           list at the end of a simulation execution.
 *           Extern vars for use in collection are initialized in load_cal()
 *           at begin of each sim_num iteration (list heads, etc)
 *
 * Usage: List is located by extern ptr to current head of list, "dt_head".
 *        Each node contains the following interval information:
 *        - cumulative downtime for the interval
 *        - interval number
 *        - cutoff time for data collection (end of interval)
 *        - ptr to next interval node
 *
 * Returns: none
 */
dt_collect( tar_time, lor, dt_start, dt_end, fstat )
double tar_time;        /* time that sets status of collection node list */
int lor;                /* LOR at which downtime occurs */
double dt_start;        /* absolute beginning time of EI downtime */
double dt_end;          /* absolute ending time of EI downtime event */
FILE *fstat;            /* ptr to statistics output file */
{
   DT_INTV *intv_ptr, *ptr;
   int span;            /* number of full intervals downtime spans, less */
                        /* start and last partial intervals */
   double last;         /* amount of downtime for last interval affected */
   int i;               /* counter */
   char *malloc();

   /* check for and remove collection nodes at beginning of list that */
   /* are no longer needed */

   while (dt_head->brkpt < tar_time)
   {
      /*
       * write to file the interval collection node being removed,
       * at this point also need to write all other stats being
       * collected
       */
      fprintf(fstat,"D %2d %7.2f %7.2f %2d %2d\n",
              dt_head->id, dt_head->brkpt, dt_head->downtime,
              dt_head->dt_count[0], dt_head->dt_count[1]);

     /*
      * removed all nodes from list, need new
      * starting node containing tar_time
      * reusing previous allocation for dt_head
      */
      if (!dt_head->next)
      {
         dt_head->downtime = 0.0;
         dt_head->id = (int) (tar_time / popln_data.intv_size) + 1;
         dt_head->brkpt = dt_head->id * popln_data.intv_size;
```

```
      dt_head->dt_count[0] = dt_head->dt_count[1] = 0;
      goto endremove;
   }
   intv_ptr = dt_head;          /* save head to free later */
   dt_head = dt_head->next;     /* set head to next allocated interval */
   free( intv_ptr );            /* free memory used by old head */
}
endremove:


/* get to interval in list which contains "dt_start" time */

intv_ptr = dt_head;
while (dt_start > intv_ptr->brkpt)
{
   if (!intv_ptr->next)          /* next node is NULL, add new interval */
   {
      intv_ptr->next = (DT_INTV *) malloc( sizeof(DT_INTV) );
      (intv_ptr->next)->downtime = 0.0;
      (intv_ptr->next)->id = intv_ptr->id + 1;
      (intv_ptr->next)->brkpt = intv_ptr->brkpt + popln_data.intv_size;
      (intv_ptr->next)->dt_count[0] = (intv_ptr->next)->dt_count[1] = 0;
      (intv_ptr->next)->next = NULL;
   }
   intv_ptr = intv_ptr->next;
}

/*
 * intv_ptr now pts to interval which contains dt_start
 */

intv_ptr->dt_count[lor] += 1;   /* increment dt instances at lor */

if (dt_end < intv_ptr->brkpt)   /* downtime occurs within single interval */
   intv_ptr->downtime += (dt_end - dt_start);
else
{
   intv_ptr->downtime += (intv_ptr->brkpt - dt_start);
   span = (int) ((dt_end - intv_ptr->brkpt) / popln_data.intv_size);
   last = dt_end - (span * popln_data.intv_size) - intv_ptr->brkpt;

   /*
    * this little redundant part of code makes sure the initializing
    * ptr (intv_ptr->next) for the "for" loop to come, is not NULL
    */
   if (!intv_ptr->next)
   {
      intv_ptr->next = (DT_INTV *) malloc( sizeof(DT_INTV) );
      (intv_ptr->next)->downtime = 0.0;
      (intv_ptr->next)->id = intv_ptr->id + 1;
      (intv_ptr->next)->brkpt = intv_ptr->brkpt + popln_data.intv_size;
      (intv_ptr->next)->dt_count[0] = (intv_ptr->next)->dt_count[1] = 0;
      (intv_ptr->next)->next = NULL;
   }

   for (i=0, ptr=intv_ptr->next; i < span; ptr=ptr->next, ++i)
   {
      ptr->downtime += popln_data.intv_size;
      if (!ptr->next)           /* make sure next interval is allocated */
      {
         ptr->next = (DT_INTV *) malloc( sizeof(DT_INTV) );
         (ptr->next)->downtime = 0.0;
         (ptr->next)->id = ptr->id + 1;
         (ptr->next)->brkpt = ptr->brkpt + popln_data.intv_size;
         (ptr->next)->dt_count[0] = (ptr->next)->dt_count[1] = 0;
         (ptr->next)->next = NULL;
      }
```

```
      }

      /*
       * "ptr" now pts to interval node in which dt_end occurs
       * will add "last" to its accumulator
       */
      ptr->downtime += last;
   }
   return;
}

/*
 * avg_ei( fstat )
 *
 * Function: Computes the average number of EIs in popln during each
 *           interval specified. If there's no MOF schedule then avg
 *           is the initial block for all intervals; otherwise the
 *           schedule is stepped through, determining a weighted avg
 *
 * Usage: Only needs to be called once, since this average does not
 *        change with each realization generated.  Output is written
 *        to the stats output file specified by the passed ptr "fstat".
 *        Best location for the call is just before closing "fstat".
 */
avg_ei( fstat )
FILE *fstat;
{
   double last_time;
   double brkpt;
   double wtd_num;
   int num_ei;
   int intv;
   MOF_SCHED *ptr;

   if (popln_data.mof != 's' && popln_data.mof != 'S')  /* 1 point for avg_ei*/
      fprintf(fstat,"N %3d\n", popln_data.mof_init);
   else                                                 /* point for each intv*/
   {
      num_ei = popln_data.mof_init;
      brkpt = popln_data.intv_size;
      last_time = 0.0;
      wtd_num = 0.0;
      intv = 1;
      for (ptr = popln_data.sched;
          (ptr) && ( (double) ptr->s_time < popln_data.sim_lgth);
           ptr = ptr->next)
      {
         while (brkpt <= (double) ptr->s_time)
         {
            wtd_num += (double) num_ei * (brkpt - last_time);
            fprintf(fstat,"M %3d %7.2f\n",
                     intv, (wtd_num / popln_data.intv_size) );
            last_time = brkpt;
            wtd_num = 0.0;
            brkpt += popln_data.intv_size;
            ++intv;
         }
         wtd_num += (double) num_ei * ( (double) ptr->s_time - last_time);
         last_time = (double) ptr->s_time;
         num_ei += ptr->s_block;
      }
      while (brkpt <= popln_data.sim_lgth)   /* for any intervals left up to */
      {                                      /* end of simulation run */
         wtd_num += (double) num_ei * (brkpt - last_time);
         fprintf(fstat,"M %3d %7.2f\n",
                  intv, (wtd_num / popln_data.intv_size) );
         last_time = brkpt;
```

```
            wtd_num = 0.0;
            brkpt += popln_data.intv_size;
            ++intv;
         }
      }
      return;
}

sp_collect( updt_time, comp_id, type, lor, rep_start, rep_end, fstat )
double updt_time;       /* time update status of list */
int comp_id;            /* id num of component list to use */
char type;              /* type of event, P or E */
int lor;                /* LOR where event occurs, STD or AUX */
double rep_start;       /* time that spares demand occurs (repair begins) */
double rep_end;         /* time that repair ends */
FILE *fstat;            /* ptr to statistics output file */
{
   SP_INTV *intv_ptr, *ptr;
   int span;
   double last;
   int i;
   char *malloc();

   /* check for and remove collection nodes at beginning of list that */
   /* are no longer needed */

   while (sp_head[comp_id]->brkpt < updt_time)
   {
      /*
       * write to file the interval collection node being removed,
       * at this point also need to write all other stats being
       * collected
       */
      fprintf(fstat,"L %2d %2d %7.2f %2d %2d %2d %2d %7.2f %7.2f\n",
            sp_head[comp_id]->id, comp_id, sp_head[comp_id]->brkpt,
            sp_head[comp_id]->er_count[0], sp_head[comp_id]->er_count[1],
            sp_head[comp_id]->pm_count[0], sp_head[comp_id]->pm_count[1],
            sp_head[comp_id]->rep_time[0], sp_head[comp_id]->rep_time[1]);

     /*
      * removed all nodes from list, need new
      * starting node containing updt_time
      * reusing previous allocation for sp_head[comp_id]
      */
      if (!sp_head[comp_id]->next)
      {
         sp_head[comp_id]->id = (int) (updt_time / popln_data.intv_size) + 1;
         sp_head[comp_id]->brkpt = sp_head[comp_id]->id * popln_data.intv_size;
         sp_head[comp_id]->er_count[0] = sp_head[comp_id]->er_count[1] = 0;
         sp_head[comp_id]->pm_count[0] = sp_head[comp_id]->pm_count[1] = 0;
         sp_head[comp_id]->rep_time[0] =
            sp_head[comp_id]->rep_time[1] = 0.0;
         goto endremove;
      }
      intv_ptr = sp_head[comp_id];        /* save head to free later */
      sp_head[comp_id] = sp_head[comp_id]->next;  /* set head to next */
                                                  /* allocated interval */

      free( intv_ptr ); /* free memory used by old head */
   }
   endremove:


   /* get to interval in list which contains "rep_start" (pnt of usage) */

   intv_ptr = sp_head[comp_id];
   while (rep_start > intv_ptr->brkpt)
```

```
{
   if (!intv_ptr->next)          /* next node is NULL, add new interval */
   {
      intv_ptr->next = (SP_INTV *) malloc( sizeof(SP_INTV) );
      (intv_ptr->next)->id = intv_ptr->id + 1;
      (intv_ptr->next)->brkpt = intv_ptr->brkpt + popln_data.intv_size;
      (intv_ptr->next)->er_count[0] = (intv_ptr->next)->er_count[1] = 0;
      (intv_ptr->next)->pm_count[0] = (intv_ptr->next)->pm_count[1] = 0;
      (intv_ptr->next)->rep_time[0] = (intv_ptr->next)->rep_time[1] = 0.0;
      (intv_ptr->next)->next = NULL;
   }
   intv_ptr = intv_ptr->next;
}

/*
 * intv_ptr now pts to interval which contains rep_start
 */
if (type == 'E' || type == 'X')     /* increment ER demand at lor */
   intv_ptr->er_count[lor] += 1;
else
if (type == 'P')                    /* increment PM demand at lor */
   intv_ptr->pm_count[lor] += 1;

/*
 * now collect the repair time expended by the server during the event
 * (data will be by component, but just sum across for totals)
 */

if (rep_end < intv_ptr->brkpt)   /* downtime occurs within single interval */
   intv_ptr->rep_time[lor] += (rep_end - rep_start);
else
{
   intv_ptr->rep_time[lor] += (intv_ptr->brkpt - rep_start);
   span = (int) ((rep_end - intv_ptr->brkpt) / popln_data.intv_size);
   last = rep_end - (span * popln_data.intv_size) - intv_ptr->brkpt;

   /*
    * this little redundant part of code makes sure the initializing
    * ptr (intv_ptr->next) for the "for" loop to come, is not NULL
    */
   if (!intv_ptr->next)
   {
      intv_ptr->next = (SP_INTV *) malloc( sizeof(SP_INTV) );
      (intv_ptr->next)->id = intv_ptr->id + 1;
      (intv_ptr->next)->brkpt = intv_ptr->brkpt + popln_data.intv_size;
      (intv_ptr->next)->er_count[0] = (intv_ptr->next)->er_count[1] = 0;
      (intv_ptr->next)->pm_count[0] = (intv_ptr->next)->pm_count[1] = 0;
      (intv_ptr->next)->rep_time[0] = (intv_ptr->next)->rep_time[1] = 0.0;
      (intv_ptr->next)->next = NULL;
   }

   for (i=0, ptr=intv_ptr->next; i < span; ptr=ptr->next, ++i)
   {
      ptr->rep_time[lor] += popln_data.intv_size;
      if (!ptr->next)          /* make sure next interval is allocated */
      {
         ptr->next = (SP_INTV *) malloc( sizeof(SP_INTV) );
         (ptr->next)->id = ptr->id + 1;
         (ptr->next)->brkpt = ptr->brkpt + popln_data.intv_size;
         (ptr->next)->er_count[0] = (ptr->next)->er_count[1] = 0;
         (ptr->next)->pm_count[0] = (ptr->next)->pm_count[1] = 0;
         (ptr->next)->rep_time[0] = (ptr->next)->rep_time[1] = 0.0;
         (ptr->next)->next = NULL;
      }
   }

   /*
```

```
        * "ptr" now pts to interval node in which rep_end occurs
        * will add "last" to its accumulator
        */
       ptr->rep_time[lor] += last;
    }

    return;
}
```

```
/*
 * File Name:  RANDSAMP.C
 */
#include <stdio.h>
#include <math.h>
#include "sm_struc.h"

/*
 * rand_samp(i)
 *
 * Usage: given that the ith component needs a sample, this
 *        routine determines the failure distribution used
 *        and calls the appropriate sampling routine with
 *        the correct parameters.
 * Returns: a double value which represents the random sample
 *          a value of -999.0 indicates an error occured.
 *          (check for negative return to signify error)
 */
double rand_samp(i)
int i;
{
   extern COMPONENT *comp_data[MAX_COMP];
   double parm1,parm2;
   double expon( ),normal( ),empir( ),uniform( ),weibull( );
   double lognormal( ),erlang( ),gamma( );

   parm1 = comp_data[i]->parm1;
   parm2 = comp_data[i]->parm2;

   switch( comp_data[i]->fail_dist )
   {
      case 'e':
      case 'E':
                  return( expon(parm1) );
      case 'n':
      case 'N':
                  return( normal(parm1,parm2) );
      case 'm':
      case 'M':
                  return( empir(comp_data[i]->emp_head) );
      case 'u':
      case 'U':
                  return( uniform(parm1,parm2) );
      case 'l':
      case 'L':
                  return( lognormal(parm1,parm2) );
      case 'w':
      case 'W':
                  return( weibull(parm1,parm2) );
      case 'r':
      case 'R':
                  return( erlang((int)parm1,parm2) );
      case 'g':
      case 'G':
                  return( gamma(parm1,parm2) );
         default:
                  return(-999.0);
   }
}

/*-----------------------------------------------------------------
   The following code contains random sampling routines for these
   probability distributions:

           Normal
           Exponential
           Uniform
```

```
            (0,1)
            (-1,1)
            General
         Weibull
         LogNormal
         Gamma
         Erlang

   References:

   [1] Banks, Jerry and J.S. Carson, Discrete-Event System
       Simulation, Prentice-Hall, Englewood Cliffs, NJ (1984).
   [2] Press, W.H., B.P. Flannery, S.A. Teukolsky and W.T. Vetterling.
       Numerical Recipes: Art of Scientific Computing, Cambridge
       University Press, Cambridge (1986).

-------------------------------------------------------------------*/

/*-------------------------------------------------------------------
   normal distribution sampling
      E[X] = mu, Var[X] = sigma¢2
      return is transform, X, of std normal N(0,1), Z;
      where Z = (X-mu)/sigma
      random number generator must be seeded prior to call
-------------------------------------------------------------------*/
double normal(mu,sigma)
double mu,sigma;
{
   double stdnorml();

   return( stdnorml() * sigma + mu );
}

/*-------------------------------------------------------------------
   normal distribution sampling [2]
      with E[X] = 0, Var[X] = 1
      random number generator must be seeded prior to 1st call
-------------------------------------------------------------------*/
double stdnorml()
{
   static double extra;
   static int f_extra = 0;
   double fac,r,v1,v2;
   double random(),pow(),sqrt(),log();

   if (f_extra == 0)
   {
      do
      {
         v1 = random(2);  /* random number in (-1,1) */
         v2 = random(2);
         r = pow(v1,2.0) + pow(v2,2.0);
      }
      while (r >= 1.0);

      fac = sqrt(-2.0*log(r)/r);
      extra = v1*fac;
      f_extra = 1;
      return(v2*fac);
   }
   else
   {
      f_extra = 0;
      return(extra);
   }
}
```

```
double lognormal(mu,sigma)
double mu;
double sigma;
{
   double x;
   double x_mu;
   double x_sigma;
   double pow( ),sqrt( ),log( ),normal( );

   x = log( 1.0 + sigma / mu );
   x_sigma = sqrt(x);
   x_mu = log(mu) - 0.5 * x;

   return( pow(2.718281828, normal(x_mu,x_sigma)) );
}

/*-----------------------------------------------------------------
    exponential distribution sampling [2]
      the normal (1-random) for the log calculation
      is replaced by (random), since rand <==> (0,1)
      random number generator must be seeded prior to 1st call
-----------------------------------------------------------------*/
double expon(mean)
double mean;
{
   double random( ),log( );
   double lam;

   lam = 1.0 / mean;

   return(-log(random(1))/lam);
}

/*-----------------------------------------------------------------
    uniform distribution sampling [1]
      random number generator must be seeded prior to 1st call
-----------------------------------------------------------------*/
double uniform(a,b)
double a,b,random( );
{
   double random( );

   return(a+(b-a)*random(1));
}

/*-----------------------------------------------------------------
   weibull distribution sampling [1]
      with scale parameter alpha and shape parameter beta
      (user supplied), and location parameter nu = 0
      the usual (1-random) for the log calculation
      is replaced by (random), since rand <==> (0,1)
      random number generator must be seeded prior to 1st call
-----------------------------------------------------------------*/
double weibull(alpha,beta)
double alpha, beta;
{
   double x,log( ),pow( ),random( );

   x = alpha * pow(-log(random(1)),(1/beta));
   return(x);
}

/*-----------------------------------------------------------------
   erlang-k distribution sampling [1]
      with shape parameters k and lambda
      mean = 1/lambda, variance = 1/(k*lambda¢2)
```

```
      random number generator must be seeded prior to 1st call
-------------------------------------------------------------------*/
double erlang(k,mean)
int k;
double mean;
{
   int i;
   double lambda;
   double r,log( ),random( );

   lambda = 1.0 / mean;

   for (i=1, r=1.0; i <= k; i++)
      r *= random(1);

   return( -1.0/((double) k * lambda) * log(r) );
}

/*-------------------------------------------------------------------
   gamma distribution sampling [1]
      with scale parameter theta and shape parameter beta
      E[X] = 1/theta, Var[X] = 1/(beta*theta¢2)
      random number generator must be seeded prior to 1st call
-------------------------------------------------------------------*/
double gamma(theta,beta)
double theta,beta;
{
   double a,b,x,r1,r2;
   double log( ),pow( ),random( ),sqrt( );

   a = sqrt(2.0*beta-1.0);
   b = 2.0*beta - log(4.0) + 1/a;

   do
   {
      r1 = random(1);
      r2 = random(1);
      x = beta * pow(r1/(1.0-r1),a);
   }
   while ( x > b-log(pow(r1,2.0)*r2) );

   return( x/(beta*theta) );
}

/*-------------------------------------------------------------------
   random(id) generates uniform random samples in (0,1) and (-1,1)
      id = 1 ==> uniform (0,1) r.v.
      id = 2 ==> uniform (-1,1) r.v.
   the call srand(seed) must have been performed prior
      to calling any routine that uses the system random
      generator, rand( )
   a return value of -999.0 indicates an invalid id #
-------------------------------------------------------------------*/
double random(id)
int id;
{
   double x,y;

   x = (double) rand( ) / 32767.0;      /* convert to 0,1 random number */
   y = 2.0 * x - 1.0;                   /* convert to -1,1 random number */
   if (id == 1)
      return(x);
   else
   if (id == 2)
      return(y);
   else
```

```
    return(-999.0);                /* error flag, invalid id */
}

/*-------------------------------------------------------------------------*
 *  empir(head)
 *
 *  Function: For generating sample random deviates from a
 *            user-specified empirical probability distribution.
 *
 *  Usage: Must pass a pointer to a linked list structure that
 *         contains cumulative probabilities and corresponding
 *         response values (along with a pointer to the next
 *         interval).
 *                              struct emp_data
 *                         ------------------------------------
 *  Example:  Interval     Probability  Response     *Next_Intv
 *            --------     -----------  --------    ------------
 *               1             0.0         2.4      Address of 2
 *               2             0.28        5.9      Address of 3
 *               3             0.67       10.8      Address of 4
 *               4             1.0        15.0          NULL
 *
 *-------------------------------------------------------------------------*/

double empir(head)
EMP_DATA *head;
{
   double random();
   double r,a;                     /* r=(0,1) r.v. a=slope of r's interval */
   register EMP_DATA *low;         /* temp ptr to low interval */
   register EMP_DATA *high;        /* temp ptr to high intervals */

   r = random(1);                  /* generate (0,1) random number */
   if (r == 0.0)                   /* check endpoints (0,1) r.v. */
      return( head->response );
/*
 * Determine interval in which (0,1) random variable lies.
 * Routine steps through the linked list defining
 * empirical data intervals, and stops when the matching
 * interval is found or a NULL ptr to the next structure
 * is found (a NULL value signifies a FALSE condition
 * the loop), which implies no matching interval exists.
 */
   for (low = head; low->next_intv; low = low->next_intv)
   {
      high = low->next_intv;
      if (r > low->probab && r <= high->probab)
      {
         a = (high->response - low->response)/(high->probab - low->probab);
         return( low->response + a * (r - low->probab) );
      }
   }
   return(-999.0);              /* error condition, interval not found */
}
```

```
/*
 * File Name:  SHOWDATA.C
 */
#include <stdio.h>
#include <wfd.h>
#include "sm_struc.h"

/*
 * edit_menu( )
 *
 * Usage: Display and process the Retrieve/Edit/Create Menu.
 *        Use to manipulate system, end item and component
 *        data. Menu is created in WFD window "wn".
 * Returns: None
 */
edit_menu( )
{
   char choice;
   extern POPLN popln_data;
   char *dat_file;
   FILE *fptr;
   void show_data( );
   void def_menu( );
   char *strcpy( );
   WINDOW wn;

   dat_file = stblank(15);              /* initialize input file name */

   defs_wn(&wn,4,16,14,47,BDR_DLNP);     /* define menu window */
   sw_name("Model Edit Menu", &wn);
   sw_plcsr(ON, &wn);
   def_menu(&wn);         /* define and set menu on screen for first time */

   for (;;)
   {
      choice = ki( );      /* get keystroke from user */
      switch ( choice )    /* branch on keystroke */
      {
         case 82 :                        /* "retrieve" */
         case 114:                        /* "R" */
                  get_data(16,19,"File name to retrieve:", def_pic('X',15),
                           F_STRING, dat_file);
                  if ( (fptr = fopen(dat_file,"r")) != NULL)
                  {
                     unset_wn(&wn);
                     strcpy( popln_data.data_file, dat_file );
                     retrv_file(fptr);
                     fclose(fptr);
                     def_menu(&wn);
                  }
                  else
                  {
                     csr_hide( );
                     bell( );
                  }
                  break;
         case 69 :                        /* "edit" */
         case 101:                        /* "E" */
                  unset_wn(&wn);
                  show_data( );
                  def_menu(&wn);
                  break;
         case 67 :                        /* "create" */
         case 99 :                        /* "C" */
                  get_data(16,19,"File name to create:", def_pic('X',15),
                           F_STRING, dat_file);
                  if ( (fptr = fopen(dat_file, "w")) != NULL)
```

```
                {
                   unset_wn(&wn);
                   strcpy( popln_data.data_file, dat_file );
                   create_data(fptr);       /* call create function */
                   fclose(fptr);
                   def_menu(&wn);
                }
                else
                {
                   csr_hide();
                   bell();
                }
                break;
        case 83 :                       /* "save" */
        case 115:                       /* "S" */
                get_data(16,19,"Save file name for data:", def_pic('X',15),
                        F_STRING, dat_file);
                if ( (fptr = fopen(dat_file, "w")) != NULL)
                {
                   unset_wn(&wn);
                   strcpy( popln_data.data_file, dat_file );
                   save_data(fptr);         /* call save function */
                   fclose(fptr);
                   def_menu(&wn);
                }
                else
                {
                   csr_hide();
                   bell();
                }
                break;
        case 81 :                       /* "quit" */
        case 113:                       /* "Q" */
        case 27 :                       /* <ESC> */
                unset_wn(&wn);
                return;                 /* exit menu sub-system */
        default :
                bell();
      }
   }
}


void def_menu(wn)
WINDOW *wn;
{
   cls();
   set_wn(wn);     /* place window on screen */

   v_plst(1,1,"Current Active Data File: ",wn);

   wn->att = LHIGHLITE;
   v_st(popln_data.data_file, wn);
   v_plst(5,15,"r",wn);
   v_plst(6,15,"e",wn);
   v_plst(7,15,"c",wn);
   v_plst(8,15,"s",wn);
   v_plst(9,15,"q",wn);
   wn->att = LNORMAL;

   v_plst(3,CENTER_TXT,"Data Action Choices",wn);
   v_plst(5,17,"- Retrieve",wn);
   v_plst(6,17,"- Edit",wn);
   v_plst(7,17,"- Create",wn);
   v_plst(8,17,"- Save",wn);
   v_plst(9,17,"- Quit",wn);
   csr_hide();
```

```
   return;
}


/*
 * void show_data( )
 *
 * Function: displays secondary menu to select which part
 *           of the data the user wishes to view and edit
 *           Process the request and returns to caller upon
 *           exit sequence from menu
 */
void show_data( )
{
   void show_popln( ), def_data( );
   char choice;
   WINDOW wn;
   int i;

   defs_wn(&wn,4,16,13,47,BDR_DLNP);      /* define menu window */
   sw_name("Data Type Menu", &wn);
   sw_plcsr(ON, &wn);
   def_data(&wn);         /* define and set menu on screen for first time */

   for (;;)
   {
      choice = ki( );      /* get keystroke from user */
      switch ( choice )   /* branch on keystroke */
      {
         case 99 :                        /* "component" */
         case 67 :                        /* "C" */
                  unset_wn(&wn);
                  for (i=0; i < _num_comp; ++i)
                     show_comp(comp_data[i],1); /* show comp i data for edit */
                  def_data(&wn);
                  break;
         case 69 :                        /* "end item" */
         case 101:                        /* "E" */
                  unset_wn(&wn);
                  show_popln(1,1);       /* show ei data for edit */
                  def_data(&wn);
                  break;
         case 108:                        /* "lor" */
         case 76 :                        /* "L" */
                  unset_wn(&wn);
                  show_popln(3,1);       /* show lor data for edit */
                  def_data(&wn);
                  break;
         case 115:                        /* "simulation" */
         case 83 :                        /* "S" */
                  unset_wn(&wn);
                  show_popln(4,1);       /* show simul data for edit */
                  def_data(&wn);
                  break;
         case 113:                        /* "quit" */
         case 81 :                        /* "Q" */
         case 27 :                        /* <ESC> */
                  unset_wn(&wn);
                  return;                /* exit menu sub-system */
                  break;
         default :
                  bell( );
      }
   }
}
```

```
void def_data(wn)
WINDOW *wn;
{
   cls( );
   set_wn(wn);     /* place window on screen */

   v_plst(1,1,"Current Active Data File: ",wn);

   wn->att = LHIGHLITE;
   v_st(popln_data.data_file, wn);
   v_plst(5,14,"c",wn);
   v_plst(6,14,"e",wn);
   v_plst(7,14,"l",wn);
   v_plst(8,14,"s",wn);
   v_plst(9,14,"q",wn);
   wn->att = LNORMAL;

   v_plst(3,CENTER_TXT,"Data Type Choices",wn);
   v_plst(5,16,"- Component",wn);
   v_plst(6,16,"- End Item",wn);
   v_plst(7,16,"- LOR",wn);
   v_plst(8,16,"- Simulation",wn);
   v_plst(9,16,"- Quit",wn);
   csr_hide( );

   return;
}

/* void show_popln(flag)
 *
 * Usage: to display/edit data specific to the external
 *        POPLN structure "popln_data" and the LOR structure.
 *        The value of "disp_flag" determines which portion of
 *        data is displayed
 *               flag            display
 *               ----        ----------------------
 *                 1         name/fail/retire/costs
 *                 3         LOR data
 *                 4         simulation data
 *               other       error
 *
 *        If "edit_flag" is passed value 0, then windows are set
 *        for "file create" operation.
 */
void show_popln(disp_flag, edit_flag)
int disp_flag;                /* IDs information to display*/
int edit_flag;                /* flags if call is for edit or create */
{
   DFORMPTR popln_form;
   DFIELDPTR def_fldw( );
   void show_sched( );
   int sched_flag = FALSE;

   cls( );

      /*
       * Layout the fields for the popln data window, as specified
       * by the display flag code. Build the display form.
       * structure defined within a component. Uses function def_fldw( )
       * which is defined by def_fldx( ) during the call.
       */

   switch ( disp_flag )
   {
      case 1:
               popln_form = def_frm(4,17,13,45,LNORMAL,BDR_DLNP);
               sw_name( "End Item Data", popln_form->wnp );
```

```
        def_fldw(1,1,"End Item Name:",def_pic('X',20), F_STRING,
                popln_data.name, popln_form);
        def_fldw(3,1,"Mode of Fielding (MOF):","X", F_CHAR,
                &popln_data.mof, popln_form);
        def_fldw(4,1,"  Initial Block of EIs:",def_pic('9',5),F_INT,
                &popln_data.mof_init, popln_form);

        if ( (popln_data.mof == 's' || popln_data.mof == 'S')
            && edit_flag == TRUE )
        {
             sched_flag = TRUE;
             def_txtx(5,1,"Note: MOF Schedule Data will follow",
                     LHIGHLITE, popln_form);
             def_txtx(6,7,"the current screen.",
                     LHIGHLITE, popln_form);
        }
        def_fldw(8,1,"EI Downtime Cost:","######@##",F_FLOAT,
                &popln_data.c_eidt, popln_form);
        def_fldw(9,1,"Time Unit for Downtime Cost:","X",F_CHAR,
                &popln_data.u_eidt, popln_form);

        rd_fldsx(popln_form);
        free_frm(popln_form);
        if ( edit_flag == FALSE && (popln_data.mof == 's'
             || popln_data.mof == 'S') )
           cre_sched();                  /* get a new sched from user */
        else
        if (sched_flag)
           show_sched(popln_data.sched, popln_data.mof_nodes);
        break;

case 3:
        popln_form = def_frm(1,17,21,45,LNORMAL,BDR_DLNP);
        sw_name( "LOR Data", popln_form->wnp );

        def_txtx(1,1,"Standard Repair Facility",LHIGHLITE,
                popln_form);
        def_fldw(2,3,"Name:",def_pic('X',20),
                F_STRING, lor_data[0].name, popln_form);
        def_fldw(3,3,"Capacity:",def_pic('9',5), F_INT,
                &lor_data[0].capacity, popln_form);
        def_fldw(4,3,"Usage Cost:","######@##", F_FLOAT,
                &lor_data[0].c_usage, popln_form);
        def_fldw(5,3,"Unit for Cost:","X", F_CHAR,
                &lor_data[0].u_usage, popln_form);
        def_fldw(6,3,"One-Way Transport Time:","######@##",
                F_FLOAT, &lor_data[0].t_trans, popln_form);
        def_fldw(7,3,"One-Way Transport Cost:","######@##",
                F_FLOAT, &lor_data[0].c_trans, popln_form);
        def_fldw(8,3,"Gang Repair Flag:","9", F_INT,
                &lor_data[0].gang_flag, popln_form);

        def_txtx(10,1,"Auxiliary Repair Facility",LHIGHLITE,
                popln_form);
        def_fldw(11,3,"Name:",def_pic('X',20),
                F_STRING, lor_data[1].name, popln_form);
        def_fldw(12,3,"Capacity:",def_pic('9',5), F_INT,
                &lor_data[1].capacity, popln_form);
        def_fldw(13,3,"Usage Cost:","######@##", F_FLOAT,
                &lor_data[1].c_usage, popln_form);
        def_fldw(14,3,"Unit for Cost:","X", F_CHAR,
                &lor_data[1].u_usage, popln_form);
        def_fldw(15,3,"One-Way Transport Time:","######@##",
                F_FLOAT, &lor_data[1].t_trans, popln_form);
        def_fldw(16,3,"One-Way Transport Cost:","######@##",
                F_FLOAT, &lor_data[1].c_trans, popln_form);
        def_fldw(17,3,"Gang Repair Flag:","9", F_INT,
```

```
                &lor_data[1].gang_flag, popln_form);
            rd_fldsx(popln_form);
            free_frm(popln_form);
            break;
      case 4:
            popln_form = def_frm(5,17,11,45,LNORMAL,BDR_DLNP);
            sw_name( "Simulation Parameters", popln_form->wnp );

            def_fldw(1,1,"Time Unit for Simulation:","X", F_CHAR,
                &popln_data.u_sim, popln_form);
            def_fldw(2,1,"Length of Simulation:","#######0##", F_FLOAT,
                &popln_data.sim_lgth, popln_form);
            def_fldw(3,1,"Number of Simulations:",def_pic('9',5), F_INT,
                &popln_data.sim_num, popln_form);
            def_fldw(4,1,"Interval Size for Stats:","#######0##", F_FLOAT,
                &popln_data.intv_size, popln_form);
            def_fldw(5,1,"OPM Flag:","9", F_INT,
                &popln_data.opm_flag, popln_form);
            def_fldw(6,1,"OPM Look-Ahead Time:","#######0##", F_FLOAT,
                &popln_data.opm_look, popln_form);
            def_fldw(7,1,"Auxiliary LOR Flag:","9", F_INT,
                &popln_data.aux_flag, popln_form);

            rd_fldsx(popln_form);
            free_frm(popln_form);
            break;
      default:
            break;
   }
   return;
}


/*
 * show_comp(tcomp, edit_flag)
 *
 * Usage: Displays and reads component "tcomp" data from the external
 *        COMPONENT array of ptrs. "tcomp" points to the memory holding
 *        data.
 *        Allows memory modification of fixed data (except
 *        empirical distributions); or flagged for creation of data
 *        if edit_flag = FALSE
 * Returns: 0 if abort key pressed by user, 1 otherwise.
 */
show_comp(tcomp, edit_flag)
COMPONENT *tcomp;
int edit_flag;
{
   DFIELDPTR def_fldw();
   DFORMPTR comp_form;
   void show_empir();
   int emp_flag = FALSE;

   cls();
   comp_form = def_frm(2,18,19,43,LNORMAL,BDR_DLNP); /* allocate form */
   sw_name("Component Data", comp_form->wnp );

   /*
    * Layout the fields for the component data window.  Will be
    * used to create, display, and edit a component structure.
    * Does not yet include a branching to the empirical distribution
    * structure defined within a component. Uses function def_fldw()
    * which is defined by def_fldx() during the call.
    */


   def_fldw(1,1,"Component name:",def_pic('X',20), F_STRING,
```

```
            tcomp->name, comp_form);
    def_fldw(3,1,"Failure Distribution:","X", F_CHAR,
            &tcomp->fail_dist, comp_form);
    def_fldw(4,1,"Time Unit of Mean:","X", F_CHAR,
            &tcomp->u_fail, comp_form);

    if ( (tcomp->fail_dist == 'm' || tcomp->fail_dist == 'M') &&
         edit_flag == TRUE)
    {
       def_txtx(5,1,"Note: Empirical Distribution Data",
               LHIGHLITE, comp_form);
       def_txtx(6,7,"Will Follow This Screen",LHIGHLITE, comp_form);
       emp_flag = TRUE;
    }
    else
    if (edit_flag == TRUE)   /* set fields for standard fail parms */
    {
       def_fldw(5,3,"Fail Parameter #1:","#####@####",F_FLOAT,
               &tcomp->parm1, comp_form);
       if (tcomp->fail_dist != 'e' && tcomp->fail_dist != 'E')
       {
          def_fldw(6,3,"Fail Parameter #2:","#####@####",F_FLOAT,
                  &tcomp->parm2, comp_form);
       }
    }
    else                  /* edit_flag == FALSE */
    {
       def_txtx(5,1,"Note: Failure Distribution Data",
               LHIGHLITE, comp_form);
       def_txtx(6,7,"Entry Follows This Screen",LHIGHLITE, comp_form);
    }
    def_fldw(8,1,"Cost of ER spares:","######@##",F_FLOAT,
            &tcomp->c_erspare, comp_form);
    def_fldw(9,1,"Cost of PM spares:","######@##",F_FLOAT,
            &tcomp->c_pmspare, comp_form);
    def_fldw(10,1,"ER Duration:","######@##",F_FLOAT,
            &tcomp->er_durn, comp_form);
    def_fldw(11,1,"PM Duration:","######@##",F_FLOAT,
            &tcomp->pm_durn, comp_form);
    def_fldw(12,1,"Time Unit for Durations:","X", F_CHAR,
            &tcomp->u_durn, comp_form);
    def_fldw(13,1,"PM Policy:","X", F_CHAR,
            &tcomp->pm_polcy, comp_form);
    def_fldw(14,1,"PM Interval:","######@##",F_FLOAT,
            &tcomp->pmintv, comp_form);
    def_fldw(15,1,"Time Unit for PM Interval:","X", F_CHAR,
            &tcomp->u_pmintv, comp_form);

    if ( rd_fldsx(comp_form) == ABORT_FORM && edit_flag == FALSE )
    {
       free_frm( comp_form );    /* free memory allocated to component form */
       return(0);                /* for creating, flag abort from form */
    }
    if (edit_flag)               /* check for displaying emp dist */
    {
       if (emp_flag)
          show_empir(tcomp->emp_head, tcomp->emp_nodes);
    }
    else
    if (!edit_flag)         /* creating and now want to get fail dist */
       cre_dist(tcomp);     /* data for comp "tcomp" from user */

    free_frm( comp_form );    /* free memory allocated to component form */
    return(1);
}
```

```
void show_sched(sched_head, num_elmnts)
MOF_SCHED *sched_head;
int num_elmnts;
{
   WINDOW list_wn;
   FREC sched_file;
   MOF_SCHED *ptr;
   char outp_row[20];
   int i;

   defs_wn(&list_wn,4,27,15,25,BDR_DLNP);              /* define menu window */
   sw_name("MOF Schedule Data", &list_wn);
   def_fr(&sched_file,NULLP,(num_elmnts+3),30);       /* define mem file */

   sti_file(" ", 0, &sched_file);
   sti_file("  Time      Quantity", 1, &sched_file);
   sti_file(" Period     Obtained", 2, &sched_file);
   for (ptr = sched_head, i = 3; ptr; ptr = ptr->next, ++i)
   {
      sprintf(outp_row, " %6d     %6d", ptr->s_time, ptr->s_block);
      sti_file(outp_row, i, &sched_file);
   }

   sw_mfile(&sched_file, &list_wn);      /* assign file to window */
   s_tbfmsg(OFF);                         /* set file top/bot markers off */
   vs_file(K_ESC, &list_wn);             /* put file in window and */
                                          /* set <esc> as exit key */
   free_file(&sched_file);
   return;
}

void show_empir(head, num_elmnts)
EMP_DATA *head;
int num_elmnts;
{
   WINDOW list_wn;
   FREC emp_file;
   EMP_DATA *ptr;
   char outp_row[27];
   int i;

   defs_wn(&list_wn,5,26,13,29,BDR_DLNP);             /* define menu window */
   sw_name("Empirical Distrib Data", &list_wn);
   def_fr(&emp_file,NULLP,(num_elmnts+3),30);        /* define mem file */

   sti_file(" ", 0, &emp_file);
   sti_file(" Cumulative    Response", 1, &emp_file);
   sti_file(" Probability    Value", 2, &emp_file);

   for (ptr = head, i = 3; ptr; ptr = ptr->next_intv, ++i)
   {
      sprintf(outp_row, "   %6.4f      %8.2f", ptr->probab, ptr->response);
      sti_file(outp_row, i, &emp_file);
   }
   sw_mfile(&emp_file, &list_wn);        /* assign file to window */
   s_tbfmsg(OFF);                        /* set file top/bot markers off */
   vs_file(K_ESC, &list_wn);             /* put file in window and */
                                         /* set <esc> as exit key */
   free_file(&emp_file);
   return;
}

/*
 * Usage: Definition of module to define fields in a form
 *        specified by the user. Adds a form pointer to the
 *        end of the standard call def_fld( ).
 * Returns: pointer to allocated form "formp".
```

```
 */
DFIELDPTR def_fldw(prmpt_rb,prmpt_cb,prompt,pic,fld_type,
                   datap,formp)
int prmpt_rb;
int prmpt_cb;
char *prompt;
char *pic;
int fld_type;
char *datap;
DFORMPTR formp;
{
   int fld_rb, fld_cb;

   fld_rb = prmpt_rb;
   if(prompt == NULLP)
      fld_cb = prmpt_cb;
   else
      fld_cb = prmpt_cb + strlen(prompt) + 1;

   return(def_fldx(prmpt_rb, prmpt_cb, prompt, pic, fld_type, datap,
                   fld_rb, fld_cb, NULLP, 0, LFIELDA, LNORMAL, NULLP,
                   NULLP, NULLFP, NULLP, formp));
}
```

```
/*
 * File Name:  CREATE.C
 */
#include <stdio.h>
#include <wfd.h>
#include "sm_struc.h"

/*
 * create_data( fptr )
 *
 * Function: Processes the "create" data file function within the
 *           Simulate program.  Prompts for filename, opens file,
 *           calls modules which set up and process data entry forms.
 *           Saves file when entry is complete.
 *
 * Returns: None
 */
create_data( fptr )
FILE *fptr;
{
   char *malloc( );
   char *strcpy( );
   char *blank_name;
   register int i = 0;
   int end_flag = 1;

   blank_name = stblank(19);     /* initialize name data field to blanks */
   do                            /* get component data */
   {
      free( comp_data[i] );
      comp_data[i] = (COMPONENT *) malloc(sizeof(COMPONENT));

      /* initialize data field values */

      strcpy ( comp_data[i]->name, blank_name );
      comp_data[i]->fail_dist = 'n';
      comp_data[i]->u_fail = 'd';
      comp_data[i]->parm1 = comp_data[i]->parm2 = 0.0;
      comp_data[i]->emp_head = NULL;
      comp_data[i]->emp_nodes = 0;
      comp_data[i]->c_pmspare = comp_data[i]->c_erspare = 0.0;
      comp_data[i]->u_durn = 'd';
      comp_data[i]->er_durn = 0.0;
      comp_data[i]->pm_policy = '1';
      comp_data[i]->pm_durn = 0.0;
      comp_data[i]->pmintv = 0.0;
      comp_data[i]->u_pmintv = 'd';
      end_flag = show_comp(comp_data[i],0);
      ++i;                          /* increment number of components read */
   }
   while ( (end_flag != 0) && (i <= MAX_COMP) );

   free( comp_data[i-1] );    /* free extra allocation */
   comp_data[i-1] = NULL;
   _num_comp = i - 1;

   /* init fields for rest of input data */
   strcpy( popln_data.name, blank_name );
   popln_data.mof = 'n';
   popln_data.mof_init = 0;
   popln_data.sched = NULL;
   popln_data.mof_nodes = 0;
   popln_data.c_eidt = 0.0;
   popln_data.u_eidt = 'd';
   popln_data.u_sim = 'd';
   popln_data.sim_lgth = 0.0;
   popln_data.sim_num = 1;
```

```
   popln_data.intv_size = 0.0;
   popln_data.opm_flag = 1;
   popln_data.opm_look = 0.0;
   popln_data.aux_flag = 1;

   strcpy( lor_data[0].name, blank_name );
   strcpy( lor_data[1].name, blank_name );
   lor_data[0].capacity = lor_data[1].capacity = 0;
   lor_data[0].c_usage = lor_data[1].c_usage = 0.0;
   lor_data[0].u_usage = lor_data[1].u_usage = 'd';
   lor_data[0].t_trans = lor_data[1].t_trans = 0.0;
   lor_data[0].c_trans = lor_data[1].c_trans = 0.0;
   lor_data[0].gang_flag = lor_data[1].gang_flag = 1;


   /* get rest of input data */
   show_popln(1,0);             /* call show ei function with create flag */
   show_popln(3,0);             /* call show lor function with create flag */
   show_popln(4,0);             /* call show simul function with create flag */

   save_data( fptr );        /* write data entered to file specified in call */
   return;
}

/*
 * save_data( fptr )
 *
 * Function: Saves current values of input data in memory to file
 *           specified by fptr
 *
 * Returns: 1, things went smoothly
 *          0, otherwise
 */
save_data( fptr )
FILE *fptr;
{
   EMP_DATA *emp_ptr;
   MOF_SCHED *mof_ptr;
   char *strcpy();
   register int i;

   if ( fptr != NULL)
   {
      fprintf(fptr,"%s\n",popln_data.name);
      fprintf(fptr,"%c\n",popln_data.mof);
      fprintf(fptr,"%d\n",popln_data.mof_init);

      /*
       * if MOF is specified as "schedule" (popln_data.mof), write the
       * schedule into file.
       *   'a' or other => initial block only
       *   's' or 'S' => schedule
       */

      if (popln_data.mof == 's' || popln_data.mof == 'S')
      {
         for (mof_ptr = popln_data.sched; mof_ptr; mof_ptr = mof_ptr->next)
            fprintf(fptr,"%d  %d\n", mof_ptr->s_time, mof_ptr->s_block);
         fprintf(fptr,"-1\n");
      }
      fprintf(fptr,"%lf\n",popln_data.c_eidt);
      fprintf(fptr,"%c\n",popln_data.u_eidt);
      fprintf(fptr,"%c\n",popln_data.u_sim);
      fprintf(fptr,"%lf\n",popln_data.sim_lgth);
      fprintf(fptr,"%d\n",popln_data.sim_num);
      fprintf(fptr,"%lf\n",popln_data.intv_size);
      fprintf(fptr,"%d\n",popln_data.opm_flag);
```

```
      fprintf(fptr,"%lf\n",popln_data.opm_look);
      fprintf(fptr,"%d\n",popln_data.aux_flag);

      for (i=0; i < 2; ++i)
      {
         fprintf(fptr,"%s\n",lor_data[i].name);
         fprintf(fptr,"%d\n",lor_data[i].capacity);
         fprintf(fptr,"%lf\n",lor_data[i].c_usage);
         fprintf(fptr,"%c\n",lor_data[i].u_usage);
         fprintf(fptr,"%lf\n",lor_data[i].t_trans);
         fprintf(fptr,"%lf\n",lor_data[i].c_trans);
         fprintf(fptr,"%d\n",lor_data[i].gang_flag);
      }

      for (i=0; i < _num_comp; ++i)
      {
         fprintf(fptr,"%s\n",comp_data[i]->name);
         fprintf(fptr,"%c\n",comp_data[i]->fail_dist);
         fprintf(fptr,"%c\n",comp_data[i]->u_fail);
         switch ( comp_data[i]->fail_dist )
         {
            case 'n':                          /* normal */
            case 'N':
            case 'w':                          /* weibull */
            case 'W':
            case 'g':                          /* gamma */
            case 'G':
            case 'u':                          /* uniform */
            case 'U':
            case 'l':                          /* lognormal */
            case 'L':
            case 'r':                          /* erlang */
            case 'R':
                     fprintf(fptr,"%lf\n",comp_data[i]->parm1);
                     fprintf(fptr,"%lf\n",comp_data[i]->parm2);
                     break;

            case 'e':                          /* exponential */
            case 'E':
                     fprintf(fptr,"%lf\n",comp_data[i]->parm1);
                     break;

            case 'm':                          /* empirical */
            case 'M':
                     for (emp_ptr = comp_data[i]->emp_head; emp_ptr;
                          emp_ptr = emp_ptr->next_intv)
                        fprintf(fptr,"%lf  %lf\n",
                                emp_ptr->probab, emp_ptr->response);
                     fprintf(fptr,"-1.0\n");
                     break;

            default :
                     break;
         }
         fprintf(fptr,"%lf\n",comp_data[i]->c_pmspare);
         fprintf(fptr,"%lf\n",comp_data[i]->c_erspare);
         fprintf(fptr,"%c\n",comp_data[i]->u_durn);
         fprintf(fptr,"%lf\n",comp_data[i]->er_durn);
         fprintf(fptr,"%c\n",comp_data[i]->pm_polcy);
         fprintf(fptr,"%lf\n",comp_data[i]->pm_durn);
         fprintf(fptr,"%lf\n",comp_data[i]->pmintv);
         fprintf(fptr,"%c\n",comp_data[i]->u_pmintv);
      }
      fprintf(fptr,"%c\n",'$');  /* mark end of segment (used with stats) */
      return(1);           /* file status OK */
   }
   else
```

```
    return(0);          /* bad file status */
}


/*
 * cre_dist( comp_ptr )
 *
 * Function: Sets up data entry forms and processes the input of
 *           an empirical distribution for some component
 *
 * Returns: None
 */
void cre_dist( comp_ptr )
COMPONENT *comp_ptr;
{
   DFORMPTR dist_form;
   char *malloc( );
   int head_flag = TRUE;
   int i = 0;
   EMP_DATA *now_emp, *emp_ptr;

   dist_form = def_frm(8,20,7,40, LNORMAL, BDR_DLNP);
   sw_name("Fail Data Entry", dist_form->wnp);

   if (comp_ptr->fail_dist == 'm' || comp_ptr->fail_dist == 'M')
   {
      while (comp_ptr->emp_head)           /* free any prev emp linked list */
      {
         now_emp = comp_ptr->emp_head;
         comp_ptr->emp_head = now_emp->next_intv;
         free ( now_emp );
      }

      now_emp = (EMP_DATA *) malloc( sizeof(EMP_DATA) );
      do
      {
         ++i;                              /* increment number of nodes read */
         if (!head_flag)
         {
            now_emp->next_intv = (EMP_DATA *) malloc( sizeof(EMP_DATA) );
            emp_ptr = now_emp;
            now_emp = now_emp->next_intv;
         }
         else
         {
            head_flag = FALSE;
            comp_ptr->emp_head = emp_ptr = now_emp;
         }
         now_emp->probab = 0.0;
         now_emp->response = 0.0;

         def_txtx(1,1,"Enter Failure Distribution Data.",
                  LHIGHLITE, dist_form);
         def_fldw(3,1,"Cumulative Probability:","#.######", F_FLOAT,
                  &now_emp->probab, dist_form);
         def_fldw(4,1,"Response Value:","#######.##", F_FLOAT,
                  &now_emp->response, dist_form);
      }
      while ( rd_fldsx(dist_form) != ABORT_FORM );

      if (now_emp == comp_ptr->emp_head)     /* user aborted at first entry */
         comp_ptr->emp_head = NULL;
      else                                   /* set NULL at end of list */
         emp_ptr->next_intv = NULL;
      free( now_emp );              /* free extra node allocated */
      comp_ptr->emp_nodes = i;
   }
```

```
   else
   {
      def_txtx(1,1,"Enter Failure Distribution Data.",
               LHIGHLITE, dist_form);
      def_fldw(3,1,"Fail Parameter #1:","###@####",F_FLOAT,
               &comp_ptr->parm1, dist_form);
      if (comp_ptr->fail_dist != 'e' && comp_ptr->fail_dist != 'E')
         def_fldw(4,1,"Fail Parameter #2:","###@####",F_FLOAT,
                  &comp_ptr->parm2, dist_form);
      rd_fldsx(dist_form);
   }
   return;
}

/*
 * cre_sched()
 *
 * Function: Sets up data entry forms and processes the input of
 *           an MOF schedule if specified in EI data
 *
 * Returns: None
 */
void cre_sched()
{
   DFORMPTR sched_form;
   char *malloc();
   int head_flag = TRUE;
   int i = 0;
   MOF_SCHED *now_mof, *mof_ptr;

   sched_form = def_frm(8,20,7,40, LNORMAL, BDR_DLNP);
   sw_name("MOF Schedule Entry", sched_form->wnp);

   while (popln_data.sched)            /* free any prev emp linked list */
   {
      now_mof = popln_data.sched;
      popln_data.sched = now_mof->next;
      free ( now_mof );
   }

   now_mof = (MOF_SCHED *) malloc( sizeof(MOF_SCHED) );
   do
   {
      ++i;                              /* increment number of nodes read */
      if (!head_flag)
      {
         now_mof->next = (MOF_SCHED *) malloc( sizeof(MOF_SCHED) );
         mof_ptr = now_mof;
         now_mof = now_mof->next;
      }
      else
      {
         head_flag = FALSE;
         popln_data.sched = mof_ptr = now_mof;
      }
      now_mof->s_time = 0;
      now_mof->s_block = 0;

      def_txtx(1,1,"Enter Acquisition Schedule.",
               LHIGHLITE, sched_form);
      def_fldw(3,1,"Time of Acquisition:","999999", F_INT,
               &now_mof->s_time, sched_form);
      def_fldw(4,1,"Number of End Items:","999", F_INT,
               &now_mof->s_block, sched_form);
   }
   while ( rd_fldsx(sched_form) != ABORT_FORM );
```

```
   if (now_mof == popln_data.sched)     /* user aborted at first entry */
      popln_data.sched = NULL;
   else                                   /* set NULL at end of list */
      mof_ptr->next = NULL;
   free( now_mof );              /* free extra node allocated */
   popln_data.mof_nodes = i;
   return;
}
```

```
/*
 * File Name:  READDATA.C
 */
#include <stdio.h>
#include <ctype.h>
#include "sm_struc.h"

/*
 * retrv_file(FILE *fileptr)
 *
 * Usage: Retrieves a specified data file and assigns data
 *        to the external data structures (POPLN, COMPONENT,
 *        Open the filename before passing for read, close after
 *        if necessary.
 *
 * Routines   read_popln( ) - reads data for extern POPLN struct
 * Called:    read_lor( ) - reads data for extern LOR struct
 *            read_comp( ) - reads component data for one COMPONENT struct
 *                           called until EOF is found or MAX_COMP is
 *                           is exceeded. Allocates space as needed.
 *                           Returns ptr to COMPONENT structure filled.
 */
retrv_file(fptr)
FILE *fptr;
{
   extern int _num_comp;
   extern POPLN popln_data;
   extern COMPONENT *comp_data[];
   COMPONENT *read_comp( );
   int i = 0;              /* counter for num of components read */

   if (fptr != NULL)
   {
      read_popln(fptr);                      /* read population data */
      read_lor(fptr);                        /* read LOR data */

      while ( (comp_data[i] = read_comp(fptr)) != NULL
                && (i < MAX_COMP) )       /* read component data */
         i += 1;                          /* set to read next component */
      _num_comp = i;
      return(i);
   }
   else
      return(0);      /* return number of components read from file */
}


/*
 * int read_popln(FILE *fptr)
 *
 * Usage: used to read a single structure of population data
 *        from file defined by the pointer to FILE passed in
 *        the call. Sample file "test.dat" contains commented
 *        source on layout of the data file. The population
 *        structure contains all necessary simulation, population
 *        and end item parameters.
 * Returns: 0 to indicate error, 1 otherwise.
 *
 */
read_popln(fptr)
FILE *fptr;
{
   extern POPLN popln_data;
   void comment( );

   comment(fptr);
   if ( fscanf(fptr,"%s",popln_data.name) != EOF )
```

```
   {
      comment(fptr);
      fscanf(fptr,"%c",&popln_data.mof);

      comment(fptr);
      fscanf(fptr,"%d",&popln_data.mof_init);  /* get initial size of popln */


      /*
       * if MOF is specified as "schedule" (popln_data.mof), read the
       * schedule into list.
       *   'a' or other => initial block only
       *   's' or 'S' => schedule
       */

      if (popln_data.mof == 's' || popln_data.mof == 'S')
      {
         comment(fptr);
         popln_data.mof_nodes = read_sched(fptr, &popln_data);
      }
      comment(fptr);
      fscanf(fptr,"%lf",&popln_data.c_eidt);
      comment(fptr);
      fscanf(fptr,"%c",&popln_data.u_eidt);
      comment(fptr);
      fscanf(fptr,"%c",&popln_data.u_sim);
      comment(fptr);
      fscanf(fptr,"%lf",&popln_data.sim_lgth);
      comment(fptr);
      fscanf(fptr,"%d",&popln_data.sim_num);
      comment(fptr);
      fscanf(fptr,"%lf",&popln_data.intv_size);
      comment(fptr);
      fscanf(fptr,"%d",&popln_data.opm_flag);
      comment(fptr);
      fscanf(fptr,"%lf",&popln_data.opm_look);
      comment(fptr);
      fscanf(fptr,"%d",&popln_data.aux_flag);
      return(1);
   }
   else
      return(0);
}

/*
 * int read_lor(FILE *fptr)
 *
 * Usage: used to read 3 structures of LOR data from file
 *        passed in the call. There must be entries for
 *        all 3 levels, use dummy names if not needed.
 *        Kind of primitive, will eventually changed to
 *        recognize end of LOR data.
 * Returns: 0 to indicate error (an early EOF), 1 otherwise.
 *
 */
read_lor(fptr)
FILE *fptr;
{
   void comment();
   int i;

   for (i=0; i < 2; ++i)
   {
      comment(fptr);
      if ( fscanf(fptr,"%s",lor_data[i].name) != EOF )
      {
         comment(fptr);
```

```
         fscanf(fptr,"%d",&lor_data[i].capacity);
         comment(fptr);
         fscanf(fptr,"%lf",&lor_data[i].c_usage);
         comment(fptr);
         fscanf(fptr,"%c",&lor_data[i].u_usage);
         comment(fptr);
         fscanf(fptr,"%lf",&lor_data[i].t_trans);
         comment(fptr);
         fscanf(fptr,"%lf",&lor_data[i].c_trans);
         comment(fptr);
         fscanf(fptr,"%d",&lor_data[i].gang_flag);
      }
      else
         return(0);
   }
   return(1);
}

/*
 * COMPONENT read_comp(FILE fptr)
 *
 * Usage: used to read a single structure of component data
 *        from file defined by the pointer to FILE passed in
 *        the call. Sample file "test.dat" contains commented
 *        source on layout of the data file.
 * Returns: pointer to COMPONENT structure containing data
 *          just read, or NULL to indicate error.
 */
COMPONENT *read_comp(fptr)
FILE *fptr;
{
   COMPONENT *comp_ptr;
   void comment( );
   char *malloc( );
   char ch;
   char temp_name[20];

   comment(fptr);
   if ( (ch = getc(fptr)) != '#' )
   {
      ungetc(ch,fptr);  /* return read character and continue */
                        /* with component read */
      fscanf(fptr,"%s",temp_name);
      comp_ptr = (COMPONENT *) malloc(sizeof(COMPONENT));

      strcpy(comp_ptr->name, temp_name);
      comment(fptr);
      fscanf(fptr,"%c",&comp_ptr->fail_dist);
      comment(fptr);
      fscanf(fptr,"%c",&comp_ptr->u_fail);  /* read time unit for dist */

      switch ( comp_ptr->fail_dist )
      {
         case 'n':                        /* normal */
         case 'N':                        /* normal */
         case 'w':                        /* weibull */
         case 'W':                        /* weibull */
         case 'g':                        /* gamma */
         case 'G':                        /* gamma */
         case 'u':                        /* uniform */
         case 'U':                        /* uniform */
         case 'l':                        /* lognormal */
         case 'L':                        /* lognormal */
         case 'r':                        /* erlang */
         case 'R':                        /* erlang */
                  comment(fptr);
                  fscanf(fptr,"%lf",&comp_ptr->parm1);
```

```
                comment(fptr);
                fscanf(fptr,"%lf",&comp_ptr->parm2);
                break;

        case 'e':                       /* exponential */
        case 'E':                       /* exponential */
                comment(fptr);
                fscanf(fptr,"%lf",&comp_ptr->parm1);
                break;

        case 'm':                       /* empirical */
        case 'M':                       /* empirical */
                comment(fptr);
                comp_ptr->emp_nodes = read_empir(fptr, comp_ptr);
                break;

        default :
                break;
      }
      comment(fptr);
      fscanf(fptr,"%lf",&comp_ptr->c_pmspare);
      comment(fptr);
      fscanf(fptr,"%lf",&comp_ptr->c_erspare);
      comment(fptr);
      fscanf(fptr,"%c",&comp_ptr->u_durn);
      comment(fptr);
      fscanf(fptr,"%lf",&comp_ptr->er_durn);
      comment(fptr);
      fscanf(fptr,"%c",&comp_ptr->pm_polcy);
      comment(fptr);
      fscanf(fptr,"%lf",&comp_ptr->pm_durn);
      comment(fptr);
      fscanf(fptr,"%lf",&comp_ptr->pmintv);
      comment(fptr);
      fscanf(fptr,"%c",&comp_ptr->u_pmintv);
      comment(fptr);
   }
   else
   {
      comment(fptr);      /* get to next non-whitespace character */
      comp_ptr = NULL;    /* no component was read */
   }
   return(comp_ptr);
}

/*
 * read_empir(FILE *ptr, COMPONENT *comp_ptr)
 *
 * Pass FILE pointer to the input file and pointer to
 * the COMPONENT structure in which the "head" pointer
 * to empirical data belongs.
 *
 * e.g. read_empir(fptr, comp_data[2])
 *
 * Returns: int number of elements read into the list;
 *          zero (0) indicates error => empir was specified
 *          but no data was found.
 */
read_empir( fp, comp_ptr)
COMPONENT *comp_ptr;
FILE *fp;
{
   char *malloc( );
   EMP_DATA *ptr;
   double x = -1.0;
   double y;
   int i = 0;
```

```
   while ( (fscanf(fp,"%lf",&x) != EOF) && (x >= 0.0) )  /* negative number */
   {                                                     /* indicates end */
      fscanf(fp,"%lf",&y);
      if (i != 0)
      {
         /* Allocate space and assign ptrs to each successive element */
         /* in the list, except the head */

         ptr->next_intv = (EMP_DATA *) malloc(sizeof(EMP_DATA));
         ptr = ptr->next_intv;
      }
      else
      {
         /* Allocate space for head list element and assign ptr in */
         /* the COMPONENT structure passed */

         comp_ptr->emp_head = (EMP_DATA *) malloc(sizeof(EMP_DATA));
         ptr = comp_ptr->emp_head;
      }
      i += 1;
      ptr->probab = x;
      ptr->response = y;
      ptr->next_intv = NULL;
   }
   if (i == 0)
       comp_ptr->emp_head = NULL;

   return(i);   /* return number of nodes read */
}

/*
 * read_sched(FILE *ptr, POPLN *popln_ptr)
 *
 * Pass FILE pointer to the input file and pointer to
 * the POPLN structure in which the "head" pointer
 * to MOF schedule data belongs.
 *
 * e.g. read_sched(fptr, &popln_data)
 *
 * Returns: int number of elements read into the list;
 *          zero (0) indicates error => sched was specified
 *          for MOF but no data was found.
 */
read_sched( fp, popln_ptr)
POPLN *popln_ptr;
FILE *fp;
{
   char *malloc( );
   MOF_SCHED *ptr;
   int x = -1;
   int y;
   int i = 0;

   while ( (fscanf(fp,"%d",&x) != EOF) && (x >= 0) )  /* negative number */
   {                                                  /* indicates end */
      fscanf(fp,"%d",&y);
      if (i != 0)
      {
         /* Allocate space and assign ptrs to each successive element */
         /* in the list, except the head */

         ptr->next = (MOF_SCHED *) malloc(sizeof(MOF_SCHED));
         ptr = ptr->next;
      }
      else
      {
```

```
         /* Allocate space for head list element and assign ptr in */
         /* the POPLN structure passed */

         popln_ptr->sched = (MOF_SCHED *) malloc(sizeof(MOF_SCHED));
         ptr = popln_ptr->sched;
      }
      i += 1;
      ptr->s_time = x;
      ptr->s_block = y;
      ptr->next = NULL;
   }
   if (i == 0)
       popln_ptr->sched = NULL;

   return(i);   /* return number of nodes read */
}


/*
 * comment(fp)
 *
 * Usage: after any file read this function can be used to
 *        detect and skip comment statements preceded by a '*'.
 *        All successive lines will be skipped until a
 *        non-whitespace character not matching the comment is
 *        found at the beginning of a new line.
 */
#define EOL  '\n'
#define COMMENT '*'
void comment(fp)
FILE *fp;
{
   register int ch = COMMENT;

   while (ch == COMMENT)
   {
      while ( isspace(ch = getc(fp)) )   /* skip whitespace before first */
         ;                               /* character                    */
      if (ch == COMMENT)
         while (getc(fp) != EOL)
            ;
      else
         ungetc(ch,fp);
   }
   return;
}
```

```
/*
 * File Name:  GR_MAIN.H
 *
 * This header file is included in the main module of the Analyze
 * system (ANALYZE.C).  A duplicate of this file, with the global
 * declarations following the last "typedef" changed to "external",
 * is included in each module of Analyze.
 */

typedef struct gr_data        /* basic set of raw stats maintained */
        {
           int init_flag;     /* flag that specifies initialization */
           double max_stat;   /* maximum value of stat */
           double min_stat;   /* minimum value of stat */
           double mean;       /* mean value of stat */
           double stddev;     /* std dev value of stat */
        }
        GR_DATA;

typedef struct gr_pstruc      /* set of statistics maintained for popln */
                              /* for each interval */
        {
           GR_DATA *dt;         /* downtime */
           GR_DATA *std_serv;   /* service time at STD LOR */
           GR_DATA *aux_serv;   /* service time at AUX LOR */
           GR_DATA *std_dt;     /* trips to STD LOR */
           GR_DATA *aux_dt;     /* trips to AUX LOR */
           double avg_ei;       /* avg number of EI in population */
        }
        GR_PSTRUC;

typedef struct gr_cstruc      /* set of statistics maintained for each */
                              /* component for each interval */
        {
           GR_DATA *std_er;   /* times to STD for ER */
           GR_DATA *aux_er;   /* times to AUX for ER */
           GR_DATA *std_pm;   /* times to STD for PM */
           GR_DATA *aux_pm;   /* times to AUX for PM */
        }
        GR_CSTRUC;

GR_DATA *plot_stat;          /* ptr to storage for plotting statistic array */

GR_PSTRUC *gr_popln;         /* ptr to storage for popln stats array */

GR_CSTRUC *gr_comp[MAX_COMP];/* ptrs to storage for comp stats arrays */

int _num_intv;               /* number of intervals in stats, global var */

int gr_type;            /* switch between (mean, mean/stddev, mean/max/min) */
int stat_type;          /* switch between (frequency, percent, dollars) */
int data_type;          /* indicate the statistic summarized in "plot_stat" */

char gr_title[51];      /* graph title string */
```

```
/*
 * File Name:  ANALYZE.C
 */
#include <stdio.h>
#include <wfd.h>
#define WFD_FLOAT
#include <wfd_glob.h>
#include "sm_main.h"
#include "gr_main.h"

/*
 * main()
 *
 * Function: Main module for ANALYZE program, presents main menu
 *           and processes user's choice
 *
 * Returns: None
 */
main()
{
   char *stat_file = "stats.out      ";   /* default file name */
   FILE *fstat;                           /* ptr to file */
   char *strcpy();
   char verify = 'n';                     /* exit menu flag */
   int choice;
   int file_read = 0;                     /* flag indicating file read */
   void anlyz_menu();
   WINDOW wn;
   WINDOW wait_wn;

   init_wfd();

   defs_wn(&wait_wn,11,20,3,39,BDR_DLNP);     /* define status window */
   wait_wn.att = LHIGHLITE;
   sw_csadv(OFF,&wait_wn);
   sw_cleor(OFF,&wait_wn);

   defs_wn(&wn,5,16,13,47,BDR_DLNP);          /* define menu window */
   sw_name("Analyze Menu", &wn);
   sw_plcsr(ON, &wn);
   strcpy(popln_data.data_file, "None Defined   ");
   anlyz_menu(&wn);         /* define and set menu on screen for first time */

   for (;;)
   {
      choice = ki();      /* get keystroke from user */
      switch ( choice )   /* branch on keystroke */
      {
         case 114:        /* Retrieve */
         case  82:
                  get_data(16,19,"Stats file to retrieve:", def_pic('X',15),
                           F_STRING, stat_file);
                  csr_hide();
                  if ( (fstat = fopen(stat_file, "r")) != NULL)
                  {
                     unset_wn(&wn);
                     set_wn(&wait_wn);       /* activate status window */
                     mv_cs(0,0,&wait_wn);
                     v_st("Wait...Retrieving/Summarizing Stats",&wait_wn);
                     strcpy(popln_data.data_file, stat_file);
                     retrv_file(fstat);
                     read_stats(fstat);
                     fclose(fstat);
                     file_read = 1;          /* indicate active data file */
                     unset_wn(&wait_wn);     /* remove status window */
                     anlyz_menu(&wn);
                  }
```

```
                else
                   bell();
                break;

        case 103:       /* graph */
        case  71:
                if (file_read)
                {
                   unset_wn(&wn);
                   whattype(0,NULL);     /* process graph types */
                   anlyz_menu(&wn);
                }
                else
                   bell();
                break;

        case 112:       /* print */
        case  80:
                if (file_read)
                {
                   unset_wn(&wn);
                   print_stats();
                   anlyz_menu(&wn);
                }
                else
                   bell();
                break;

        case 113:
        case  81:
                if ( (verify = verify_exit(16,18)) == YES)
                {
                   unset_wn(&wn);
                   csr_show();
                   cls();
                   exit();
                }
                else
                   csr_hide();
                break;

        default:
                bell();
     }
   }
}

void anlyz_menu(wn)
WINDOW *wn;
{
   cls();
   set_wn(wn);     /* place window on screen */

   v_plst(1,1,"Current Active Data File: ",wn);
   wn->att = LHIGHLITE;
   v_st(popln_data.data_file, wn);
   v_plst(5,13,"r",wn);
   v_plst(6,13,"g",wn);
   v_plst(7,13,"p",wn);
   v_plst(8,13,"q",wn);
   wn->att = LNORMAL;

   v_plst(3,CENTER_TXT,"Menu Choices",wn);
   v_plst(5,15,"- Retrieve file",wn);
   v_plst(6,15,"- Graph data",wn);
   v_plst(7,15,"- Print data",wn);
   v_plst(8,15,"- Quit program",wn);
```

```
   csr_hide( );

   return;
}

/*
 * whattype( print_flag, fprt )
 *
 * Function: Processes user's choice of what data to graph or print
 *
 * Returns: None
 */
whattype(print_flag,fprt)
int print_flag;
FILE *fprt;
{
   WINDOW wn;
   int choice;
   void type_menu( );

   defs_wn(&wn,5,16,12,47,BDR_DLNP);      /* define menu window */
   if (print_flag == FALSE)
      sw_name("Main Graph Menu", &wn);
   else
      sw_name("Print Stat Menu", &wn);
   sw_plcsr(ON, &wn);
   type_menu(&wn,print_flag);/* define and set menu on screen for first time */

   for (;;)
   {
      choice = ki( );      /* get keystroke from user */
      switch ( choice )   /* branch on keystroke */
      {
         case 99:             /* "c" */
         case 67:             /* "C" */
                  unset_wn(&wn);
                  comp_graph(print_flag,fprt);
                  type_menu(&wn,print_flag);
                  break;
         case 108:            /* "l" */
         case  76:            /* "L" */
                  unset_wn(&wn);
                  lor_graph(print_flag,fprt);
                  type_menu(&wn,print_flag);
                  break;
         case 112:            /* "p" */
         case  80:            /* "P" */
                  unset_wn(&wn);
                  popln_graph(print_flag,fprt);
                  type_menu(&wn,print_flag);
                  break;
         case 113:            /* "q" */
         case  81:            /* "Q" */
                  unset_wn(&wn);
                  return;
         default:
                  bell( );
      }
   }
}

void type_menu(wn,print_flag)
WINDOW *wn;
int print_flag;
{
   cls( );
   set_wn(wn);     /* place window on screen */
```

```
   v_plst(1,1,"Current Active Data File: ",wn);
   wn->att = LHIGHLITE;
   v_st(popln_data.data_file, wn);
   v_plst(5,14,"c",wn);
   v_plst(6,14,"l",wn);
   v_plst(7,14,"p",wn);
   v_plst(8,14,"q",wn);
   wn->att = LNORMAL;

   if (print_flag == FALSE)
      v_plst(3,CENTER_TXT,"Graph Data Options",wn);
   else
      v_plst(3,CENTER_TXT,"Print Data Options",wn);
   v_plst(5,16,"- Component",wn);
   v_plst(6,16,"- LOR",wn);
   v_plst(7,16,"- Population",wn);
   v_plst(8,16,"- Quit",wn);
   csr_hide();

   return;
}

/*
 * print_stats()
 *
 * Function: Displays the main print menu and processes user's
 *           choice of statistics category (echo input, raw data,
 *           or displayed stats)
 *
 * Returns: None
 */
print_stats()
{
   WINDOW wn;
   int choice;
   FILE *fprt;              /* ptr to output device (file or printer) */
   FILE *prt_type();
   void print_menu();

   defs_wn(&wn,5,16,13,47,BDR_DLNP);      /* define menu window */
   sw_name("Main Print Menu", &wn);
   sw_plcsr(ON, &wn);
   print_menu(&wn);         /* define and set menu on screen for first time */

   for (;;)
   {
      choice = ki();      /* get keystroke from user */
      switch ( choice )   /* branch on keystroke */
      {
         case 105:              /* "i" */
         case  73:              /* "I" */
                  fprt = prt_type(16,18);
                  if (fprt == NULL)
                     goto bad_select;
                  unset_wn(&wn);
                  prt_input(fprt);
                  fclose(fprt);
                  print_menu(&wn);
                  break;
         case 114:              /* "r" */
         case  82:              /* "R" */
                  fprt = prt_type(16,18);
                  if (fprt == NULL)
                     goto bad_select;
                  unset_wn(&wn);
                  prt_raw(fprt);
```

```
                fclose(fprt);
                print_menu(&wn);
                break;
        case 115:               /* "s" */
        case  83:               /* "S" */
                fprt = prt_type(16,18);
                if (fprt == NULL)
                   goto bad_select;
                unset_wn(&wn);
                whattype(1,fprt);
                fclose(fprt);
                print_menu(&wn);
                break;
        case 113:               /* "q" */
        case  81:               /* "Q" */
                unset_wn(&wn);
                return;
        default:
        bad_select:
                bell();
     }
   }
}


void print_menu(wn)
WINDOW *wn;
{
   cls();
   set_wn(wn);     /* place window on screen */

   v_plst(1,1,"Current Active Data File: ",wn);
   wn->att = LHIGHLITE;
   v_st(popln_data.data_file, wn);
   v_plst(5,11,"i",wn);
   v_plst(6,11,"r",wn);
   v_plst(7,11,"s",wn);
   v_plst(8,11,"q",wn);
   wn->att = LNORMAL;

   v_plst(3,CENTER_TXT,"Data Type Choices",wn);
   v_plst(5,13,"- Input Summary",wn);
   v_plst(6,13,"- Raw Stats Summary",wn);
   v_plst(7,13,"- Stats Summary",wn);
   v_plst(8,13,"- Quit",wn);
   csr_hide();

   return;
}

/*
 * verify_exit( row, col )
 *
 * Function: Displays message at screen location (row,col) and prompts
 *           user to answer if an exit request was intentional
 *
 * Returns: 1, if continue with exit
 *          0, otherwise
 */
verify_exit(row, col)
int row;             /* screen row to start message at */
int col;             /* screen column to start message at */
{
   int choice;
   WINDOW exit_wn;

   defs_wn(&exit_wn,row,col,1,37,BDR_OP);     /* define status window */
```

```
   exit_wn.att = LHIGHLITE;
   sw_popup(ON,&exit_wn);
   sw_plcsr(ON, &exit_wn);
   set_wn(&exit_wn);
   v_plst(0,0,"Do you really want to exit? (y/n) ",&exit_wn);
   for (;;)
   {
      choice = ki();
      switch ( choice )
      {
         case 121:            /* yes */
         case  89:
                  unset_wn(&exit_wn);
                  return(1);
         case 110:
         case  78:            /* no */
                  unset_wn(&exit_wn);
                  return(0);
         default:
                  bell();
      }
   }
}

/*
 * prt_type(row,col)
 *
 * Function: Prompts user to select output to file or printer
 * if file, prompts for filename, then attempts to open device
 *
 * Returns: pointer to selected output device
 */
FILE *prt_type(row, col)
int row;             /* screen row to start message at */
int col;             /* screen column to start message at */
{
   int choice;
   char *prt_file = "data.prn       ";
   FILE *fprt;
   WINDOW prt_wn;

   defs_wn(&prt_wn,row,col,1,37,BDR_OP);     /* define status window */
   prt_wn.att = LHIGHLITE;
   sw_popup(ON,&prt_wn);
   sw_plcsr(ON, &prt_wn);
   set_wn(&prt_wn);
   v_plst(0,0,"Output to printer or file? (p/f) ",&prt_wn);
   for (;;)
   {
      choice = ki();
      switch ( choice )
      {
         case 112:            /* p */
         case  80:
                  unset_wn(&prt_wn);
                  fprt = fopen("lpt1","w");
                  return(fprt);
         case 102:
         case  70:            /* f */
                  unset_wn(&prt_wn);
                  get_data(row,col+1,"Name for output file:", def_pic('X',15),
                           F_STRING, prt_file);
                  csr_hide();
                  fprt = fopen(prt_file,"w");
                  return(fprt);
         default:
                  bell();
```

```
        }
    }
}
```

```
/*
 * File Name:  READSTAT.C
 */
#include <stdio.h>
#include "sm_struc.h"
#include "gr_struc.h"

/*
 * read_stats(FILE *fstat)
 *
 * Function: Performs the stats "retrieve" operation from the main
 *           analyze menu (after global popln and comp structure
 *           data has been read). Initializes interval structures,
 *           reads and sums interval variables, then computes the
 *           means and std deviations for each interval var.
 * Returns: none
 */
read_stats( fstat )
FILE *fstat;
{
   void comment( );   /* file garbage remover function */
   char ch;           /* record identifier */
   double avg_ei;     /* avg num ei, for "N" or "M" record */
   int intv_id;       /* interval id for "M" record */
   register int i;    /* counter */
   register int j;    /* counter */

   init_stats( );     /* initialize stat collect structures */

   /*
    * read and process input statistics file
    */
   comment(fstat);         /* get to first char of stats file */
   while ( (ch = getc(fstat)) != EOF )
   {
      switch ( ch )
      {
         case '*':               /* skip past comment line */
                  ungetc(ch,fstat);
                  comment(fstat);
                  break;
         case 'D':               /* process "downtime" record */
                  dt_process(fstat);
                  break;
         case 'L':               /* process "LOR" record */
                  lor_process(fstat);
                  break;
         case 'M':               /* process "mof_schedule" record */
                  fscanf(fstat,"%d %lf", &intv_id, &avg_ei);
                  if (intv_id <= _num_intv)
                  {
                     intv_id -= 1;
                     (gr_popln + intv_id)->avg_ei = avg_ei;
                  }
                  break;
         case 'N':               /* process "no_mof_schedule" record */
                  fscanf(fstat,"%lf", &avg_ei);
                  for (i = 0; i < _num_intv; ++i)
                     (gr_popln + i)->avg_ei = avg_ei;
                  break;
         default:                /* shouldn't have any! */
                  break;
      }
   }
   /*
    * now must compute the mean and stddev in each GR_DATA structure
    * with number samples, n = popln_data.sim_num
```

```
    */
   for (i = 0; i < _num_intv; ++i)
   {
      avg_stats( (gr_popln + i)->dt, 0);
      avg_stats( (gr_popln + i)->std_serv, 1);
      avg_stats( (gr_popln + i)->aux_serv, 1);
      avg_stats( (gr_popln + i)->std_dt, 0);
      avg_stats( (gr_popln + i)->aux_dt, 0);

      for (j = 0; j < _num_comp; ++j)
      {
         avg_stats( (gr_comp[j] + i)->std_er, 0);
         avg_stats( (gr_comp[j] + i)->aux_er, 0);
         avg_stats( (gr_comp[j] + i)->std_pm, 0);
         avg_stats( (gr_comp[j] + i)->aux_pm, 0);

      }
   }
   return;
}

/*
 * init_stats()
 *
 * Function: Allocates and initializes the structures to contain interval
 *           data for the global vars "gr_popln" and "gr_comp[MAX_COMP]"
 *           If previously called, routine first frees memory associated
 *           with previous allocation, then performs a new allocation
 *           using what might be new parameters.
 * Returns: None
 */
init_stats()
{
   static int called = FALSE;     /* if TRUE, flags previous alloc performed */
   static int last_num_comp;      /* saves "_num_comp" for next dealloc read */
   register int i;                /* loop counter */
   register int j;                /* loop counter */
   char *calloc();

   if (called == TRUE)    /* free memory from previous allocation */
   {
      for (i = 0; i < _num_intv; ++i)
      {
         free( (gr_popln + i)->dt );
         free( (gr_popln + i)->std_serv );
         free( (gr_popln + i)->aux_serv );
         free( (gr_popln + i)->std_dt );
         free( (gr_popln + i)->aux_dt );

         for (j = 0; j < last_num_comp; ++j)
         {
            free( (gr_comp[j] + i)->std_er );
            free( (gr_comp[j] + i)->aux_er );
            free( (gr_comp[j] + i)->std_pm );
            free( (gr_comp[j] + i)->aux_pm );
         }
      }
      for (j = 0; j < last_num_comp; ++j)
         free( gr_comp[j] );
      free( gr_popln );
      free( plot_stat );
   }
   else
      called = TRUE;

   last_num_comp = _num_comp;     /* save component count from current read */
                                  /* (used for any subsequent "free" above) */
```

```
   /*
    *initialize graphing structures for read and collect
    */
   _num_intv = (int) (popln_data.sim_lgth / popln_data.intv_size);

   if ( (gr_popln = (GR_PSTRUC *) calloc(_num_intv,sizeof(GR_PSTRUC))) == NULL)
      abort();        /* abort due to insufficient memory */

   for (i = 0; i < _num_intv; ++i)
   {
      init_data( &((gr_popln +i)->dt) );
      init_data( &((gr_popln +i)->std_serv) );
      init_data( &((gr_popln +i)->aux_serv) );
      init_data( &((gr_popln +i)->std_dt) );
      init_data( &((gr_popln +i)->aux_dt) );

      (gr_popln + i)->avg_ei = 0.0;
   }

   for (j = 0; j < _num_comp; ++j)
   {
      if ( (gr_comp[j] = (GR_CSTRUC *) calloc(_num_intv,sizeof(GR_CSTRUC)))
            == NULL)
         abort();     /* abort due to insufficient memory */

      for (i = 0; i < _num_intv; ++i)
      {
         init_data( &((gr_comp[j] + i)->std_er) );
         init_data( &((gr_comp[j] + i)->aux_er) );
         init_data( &((gr_comp[j] + i)->std_pm) );
         init_data( &((gr_comp[j] + i)->aux_pm) );
      }
   }
   if ( (plot_stat = (GR_DATA *) calloc(_num_intv, sizeof(GR_DATA))) == NULL)
      abort();
   return;
}

/*
 * init_data( GR_DATA **ptr )
 *
 * Function: Takes address of allocated ptr to a GR_DATA structure,
 *           allocates memory for the GR_DATA structure, and initializes
 *           necessary vars.
 * Returns: none
 */
init_data(ptr)
GR_DATA **ptr;
{
   char *malloc();

   if ( (*ptr = (GR_DATA *) malloc(sizeof(GR_DATA))) == NULL)
      abort();     /* abort due to insufficient memory */
   (*ptr)->init_flag = FALSE;
   (*ptr)->max_stat = 0.0;
   (*ptr)->min_stat = 0.0;
   (*ptr)->mean = 0.0;
   (*ptr)->stddev = 0.0;
   return;
}


/*
 * dt_process(FILE *fstat)
 *
 * Function: Reads a "D" record from the stats input file and calls
 *           summing routine for each var read, based on the interval
```

```
 *           contained in the record. Checks for interval in range
 *           and ignores if beyond _num_intv.
 * Returns: none
 */
dt_process(fstat)
FILE *fstat;
{
   int intv_id;          /* interval id */
   double brkpt;         /* interval breakpoint */
   double downtime;      /* downtime */
   int std_dt;           /* std dt instances */
   int aux_dt;           /* aux dt instances */

   fscanf(fstat,"%d %lf %lf %d %d",
          &intv_id, &brkpt, &downtime, &std_dt, &aux_dt);

   if (intv_id > _num_intv)     /* ignore interval data beyond _num_intv */
      return;

   --intv_id;                   /* convert intv_id for array indexing */

   sum_stats( (gr_popln + intv_id)->dt, downtime);
   sum_stats( (gr_popln + intv_id)->std_dt, (double) std_dt);
   sum_stats( (gr_popln + intv_id)->aux_dt, (double) aux_dt);

   return;
}

/*
 * lor_process(FILE *fstat)
 *
 * Function: Reads a "L" record from the stats input file and calls
 *           summing routine for each var read, based on the interval
 *           and comp_id contained in the record. Checks for interval
 *           in range and ignores if beyond _num_intv.
 * Returns: none
 */
lor_process(fstat)
FILE *fstat;
{
   int intv_id;          /* interval id */
   int comp_id;          /* component id */
   double brkpt;         /* interval breakpoint */
   int std_er;           /* trips to std for ER */
   int aux_er;           /* trips to aux for ER */
   int std_pm;           /* trips to std for PM */
   int aux_pm;           /* trips to aux for PM */
   double std_serv;      /* service time used at std */
   double aux_serv;      /* service time used at aux */

   fscanf(fstat,"%d %d %lf %d %d %d %d %lf %lf",
          &intv_id, &comp_id, &brkpt, &std_er, &aux_er, &std_pm, &aux_pm,
          &std_serv, &aux_serv);

   if (intv_id > _num_intv)     /* ignore intervals past _num_intv */
      return;

   --intv_id;                   /* convert intv_id for array indexing */

   sum_stats( (gr_comp[comp_id] + intv_id)->std_er, (double) std_er);
   sum_stats( (gr_comp[comp_id] + intv_id)->aux_er, (double) aux_er);
   sum_stats( (gr_comp[comp_id] + intv_id)->std_pm, (double) std_pm);
   sum_stats( (gr_comp[comp_id] + intv_id)->aux_pm, (double) aux_pm);

   sum_stats( (gr_popln + intv_id)->std_serv, std_serv);
   sum_stats( (gr_popln + intv_id)->aux_serv, aux_serv);
```

```
   return;
}

/*
 * sum_stats( GR_DATA *ptr, double stat )
 *
 * Funtion: Performs the summing/collection operation for a
 *          GR_DATA structure. If first call for a particular
 *          structure initializes "max" and "min" vars, otherwise
 *          maintains max/min. Sums passed stat and stat¢2 in the
 *          "mean" and "stddev" vars, in preparation for computing
 *          mean and stddev once all data has been read.
 * Usage: Pass ptr to the GR_DATA for maintenance, and the stat value
 *        to be processed.
 * Returns: none
 */
sum_stats( ptr, stat )
GR_DATA *ptr;
double stat;
{
   double pow( );

   ptr->mean += stat;
   ptr->stddev += pow(stat,2.0);

   if (ptr->init_flag == FALSE)
   {
      ptr->max_stat = ptr->min_stat = stat;
      ptr->init_flag = TRUE;
      return;
   }
   if (stat > ptr->max_stat)
      ptr->max_stat = stat;
   else
   if (stat < ptr->min_stat)
      ptr->min_stat = stat;

   return;
}

/*
 * avg_stats( GR_DATA *ptr, int flag )
 *
 * Function: Computes the std deviation and mean for any set
 *           of collected GR_DATA containing a mean = sum(x(i))
 *           and stddev = sum(x(i)¢2).  Uses popln_data.sim_num
 *           as sample size unless flag = TRUE, then sample size
 *           equals (_num_comp * sim_num)
 * Usage: Pass ptr to the GR_DATA structure to perform operation on.
 * Returns: none
 */
avg_stats( ptr, flag )
GR_DATA *ptr;
int flag;
{
   double pow( ),sqrt( );
   double variance;
   double n_mean;            /* sample size for mean calc */
   double n_var;             /* sample for variance calc */

   n_mean = (double) popln_data.sim_num;
   if (flag == FALSE)
      n_var = (double) popln_data.sim_num;
   else
   if (flag == TRUE)
      n_var = (double) (popln_data.sim_num * _num_comp);
```

```
   variance = ((n_var * ptr->stddev) - pow(ptr->mean,2.0)) /
              (n_var * (n_var - 1.0));

   if (flag == TRUE)
      variance *= pow( (double) _num_comp, 2.0 );  /* transform var for */
                                                   /* std and aux service */
   ptr->stddev = sqrt(variance);
   ptr->mean = ptr->mean / n_mean;

   return;
}
```

```
/*
 * File Name:  GRAPHMNU.C
 */
#include <stdio.h>
#include <wfd.h>
#include "sm_struc.h"
#include "gr_struc.h"

/*
 * popln_graph( print_flag, fprt )
 *
 * Function: Displays stat types avail for population, and process
 *           user's choice.  Either prints or graphs depending on
 *           "print_flag"
 * Returns: None
 */
popln_graph(print_flag,fprt)
int print_flag;            /* if TRUE, indicates menu processing to be */
                           /* performed for printing instead of graph */
FILE *fprt;                /* ptr to print file, if flag TRUE */
{
   WINDOW wn;
   int choice;
   double max_stat, min_stat;
   void g_popln_menu( );

   defs_wn(&wn,5,16,15,47,BDR_DLNP);      /* define menu window */
   if (print_flag == FALSE)
      sw_name("Population Graphs", &wn);
   else
      sw_name("Population Printing",&wn);
   sw_plcsr(ON, &wn);

   /* define and set menu on screen for first time */
   g_popln_menu(&wn,print_flag);

   for (;;)
   {
      choice = ki( );      /* get keystroke from user */
      switch ( choice )    /* branch on keystroke */
      {
         case 97:             /* "a" */
         case 65:             /* "Availability" */
                  data_type = 2;
                  goto finish_up;
         case 100:             /* "d" */
         case  68:             /* "Downtime Instances" */
                  data_type = 3;
                  goto finish_up;
         case 99:              /* "c" */
         case 67:              /* "C", downtime cost */
                  data_type = 11;
                  goto finish_up;
         case 112:            /* "p" */
         case  80:            /* "sPares cost" */
                  data_type = 5;
                  goto finish_up;
         case 115:            /* "s" */
         case  83:            /* "Service cost" */
                  data_type = 4;
                  goto finish_up;
         case 116:            /* "t" */
         case  84:            /* "Total cost" */
                  data_type = 1;
                  goto finish_up;
         finish_up:
                  unset_wn(&wn);
```

```
                prep_plot(&max_stat, &min_stat, 0, 0);
                if (print_flag == FALSE)
                   graph_it(max_stat, min_stat);
                else
                {
                   print_msg(ON);
                   print_it(fprt);
                   print_msg(OFF);
                }
                g_popln_menu(&wn,print_flag);
                break;
        case 113:             /* "q" */
        case  81:             /* "Quit" */
                unset_wn(&wn);
                return;
        default:
                bell();
     }
   }
}

void g_popln_menu(wn,print_flag)
WINDOW *wn;
int print_flag;             /* if TRUE, indicates menu processing to be */
                            /* performed for printing instead of graph */
{
   cls();
   set_wn(wn);     /* place window on screen */

   v_plst(1,1,"Current Active Data File: ",wn);
   wn->att = LHIGHLITE;
   v_st(popln_data.data_file, wn);
   v_plst(5,10,"a",wn);
   v_plst(6,10,"d",wn);
   v_plst(7,10,"c",wn);
   v_plst(8,10,"p",wn);
   v_plst(9,10,"s",wn);
   v_plst(10,10,"t",wn);
   v_plst(11,10,"q",wn);
   wn->att = LNORMAL;

   if (print_flag == FALSE)
      v_plst(3,CENTER_TXT,"Population Graph Types",wn);
   else
      v_plst(3,CENTER_TXT,"Population Print Types",wn);
   v_plst(5,12,"- Availability",wn);
   v_plst(6,12,"- Downtime Instances",wn);
   v_plst(7,12,"- Downtime Cost",wn);
   v_plst(8,12,"- Spares Cost",wn);
   v_plst(9,12,"- Service Cost",wn);
   v_plst(10,12,"- Total Cost",wn);
   v_plst(11,12,"- Quit",wn);
   csr_hide();

   return;
}

/*
 * lor_graph( print_flag, fprt )
 *
 * Function: Displays lor stat type menu and processes user's choice
 *           Either prints or graphs, depending on "print_flag"
 * Returns: None
 */
lor_graph(print_flag,fprt)
int print_flag;             /* if TRUE, indicates menu processing to be */
```

```
                        /* performed for printing instead of graph */
FILE *fprt;
{
   WINDOW wn;
   int choice;
   double max_stat, min_stat;
   void g_lor_menu( );

   defs_wn(&wn,5,16,12,47,BDR_DLNP);      /* define menu window */
   if (print_flag == FALSE)
      sw_name("LOR Graphs", &wn);
   else
      sw_name("LOR Printing", &wn);
   sw_plcsr(ON, &wn);
   g_lor_menu(&wn,print_flag);/* define and set menu on screen for first time */

   for (;;)
   {
      choice = ki( );      /* get keystroke from user */
      switch ( choice )    /* branch on keystroke */
      {
         case 117:            /* "u" */
         case  85:            /* "Utilization" */
                  unset_wn(&wn);
                  data_type = 6;
                  prep_plot(&max_stat, &min_stat, 0, 0); /* begin with STD */
                  if (print_flag == FALSE)
                     graph_it(max_stat, min_stat);
                  else
                  {
                     print_msg(ON);
                     print_it(fprt);
                     if (popln_data.aux_flag == TRUE)
                     {
                        prep_plot(&max_stat, &min_stat, 0, 1); /* print aux */
                        print_it(fprt);
                        prep_plot(&max_stat, &min_stat, 0, 2); /* print total */
                        print_it(fprt);
                     }
                     print_msg(OFF);
                  }
                  g_lor_menu(&wn,print_flag);
                  break;
         case 111:            /* "o" */
         case  79:            /* "Other" */
                  data_type = 7;
                  break;
         case 113:            /* "q" */
         case  81:            /* "Q" */
                  unset_wn(&wn);
                  return;
         default:
                  bell( );
      }
   }
}
void g_lor_menu(wn,print_flag)
int print_flag;            /* if TRUE, indicates menu processing to be */
                           /* performed for printing instead of graph */
WINDOW *wn;
{
   cls( );
   set_wn(wn);     /* place window on screen */

   v_plst(1,1,"Current Active Data File: ",wn);
   wn->att = LHIGHLITE;
   v_st(popln_data.data_file, wn);
```

```
   v_plst(5,14,"u",wn);
   v_plst(6,14,"o",wn);
   v_plst(7,14,"q",wn);
   wn->att = LNORMAL;

   if (print_flag == FALSE)
      v_plst(3,CENTER_TXT,"LOR Graph Types",wn);
   else
      v_plst(3,CENTER_TXT,"LOR Print Types",wn);
   v_plst(5,16,"- Utilization",wn);
   v_plst(6,16,"- Other",wn);
   v_plst(7,16,"- Quit",wn);
   csr_hide( );

   return;
}

/*
 * comp_graph( print_flag, fprt )
 *
 * Function: Displays menu of component stat types and processes
 *           user's choice. Either prints or graphs, depending upon
 *           value of "print_flag"
 * Returns: None
 */
comp_graph(print_flag,fprt)
int print_flag;            /* if TRUE, indicates menu processing to be */
                           /* performed for printing instead of graph */
FILE *fprt;
{
   WINDOW wn;
   int choice;
   int comp_id;            /* counter for printing component stats */
   double max_stat, min_stat;
   void g_comp_menu( );

   defs_wn(&wn,5,16,12,47,BDR_DLNP);      /* define menu window */
   if (print_flag == FALSE)
      sw_name("Component Graphs", &wn);
   else
      sw_name("Component Printing", &wn);
   sw_plcsr(ON, &wn);
   g_comp_menu(&wn,print_flag);     /* define and set menu for first time */

   for (;;)
   {
      choice = ki( );      /* get keystroke from user */
      switch ( choice )    /* branch on keystroke */
      {
         case 101:              /* "e" */
         case  69:              /* "Er" */
                  data_type = 9;
                  goto finish_up;
         case 112:              /* "p" */
         case  80:              /* "Pm" */
                  data_type = 8;
                  goto finish_up;
         case 116:              /* "t" */
         case  84:              /* "Total" */
                  data_type = 10;
         finish_up:
                  unset_wn(&wn);
                  prep_plot(&max_stat, &min_stat, 0, 0); /* start with comp 0 */
                  if (print_flag == FALSE)
                     graph_it(max_stat, min_stat);
                  else
                  {
```

```
                    print_msg(ON);
                    print_it(fprt);
                    for (comp_id = 1; comp_id < _num_comp; ++comp_id)
                    {
                       prep_plot(&max_stat,&min_stat,comp_id,0);
                       print_it(fprt);
                    }
                    print_msg(OFF);
                 }
                 g_comp_menu(&wn,print_flag);
                 break;
          case 113:              /* "q" */
          case  81:              /* "Quit" */
                 unset_wn(&wn);
                 return;
          default:
                 bell();
       }
    }
}
void g_comp_menu(wn,print_flag)
int print_flag;              /* if TRUE, indicates menu processing to be */
                             /* performed for printing instead of graph */
WINDOW *wn;
{
   cls();
   set_wn(wn);     /* place window on screen */

   v_plst(1,1,"Current Active Data File: ",wn);
   wn->att = LHIGHLITE;
   v_st(popln_data.data_file, wn);
   v_plst(5,9,"e",wn);
   v_plst(6,9,"p",wn);
   v_plst(7,9,"t",wn);
   v_plst(8,9,"q",wn);
   wn->att = LNORMAL;

   if (print_flag == FALSE)
      v_plst(3,CENTER_TXT,"Component Graph Types",wn);
   else
      v_plst(3,CENTER_TXT,"Component Print Types",wn);
   v_plst(5,11,"- ER Instances",wn);
   v_plst(6,11,"- PM Instances",wn);
   v_plst(7,11,"- Total Repair Instances",wn);
   v_plst(8,11,"- Quit",wn);
   csr_hide();

   return;

}
```

```
/*
 * File Name:  GRAPHDRV.C
 */
#include <stdio.h>
#include <wfd.h>
#include "sm_struc.h"
#include "gr_struc.h"

/*
 * graph_it( double max_stat, double min_stat )
 *
 * Function: Drives the display and user manipulation of statistic
 *           graphs, the actual drawing/labeling routines are elsewhere,
 *           but this routine displays the first graph selected and
 *           then processes user keystrokes to determine future wants
 *
 * Usage: Pass the max and min value across all time intervals of the
 *        statistic being graphed
 *
 * Returns: none
 */

graph_it(max_stat, min_stat)
double max_stat;
double min_stat;
{
   int choice;          /* key pressed by user */
   int cur_intv = 0;    /* current interval at begin of graph */
   int lor = 0;         /* current LOR displayed */
   int comp_id = 0;     /* current component displayed */
   int grid = OFF;      /* status of graph grid lines */
   int prt_status;
   double temp_max;     /* current setting of max scale value */
   double temp_min;     /* current setting of min scale value */
   double new_max;      /* var to accept changed max */
   double new_min;      /* var to accept changed min */


   if (stat_type == 1)
   {
      temp_max = 100.0;
      temp_min = 0.0;
   }
   else
   {
      temp_max = max_stat;
      temp_min = min_stat;
   }
   /*
    * begin the graph at the first interval (cur_intv = 0)
    */
   draw_graph(temp_max, temp_min, cur_intv);     /* produce axes and labels */
   draw_lines(temp_max, temp_min, cur_intv, 1);  /* draw stat lines */
   show_status();

   for (;;)
   {
      choice = ki();    /* get keystroke from user (during graph view) */
      if (choice >= 0)
      {
         switch (choice)
         {
            case 103:      /* "g" */
            case  71:      /* "G", draw grid lines */
                    if ( grid == OFF )
                    {
                       grid = ON;
```

```
                draw_grid(1);
             }
             else
             {
                grid = OFF;
                draw_grid(0);
                draw_lines(temp_max, temp_min, cur_intv, 1);
             }
             break;
      case 115:      /* "s", set scales */
      case  83:      /* "S" */
             /* get new y-axis scales from user */
             /*
              *<cr> or <end> to end number input,
              * <bs> and <del> active
              */
             clear_status();
             grxlab("New Y-Axis Min: ", 1, 3, 24);
             getdoub(8, 2, 65, '#', 1, 2, &new_min, 1, 0.0,
                     99999999.99);
             clear_status();
             grxlab("New Y-Axis Max: ", 1, 3, 24);
             getdoub(8, 2, 65, '#', 1, 2, &new_max, 1, 0.0,
                     99999999.99);
             clear_status();
             if ( (new_max < max_stat) || (new_max <= 0.0) ||
                  (new_min > min_stat) )
             {
                gr_errmsg(1);       /* graph would be off screen */
             }
             else
             {
                temp_max = new_max;
                temp_min = new_min;
                draw_graph(temp_max, temp_min, cur_intv);
                draw_lines(temp_max, temp_min, cur_intv, 1);
                if (grid == ON)
                   draw_grid(1);
             }
             show_status();
             break;
      case 35:                    /* "#", print graphics screen */
             clear_status();
             grxlab("File: ", 1, 3, 24);
             grxlab(popln_data.data_file, 1, 9, 24);
             /* essn util function for PrtSc */
             if ( (prt_status = prtsc()) != 0)
                gr_errmsg(99);
             clear_status();
             show_status();
             break;
      case 113:                   /* "q", quit viewing graph */
      case  81:                   /* "Q" */
      case  27:                   /* "<esc>" */
             initgraf(0,0,0);
             return;
      default:
             bell();
   }
}
else
{
   choice = (-choice);
   switch ( choice )
   {
      case 77:       /* cursor right, next group of intervals */
      case 75:       /* cursor left */
```

```
            if (choice == 77)
            {
               if ( (cur_intv + 12) >= _num_intv)
               {
                  bell();              /* no more interval groups */
                  goto too_far;
               }
               draw_lines(temp_max, temp_min, cur_intv, 0);
               cur_intv += 6;
            }
            else
            if (choice == 75)
            {
               if (cur_intv == 0)
               {
                  bell();
                  goto too_far;
               }
               draw_lines(temp_max, temp_min, cur_intv, 0);
               cur_intv -= 6;
            }
            /* relabel x-axis with new cur_intv */
            label_x(cur_intv);
            draw_lines(temp_max, temp_min, cur_intv, 1);
            if (grid == ON)
               draw_grid(1);

         too_far:
            break;

   case 72:       /* cursor up, next component */
   case 80:       /* cursor down, prev. component */
            switch (data_type)
            {
              case 8:            /* component graph cases */
              case 9:
              case 10:
                     if (choice == 72)    /* next comp */
                     {
                        if ( (comp_id + 1) == _num_comp )
                           comp_id = 0;
                        else
                           ++comp_id;
                     }
                     else
                     if (choice == 80)    /* prev comp */
                     {
                        if ( comp_id == 0 )
                           comp_id = _num_comp - 1;
                        else
                           --comp_id;
                     }
                     prep_plot(&max_stat,&min_stat,comp_id,0);
                     temp_max = max_stat;
                     temp_min = min_stat;
                     draw_graph(temp_max,temp_min,cur_intv);
                     draw_lines(temp_max,temp_min,cur_intv,1);
                     if (grid == ON)
                        draw_grid(1);
                     show_status();
                     break;
              default:
                     bell();
            }
            break;

   case 73:            /* PgUp, switch STD/AUX/TOT or ER/PM/TOT */
```

```
case 81:            /* PgDn */
        switch ( data_type )
        {
          case 6:        /* switch to next/prev LOR graph type */
          case 7:
                 if ( popln_data.aux_flag != 0 &&
                      lor_data[AUX_LOR].capacity != 0 )
                 {
                    if (choice == 81)  /* PgDn */
                    {
                       if (lor == 0)
                          lor = 2;
                       else
                          --lor;
                    }
                    else               /* PgUp */
                    if (choice == 73)
                    {
                    if (lor == 2)
                       lor = 0;
                    else
                       ++lor;
                    }
                    prep_plot(&max_stat, &min_stat, 0,lor);
                    temp_max = 100.0;
                    temp_min = 0.0;
                    draw_graph(temp_max, temp_min, cur_intv);
                    draw_lines(temp_max, temp_min, cur_intv, 1);
                    if (grid == ON)
                       draw_grid(1);
                    show_status();
                 }
                 else
                    bell();
                 break;
          case 8:
          case 9:      /* switch to next/prev comp graph type */
          case 10:
                 if (choice == 81)    /* PgDn */
                 {
                    if (data_type == 10)
                       data_type = 9;
                    else
                    if (data_type == 9)
                       data_type = 8;
                    else
                       data_type = 10;
                 }
                 else                 /* PgUp */
                 if (choice == 73)
                 {
                    if (data_type == 8)
                       data_type = 9;
                    else
                    if (data_type == 9)
                       data_type = 10;
                    else
                       data_type = 8;
                 }
                 prep_plot(&max_stat,&min_stat, comp_id,0);
                 temp_max = max_stat;
                 temp_min = min_stat;
                 draw_graph(temp_max,temp_min,cur_intv,1);
                 draw_lines(temp_max,temp_min,cur_intv,1);
                 if (grid == ON)
                    draw_grid(1);
                 show_status();
```

```
                        break;
                    default:
                        bell();
                }
                break;
            default:
                bell();
        }
    }
}

/*
 * Function: Clears status line on graph
 */
clear_status()
{
    register int i;
    for (i = 3; i < 75; i += 3)    /* clear graph status line */
        grxlab("   ", 1, i, 24);
    return;
}

/*
 * Function: Displays status line on graph (command options)
 */
show_status()
{
    grxlab("Options:", 1, 3, 24);
    grxlab("g-Grid", 1, 13, 24);
    grxlab("s-Scale", 1, 21, 24);
    grxlab("#-Print", 1, 59, 24);
    grxlab("q-Quit", 1, 68, 24);
    return;
}

/*
 * processes and displays DOS printer error messages to
 * user, line 24 of the graphics print display
 */
gr_errmsg(err_type)
int err_type;
{
    char err_msg[71];
    char *strcpy(), *strcat();

    clear_status();
    switch (err_type)
    {
        case 1:  strcpy( err_msg, "Scale value out of range, ");
                 break;
        case 99: strcpy( err_msg, "Check printer and retry request, ");
                 break;
        default: strcpy( err_msg, "Program passed bad 'err_type', ");
    }
    strcat( err_msg, "press any key to continue.");
    grxlab( err_msg, 1, 3, 24);
    bell();
    ki();
    clear_status();
    return;
}
```

```
/*
 * File Name:  PREPARE.C
 */
#include <stdio.h>
#include "sm_struc.h"
#include "gr_struc.h"

/*
 * prep_plot(max_stat, min_stat, comp_id, lor)
 *
 * Function: Prepares the selected statistic for plotting. Computes
 *           the statistic as necessary, from the external vars
 *           "gr_popln" and "gr_comp" and determines the max/min
 *           value along the entire interval. Copies the new statistic
 *           into the "plot_stat" array and returns ("plot_stat" was/is
 *           allocated during the file retrieve).  Sets titles for
 *           graphic display in global "gr_title[2]"; and global vars
 *           "gr_type" and "stat_type".
 *
 * Usage: References the global var "data_type" which is set during
 *        user selection from menu.
 *
 * Returns: modifies the max/min ptrs passed to contain the max/min
 *          values for the new statistic set. Also sets titles in
 *          char array ptr passed.
 */
prep_plot( max_stat, min_stat, comp_id, lor )
double *max_stat;         /* ptr to contain modified max value */
double *min_stat;         /* ptr to contain modified min value */
int comp_id;          /* for which component, if applicable */
int lor;              /* for which LOR, if applicable */
{
   register int i;
   register int j;
   double avail;        /* total available popln uptime for an interval */
   char *strcpy( );
   char *strcat( );

   switch (data_type)
   {
      case 1:              /* popln, total cost (mean) */
                           /* (Downtime + Service + Travel + Spares) */
               gr_type = 1;
               stat_type = 2;
               strcpy(gr_title,"Program Cost, Total Population");

               for (i = 0; i < _num_intv; ++i)
               {
                  (plot_stat + i)->mean = (popln_data.c_eidt *
                     (gr_popln + i)->dt->mean) + (lor_data[0].c_usage *
                     (gr_popln + i)->std_serv->mean) + (lor_data[1].c_usage *
                     (gr_popln + i)->aux_serv->mean) + (lor_data[0].c_trans *
                     (gr_popln + i)->std_dt->mean * 2) + (lor_data[1].c_trans *
                     (gr_popln + i)->aux_dt->mean * 2);
                  for (j = 0; j < _num_comp; ++j)
                  {
                     (plot_stat + i)->mean += ((gr_comp[j]+i)->std_er->mean +
                                               (gr_comp[j]+i)->aux_er->mean) *
                                               comp_data[j]->c_erspare +
                                               ((gr_comp[j]+i)->std_pm->mean +
                                               (gr_comp[j]+i)->aux_pm->mean) *
                                               comp_data[j]->c_pmspare;
                  }
                  set_limits(i, max_stat, min_stat);
               }
               break;
      case 2:              /* popln, availability (mean,stddev) */
```

```
            gr_type = 2;
            stat_type = 1;
            strcpy(gr_title,"Availability, Total Population");

            for (i = 0; i < _num_intv; ++i)
            {
               avail = popln_data.intv_size * (gr_popln+i)->avg_ei;
               (plot_stat+i)->mean = 100. - (100. *
                                     (gr_popln+i)->dt->mean / avail);

               (plot_stat+i)->stddev = 0.0;
               set_limits(i, max_stat, min_stat);
            }
            break;
case 3:             /* popln, downtime instances (mean,stddev) */
            gr_type = 2;
            stat_type = 0;
            strcpy(gr_title,"Downtime Instances, Total Population");
            for (i = 0; i < _num_intv; ++i)
            {
               (plot_stat+i)->mean = (gr_popln+i)->std_dt->mean +
                                     (gr_popln+i)->aux_dt->mean;
               (plot_stat+i)->stddev = 0.0;
               set_limits(i,max_stat,min_stat);
            }
            break;
case 4:             /* popln, repair costs (mean,stddev) */
            gr_type = 2;
            stat_type = 2;
            strcpy(gr_title,"Service Costs, Total Population");
            for (i = 0; i < _num_intv; ++i)
            {
               (plot_stat+i)->mean = ((gr_popln+i)->std_serv->mean *
                                     lor_data[0].c_usage) +
                                     ((gr_popln+i)->aux_serv->mean *
                                     lor_data[1].c_usage);
               (plot_stat+i)->stddev = 0.0;
               set_limits(i,max_stat,min_stat);
            }

            break;
case 5:             /* popln, spares costs (mean,stddev) */
            gr_type = 2;
            stat_type = 2;
            strcpy(gr_title,"Spares Costs, Total Population");
            for (i = 0; i < _num_intv; ++i)
            {
               (plot_stat+i)->mean = 0.0;
               for (j = 0; j < _num_comp; ++j)
               {
                  (plot_stat+i)->mean += ((gr_comp[j]+i)->std_er->mean +
                                         (gr_comp[j]+i)->aux_er->mean) *
                                         comp_data[j]->c_erspare +
                                         ((gr_comp[j]+i)->std_pm->mean +
                                         (gr_comp[j]+i)->aux_pm->mean) *
                                         comp_data[j]->c_pmspare;
               }
               (plot_stat+i)->stddev = 0.0;
               set_limits(i,max_stat,min_stat);
            }
            break;
case 6:             /* lor, server utilization (lor = 2 --> both) */
                    /* (mean,stddev) */
            gr_type = 2;
            stat_type = 1;
            strcpy(gr_title,"Server Utilization - ");
            if (lor == 0)
```

```
            {
               strcat(gr_title,lor_data[0].name);
               for (i = 0; i < _num_intv; ++i)
               {
                  (plot_stat+i)->mean = 100.0 *
                                        (gr_popln+i)->std_serv->mean /
                                        (lor_data[0].capacity *
                                        popln_data.intv_size);
                  (plot_stat+i)->stddev = 0.0;
                  set_limits(i,max_stat,min_stat);
               }
            }
            else
            if (lor == 1)
            {
               strcat(gr_title,lor_data[1].name);
               for (i = 0; i < _num_intv; ++i)
               {
                  (plot_stat+i)->mean = 100.0 *
                                        (gr_popln+i)->aux_serv->mean /
                                        (lor_data[1].capacity *
                                        popln_data.intv_size);
                  (plot_stat+i)->stddev = 0.0;
                  set_limits(i,max_stat,min_stat);
               }
            }
            else
            if (lor == 2)
            {
               strcat(gr_title,"Both LORs");
               for (i = 0; i < _num_intv; ++i)
               {
                  (plot_stat+i)->mean = 100.0 *
                                        ((gr_popln+i)->std_serv->mean +
                                        (gr_popln+i)->aux_serv->mean) /
                                        ((lor_data[0].capacity +
                                        lor_data[1].capacity) *
                                        popln_data.intv_size);
                  (plot_stat+i)->stddev = 0.0;
                  set_limits(i,max_stat,min_stat);
               }
            }
            break;

      case 7:  break;      /* lor, whatever (lor = 2 --> both) */

      case 8:              /* component pm trips (both LOR, mean, stddev */
            gr_type = 2;
            stat_type = 0;
            strcpy(gr_title,"Instances of PM - ");
            strcat(gr_title, comp_data[comp_id]->name);
            prep_comp(max_stat, min_stat, comp_id);
            break;
      case 9:             /* component er trips (both LOR, mean, stddev) */
            gr_type = 2;
            stat_type = 0;
            strcpy(gr_title,"Instances of ER - ");
            strcat(gr_title, comp_data[comp_id]->name);
            prep_comp(max_stat, min_stat, comp_id);
            break;
      case 10:             /* component total trips (both LOR, mean, stddev) */
            gr_type = 2;
            stat_type = 0;
            strcpy(gr_title,"Instances of PM/ER - ");
            strcat(gr_title, comp_data[comp_id]->name);
            prep_comp(max_stat, min_stat, comp_id);
            break;
```

```
      case 11:                                  /* popln downtime cost */
               gr_type = 2;
               stat_type = 2;
               strcpy(gr_title,"Downtime Cost, Total Population");
               for (i = 0; i < _num_intv; ++i)
               {
                  (plot_stat+i)->mean = (gr_popln+i)->dt->mean *
                                        popln_data.c_eidt;
                  (plot_stat+i)->stddev = 0.0;
                  set_limits(i,max_stat,min_stat);
               }
               return;
      default:
               bell();
   }
   return;
}

/*
 * set_limits(int i, double *max_stat, double *min_stat)
 *
 * Functions: Passed a ptr to "max" and "min" vars, these vars are
 *            are compared and adjusted based on value of
 *            "(plot_stat + i)->mean"
 */
set_limits(i, max_stat, min_stat)
int i;
double *max_stat;
double *min_stat;
{
   if (i == 0)
      *max_stat = *min_stat = plot_stat->mean;
   else
   if (*max_stat < (plot_stat+i)->mean)
      *max_stat = (plot_stat+i)->mean;
   else
   if (*min_stat > (plot_stat+i)->mean)
      *min_stat = (plot_stat+i)->mean;
   return;
}

/*
 * Function: Prepares the internal plotting data structure for
 *           component statistic selected
 */
prep_comp(max_stat, min_stat, comp_id)
double *max_stat;
double *min_stat;
int comp_id;
{
   register int i;

   switch (data_type)
   {
      case 8:                                   /* comp pm instances */
               for (i = 0; i < _num_intv; ++i)
               {
                  (plot_stat+i)->mean = (gr_comp[comp_id]+i)->std_pm->mean +
                                        (gr_comp[comp_id]+i)->aux_pm->mean;
                  (plot_stat+i)->stddev = 0.0;
                  set_limits(i,max_stat,min_stat);
               }
               return;
      case 9:                                   /* comp er instances */
               for (i = 0; i < _num_intv; ++i)
               {
                  (plot_stat+i)->mean = (gr_comp[comp_id]+i)->std_er->mean +
```

```
                                    (gr_comp[comp_id]+i)->aux_er->mean;
                  (plot_stat+i)->stddev = 0.0;
                  set_limits(i,max_stat,min_stat);
               }
               return;
      case 10:                                  /* tot comp maint instances */
               for (i = 0; i < _num_intv; ++i)
               {
                  (plot_stat+i)->mean = (gr_comp[comp_id]+i)->std_pm->mean +
                                        (gr_comp[comp_id]+i)->aux_pm->mean +
                                        (gr_comp[comp_id]+i)->std_er->mean +
                                        (gr_comp[comp_id]+i)->aux_er->mean;
                  (plot_stat+i)->stddev = 0.0;
                  set_limits(i,max_stat,min_stat);
               }
               return;
      default:
               bell();
               return;
   }
}
```

```
/*
 * File Name:  GRAPH.C
 */
#include <stdio.h>
#include "sm_struc.h"
#include "gr_struc.h"

/*
 * draw_lines( high_stat, low_stat, low_intv, color )
 *
 * Function: References the extern "plot_stat" structure and draws the
 *           set of lines selected by extern var "gr_type" and in
 *           the color specified by "color".
 *
 * Usage:  gr_type     action             color     att
 *         -------  ------------          -----    -----
 *            1        mean                 0      black
 *            2     mean,stddev             1      white
 *            3     mean,max,min
 *
 * Returns: 0, to indicate bad "type"
 *          1, otherwise
 */

draw_lines( high_stat, low_stat, low_intv, color )

double high_stat; /* highest label on y-axis */
double low_stat;  /* lowest label on y-axis */
int low_intv;     /* lowest interval label on x-axis */
int color;        /* used to reverse color and erase previous line */
{
   double pix_per_stat;

   pix_per_stat = 160.0 / (high_stat - low_stat);

   switch ( gr_type )
   {
      case 1:
               plot_mean(low_intv, low_stat, pix_per_stat, color);
               return(1);
      case 2:
               plot_mean(low_intv, low_stat, pix_per_stat, color);
               plot_stddev(low_intv, low_stat, pix_per_stat, color);
               return(1);
      case 3:
               plot_mean(low_intv, low_stat, pix_per_stat, color);
               plot_stddev(low_intv, low_stat, pix_per_stat, color);
               return(1);
      default:
               return(0);
   }
}

/*
 * plot_mean(low_intv, low_stat, pix_per_stat, color)
 *
 * Function: Plots the "mean" statistic graphics line.
 *
 * Usage: Pass the low x,y values; and the pix_per_stat =
 *        (tot_y_pix)/(high_stat - low_stat) with tot_y_pix = 10 * 16 = 160
 *
 */

plot_mean(low_intv, low_stat, pix_per_stat, color)
int low_intv;              /* interval that starts x-axis */
double low_stat;           /* y value that begins y-axis */
double pix_per_stat;    /* equals (176 / (high_stat - low_stat)) */
```

```
int color;
{
   register int i;
   int high_intv;
   int new_mean;
   int old_mean;
   int old_intv;

   if ( (low_intv + 12) >= _num_intv)
      high_intv = _num_intv;           /* set upper limit of x-axis plotting */
   else
      high_intv = low_intv + 12;

   old_mean = 171 - (int)(((plot_stat+low_intv)->mean - low_stat) *
                          pix_per_stat);

   for (i = low_intv + 1, old_intv = 135;
        i < high_intv;
        ++i, old_intv += 40)
   {
      new_mean = 171 - (int)(((plot_stat+i)->mean - low_stat) * pix_per_stat);
      grline(old_intv, old_mean, (old_intv + 40), new_mean, color);
      old_mean = new_mean;
   }
   return;
}

plot_maxmin(low_intv,low_stat,pix_per_stat,color)
int low_intv;
double low_stat;
double pix_per_stat;
int color;
{
   register int i;
   int high_intv;
   int new_max;
   int new_min;
   int old_intv;
   int old_max;
   int old_min;

   if ( (low_intv + 12) >= _num_intv)
      high_intv = _num_intv;    /* set upper limit of x-axis plotting */
   else
      high_intv = low_intv + 12;

   old_max = 171 - (int)(((plot_stat + low_intv)->max_stat - low_stat) *
             pix_per_stat);
   old_min = 171 - (int)(((plot_stat + low_intv)->min_stat - low_stat) *
             pix_per_stat);

   for (i = low_intv + 1, old_intv = 135;
        i < high_intv;
        ++i, old_intv += 40)
   {
      new_max = 171 - (int)(((plot_stat + i)->max_stat - low_stat) *
                pix_per_stat);
      new_min = 171 - (int)(((plot_stat + i)->min_stat - low_stat) *
                pix_per_stat);

      grdash(old_intv, old_max, (old_intv + 40), new_max, color);
      grdash(old_intv, old_min, (old_intv + 40), new_min, color);
      old_max = new_max;
      old_min = new_min;
   }
   return;
}
```

```
plot_stddev(low_intv,low_stat,pix_per_stat,color)
int low_intv;
double low_stat;
double pix_per_stat;
int color;
{
   register int i;
   int high_intv;
   int new_stdl;
   int new_stdh;
   int old_stdl;
   int old_stdh;
   int old_intv;

   if ( (low_intv + 12) >= _num_intv)
      high_intv = _num_intv;    /* set upper limit of x-axis plotting */
   else
      high_intv = low_intv + 12;

   old_stdl = 171 - (int)(((plot_stat + low_intv)->mean - low_stat) *
              pix_per_stat) + (int)((plot_stat + low_intv)->stddev *
              pix_per_stat);

   if (old_stdl > 171)     /* check for stddev plot below y-axis */
      old_stdl = 171;

   old_stdh = 171 - (int)(((plot_stat + low_intv)->mean - low_stat) *
              pix_per_stat) - (int)((plot_stat + low_intv)->stddev *
              pix_per_stat);

   for (i = low_intv + 1, old_intv = 135;
        i < high_intv;
        ++i, old_intv += 40)
   {
      new_stdl = 171 - (int)(((plot_stat + i)->mean - low_stat) *
                 pix_per_stat) + (int)((plot_stat + i)->stddev * pix_per_stat);

      if (new_stdl > 171)     /* check for stddev plot below y-axis */
         new_stdl = 171;

      new_stdh = 171 - (int)(((plot_stat + i)->mean - low_stat) *
                 pix_per_stat) - (int)((plot_stat + i)->stddev * pix_per_stat);

      grdotb(old_intv, old_stdl, (old_intv + 40), new_stdl, color);
      grdotb(old_intv, old_stdh, (old_intv + 40), new_stdh, color);

      old_stdl = new_stdl;
      old_stdh = new_stdh;
   }
   return;
}

/*
 * draw_graph( high_stat, low_stat, low_intv )
 *
 * Function: Sets graphics mode, draws and labels graph axes based
 *           on passed parameters.  Sets any titles on screen.
 *
 * Usage: Should have previously computed the plotting statistic and
 *        placed it in the external array "plot_stat" (although the points
 *        are not plotted in this routine) and determined the
 *        max/min y-axis value across all intervals. Must keep track of
 *        the beginning interval to display (that and the next 11 are
 *        displayed).
 *        Settings for type: stat_type     y-axis action       format
 *                           ---------    ---------------  ---------------
```

```
 *                              1         counting stat      (xxxxxx)
 *                              2          dollar stat     (xxxxxxx.xx)
 *                              3        percentage stat  (0.00 - 100.00)
 * Returns: none
 */

draw_graph(high_stat, low_stat, low_intv)

double high_stat;          /* max value for y-axis */
double low_stat;           /* min value for y-axis */
int low_intv;              /* lowest interval value for x-axis */
{
   register int i;      /* loop counter */
   register int j;      /* loop counter */
   char intv_label[30];
   char num_units[5];
   char stat_label[10];
   char *strcpy( );
   char *strcat( );
   double next_label;   /* current label for printing on stat axis */
   double step;         /* size of stats axis steps (high-low)/10 */
   char stat_pnt[11][10];  /* array to hold y-axis label strings */

   initgraf(2,0,0);                        /* init high-res graphics mode */

   grline(135,171,576,171,1);              /* draw x-axis */
   for (i = 135; i < 615; i += 40)         /* x-axis hash marks */
      grline(i, 171, i, 173, 1);

   grline(135,171,135,11,1);                /* draw y-axis */
   grline(576,171,576,11,1);                /* echo y-axis to right side */
   for (i = 171; i > 0; i -= 16)            /* draw y-axis hash marks */
      grline(135, i, 131, i, 1);
   grxlab(gr_title, 1, 27, 0);             /* write graph title */

   /*
    * assemble interval title line
    */
   strcpy( intv_label, "Period (" );
   if (popln_data.intv_size < 10.0)
      sprintf( num_units, "%1d ", (int) popln_data.intv_size );
   else
   if (popln_data.intv_size < 100.0)
      sprintf( num_units, "%2d ", (int) popln_data.intv_size );
   else
   if (popln_data.intv_size < 1000.0)
      sprintf( num_units, "%3d ", (int) popln_data.intv_size );

   strcat( intv_label, num_units);

   switch ( popln_data.u_sim )
   {
      case 'h':
      case 'H':
               strcat( intv_label, "hours ");
               break;
      case 'd':
      case 'D':
               strcat( intv_label, "days ");
               break;
      case 'w':
      case 'W':
               strcat( intv_label, "weeks ");
               break;
      case 'm':
      case 'M':
               strcat( intv_label, "months ");
```

```
            break;
      case 'y':
      case 'Y':
            strcat( intv_label, "years ");
   }
   strcat( intv_label, "each)");
   grxlab(intv_label, 1, 32, 23);          /* label x-axis */
   label_x( low_intv );

   /*
    * use sprintf( ) to produce x and y axis labels
    * before the following print to screen, format for y
    * based on type var passed (0 => count, 1 => percentage,
    * 2 => dollars)
    */
   step = (high_stat - low_stat) / 10;
   switch ( stat_type )
   {
      case 0:                                    /* quantity format */
            for (i = 0, next_label = low_stat;
                 i < 10;
                ++i, next_label += step)
            {
                 sprintf(stat_pnt[i], "%9.1f", next_label);
            }
            sprintf( stat_pnt[10], "%9.1f", high_stat);
            strcpy(stat_label,"Frequency");
            break;
      case 1:                                    /* percentage format */
            for (i = 0, next_label = low_stat;
                 i < 10;
                ++i, next_label += step)
            {
               sprintf(stat_pnt[i], "%9.1f", next_label);
            }
            sprintf( stat_pnt[10], "%9.1f", high_stat);
            strcpy(stat_label,"Percent");
            break;
      case 2:                                    /* dollar format */
            for (i = 0, next_label = low_stat;
                 i < 10;
                ++i, next_label += step)
            {
               sprintf(stat_pnt[i], "%9.0f", next_label);
            }
            sprintf( stat_pnt[10], "%9.0f", high_stat);
            strcpy(stat_label,"Dollars");
            break;
   }

   grylab(stat_label, 1, 4, 8);            /* label y-axis */
   for (i = 0, j = 21; i < 11; ++i, j -= 2)
      grxlab(stat_pnt[i], 1, 7, j);        /* label y-axis points */

   return;
}

/*
 * label_x( cur_intv )
 *
 * Function: Labels x-axis of graph beginning with cur_intv
 *
 * Returns: None
 */
label_x( cur_intv )
int cur_intv;
{
```

```
   register int i;          /* label array index */
   register int j;          /* interval loop var */
   int high_intv;
   char intv_pnt[12][3];    /* array to hold x-axis label strings */

   /*
    * redraw x and y axes (in case line erase
    * before call removes axis line)
    */
   grline(135,171,576,171,1);
   grline(135,171,135,11,1);

   if ( (cur_intv + 12) >= _num_intv)
      high_intv = _num_intv;           /* set upper limit of x-axis plotting */
   else
      high_intv = cur_intv + 12;

   for (i = 0, j = cur_intv; j < high_intv; ++i, ++j)
      sprintf(intv_pnt[i], "%2d", j + 1);          /* load interval labels */
   for ( ; i < 12; ++i)
      strcpy(intv_pnt[i],"  ");                   /* set remaining to blank */

   for (i = 0, j = 16; i < 12; ++i, j += 5)
      grxlab(intv_pnt[i], 1, j, 22);         /* label x-axis points */

   return;
}

/*
 * draw_grid( color )
 *
 * Function: Draws grid lines on displayed graph. Color is used to
 *           remove previously drawn grid, i.e. color = 1 draws
 *           grid in white, color = 0 draws grid in black.\
 */
draw_grid(color)
int color;
{
   register int i;

   for (i = 155; i > 0; i -= 16)          /* draw y-axis grid lines */
      grline(136, i, 575, i, color);
   return;
}
```

```
/*
 * File Name:  PRINTDRV.C
 */
#include <stdio.h>
#include <wfd.h>
#include "sm_struc.h"
#include "gr_struc.h"

/*
 * print_it( fprt )
 *
 * Function: Drives the printing of user requested data. Uses the
 *           plot_stat structure loaded using prep_plot()
 *           to print one set of interval data to the output device.
 *           Titles the printout with title in gr_title.
 * Usage: Pass ptr to output file/printer.
 * Returns: none
 */
print_it(fprt)
FILE *fprt;
{
   register int i;

   fprintf(fprt,"%s\n", gr_title);
   fprintf(fprt,"Interval        Mean\n");
   for (i = 0; i < _num_intv; ++i)
      fprintf(fprt,"%5d      %10.2f\n",
               i + 1, (plot_stat+i)->mean);
   fprintf(fprt,"\n");
   fflush(fprt);
   return;
}

/*
 * prt_raw(fprt)
 *
 * Function: Prints the raw internal associated with the population
 *           to the output device specified by "fprt". Does not print
 *           raw component data, that can be obtained in part by
 *           printing the "generated" stats by component
 * Returns: none
 */
prt_raw(fprt)
FILE *fprt;
{
   register int i;
   register int j;
   char *strcpy();

   print_msg(ON);
   for (i = 0; i < 5; ++i)
   {
      for (j = 0; j < _num_intv; ++j)
      {
         switch ( i )
         {
            case 0:  strcpy(gr_title,"Total Downtime");
                     fprintf(fprt,"%s\n",gr_title);
                     fprintf(fprt,"Interval     Mean         Max         ");
                     fprintf(fprt,"Min      Std.Dev.\n");
                     for (j = 0; j < _num_intv; ++j)
                     {
                        fprintf(fprt,"%5d   %10.2f  %10.2f  %10.2f  %9.3f\n",
                                j+1, (gr_popln+j)->dt->mean,
                                (gr_popln+j)->dt->max_stat,
                                (gr_popln+j)->dt->min_stat,
                                (gr_popln+j)->dt->stddev);
```

```
                }
                break;
        case 1: strcpy(gr_title,"Service Time at Standard LOR");
                fprintf(fprt,"%s\n",gr_title);
                fprintf(fprt,"Interval      Mean\n");
                for (j = 0; j < _num_intv; ++j)
                {
                   fprintf(fprt,"%5d   %10.2f\n",
                           j+1, (gr_popln+j)->std_serv->mean);
                }
                break;
        case 2: if (popln_data.aux_flag == TRUE)
                {
                   strcpy(gr_title,"Service Time at Auxiliary LOR");
                   fprintf(fprt,"%s\n",gr_title);
                   fprintf(fprt,"Interval      Mean\n");
                   for (j = 0; j < _num_intv; ++j)
                   {
                      fprintf(fprt,"%5d   %10.2f\n",
                              j+1, (gr_popln+j)->aux_serv->mean);
                   }
                }
                break;
        case 3: strcpy(gr_title,"Trips to Standard LOR");
                fprintf(fprt,"%s\n",gr_title);
                fprintf(fprt,"Interval     Mean         Max         ");
                fprintf(fprt,"Min      Std.Dev.\n");
                for (j = 0; j < _num_intv; ++j)
                {
                   fprintf(fprt,"%5d   %10.2f  %10.2f  %10.2f  %9.3f\n",
                           j+1, (gr_popln+j)->std_dt->mean,
                           (gr_popln+j)->std_dt->max_stat,
                           (gr_popln+j)->std_dt->min_stat,
                           (gr_popln+j)->std_dt->stddev);
                }
                break;
        case 4: if (popln_data.aux_flag == TRUE)
                {
                   strcpy(gr_title,"Trips to Auxiliary LOR");
                   fprintf(fprt,"%s\n",gr_title);
                   fprintf(fprt,"Interval     Mean         Max         ");
                   fprintf(fprt,"Min      Std.Dev.\n");
                   for (j = 0; j < _num_intv; ++j)
                   {
                      fprintf(fprt,"%5d   %10.2f  %10.2f  %10.2f  %9.3f\n",
                              j+1, (gr_popln+j)->aux_dt->mean,
                              (gr_popln+j)->aux_dt->max_stat,
                              (gr_popln+j)->aux_dt->min_stat,
                              (gr_popln+j)->aux_dt->stddev);
                   }
                }
                break;
      }
    }
    fprintf(fprt,"\n");
  }
  fprintf(fprt,"Average Number End Items in Population\n");
  fprintf(fprt,"Interval      Avg. End Items\n");
  for (j = 0; j < _num_intv; ++j)
     fprintf(fprt,"%5d        %10.2f\n", j+1, (gr_popln+j)->avg_ei);

  fprintf(fprt,"\f");     /* form feed */
  fflush(fprt);
  print_msg(OFF);
  return;
}
```

```
/*
 * prt_input(fprt)
 *
 * Function: Used to echo the original input data from a stats
 *           output file
 * Returns: None
 */
prt_input( fprt )
FILE *fprt;
{
   EMP_DATA *emp_ptr;
   MOF_SCHED *mof_ptr;
   char *strcpy( );
   register int i;

   print_msg(ON);
   fprintf(fprt,"Population Name: %s\n",popln_data.name);
   fprintf(fprt,"MOF Type: %c\n",popln_data.mof);
   fprintf(fprt,"MOF Initial Block: %3d\n",popln_data.mof_init);

   /*
    * if MOF is specified as "schedule" (popln_data.mof), write the
    * schedule into file.
    *   'a' or other => initial block only
    *   's' or 'S' => schedule
    */

   if (popln_data.mof == 's' || popln_data.mof == 'S')
   {
      fprintf(fprt,"MOF Schedule Specified\n");
      fprintf(fprt," Time      Number Acquired\n");
      for (mof_ptr = popln_data.sched; mof_ptr; mof_ptr = mof_ptr->next)
         fprintf(fprt,"%6d       %3d\n", mof_ptr->s_time, mof_ptr->s_block);
   }
   fprintf(fprt,"Cost of Downtime: %10.2f\n",popln_data.c_eidt);
   fprintf(fprt," per: %c\n",popln_data.u_eidt);
   fprintf(fprt,"Simulation Time Unit: %c\n",popln_data.u_sim);
   fprintf(fprt,"Simulation Time Length: %10.2f\n",popln_data.sim_lgth);
   fprintf(fprt,"Number of Simulations to Run: %3d\n",popln_data.sim_num);
   fprintf(fprt,"Statistics Collection Interval: %9.2f\n",popln_data.intv_size);
   fprintf(fprt,"OPM Flag: %1d\n",popln_data.opm_flag);
   fprintf(fprt,"OPM Look-Ahead Time: %9.2f\n",popln_data.opm_look);
   fprintf(fprt,"Auxiliary LOR Flag: %1d\n",popln_data.aux_flag);
   fprintf(fprt,"\n");

   for (i=0; i < 2; ++i)
   {
      fprintf(fprt,"LOR Name: %s\n",lor_data[i].name);
      fprintf(fprt,"LOR Capacity: %3d\n",lor_data[i].capacity);
      fprintf(fprt,"Cost of Usage: %10.2f\n",lor_data[i].c_usage);
      fprintf(fprt," per: %c\n",lor_data[i].u_usage);
      fprintf(fprt,"Time to Transport: %10.2f\n",lor_data[i].t_trans);
      fprintf(fprt,"Cost to Transport: %10.2f\n",lor_data[i].c_trans);
      fprintf(fprt,"Gang Repair Flag: %1d\n",lor_data[i].gang_flag);
      fprintf(fprt,"\n");
   }

   for (i=0; i < _num_comp; ++i)
   {
      fprintf(fprt,"Component Name: %s\n",comp_data[i]->name);
      fprintf(fprt,"Failure Distribution: %c\n",comp_data[i]->fail_dist);
      fprintf(fprt,"Unit of Time: %c\n",comp_data[i]->u_fail);
      switch ( comp_data[i]->fail_dist )
      {
         case 'n':                        /* normal */
         case 'w':                        /* weibull */
```

```
         case 'g':                          /* gamma */
         case 'u':                          /* uniform */
         case 'r':                          /* erlang */
         case 'l':                          /* lognormal */
                  fprintf(fprt,"Fail Parm 1: %10.2f\n",comp_data[i]->parm1);
                  fprintf(fprt,"Fail Parm 2: %10.2f\n",comp_data[i]->parm2);
                  break;

         case 'e':                          /* exponential */
                  fprintf(fprt,"Fail Parm 1: %10.2f\n",comp_data[i]->parm1);
                  break;

         case 'm':                          /* empirical */
                  fprintf(fprt,"Empirical Distribution Data\n");
                  fprintf(fprt," Probability   Response \n");
                  for (emp_ptr = comp_data[i]->emp_head; emp_ptr;
                       emp_ptr = emp_ptr->next_intv)
                     fprintf(fprt,"    %6.4f    %10.2f\n",
                           emp_ptr->probab, emp_ptr->response);
                  break;

         default :
                  break;
      }
      fprintf(fprt,"Cost of PM Spare: %10.2f\n",comp_data[i]->c_pmspare);
      fprintf(fprt,"Cost of ER Spare: %10.2f\n",comp_data[i]->c_erspare);
      fprintf(fprt,"Time Unit for Durations: %c\n",comp_data[i]->u_durn);
      fprintf(fprt,"Repair Time for ER: %10.2f\n",comp_data[i]->er_durn);
      fprintf(fprt,"PM Policy Flag: %c\n",comp_data[i]->pm_polcy);
      fprintf(fprt,"Repair Time for PM: %10.2f\n",comp_data[i]->pm_durn);
      fprintf(fprt,"Interval Between PMs: %10.2f\n",comp_data[i]->pmintv);
      fprintf(fprt,"Time Unit for PM Interval: %c\n",comp_data[i]->u_pmintv);
      fprintf(fprt,"\n");
   }
   fprintf(fprt,"\f");
   fflush(fprt);
   print_msg(OFF);
   return;
}

/*
 * print_msg( status )
 *
 * Function: Places user message on screen notifying that printing
 *           is occuring.  Status is used to turn message on (status = ON)
 *           or turn off (status = OFF)
 * Returns: None
 */
print_msg( request )
int request;
{
   static WINDOW wait_wn;
   static int used = FALSE;

   if (used == FALSE)
   {
      defs_wn(&wait_wn,11,24,3,31,BDR_DLNP);     /* define message window */
      wait_wn.att = LHIGHLITE;
      sw_csadv(OFF,&wait_wn);
      sw_cleor(OFF,&wait_wn);
      used = TRUE;
   }

   if (request == ON)
   {
      set_wn(&wait_wn);       /* activate message window */
      csr_hide();
```

```
      mv_cs(0,0,&wait_wn);
      v_st("Wait...Printing Output Data",&wait_wn);
   }
   else
   if (request == OFF)
      unset_wn(&wait_wn);      /* remove message window */
   return;
}
```

The vita has been removed from the scanned document