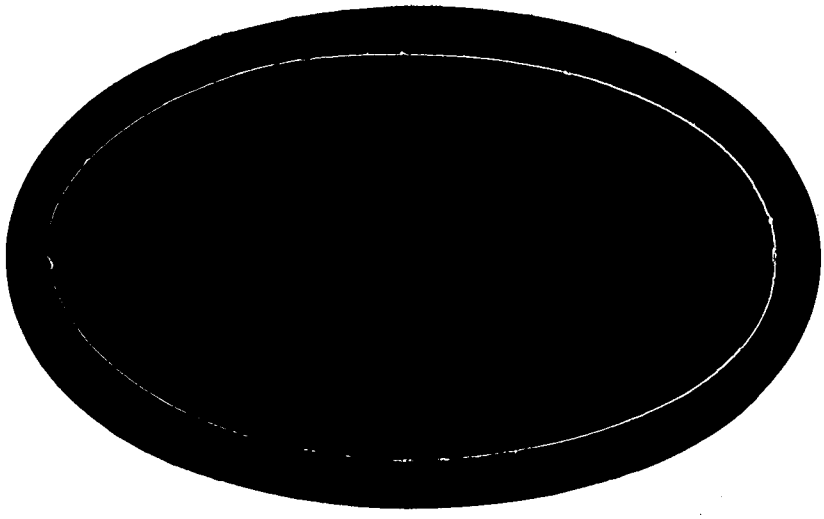
2(mix)

CR-128513



(NASA-CR-128513) ADVANCED DATA MANAGEMENT SYSTEM ANALYSIS TECHNIQUES STUDY Final Report (Intermetrics, Inc.) 30 Jul. 1972 ~~470~~ 362 p
CSCL 09B
G3/08

N72-31228
Unclas
39632

INTERMETRICS

pc

FINAL REPORT

CONTRACT NAS-9-12119

ADVANCED DATA MANAGEMENT SYSTEM ANALYSIS TECHNIQUES STUDY

30 JULY 1972

Submitted to:

National Aeronautics and Space Administration
Manned Spacecraft Center
Houston, Texas 77058

Submitted by:

Intermetrics, Inc.
701 Concord Avenue
Cambridge, Massachusetts 02138

Note: The Project Manager for NASA was Dr. William J. B. Oldham, Information Systems Directorate, MSC. The Project Manager for Intermetrics initially was Mr. Joseph Saponaro, who was succeeded by Mr. Edward Copps.

ACKNOWLEDGEMENTS

Intermetrics personnel contributing to this study were:

Neal A. Carlson, Ph.D.	-	Chapter 3
Edward M. Copps	-	Chapters 2, 5, 6, 7, 8, 13
James T. Pepe, Ph.D.	-	Chapters 1, 4, 9, 10, 11, 12, 14, 16, 17, 19, Appendices A and B
Joseph A. Saponaro	-	Chapters 15, 18

TABLE OF CONTENTS

TABLE OF ILLUSTRATIONS

PART I

INTRODUCTION

INTERMETRICS INCORPORATED • 701 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

CHAPTER 1

INTRODUCTION TO SYSTEM ANALYSIS

1-1.0 Introduction

With the continuous advance of technology more and more automation is being introduced into our society. Combined electrical and mechanical systems have been implemented which reach all of us; e.g., billing by computer, satellite communications, medical information, air traffic control, etc. The extent to which these large systems have evolved and influenced our society should not be a surprise. They were predicted by Norbert Wiener [1-3] over ten years ago. Moreover, the future should see more of these systems designed and built to assist man in his everyday life.

The methods that have been developed and used in designing these systems constitute a major part of our engineering literature. However, the system designer wishing to learn these methods faces an enormous literature search to find the significant papers. Hundreds of journals and textbooks dealing with system design have already been published, and many more appear each year. To investigate this entire literature would be a large, time consuming task. So far no publication has summarized the major design techniques and experience in a single volume to aid the student of system design. Indeed such an undertaking would be very ambitious considering the diverse areas in which large systems have been applied. Yet, some of the known design methods in a specific discipline, such as aerospace computer design, could be consolidated in a single volume. If this information is supplemented with practical experience gained in implementing such systems, the resulting publication would be a valuable aid to the aerospace computer designer. He could easily find specific design methods related to his field of interest and the results of having used these methods in the past.

The purpose of this document is to provide such an aid for the system designer. Specifically, we will concentrate on analysis techniques for aerospace data management systems. The known analysis and design methods in this area will then be combined with our own experience in designing and implementing the Apollo Guidance System to produce the final contents of this document.

1-2.0 Content of Study

Trying to answer the question of "what is system design", in all but the trivial sense, is perhaps as difficult as trying to answer the question of "what is a system". The answer is ellusive because systems take many forms and to include all these in a single definition is difficult. A general purpose computer with all its peripherals is a system, as is its software operating system. Then again, the combined hardware/software may also be considered a system but on a higher level. As a further example consider the navigation and guidance hardware for a spacecraft which includes gyros, accelerometers, autopilot,etc. This is certainly a system composed of each of its subsystems.

We will not attempt a precise definition of a system here. Instead, we will rely on our intuitive concepts of what a system is when discussing data management system analysis and design, the main focus of this document. That intuition tells us that in a trivial sense a system is an organized collection of interacting elements [4]. The word analysis stems from the ancient Greek words that mean "to break up", and the modern usage is best described as "separating a complex into its constituants and examining these elements and their inter-relationships in forming the whole." In this sense, analysis is a feedback procedure in the design of engineering systems: The system must exist before it can be analyzed.

Synthetic procedures for data management systems are not complete enough to be described as a collection of precise procedures. The synthesis or creative process must be, for the most part, left to the genius of the designer. Where synthetic techniques exist, we have included them in this report.

Since systems take many forms; e.g., computation, communication, control, etc., many methods for system design have been developed. Not all these methods are applicable to every system that is to be implemented. However, each method has proven its value under specific design conditions. Examples of these methods are:

a. simulation

b. performance monitoring

c. reliability analysis

d. redundancy

e. benchmarks, etc.

In our study we will survey the state of the art of 12 such design techniques. Each chapter will deal with a specific technique, and numerous examples will be cited of how these techniques have been successfully used implementing large systems. No chapter exhausts the known information on a topic since this would require a text devoted to each topic. Hence, a bibliography is provided with each chapter to direct the reader to further information on these design methods. In addition, two appendices provide bibliographies on important areas in which the designer should be knowledgeable but are beyond the scope of this contract to include as chapters of this document.

1-3.0 A Step Toward Unity

There is a certain danger in presenting the variety of topics that we will present in this report. Such a presentation can lead the reader to the conclusion that system analysis is an incohesive process involving the application of seemingly unrelated methods. To avoid this thought we now stress the overall unity that the system design process should have. Each analysis technique that we will present is a part of this overall process, and the designer uses these as appropriate system questions arise. Each step of analysis that he performs using these methods, with which he should be familiar, is an important step bringing him closer to achieving the complete design of his system. We hope the reader will keep this point of view in mind when reading this study.

To unify the system design methods which we will present, we begin by describing a hypothetical system that we wish to build. The statement of the problem will lead us to an understanding of the necessary elements of the system. Then we will see which design methods enable us to optimize the specification of these elements and their interactions.

1-3.1 Statement of Problem

A manned scientific laboratory is to be put in orbit around Mars. Its purpose is to observe the planet and conduct scieitific experiments. Visual data of the planet's surface as well as the results of the experiments will be transmitted back to Earth. In addition, the crew on board the laboratory will receive communications sent from Earth to start, stop or modify the experiments based upon previously collected data.

This laboratory must be designed for at least five years of use and have safety features to safeguard the lives of its crew. Some of the experiments it will conduct are the following:

a. study atmospheric phenomena of Mars;

b. study terrain for possible future landing site;

c. identify and investigate any life forms on the planet;

d. monitor quasi-stellar objects existing beyond our solar system;

e. study the composition and distribution of interstellar matter; etc.

In general, this laboratory will conduct experiments to try to answer some of the important questions of planetary geology and astrophysics [5]. Our problem is to determine the computational systems necessary to support this laboratory. A first step is to understand its computational requirements, and then we can decide upon the needed hardware and software. We begin this analysis by further reflection upon the above problem.

1-3.2 Discussion of Problem

The requirements for at least five years of use demands that the system design must employ reliability analysis. Failure detection methods must be built into the system and recovery algorithms established should a failure occur. This detection and recovery is especially important for the life support systems. Since the laboratory will be placed in orbit, size, weight and power consumption might well be constraints around which the overall system must be designed. This fact implies a cost constrained reliability analysis is necessary. That is, the most reliable system is desired which does not exceed the given design constraints.

The large amounts of scientific data that must be transmitted back to Earth could pose a communication bandwidth problem. A solution is to process as much of the data as possible on an airborne computer in the laboratory. Then only

the computed results have to be communicated, thus reducing the amount of information that needs to be transmitted. In addition, these results will tell the crew of the laboratory how to optimally conduct their future experiments.

The algorithms for the airborne computer must be designed and their execution rates determined. This information along with reliability requirements will help determine what computer should be chosen. (Computer reliability is crucial since experiments conducted in real time can be invalidated because of a computer failure for a sufficiently long time.) The choice can be aided by constructing a data book containing the characteristics of available aerospace computers. Part of this data would be the results of running benchmark programs on each of the candidate computers. These benchmarks measure performance characteristics under various computational loads. All of this data will then enable an intelligent choice of the airborne computer chosen.

The data may alternatively indicate that there is no available computer that can do the given job. In this case an aerospace computer with the designed characteristics will have to be designed and built.

The use of an airborne computer raises several software system issues. An executive system must be designed and implemented to control execution of the necessary processes. In addition, the data formats to represent scientific information within the computer must be specified so that program design can be optimized. Finally, the software system designer must decide if a higher order language will be used to encode the computer's algorithms. When all these issues have been settled, work on the software system can proceed.

The communication system between the laboratory and Earth raises additional questions. Visual and scientific data will be transmitted to Earth, and the laboratory will receive information from Earth partially directing its experiments. To insure economy in the transmitted data and reduce signal distortion from atmospheric and galactic sources, transform methods might be needed to encode transmitted data. If this is the case, more decisions must be made concerning what transform to use and whether to implement it in hardware or software.

On the other side of the communication link the data received on Earth must be further processed to provide

scientists with valuable information on the nature of Mars, the solar system and the galaxy. Since large amounts of data can be expected, a sufficiently powerful computer is necessary to process this data. This computer will probably be a commercial machine with an already existing operating system. However, the software for processing the received data will have to be designed and implemented, depending upon the algorithms provided by the scientists directing the experiments.

Once these systems have been specified, their overall design and performance must be verified. Microsimulation is the best method for the airborne computer. This is a standard way of verifying aerospace computer software. On the other hand, the overall system can be modeled using Markov analysis and queueing theory. Then the model can be simulated on a large commercial computer yielding valuable data on the expected performance of the entire system. Most importantly, this simulation will show if the system has been properly designed by showing whether it can successfully meet the computational needs of the orbiting laboratory.

1-4.0 Summary

The overall communication and computation system required is shown in Figure 1.1. We have arrived at the necessary elements and their interactions from an analysis of the given problem combined with our knowledge of system analysis techniques. As we have seen, the design of this entire system has many phases and requires many design and analysis methods. The system designers must be familiar with the proper use of all of these methods. This knowledge combined with an understanding of the given problem will help insure system design.

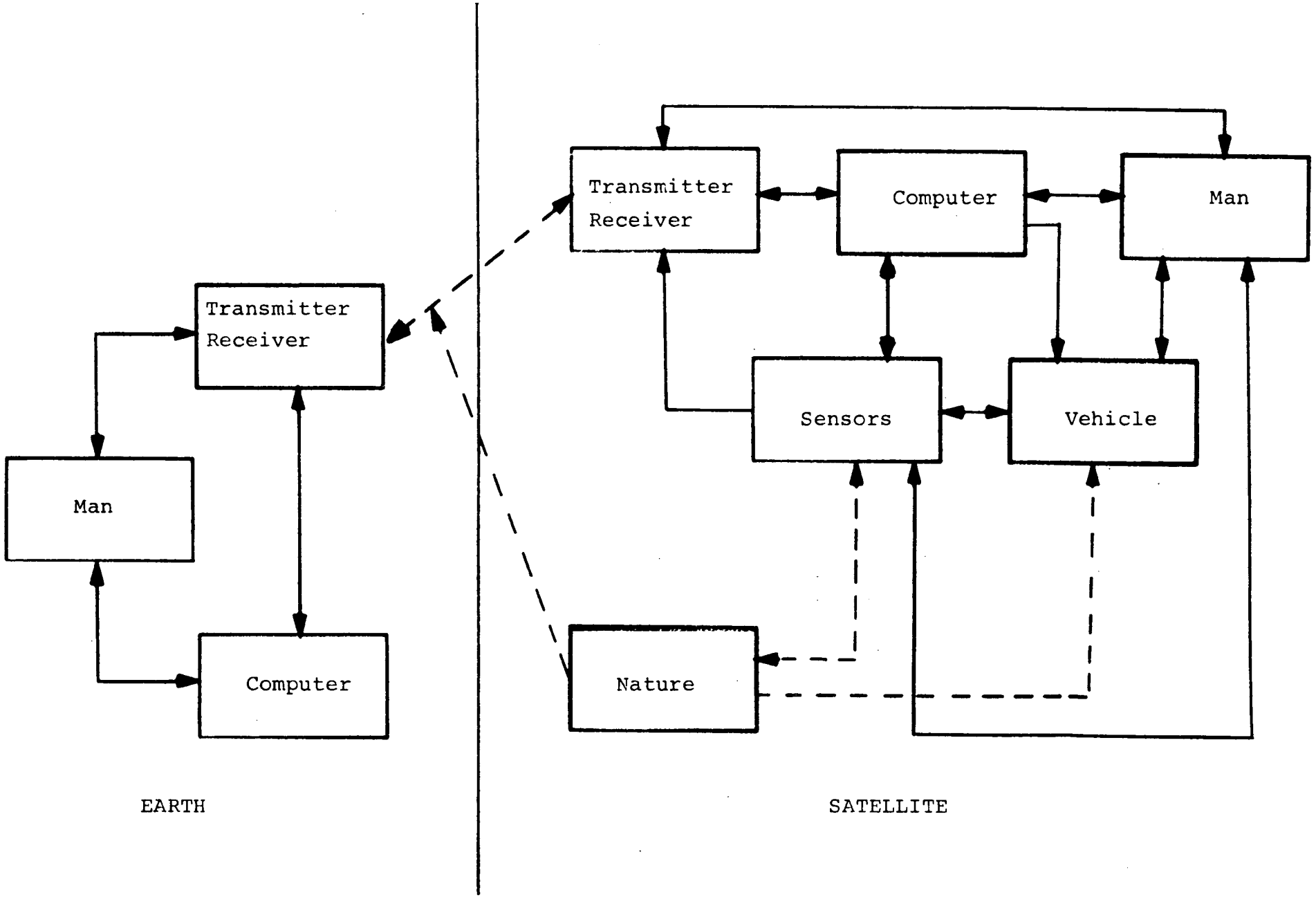


Figure 1.1 Communication and Computation System
(Elements and Interactions)

REFERENCES

1. Wiener, N., The Human Use of Human Beings, (Houghton Mifflin Co., Boston, 1950).

2. Wiener, N., God and Golem, Inc., (M.I.T. Press, Cambridge, Mass., 1964).

3. Wiener, N., Cybernetics, 2nd ed., (M.I.T. Press, Cambridge, Mass., 1961).

4. Von Bertalanffy, L., "General System Theory", in Main Currents in Modern Thought, 71(75), 1955.

5. Sciama, D.W., Modern Cosmology, (Cambridge University Press, New York, 1971).

PART II

ANALYTIC TECHNIQUES

CHAPTER 2

RELIABILITY ANALYSIS

2.1.0 Introduction

Reliability analysis is one tool in the kit bag of the system designer. It is useful when considering many problems associated with Data Management such as redundancy, on board checkout, switch over policy and so forth. This chapter develops an approach to reliability analysis that is useful under circumstances that arise frequently. These results will then be expanded in Chapter 3.

A typical problem in reliability analysis is the computation of the probability of failure as a function of time of a system of components, each subject to random failure, characterized by a failure rate λ or its inverse, the mean time to failure, τ. Solutions to this problem can be obtained in the general literature. This chapter considers a somewhat different problem statement: Suppose that a system of m components are arranged in parallel (redundantly). What is the average time it will take for only n of these components to be left. We will develop a general solution to this problem, a simple approximate solution, and an illustration problem that shows that the formulae derived are useful for a class of problems encountered in space flight.

2.2.0 Derivation

We will start with a formula that describes the probability that exactly n systems remain out of m, as a function of time.

$$P_{n/m} = \binom{m}{n} \left(1-e^{-\lambda t}\right)^{m-n} \left(e^{-n\lambda t}\right) \qquad (2.1)$$

The three terms in this product are first: the binomial coefficient which enumerates the number of ways that n specific components can be permutated out of m components, second: the probability that at least m-n components fail, third: the probability that at least n components have not failed.

We now note that the probability that the $(n+1)^{th}$ component fails in time $t+\Delta t$, as a function of t is the probability distribution function for the random variable, the failure of the $(n+1)^{th}$ component.

This function is the product of the probability that exactly n+1 systems exist at t and that one fails in t+Δt. Using equation (2.1) this can be seen to be

$$P_{[n/m]} = \binom{m}{n+1} \left(1-e^{-\lambda t}\right)^{m-(n+1)} e^{-t(n+1)} (n+1)\lambda, \quad (2.2)$$

where we let $\Delta t = 1$.

The terminology P [n/m] is used because the event that the $(n+1)^{th}$ component fails is the event that the system makes the transition to n systems remaining. Note that P n/m and P [n/m] mean different things and have different formulae. We now compute the mean time to "come down to" n components from m. The mean is by definition:

$$\tau_{n/m} = \int_0^\infty t\, P_{[n/m]}(t)\, dt \quad (2.3)$$

The integral need only be taken over positive time since $P_{[n/m]} = 0$ for $t<0$.

Substituting (2.2) into (2.3) we obtain

$$\tau_{n/m} = \lambda \binom{m}{n+1} (n+1) \int_0^\infty t\, (1-e^{-\lambda t})^{m-n-1} e^{\lambda t(-n-1)}\, dt. \quad (2.4)$$

Rearrangements yield

$$\tau_{n/m} = \binom{m}{n+1} (n+1) \int_0^\infty t\, (e^{\lambda t}-1)^{m-n-1} e^{-\lambda t m}\, dt, \text{ and} \quad (2.5)$$

the term $(e^{\lambda t}-1)^{m-n-1}$ can be expanded in a binomial series as

$$(e^{\lambda t}-1)^{m-n-1} = \sum_{j=0}^{m-n-1} \binom{m-n-1}{j} e^{\lambda t(m-n-1-j)} (-1)^j \quad (2.6)$$

This permits after more manipulation,

$$\tau_{n/m} = \lambda \binom{m}{n+1} (n+1) \sum_{j=0}^{m-n-1} \left\{ \binom{m-n-1}{j} (-1)^j \int_0^\infty t e^{\lambda t(-n-1-j)}\, dt \right\} \quad (2.7)$$

The integral $\int_0^\infty te^{-wt}\,dt$ can be shown to be equal to $\frac{1}{w^2}$.

Using this fact, we can write

$$\tau_{n/m} = \binom{m}{n+1} (n+1) \sum_{j=0}^{m-n-1} \left[\binom{m-n-1}{j} (-1)^j \frac{1}{(n+1+j)^2} \right] \qquad (2.8)$$

In equation (2.8) we have stated the desired result non-dimensionalized by dividing by τ, the mean time to failure. Table 1 presents values of $\tau_{n/m}$ for values of m, the number of components started with, and n, the number of components which defines the time when the mission must be ended.

→ Start with

End with ↓ n \ m	1	2	3	4
0	1	1.5	1.83	2.08
1	0	.5	.84	1.1
2		0	.33	.60
3			0	.25

TABLE 1: Values of $\tau_{n/m}$ for small m, n

Equation (2.8) is not difficult to evaluate numerically for small m and n. However, it can be tedious for large values. We will therefore turn to the evaluation of a somewhat similar parameter which has the virtue of being computable using natural log tables.

We begin again with equation (2.2). This time we compute the time that it is most likely that the system makes the transition from n+1 components unfailed to n components unfailed. The maximum likelyhood occurs where

$$dP_{[n/m]} / dt = 0 \qquad (2.9)$$

We solve for condition (2.9) and obtain the desired relationship

$$T_{ml} / \tau = \ln\left(\frac{m}{n+1}\right) \qquad (2.10)$$

We note that T_{m1}/τ is single valued, and therefore no question need arrise about relative maxima. Table 2 presents values of T_{m1}/τ in a manner similar to Table 1.

→ Start with →

End With ↓ / m →	1	2	3	4
0	0	.69	1.1	1.38
1		0	.40	.69
2			0	.28
3				0

TABLE 2: Values of T_{m1}/τ for small m, n

It will be seen that the maximum likelyhood formula yields a smaller T/τ than does the mean. In fact, the value $\tau_{n/m}/\tau$ lies somewhere between $\ln(\frac{m}{n+1})$ and $\ln(\frac{m}{n})$, which is proved in Chapter 3.

The logarithmic character of the T_{m1}/τ formula points out quite clearly the well known difficulty associated with obtaining lifetimes long with respect to τ. The ratio of m to n must be squared to double the "lifetime of the system", and so forth.

We now turn to an example problem that illustrates the significance of these results. It has been determined that a piece of equipment is necessary during re-entry from space. It has a τ = 1000 hours. Re-entry takes one hour. The probability that the equipment fails prior to or during re-entry must be kept below .001. The equipment can be considered to be redundantly spared with no reliability associated with switch over. Compare the redundancy required for the following missions:

Mission Ⓐ : A space flight of 1000 hours. Once the trip begins it cannot be shortened.

Mission Ⓑ : A space flight which can be terminated at any time (by a one hour re-entry) if necessary because of equipment failure. The equipment must be spared such that the average time until termination due to equipment failures is 1000 hours.

We consider mission Ⓑ first.

Step 1: Calculate the number of components necessary at the beginning of re-entry to assure no worse than .001 probability of failure.

$$P_f = (1-e^{-t/\tau})^n \quad .$$

Try n=1:

$$P_f = .001; \text{ which is o.k.}$$

Step 2: We now compute how many we have to start with (m) to end up with the one we need. Table 1 indicates that 4 systems will yield an average lifetime of 1100 hours.

We now turn to mission Ⓐ . We compute the probability that all m systems fail in 1001 hours.

$$P_f = (1-e^{-t/\tau})^m$$

For our problem $P_f \leq .001$. We compute:

$$m \leq \frac{\log (P_f)}{\log (1-e^{-1000/1001})}$$

$$m \leq 16 + = 17 \text{ systems.}$$

We see that 17 systems are required to achieve specification Ⓐ , while 4 are required to achieve specification Ⓑ . It makes a difference how the requirements are stated. Mission Ⓑ is more like what might be desired for a near earth mission such as a space shuttle.

2.3.0 Conclusions

This chapter illustrates an approach to reliability analysis which emphasizes the situation where the length of the mission is a random variable whose length can be characterized by statistical parameters. This approach permits the designer some increased latitude in formulating redundancy policy, as shown in the examples. However, one should examine higher statistical moments of equation (2.2) in order to be certain that the standard deviation is stable and that the mean is useful in characterizing the distribution. This has not been done although the method of integration by parts described after equation (2.7) would apply as well to higher moments. Therefore, the obtaining of higher moments is straight forward.

CHAPTER 3

REDUNDANT COMPONENT FAILURE ANALYSIS

3-1.0 Introduction

Required system reliability often must be achieved through redundancy of critical subsystems or components. Typical questions then to be answered by the system designer are these. How many redundant components are necessary to provide a specified operating lifetime with a prescribed probability of success? Conversely, what minimum lifetime can be guaranteed with a prescribed confidence level for a given number of redundant components? What mean lifetime can be expected for this number of components? The design parameters of interest are clearly interrelated, and include the following:

- m original number of redundant components
- n number required for successful operation
- T operating lifetime
- τ mean time to failure
- P probability of success (operation)
- Q 1-P, probability of failure

3-2.0 Summary of Results

A single component exhibiting an exponential failure rate has the following probabilities of successful operation p and failure q during a time interval t,

$$p = e^{-\lambda t} \tag{3.1}$$

$$q = 1 - p = 1 - e^{-\lambda t} \tag{3.2}$$

where λ is the mean rate of failure. The mean time to failure τ is

$$\tau = 1/\lambda \tag{3.3}$$

The minimum component lifetime T during which the probability of success is $\geq$ P, or probability of failure is $\leq$ Q = 1 - P, is

$$T = -\tau \ \ell n \ P = \tau(Q + \tfrac{1}{2}Q^2 + \tfrac{1}{3}Q^3 + \ldots) \tag{3.4}$$

Given m identical redundant components characterized by the exponential failure rate above, the probability that exactly n remain in operation (i.e., that m-n fail) during a time interval t is

$$P_{n/m} = \binom{m}{n} p^n q^{m-n} \tag{3.5}$$

$$= \binom{m}{n} (e^{-\lambda t})^n (1-e^{-\lambda t})^{m-n}$$

The probability that <u>at least</u> n components remain operational during t (or that at <u>most m-n</u> fail) is

$$p^+_{n/m} = \sum_{k=n}^{m} \binom{m}{k} p^k q^{m-k} \tag{3.6}$$

$$= 1 - \sum_{k=0}^{n-1} \binom{m}{k} p^k q^{m-k}$$

The time of maximum probability that exactly n components remain unfailed, and that probability, are

$$t^*_{n/m} = \tau \ \ell n \frac{m}{n} \tag{3.7}$$

$$(p_{n/m})_{max} = \binom{m}{n} n^n (m-n)^{m-n}/m^m \tag{3.8}$$

The mean time to failure of one of the remaining n components, i.e., the mean time for the system to degenerate to n-1 components, is given by

$$\tau_{n-1/m} = \tau \left(\frac{1}{n} + \frac{1}{n+1} + \ldots + \frac{1}{m} \right) \tag{3.9}$$

This time can be bracketed by logarithmic bounds,

$$\tau \ \ell n \ \frac{m+1}{n} < \tau_{n-1/m} < \tau \ \ell n \ \frac{m}{n-1} \tag{3.10}$$

The standard deviation of this degeneration time is

$$\sigma_{t_{n-1/m}} = \tau \left(\frac{1}{n^2} + \frac{1}{(n+1)^2} + \ldots + \frac{1}{m^2} \right)^{1/2} \tag{3.11}$$

which can be bracketed by similar bounds,

$$\tau \sqrt{\frac{m-n+1}{(m+1)n}} < \sigma_{t_{n-1/m}} < \tau \sqrt{\frac{m-n+1}{m(n-1)}} \tag{3.12}$$

The minimum system lifetime T during which the probability that at least n components remain equals P, or during which the probability of at most m-n failures equals Q = 1-P, is obtained from

$$Q = \sum_{k=0}^{n-1} \binom{m}{k} (1-q)^k \, q^{m-k} \quad \rightarrow \quad q(Q) \tag{3.13}$$

$$T_{n/m} = -\tau \ \ell n \ (1-q) \tag{3.14}$$

When system failure corresponds to all components failed, (i.e., operation requires $n \geq 1$), the minimum system lifetime with probability P = 1 - Q) is

$$T_{1/m} = -\tau \ \ell n \ (1 - Q^{1/m}) \qquad (3.15)$$

$$= \tau \ (Q^{1/m} + \tfrac{1}{2}Q^{2/m} + \tfrac{1}{3}Q^{3/m} + \ldots)$$

3-3.0 Component Failure Statistics

To begin, we presume that the m redundant components are identical, and that they operate (and fail) independently. The probability of success or failure of each component is assumed to be a known function of time:

$$p = p(t) = \text{probability of successful operation}$$

$$q = q(t) = \text{probability of failure} \qquad (3.16)$$

with

$$p + q \equiv 1$$

In the most commonly observed case, operating components exhibit an exponential failure rate. The probability that a component will fail in the next instant dt, given that it is operating at t, is a constant independent of how long it has been operating:

$$\text{Prob}\ [t \leq t_f < t + dt \mid t_f \geq t] = \lambda \ dt \qquad (3.17)$$

where t_f represents the time of failure and λ is a constant which equals the mean failure rate. This probability relation can be expressed as follows,

$$dq = p \ \lambda \ dt = (1-q) \ \lambda \ dt \qquad (3.18)$$

That is, the incremental probability of failure between t and t+dt equals the probability that no failure has yet occurred, multiplied by a constant times the interval dt. The consequence of (3.18) is evidently

$$p(t) = e^{-\lambda (t-t_o)}$$

$$q(t) = 1 - e^{-\lambda (t-t_o)} \tag{3.19}$$

(For simplicity, we henceforth set $t_o = 0$.) Hence, the incremental failure probability in the interval dt is

$$dq = e^{-\lambda t} \lambda \ dt \tag{3.20}$$

We next define the one-component failure probability density f(t) as follows:

$$f(t) \equiv \frac{d}{dt} q(t) = \lambda \ e^{-\lambda t} \tag{3.21}$$

In terms of f, the probabilities of failure and success can be written

$$q(t) = \int_0^t f(t') \ dt'$$

$$p(t) = 1 - \int_0^t f(t') \ dt' = \int_t^\infty f(t') \ dt' \tag{3.22}$$

Further, the mean time to failure τ for a single component is defined and obtained as

$$\tau \equiv \int_0^\infty t \ f(t) \ dt = \int_0^\infty t\lambda \ e^{-\lambda t} \ dt = \frac{1}{\lambda} \tag{3.23}$$

For sake of interest we note in Table 1 the success and failure probabilities for a single component as functions of time measured in τ-units:

p(t)	q(t)	t/τ
.999	.001	.001
.99	.01	.010
.95	.05	.051
.90	.10	.105
.50	.50	.693
.368	.632	1.000
.10	.90	2.303
.05	.95	2.996
.01	.99	4.605

Table 1. Probabilities of success and failure of a single component as functions of operating time

The "half-life" of a large number of single components is

$$t_{.50} = .693\ \tau \tag{3.24}$$

3-4.0 Redundant Computer Failure Statistics

Consider now m redundant components beginning independent operation at t=0. The joint failure probability density for these m components is the product of the individual densities:

$$f_m(t_1,t_2,\ldots,t_m) = f(t_1)f(t_2)\ldots f(t_m) \tag{3.25}$$

$$= \lambda^m\, e^{-\lambda(t_1+t_2+\ldots+t_m)}$$

The probability that specific components 1 to n operate successfully, and components n+1 to m fail, during time t is

$$p_{1\leftarrow n}(t) = \int_t^\infty dt_1 \ldots \int_t^\infty dt_n \int_0^t dt_{n+1}\ldots \int_0^t dt_m\, f_m(t_1,\ldots,t_m)$$

$$= p(t)^n\, q(t)^{m-n} \tag{3.26}$$

The probability that any n components operate successfully, and the remaining m-n fail, during time t is then

$$p_{n/m} = \binom{m}{n} p^n q^{m-n} \tag{3.27}$$

where the binomial coefficient

$$\binom{m}{n} \equiv \frac{m!}{n!(m-n)!} = \frac{m(m-1)\ldots(m-n+1)}{n(n-1)\ldots 1} \tag{3.28}$$

gives the number of different ways that n components can be selected from a total of m. Note that $p_{n/m}$ is the k=n term in the binomial expansion

$$(p+q)^m = \sum_{k=0}^{m} \binom{m}{k} p^k q^{m-k} \equiv 1 \tag{3.29}$$

Equation (3.27) is valid in general for any component failure probabilities q = 1-p. For the exponential failure probabilities (3.19), eq. 3.27) becomes

$$p_{n/m}(t) = \binom{m}{n} \left(e^{-\lambda t}\right)^n \left(1-e^{-\lambda t}\right)^{m-n} \tag{3.30}$$

In either case, $p_{n/m}$ represents the probability that <u>exactly</u> n of m components remain operational after time t. It is evident that, since no components can fail in zero time,

$$p_{n/m}(0) = \begin{cases} 1 & n=m \\ 0 & n<m \end{cases} \tag{3.31}$$

The most probable time at which exactly n of m components remain can be obtained from eqs. (3.16) and (3.27) as follows:

$$\frac{d}{dq} p_{n/m} = \binom{m}{n} p^{n-1} q^{m-n-1} (m-n-mq) = 0$$

$$q^*_{n/m} = \frac{m-n}{m} \quad \rightarrow \quad t^*_{n/m} \tag{3.32}$$

The corresponding maximum probability density at $t^*_{n/m}$ is

$$(p_{n/m})_{max} = \binom{m}{n} n^n (m-n)^{m-n}/m^m \tag{3.33}$$

For the exponential failure probability (3.19), the time of maximum probability is

$$t^*_{n/m} = \tau \ln \frac{m}{n} \tag{3.34}$$

Also of interest is the probability that at least n of m components remain in operation at time t. This probability, denoted $p^+_{n/m}$, is equal to

$$\begin{aligned} p^+_{n/m} &= p_{n/m} + p_{n+1/m} + \cdots + p_{m/m} \\ &= \sum_{k=n}^{m} \binom{m}{k} p^k q^{m-k} \\ &= 1 - \sum_{k=0}^{n-1} \binom{m}{k} p^k q^{m-k} \end{aligned} \tag{3.35}$$

Hence $p^+_{n/m}$ is the sum of the last m-n+1 terms of the binomial expansion (3.29), or unity minus the first n terms. Note that, in this case,

$$p^+_{n/m}(0) \equiv 1 \qquad \text{all } n \leq m \tag{3.36}$$

We next determine the incremental probability that one of the remaining n components fails during the interval dt following time t. From eqs. (3.27) and (3.21), this probability is

$$P_{[n-1/m]} = dq_{n-1/m} = p_{n/m}(t)\ n\ \lambda\ dt$$

$$= \binom{m}{n}\ p^n\ q^{m-n}\ n\ \lambda\ dt \tag{3.37}$$

where $p_{n/m}$ is the probability that exactly n of m components remain at t, and $n\lambda dt$ is the incremental probability that any one of these n components then fails between t and t+dt. Hence the probability density of the $(m-n+1)^{th}$ failure (i.e., degeneration to n-1 components) occurring at time t is

$$f_{n-1/m}(t) \equiv \frac{dq_{n-1/m}}{dt} = \binom{m}{n}\ p(t)^n\ q(t)^{m-n}\ n\ \lambda \tag{3.38}$$

For the exponential component failure (3.19), this probability density becomes

$$f_{n-1/m}(t) = \binom{m}{n}\ (e^{-\lambda t})^n\ (1-e^{-\lambda t})^{m-n}\ n\ \lambda \tag{3.39}$$

The mean time to the $(m-n+1)^{th}$ failure, i.e., the mean time for the system to degenerate to n-1 components remaining, is obtained for the exponential case from eq. (3.39) as follows:

$$\tau_{n-1/m} \equiv \int_0^\infty t\ f_{n-1/m}\ (t)\ dt$$

$$= \binom{m}{n}\ n \int_0^\infty \lambda t\ (e^{-\lambda t})^n\ (1-e^{-\lambda t})^{m-n}\ dt \tag{3.40}$$

$$= \tau \binom{m}{n}\ n \int_0^\infty x\ e^{-nx}\ (1-e^{-x})^{m-n}\ dx$$

This integral can be evaluated in series form by utilizing a binomial expansion of the last term in the integrand, as in Chapter 2:

$$(1-e^{-x})^{m-n} = \sum_{k=0}^{m-n} \binom{m-n}{k} (-e^{-x})^k \tag{3.41}$$

The result of substituting this expansion into eq. (3.40) and integrating is

$$\tau_{n-1/m} = \tau \binom{m}{n} n \sum_{k=0}^{m-n} \binom{m-n}{k} (-1)^k \int_0^\infty x\, e^{-(n+k)x}\, dx$$

$$= \tau \binom{m}{n} n \sum_{k=n}^{m} \binom{m-n}{k-n} (-1)^{k-n} \frac{1}{k^2} \tag{3.42}$$

However, the integral (3.40) can be evaluated through integration by parts to yield a simpler form for the same result, namely,

$$\tau_{n-1/m} = \tau \sum_{k=n}^{m} \frac{1}{k} = \tau\left(\frac{1}{n} + \frac{1}{n+1} + \ldots + \frac{1}{m}\right) \tag{3.43}$$

The integration can be performed most readily by use of an auxiliary integral, considered to be a function of two exponents:

$$I(n,k) \equiv \int_0^\infty e^{-nx} (1-e^{-x})^k\, dx \tag{3.44}$$

This integral is evaluated by transforming variables and integrating by parts:

$$y = e^{-x},\ dy = -e^{-x}\, dx$$

$$I(n,k) = \int_0^1 y^{n-1} (1-y)^k\, dy$$

$$= (1-y)^k \frac{y^n}{n}\Big|_0^1 + \frac{k}{n} \int_0^1 y^n (1-y)^{k-1}\, dy$$

$$= \frac{k}{n} \int_0^1 y^n (1-y)^{k-1}\, dy \tag{3.45}$$

Successive integration by parts then yields

$$I(n,k) = \frac{k(k-1)\ldots 1}{n(n+1)\ldots(n+k-1)} \int_0^1 y^{n+k-1}\, dy$$

$$= \frac{k(k-1)\ldots 1}{n(n+1)\ldots(n+k)} = \frac{k!\,(n-1)!}{(n+k)!} = 1/n \binom{n+k}{n} \qquad (3.46)$$

Note that the integral of the probability density (3.39) is readily obtained from this formula:

$$\int_0^\infty f_{n-1/m}(t)dt = \binom{m}{n} n \int_0^1 e^{-nx}(1-e^{-x})^{m-n}\, dx$$

$$= \binom{m}{n} n\, I(n,m-n) \equiv 1 \qquad (3.47)$$

To evaluate the mean degeneration time $\tau_{n-1/m}$, we define a second integral function of the two exponents,

$$J(n,k) \equiv \int_0^\infty x\, e^{-nx}\, (1-e^{-x})^k\, dx \qquad (3.48)$$

and note that $\tau_{n-1/m}$ can then be expressed as

$$\tau_{n-1/m} = \tau \binom{m}{n} n\, J(n,m-n) \qquad (3.49)$$

The function J(n,k) can be integrated directly by parts (see Section 3-6.0),or more simply as follows. We first note that

$$\frac{\partial}{\partial n} I(n,k) = \int_0^\infty \frac{\partial}{\partial n} (e^{-nx})(1-e^{-x})^k \, dx$$

$$= \int_0^\infty (-x)\, e^{-nx}\, (1-e^{-x})^k \, dx$$

$$\equiv - J(n,k) \tag{3.50}$$

However, from eq. (3.46) we see that

$$\ell n\, I(n,k) = \ell n\, k! - \ell n n - \ell n(n+1) - \ldots - \ell n(n+k)$$

$$\frac{1}{I} \frac{\partial}{\partial n} I(n,k) = - \left(\frac{1}{n} + \frac{1}{n+1} + \ldots + \frac{1}{n+k} \right) \tag{3.51}$$

Hence, from (3.50) and (3.51)

$$J(n,k) = I(n,k) \left(\frac{1}{n} + \frac{1}{n+1} + \ldots + \frac{1}{n+k} \right)$$

$$= \frac{k!\,(n-1)!}{(n+k)!} \left(\frac{1}{n} + \frac{1}{n+1} + \ldots + \frac{1}{n+k} \right) \tag{3.52}$$

Letting k=m-n and substituting (3.52) into (3.49) then yield the desired result:

$$\tau_{n-1/m} = \tau \left(\frac{1}{n} + \frac{1}{n+1} + \ldots + \frac{1}{m} \right) \tag{3.53}$$

It is easy to show, next, by considering a rectangular integration rule, that the following logarithmic bounds apply to the series of inverse numbers:

$$\sum_{i=1}^{k} \frac{\Delta x}{a+i\Delta x} < \int_a^b \frac{dx}{x} < \sum_{i=0}^{k-1} \frac{\Delta x}{a+i\Delta x} \; ; \; \Delta x = \frac{b-a}{k}$$

$$\frac{1}{a+1} + \frac{1}{a+2} + \ldots \frac{1}{b} < \ell n \frac{b}{a} < \frac{1}{a} + \frac{1}{a+1} + \ldots + \frac{1}{b-1} \; ; \quad \Delta x = 1 \tag{3.54}$$

Thus,

$$\ell n \frac{m+1}{n} < \frac{1}{n} + \frac{1}{n+1} + \ldots + \frac{1}{m} < \ell n \frac{m}{n-1}$$

$$\tau \ell n \frac{m+1}{n} < \tau_{n-1/m} < \tau \ell n \frac{m}{n-1} \qquad (3.55)$$

By comparison with eq. (3.34), we see that $\tau_{n-1/m}$ lies between $t^*_{n/m}$ and $t^*_{n-1/m}$, the times of greatest probability that n and n-1 components remain, respectively. Table 2 presents values of $\tau_{n-1/m}$, measured in τ-units, for various initial and final numbers of components, m and n-1.

$$\tau_{n-1/m} \; / \; \tau$$

Start with ⟶

End with ↓ \ m	1	2	3	4	5
0	1.0	1.5	1.833	2.083	2.283
1	0	.5	.833	1.083	1.283
2		0	.333	.583	.783
3			0	.250	.450
4				0	.200

Table 2 Mean time to degenerate from m to n-1 components

As an aside, we note that the time $\tau_{n-1/m}$ is also the mean time to at least n systems remaining. Equation (35) for $p^+_{n/m}$ gives the probability that at least n components remain at t. The converse probability, that at most n-1 components remain (or at least m-n+1 have failed), is

$$q_{n-1/m} = 1 - p^{+}_{n/m} = 1 - \sum_{k=n}^{m} \binom{m}{k} p^k q^{m-k} \tag{3.56}$$

$$= \sum_{k=0}^{n-1} \binom{m}{k} p^k q^{m-k} \tag{3.57}$$

The failure probability density is then

$$f_{n-1/m} = \frac{d}{dt} q_{n-1/m} = \sum_{k=0}^{n-1} \binom{m}{k} p^{k-1} q^{m-k-1} [(m-k)p - kq] \frac{dq}{dt}$$

$$= \sum_{k=0}^{n-1} \left[\binom{m}{k+1} (k+1) p^k q^{m-k-1} - \binom{m}{k} k p^{k-1} q^{m-k} \right] \frac{dq}{dt}$$

$$= \frac{dq}{dt} \binom{m}{n} n p^{n-1} q^{m-n} \tag{3.58}$$

For the exponential failure probability, $dq/dt = p\lambda$, and eq. (3.58) becomes

$$f_{n-1/m} = \binom{m}{n} p^n q^{m-n} n\lambda \tag{3.59}$$

This relation is identical to eq. (3.38). Consequently, the mean time during which at least n components remain is identical to the mean time $\tau_{n-1/m}$ at which the system degenerates to n-1 components, given by (3.40). (That these two times should be identical perhaps is obvious.)

The mean-squared time of degeneration to n-1 components, and hence the standard deviation of the degeneration time, can also be determined by the previous method. First, we note that

$$\frac{\partial^2}{\partial n^2} I(n,k) = \int_0^\infty x^2 e^{-nx} (1-e^{-x})^k \, dx \tag{3.60}$$

and that the mean-squared degeneration time to n-1 components is

$$\overline{t^2}_{n-1/m} \equiv \int_0^\infty t^2 \, f_{n-1/m}(t)\, dt$$

$$= \tau^2 \binom{m}{n} n \int_0^\infty x^2 \, e^{-nx} \, (1-e^{-x})^{m-n} \, dx \qquad (3.61)$$

From eqs. (3.50) and (3.52),

$$\frac{\partial^2}{\partial n^2} I = -\frac{\partial}{\partial n}\left[I\left(\frac{1}{n} + \ldots + \frac{1}{n+k}\right)\right] \qquad (3.62)$$

$$= I\left[\frac{1}{n^2} + \ldots + \frac{1}{(n+k)^2}\right] + I\left[\frac{1}{n} + \ldots + \frac{1}{n+k}\right]^2$$

Letting k=m-n and substituting (3.62) into (3.61) then yield

$$\overline{t^2}_{n-1/m} = \tau^2\left[\frac{1}{n^2} + \ldots + \frac{1}{m^2}\right] + \tau^2\left[\frac{1}{n} + \ldots + \frac{1}{m}\right]^2 \qquad (3.63)$$

The second time is simply the square of the mean, $\tau_{n-1/m}$. Hence the first term represents the variance, or

$$\sigma^2_{t_{n-1/m}} \equiv \overline{t^2}_{n-1/m} - (\tau_{n-1/m})^2$$

$$= \tau^2\left[\frac{1}{n^2} + \ldots + \frac{1}{m^2}\right] \qquad (3.64)$$

The standard deviation of the degeneration time to n-1 components remaining is thus

$$\sigma_{t_{n-1/m}} = \tau\left[\frac{1}{n^2} + \ldots + \frac{1}{m^2}\right]^{1/2} \qquad (3.65)$$

Again, it is relatively easy to establish the following bounds:

$$\int_a^b \frac{dx}{x^2} = \frac{b-a}{ab} \begin{cases} > \dfrac{1}{(a+1)^2} + \dfrac{1}{(a+2)^2} + \ldots + \dfrac{1}{b^2} \\ \\ < \dfrac{1}{a^2} + \dfrac{1}{(a+1)^2} + \ldots + \dfrac{1}{(b-1)^2} \end{cases} \tag{3.66}$$

such that

$$\tau \sqrt{\frac{m-n+1}{(m+1)n}} < \sigma_{t_{n-1/m}} < \tau \sqrt{\frac{m-n+1}{m(n-1)}} \tag{3.67}$$

The times at which the probability $p^+_{n/m}$ has certain prescribed values are also of interest. (Recall that $p^+_{n/m}$ represents the probability that at least n of the m systems remain operational.) Define $T_{n/m}$ to be that time at which

$$p^+_{n/m}(t) = p^+_{n/m}(T_{n/m}) = P \tag{3.68}$$

We determine in τ-units as a function of P, m, and n from eqs. (3.35) and (3.19):

$$Q = 1 - P = \sum_{k=0}^{n-1} \binom{m}{k} (1-q)^k q^{m-k} \tag{3.69}$$

$$\rightarrow q = q(Q,m,n)$$

$$T_{n/m} = -\tau \ln(1-q) \tag{3.70}$$

When P is near unity, such that Q<<1, and q<<1, q and $T_{n/m}$ are approximately equal to

$$q \simeq \left[Q/\binom{m}{n-1}\right]^{\frac{1}{m-n+1}} \tag{3.71}$$

$$T_{n/m} \simeq \tau\, q \tag{3.72}$$

Perhaps of greatest interest is the case n=1, for which n-1=0 corresponds to all components failed. For this case, eq. (3.69) is simply

$$Q = q^m \tag{3.73}$$

such that q and $T_{n/m}$ are given by

$$q = Q^{1/m} = (1-P)^{1/m} \tag{3.74}$$

$$T_{1/m} = -\tau\, \ell n(1-Q^{1/m}) \tag{3.75}$$

The values of $T_{1/m}$ for several probabilities P and numbers of original components m are given in Table 3.

$T_{1/m} / \tau$ ⟶ Increasing redundancy

Decreasing confidence ↓

P \ m	1	2	3	4	5
.999	.0010	.0032	.1054	.1958	.2893
.99	.0101	.1054	.2427	.3802	.5077
.95	.0513	.2528	.4594	.6403	.7969
.90	.1054	.3802	.6239	.8263	.9969
.50	.6932	1.228	1.578	1.838	2.044

Table 3 Probable lifetimes and confidence levels for at least 1 of m components operating

Similar tables can be constructed for the cases n=2, n=3, etc. For example, the case n=2 corresponds to mission termination when all but one component fail. Equation (3.69) becomes

$$Q = q^m + m(1-q)q^{m-1}$$

$$= mq^{m-1}\left(1 - \frac{m-1}{m} q\right) \qquad (3.76)$$

Here, q is approximately equal to

$$q \approx (Q/m)^{\frac{1}{m-1}} \qquad (3.77)$$

and can readily be obtained from eq. (3.76) by successive approximations. The values of $T_{2/m}$ for a number of probabilities P and original components m are given in Table 3.

$T_{2/m} / \tau$

P \ m	2	3	4
.999	.0005	.0185	.0299
.99	.0050	.0606	.1520
.95	.0256	.1454	.2859
.90	.0527	.2179	.3864
.50	.3460	.2500	.9544

Increasing redundancy →

Decreasing confidence ↓

Table 4 Probable lifetimes and confidence levels for at least 2 of m components operating

3-5.0 Sample Problem

To illustrate the application of the previous results, we consider the example of Chapter 2. A particular system component is known to be essential during re-entry from space. This component has a mean time to failure of τ=1000 hr., and re-entry takes one hr. The probability that the component fails prior to or during re-entry must be kept below .001, i.e., the probability of safe return must be $\geq$ .999.

The probability of failure of this one component during re-entry itself is

$$q = 1 - e^{-1/1000} = .001$$

The probability of failure of all of the m components during a 1000 hour mission plus re-entry, or 1001 hours, is

$$q_{o/m} = \left(1-e^{-1001/1000}\right)^m$$

$$= .633^m \leq .001$$

which leads to

$$m \geq 15.1 \text{ or } 16 \text{ components required}$$

If the mission is terminated and re-entry initiated whenever the system degenerates to 1 component remaining, the mean mission lifetime for m=16 will be

$$\tau_{1/16} = 1000\left(\frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{16}\right) = 2381 \text{ hrs.}$$

However, from Table 2 we see that only 4 original systems are required to provide a mean mission lifetime (to one component remaining) of slightly over 1000 hours:

$$\tau_{1/4} = 1000\left(\frac{1}{2} + \frac{1}{3} + \frac{1}{4}\right) = 1083 \text{ hrs.}$$

What is the probability that the mission thus defined will last at least 1000 hours? Using eqs. (3.19) and (3.35), the probability that two or more components operate at least for 1000 hrs. is determined as follows:

$$p = e^{-1000/1000} = .368$$

$$q = 1 - p = .632$$

$$p^{+}_{2/4} = p_{2/4} + p_{3/4} + p_{4/4}$$

$$= 6p^2q^2 + 4p^3q + p^4$$

$$= .324 + .128 + .018 = .470$$

$$\text{(or)} \quad = 1 - p_{0/4} - p_{1/4}$$

$$= 1 - q^4 - 4\,pq^3$$

$$= 1 - .159 - .371 = .470$$

Thus, the probabilities that exactly 4, 3, or 2 components remain after 1000 hrs. are, respectively, .018, .128, and .324. Hence, the probability that at least 2 remain is .470. From Table 4, we note the following probable mission lifetimes for m=4 with $n \geq 2$ components operational:

P = .999	T $\geq$ 30 hrs.
P = .99	T $\geq$ 152 hrs.
P = .95	T $\geq$ 286 hrs.
P = .90	T $\geq$ 386 hrs.
P = .50	T $\geq$ 954 hrs.

Note that the probability of safe re-entry is always .999, and with m=4 original components, the probability is .47

that the mission will last (i.e., will not be terminated by degeneration to n=1 component) for at least 1000 hours.

On the other hand, note that if the mission cannot be terminated at any desired time, its pre-determined duration must be limited to 196-1 = 195 hours for P = .999 (see Table 3) probability of safe return, if only m=4 redundant components are utilized. As determined previously, a 1000-hour mission duration with no possibility of early termination necessitates m=16 redundant components for $P \geq .999$ overall.

3-6.0 Integration by Parts

We have given the integral (eq 3.46 of the text)

$$I(n,k) = \int_0^\infty e^{-nx} (1-e^{-x})^k \, dx = \frac{k!(n-1)!}{(n+k)!} \tag{3.78}$$

and desire to evaluate the integral

$$J(n,k) = \int_0^\infty x \, e^{-nx} (1-e^{-x})^k \, dx \tag{3.79}$$

We first note the indefinite integral,

$$\int x \, e^{-nx} \, dx = -\frac{1}{n} e^{-nx} \left(x+\frac{1}{n}\right) \tag{3.80}$$

and then integrate eq. (3.79) by parts:

$$J(n,k) = -\frac{1}{n} e^{-nx} \left(x+\frac{1}{n}\right)(1-e^{-x})^k \Big|_0^\infty$$

$$+ \frac{k}{n} \int_0^\infty \left(x+\frac{1}{n}\right)(e^{-x})^{n+1}(1-e^{-x})^{k-1} \, dx$$

$$= \frac{k}{n^2} I(n+1,k-1) + \frac{k}{n} J(n+1,k-1)$$

$$= \frac{1}{n} I(n,k) + \frac{k}{n} J(n+1, k-1) \tag{3.81}$$

where the transformation of I(n+1, k-1) to I(n,k) follows from eq. (3.78) Integrating J(n+1, k-1) next by parts yields

$$J(n,k) = \frac{1}{n} I(n,k) + \frac{k}{n}\left[\frac{1}{n+1} I(n+1,k-1) + \frac{k-1}{n+1} J(n+2,k-2)\right]$$

$$= \left[\frac{1}{n} + \frac{1}{n+1}\right] I(n,k) + \frac{k(k-1)}{n(n+1)} J(n+2,k-2) \qquad (3.82)$$

Proceeding in this fashion leads, after k-2 more integrations, to

$$J(n,k) = \left[\frac{1}{n} + \frac{1}{n+1} + \ldots \frac{1}{n+k-1}\right] I(n,k)$$

$$+ \frac{k(k-1)\ldots 1}{n(n+1)\ldots(n+k-1)} J(n+k,0) \qquad (3.83)$$

The last integral can be evaluated directly using eq. (3.79)

$$J(n+k,0) = \int_0^\infty x\, e^{-(n+k)x}\, dx$$

$$= -\frac{1}{n+k} e^{-(n+k)x} \left(x+\frac{1}{n+k}\right)\Big|_0^\infty = \frac{1}{(n+k)^2} \qquad (3.84)$$

Hence the second term in (3.83) is

$$\frac{k(k-1)\ldots 1}{n(n+1)\ldots(n+k)} \frac{1}{n+k} = I(n,k) \frac{1}{n+k} \qquad (3.85)$$

and the desired integral J(n,k) is equal to

$$J(n,k) = I(n,k) \left[\frac{1}{n} + \frac{1}{n+1} + \ldots + \frac{1}{n+k}\right] \qquad (3.86)$$

in agreement with eq. (3.52).

CHAPTER 4

COST CONSTRAINED RELIABILITY ANALYSIS

4-1.0 Introduction

We now turn to a slightly different reliability question. How does one design the most reliable system within given cost constraints? In fact, the fundamental problem of cost contrained reliability analysis may be stated very succinctly: How should redundancy within a system be allocated to achieve maximum reliability without exceeding one or more cost constraints? Several solutions have been proposed for this problem [1-6], most of which involve non-linear programming techniques [7-8] and have been implemented on digital computers.

This chapter will examine two specific solutions to this problem and present numerical examples of reliability optimization employing these methods. However, before pursuing this goal, we will examine some fundamental redundancy engineering principles.

4-2.0 General Redundancy Considerations

Redundant units are often added to hardware systems to increase the overall system reliability. Should one unit fail, a duplicate unit can be used as a replacement. Through redundancy the life span of the entire system can be extended. However, there are several modes of operation and configuration that these redundant units can assume.

A. Active Mode of Operation

Within this mode the redundant equipment is powered up and is fully operational for the duration of system operation. When a unit failure is detected, the faulty unit can be immediately switched out of the circuit, and an active spare switched in, enabling uninterrupted system operation.

Two disadvantages of active redundancy are:

1. increased system power consumption; and
2. the failure rate of an active unit may increase with time in operation.

B. Standby Mode of Operation

In standby mode the redundant units are powered down until the active unit fails. At this time a spare is powered up and if necessary, initialized so that system operation can continue. The amount of time necessary to power up and initialize a standby unit can be a disadvantage in using this mode of operation.

C. Series - Parallel Configuration

A typical series-parallel configuration is shown in Figure 4.1. In this configuration each unit A_{ij} has (m_j-1) spares. The system operates successfully if at least one unit at each stage is operational. Although this configuration is the most reliable in terms of organizing the individual components, it requires much more switching equipment than other configurations.

D. Parallel - Series Configuration

A system can also be organized in a parallel-series configuration as shown in Figure 4.2. Although there is less switching equipment needed than in the series-parallel configuration, one failed unit A_{ij} will put the entire row i out of operation. Thus, the saving realized in switching equipment is compensated for by the fact that this organization is not as reliable as a series-parallel organization.

4-2.1 Reliability Equations

In developing the equations for system reliability there are five redundancy models to be considered:

Model a: series-parallel, active redundancy

Model b: series-parallel, standby redundancy

Model c: parallel-series, active redundancy

Model d: parallel-series, standby redundancy

Model e: no redundancy

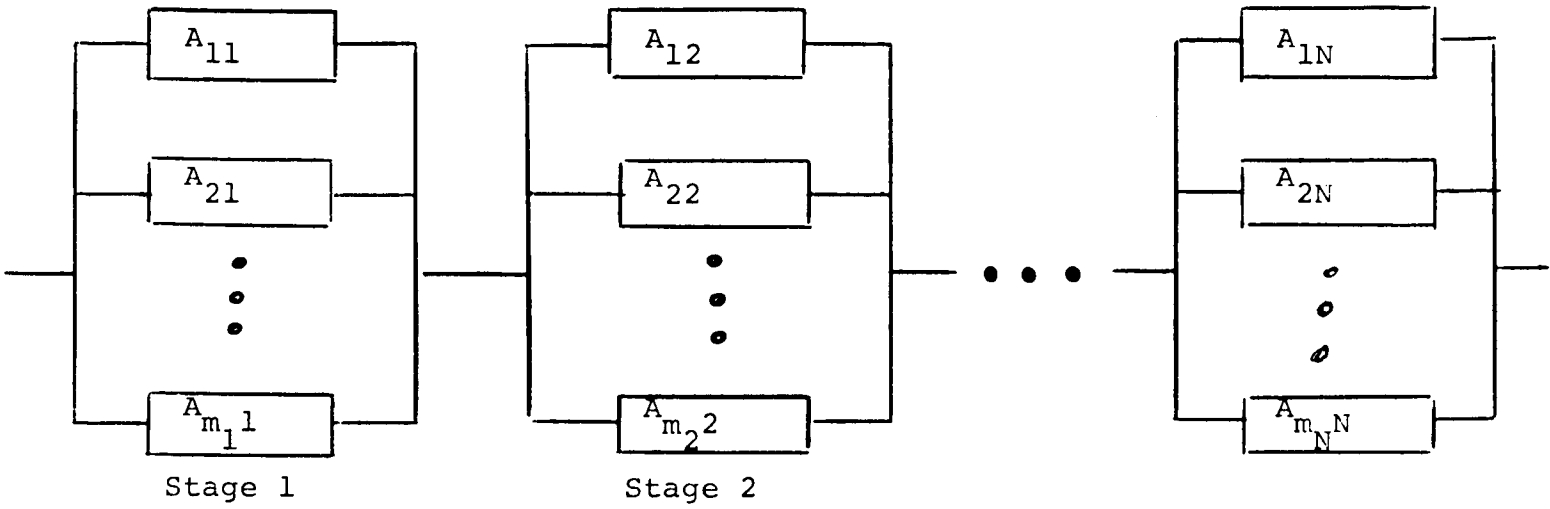


Figure 4.1: Series-Parallel Configuration

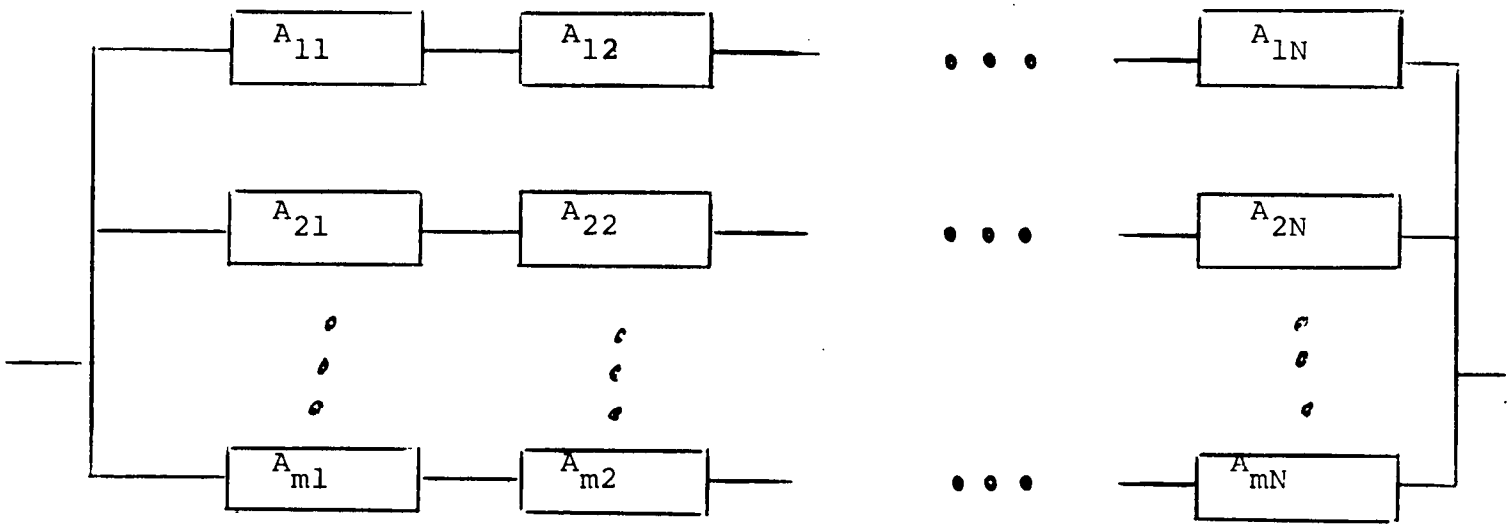


Figure 4.2: Parallel-Series Configuration

Now let us define the following notation:

$p_{ij}(t)$ = the probability that unit A_{ij} operates successfully at time t;

$f(t)$ = the time to failure density function;

$R(t)$ = system reliability at time t;

$\lambda(t)$ = instantaneous failure rate at time t;

$q_{ij}(t) = 1-p_{ij}(t)$

Note that

$$R(t) = 1-\int_0^t f(t)dt = \int_t^\infty f(t)dt$$

We assume that $f(t)$ follows an exponential distribution with $f(t) = \lambda e^{-\lambda t}$ and that λ is constant for all units and constant in time. Then $p_{ij}(t) = \int_t^\infty f(t)dt = e^{-\lambda t}$.

We may now proceed to state the reliability equations for each of the redundancy models. The derivation of these equations may be found in Barlow's Text [6].

$$R_a(t) = \prod_{j=1}^{N} (1 - (1-e^{-\lambda t})^{m_j})$$

$$R_b(t) = \prod_{j=1}^{N} [e^{-\lambda t} \sum_{r=0}^{m_j-1} \frac{(\lambda t)^r}{r!}]$$

$$R_c(t) = 1 - (1-e^{-\lambda Nt})^m$$

$$R_d(t) = e^{-\lambda Nt} \sum_{r=0}^{m-1} \frac{(N\lambda t)^r}{r!}$$

$$R_e(t) = e^{-\lambda Nt}$$

These equations do not take into account the unreliability of failure detectors or switching units used in the given systems. When these factors are included in the equations, a more complex set results. These equations are presented in Barlow's Text [6].

With these equations, we can now give a numerical example and examine the reliability of each model. Suppose a system is designed for use in a space vehicle with an expected mission time of five (5) years. Then t_o=43,800 hours. Let the subsystem failure rate be $\lambda = 2.5 \times 10^{-6}$ failures/hour. Finally, suppose $m = m_i = 3$ for all i=1, ..., N and N = 4. Then

$$e^{-\lambda t_o} = e^{-.10950} = .89628$$

$$R_a(t_o) = .99554$$

$$R_b(t_o) = .99919$$

$$R_c(t_o) = .95538$$

$$R_d(t_o) = .98988$$

$$R_e(t_o) = .64533$$

We see that the greatest reliability is achieved with series-parallel organization and standby redundancy. In fact, series-parallel organization shows a higher reliability than parallel-series, as we would expect. All four redundancy models are more reliable than a non-redundant system.

4-3.0 The Cost Constraint Problem

We will now turn to our main problem, that of determining how many levels of redundancy to use at each system stage so as to achieve the most reliable design while not exceeding certain constraint criteria. These criteria can include system cost, weight, power consumption, volume, etc. However, let us now state the problem more analytically.

Suppose we are given a series-parallel system with active redundancy. There are N stages and m_j identical units in each stage.

$$\text{System Reliability, } R = \prod_{j=1}^{N} (1-q_j^{m_j})$$

The constraints imposed on the system are $C_1, C_2, \ldots, C_r$, where r can be any integer greater than or equal to 1. c_{ij} represents the i<u>th</u> cost parameter for a unit in stage j. We seek to find the m_j, j=1, ..., N such that we maximize R and satisfy the inequalities

$$\sum_{j=1}^{N} c_{ij} m_j \leq C_i, \quad \forall i = 1,2,\ldots,r.$$

The first solution to this problem that we will explore is due to Proschan and Bray [1]. It uses a non-linear programming technique iterating on the number of units allocated to each system stage while not violating any of the constraints. A heuristic is employed to prevent this process from being an exhaustive search procedure.

Fan, et al. [2] employ a maximum principle technique to maximize R. They assume that there is only one linear constraint that cannot be violated. Then to find the m_j, j=1, ..., N, a system of N equations in the N unknowns m_j is numerically solved. The resulting solution yields the optimal redundancy allocation. This method is the second solution to the cost constraint problem that we will present.

4-4.0 Non-Linear Programming Solution

Let $\bar{m}$ represent a reliability allocation $(m_1, m_2, \ldots, m_n)$, and let $C_i(\bar{m}) = \sum_{j=1}^{N} c_{ij} m_j$ be the i<u>th</u> cost of $\bar{m}$. We say $\bar{m}_1$ dominates $\bar{m}_2$ if $C_i(\bar{m}_1) \leq C_i(\bar{m}_2)$, $\forall i=1,\ldots,r$ and $R(\bar{m}_1) \not\geq R(\bar{m}_2)$.

Moreover, $\bar{m}_1$ strictly dominates $\bar{m}_2$ if at least one of the inequalities is strict. We define a dominating sequence S of redundancy allocations as a sequence $\bar{m}_k$, k=1, 2, ..., such that

a. no $\bar{m}_k$ in S is strictly dominated by another in S, and
b. all $\bar{m}_k$ satisfying condition (a) and the r constraints are in S.

Thus, the solution $\bar{m}_j$ to our problem is a member of the dominating sequence S. We must present an algorithm for constructing S and then choosing that member $\bar{m}_j$ that maximizes R.

C.2

We may use an induction principle to construct the dominating sequence for an N stage system. That is, we first construct the sequence for a 2 stage system. Then using the dominating sequence for an h stage system we may construct the sequence for an (h+1) stage system. In this way the full dominating sequence for an N stage system can be constructed. This procedure is an application of the Principle of Optimality, as defined by Bellman [11].

4-4.1 Construction of the Dominating Sequence

Let $Q(\bar{m}) = 1 - R(\bar{m}) = 1 - \prod_{i=1}^{N} (1-q_i^{m_i}) \approx \sum_{i=1}^{N} q_i^{m_i}$. To construct a dominating sequence for the first two stages of the system we set up a table as shown in Figure 4.3. The entry in row m_i and column m_2 is the vector $(C_1(m_1,m_2), C_2(m_1,m_2), \ldots, C_r(m_1,m_2), Q(m_1,m_2))$, computed from the equations for C_i and Q given above. Any strictly dominated entries are then removed from the table. The result is a dominating sequence for the two stage system.

Now suppose we have constructed a dominating sequence for an h stage system, and we wish to go to (h + 1) stages. We construct a similar table. The entry at row m_{h+1} corresponds to m_{h+1} units at stage (h + 1). The entry at column k corresponds to the kth member of the dominating sequence for the first h stages. A table entry at row m_{h+1} and column k is the vector $(C_1(\bar{m}_k, m_{h+1}), C_2(\bar{m}_k, m_{h+1}), \ldots, C_r(\bar{m}_k, m_{h+1}), Q(\bar{m}_k, m_{h+1}))$. As before, we remove any strictly dominated entries from the table and then have a dominating sequence for the first (h+1) system stages. We continue in this manner until the dominating sequence for the full N stages is constructed. From this sequence we find the entry with the lowest unreliability that does not violate a system constraint. This entry corresponds to the optimal value of m_N and a member of the dominating sequence for the (N-1) stage system. We use this latter entry to trace back through the dominating sequences to find the remaining optimal values of $\bar{m}_j$.

Before proceeding to a numerical example using this method, we should mention that several approximations can be employed to shorten the computation time for this algorithm. One such approximation is to limit the size of the dominating sequences to be handled. This may be done by choosing a large initial value of $\bar{m}$ to begin the algorithm. We will call this initial value $\bar{m}^\circ$. A method of computing $\bar{m}^\circ$ will be presented in the next section.

4-4.2 Computation of $\bar{m}^\circ$

To find $\bar{m}^\circ$ we add one unit at each stage of the system until a constraint will be violated by one more addition. We then have a vector $\bar{m}'$. Next compute $R(\bar{m}')$ for the system using the above equation for R. Each m°_i is then found by computing the minimum m_i such that

$$(1 - q_i^{m_i}) \geqq R(\bar{m}').$$

The solution to our cost constraint problem will require an $m_i \geqq m_i^\circ$, $\forall i=1,2,\ldots,N$. We can be sure that m_i will be at least as large as the value obtained by this procedure, because otherwise, the system will be less reliable than that described by $\bar{m}'$ even if all other elements were redundant to the point of perfect reliability. We will now demonstrate this algorithm with a numerical example.

4-5.0 Numerical Example

Suppose we are designing a 3 stage system having series-parallel organization and active redundancy. We are told that cost and weight will be the two constraints within which we must design the system. The unit parameters are the following:

Stage, i	1	2	3	Constraint
Cost	1	2	3	33
Weight	1	1	1	14
q_i	.2	.3	.25	

Thus, we must find $\bar{m}$ such that $R(\bar{m}) = (1-.2^{m_1}), (1-.3^{m_2})$ $(1-.25^{m_3})$ is maximized and

$$m_1 + 2m_2 + 3m_3 \leqq 33$$

$$\text{and } m_1 + m_2 + m_3 \leqq 14$$

We first compute $\bar{m}^\circ$. From Table 1 we see that a constraint is first violated for $\bar{m} = (5,5,5)$. Thus, $\bar{m}' = (5,5,4)$ and $R(\bar{m}')$ $=.9933$. Computing the minimum m_i such that $(1-q_i^{m_i}) \geq .9933$ yields $\bar{m}^\circ = (4,5,4)$.

Using $\bar{m}^\circ$ we now construct a dominating sequence for stages 1 and 2. The result of this construction is shown in Figure 4.3. Each triple in the table represents (cost, weight, unreliability). A crossed out entry means that the entry is strictly dominated by another entry. Hence, the former is removed from the table.

TABLE I

$\bar{m}$	Cost	Weight
111	6	3
211	7	4
221	9	5
222	12	6
322	13	7
332	15	8
333	18	9
433	19	10
443	21	11
444	24	12
544	25	13
554	27	14
555	30	15

Stage 1

Stage 2	(m_1=4)	(m_1=5)	(m_1=6)
(m_2=5)			
(10,5,.00243)	(14,9,.00403)	(15,10,.00275)	~~(16,11,.002494)~~
(m_2=6)			
(12,6,.000729)	(16,10,.002329)	(17,11,.001049)	(18,12,.000793)
(m_2=7)			
(14,7,.0002187)	~~(18,11,.0018187)~~	(19,12,.0005387)	(20,13,.0002827)
			etc.

Figure 4.3: Dominating Sequence Table for Stages 1 and 2

This dominating sequence then becomes the column headings for the table in Figure 4.4. In this table we compute the dominating sequence for the 3 stage system. Only the 4 entries shown in the table are possible. Any other entries would violate a system constraint. Our solution for the optimal m_3 is the entry with lowest unreliability; namely, (29, 14, .005005) which implies the optimal $m_3=5$. Now from Figure 3 we find the optimal $m_1=4$ and $m_2=5$. Thus, the most reliable system that we can design observing the given constraints is given by $\bar{m} = (4,5,5)$.

4-6.0 Maximum Principle Procedure

A different method of solving the cost constraint problem was proposed by Fan et al. [2]. This method uses the Pontryagin Maximum Principle [9] to maximize the net profits for a system while observing a single cost constraint C. Again we assume a system with series-parallel organization.

$$\text{Let } S = PR - \sum_{i=1}^{N} c_i m_i \tag{4.1}$$

where P = profit for reliable system operation and R = system reliability. We seek the $\bar{m}$ which maximizes S and does not violate

$$C \leq \sum_{i=1}^{N} c_i m_i$$

Let us now make some definitions. Let p_i = the reliability of a component in stage i of the system ; then we define

$$\alpha_i = \alpha_{i-1} [1 - (1-p_i)^{m_i}], \quad i=1, \ldots, N$$

$$\alpha_o = 1$$

$$\beta_i = \beta_{i-1} + c_i m_i, \quad i=1, \ldots, N$$

$$\beta_o = 0$$

Now equation (4.1) can be written as

$$S = P\alpha_N - \beta_N. \tag{4.2}$$

Stages 1 and 2

Stage 3	(14,9,.00403)	(15,10,.00275)	(16,10,.002329)	. . .
(12,4,.003906)	(26,13,.00793)	(27,14,.00665)	(28,14,.006235)	
(15,5,.000975)	(29,14,.005005)			
(18,6,.00024)				
.				
.				
.				

Figure 4.4: Dominating Sequence Table for all 3 Stages

We will now state the discrete form of the Maximum Principle here since it is rather lengthy and can be found in Reference 9. However, we will proceed with its application to equation 4.2. The Principle states that the $\bar{m}$ which makes S maximal is the $\bar{m}$ which makes a Hamiltonian function $\bar{H}$ maximal. For this problem the Hamiltonian is

$$H_i = \sigma_i \cdot \alpha_{i-1} \cdot [1-(1-p_i)^{m_i}] - (\beta_{i-1}+c_i m_i),$$

where $\sigma_{i-1}=\sigma_i [1-(1-p_i)^{m_i}]$, $\sigma_N=P$, and $i=1, \ldots, N$.

We will not derive the equation for $\bar{H}$ since it can be found in Reference 9. It is our job then to find $\bar{m}$ such that $\bar{H}$ is maximal. We will do this by solving the set of equations

$$\frac{\partial H_i}{\partial m_i} = 0 \quad \text{for } i=1, \ldots, N.$$

Taking the partial derivatives of H_i with respect to m_i and equating the results to 0, gives us

$$\sigma_i \cdot \alpha_{i-1} \cdot (1-p_i)^{m_i} \ln (1-p_i) + c_i = 0 \qquad (4.3)$$

Note that σ_i can be written

$$\sigma_i = P \prod_{j=i+1}^{N} [1 - (1-p_j)^{m_j}], \qquad (4.4)$$

and α_i can be written

$$\alpha_i = \prod_{j=1}^{i} [1 - (1-p_j)^{m_j}] \qquad (4.5)$$

Substituting (4.4) and (4.5) into (4.3) yields

$$\left[\prod_{\substack{j=1 \\ j\neq i}}^{N} (1-(1-p_j)^{m_j})\right] (1-p_i)^{m_i} = \frac{-c_i}{P \ln(1-p_i)}, \qquad (4.6)$$

for $i=1, \ldots, N$

We now have a set of N equations in the N unknowns, m_j.

To simplify solving (4.6)for the m_j, let $\gamma_i = (1-p_i)^{m_i}$ and

$$\delta_i = \frac{-c_i}{\ln(1-p_i)} . \tag{4.7}$$

Then (4.6) becomes

$$P\gamma_i \prod_{\substack{j=1 \\ j\neq i}}^{N} (1-\gamma_j) = \delta_i \tag{4.8}$$

Note that (4.8) can be rewritten as

$$P \prod_{j=1}^{N} (1-\gamma_j) = \frac{\delta_1(1-\gamma_1)}{\gamma_1} = \ldots = \frac{\delta_i(1-\gamma_i)}{\gamma_i} = \ldots = \frac{\delta_N(1-\gamma_N)}{\gamma_N} \tag{4.9}$$

Numerical solution of (4.9) will yield $\bar{\gamma}$, and hence, $\bar{m}$ follows readily.

4-6.1 Regula Falsi Iteration

A regula falsi iteration [10] can be employed to solve (4.8) for the γ_i which in turn yield the m_i. Let $\gamma_{i,j}$ be the jth iteration of γ_i. With this method each two successive iterative values of γ_i are of opposite sign. (See Figure 4.5). The intersection of the secant line joining these points with the x-axis determines the next iterative value of γ_i. Each successive value $\gamma_{i,j}$ is closer to the desired value γ_i than the last iteration $\gamma_{i,j-1}$. Finally, when an iteration is within a given error bound of γ_i, the process ends. Now let us begin applying this procedure to the problem of the last section. We begin the iteration procedure by choosing values for $\gamma_{1,1}$ and $\gamma_{1,2}$ where $0 < \gamma_i < 1$. Next we compute

$$\rho = \frac{\delta_1(1-\gamma_1)}{\gamma_1} \quad \text{and}$$

$$\gamma_i = \frac{\delta_i}{\rho + \delta_i} \quad ,i=2, \ldots, N., \text{ where each } \delta_i \text{ is given by (4.7).}$$

Finally, S can be computed from

$$S = P \prod_{i=1}^{N} (1-\gamma_i).$$

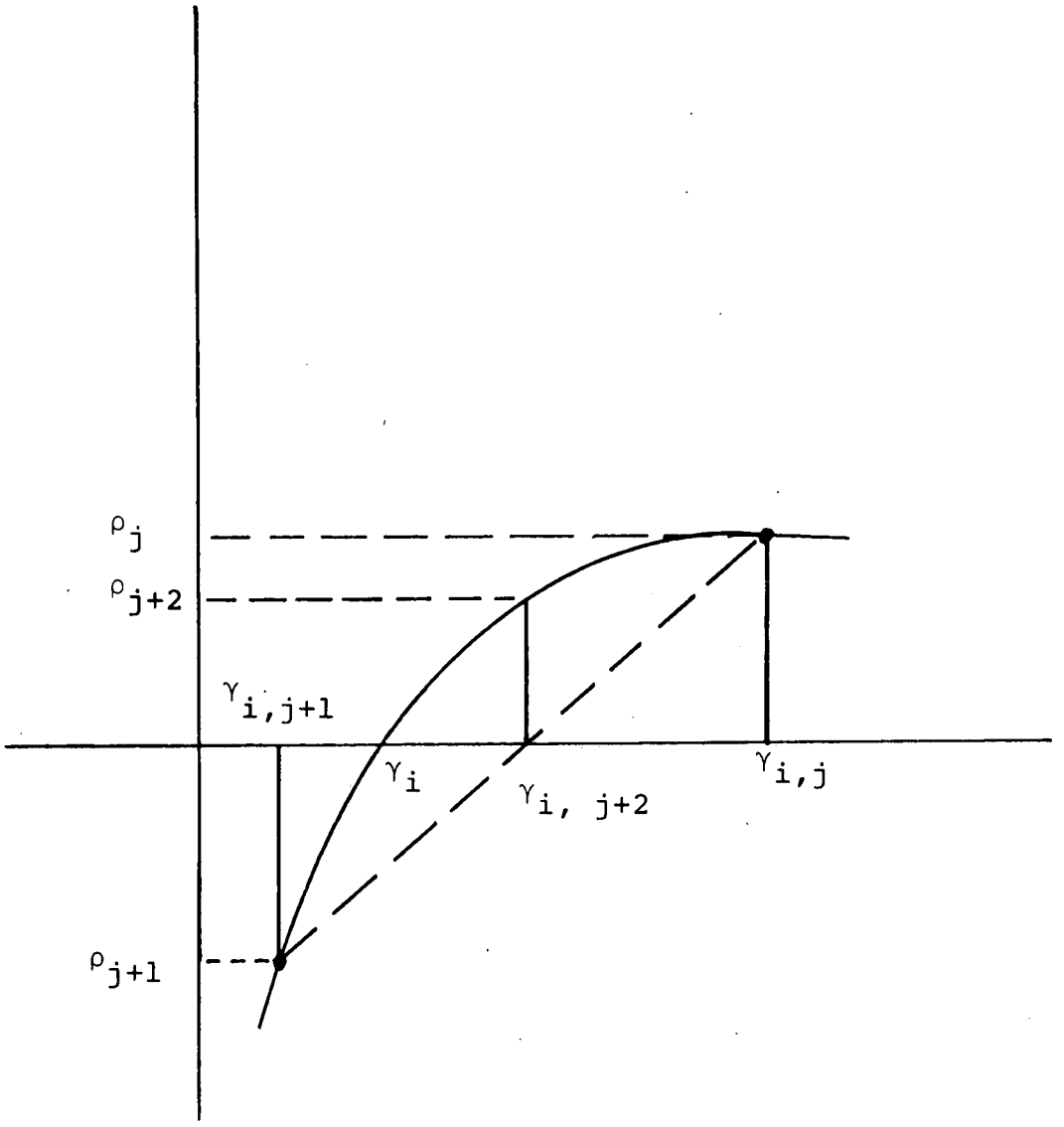


Figure 4.5: Regula Falsi Iteration

If the error $\varepsilon = S-\rho$ is within bounds, we are done and can compute m_i from γ_i. Otherwise, we begin the next step of the iteration by calculating

$$\gamma_{1,j+2} = \frac{\gamma_{1,j}\ \rho_{j+1} - \gamma_{1,j+1}\rho_j}{\rho_{j+1} - \rho_j}$$

where $$\rho_j = \frac{\delta_1\ (1-\gamma_{1,j})}{\gamma_{1,j}}$$

We then repeat the above procedure using $\gamma_{1,2}$ and $\gamma_{1,3}$ and again determine the error.

Finally, when we have iterated sufficiently many times so that the error is within bounds, we are done. We can then easily calculate $\bar{m}$ from $\bar{\gamma}$.

The advantage of this cost constraint algorithm is that lengthy dominating sequences do not have to be calculated. $\bar{m}$ is found by solving N equations in N unknowns by numerical methods. This algorithm can easily be implemented on a digital computer. The disadvantage is that only one cost constraint can be taken into account in designing the system.

4-7.0 Example

This algorithm was applied to an 8 stage system having the following characteristics:

Stage, i	p_i	c_i
1	.90	.5
2	.75	.4
3	.65	.9
4	.80	.7
5	.85	.7
6	.95	.4
7	.75	1.0
8	.60	.8

with P = 100.0

The solution found was

$\bar{m}$ = (3,4,5,3,3,2,4,5) and S = 75.64.

REFERENCES

1. Proschan, F. and Bray, T., "Optimum Redundancy under Multiple Constraints", (Operations Research, 13, Sept. - Oct. 1965), pp. 800-814.

2. Fan, L.T., et al., "Optimization of Systems Reliability", (IEEE Trans on Reliability, R-16 (2), Sept. 1967), pp. 81-86.

3. Bellman, R., and Dreyfus, S., "Dynamic Programming and the Reliability of Multicomponent Devices", (Operations Research, 6, March - April 1958), pp. 200-206.

4. Black, G., and Proschan, F., "On Optimal Redundancy", (Operations Research, 7, Sept. - Oct. 1959), pp. 581-588.

5. Selman, V., and Grisamore, N., "Optimum System Analysis by Linear Programming", (Proc., 1966 Annual Symposium on Reliability), pp. 696-703.

6. Barlow, R., and Proschan, F., Mathematical Theory of Reliability, (John Wiley and Sons, New York, 1965).

7. Garvin, W.W., Introduction to Linear Programming, (McGraw-Hill, New York, 1960).

8. Arrow, K. L., et al., Studies in Linear and Nonlinear Programming, (Stanford Univ. Press, Stanford, California, 1958).

9. Fan, L.T., and Wang, C.S., The Discrete Maximum Principle, (John Wiley and Sons, New York, 1964).

10. Hildebrand, F., Introduction to Numerical Analysis, (McGraw-Hill, New York, 1956).

11. Bellman, R., and Dreyfus, S., Applied Dynamic Programming, (Princeton Univ. Press, Princeton, New Jersey, 1962).

CHAPTER 5

QUEUEING THEORY

5-1.0 Discussion

Queueing theory deals with congestion in systems consisting of networks of service facilities. It is a relatively recent branch of probability theory, generally considered to date from the work of A. K. Erlang in the period 1900-1920, in the service of the Copenhagen Telephone Company. The most important abstractions involved in queueing theory involve customers, servicers and queues.

Customers arrive, wait in a queue and when the servicer is free, are themselves serviced and then leave the system. Queueing theory obtains analytic solutions to the probability distributions of variables that measure congestion in the system such as length of queues, time spent waiting and so forth.

Computing systems have been successfully modelled as queueing systems. The purpose is to predict performance, and to identify and correct efficiency problems in the system. In a computing system, the customers are the stream of jobs arriving at the system interface, and the servicers are the collection of equipment, processors, file storages, data links, printers, etc., and their interconnections.

Using queueing theory, partial analytic solutions have been developed for quite a wide variety of problems. This chapter will serve to acquaint the systems analyst with some of the most important analytic results and with the methodology associated with the theory.

For many applications an analyst can be satisfied with solutions which describe the behavior of a model in statistical equilibrium. Statistical equilibrium occurs when the probability distribution functions which describe the random variables in the system are static. Statistical equilibrium does not mean that the variables themselves are static. In an ergodic system, the variables will undergo constant and unpredictable dynamic change, "filling in" over a sufficiently long period, a pattern whose statistics approach a static distribution. It is fortunate that statistical equilibrium solutions are useful because with few exceptions, that is all that is available. We will discuss this briefly later in the chapter.

Furthermore, it is often considered acceptable to know the first and second moments of a distribution, rather than a more complete description. Even so, the mathematics involved in developing analytic solutions is burdensome, and for complicated problems, overwhelming. At some point, the analyst must choose between reducing the fidelity of his model, or abandoning analytic approaches and using iterative or simulation techniques to obtain the data he needs. Subsequent chapters discuss the second choice.

5-2.0 Erlang's Model

Many of the concepts and techniques of queueing theory appear in a discussion of a model consisting of a single server, servicing a waiting line fed by a randomly arriving customer stream. It is convenient, and often realistic to assume that customer arrivals satisfy a Poisson probability distribution function. We will describe the Poisson process here because it exposes some of the mathematical techniques of queueing theory.

A Poisson distribution results from assuming that the chance that a customer arrives in a short time interval Δt is $\lambda\Delta t$, where λ is constant. The chance that two or more customers arrive in Δt is presumed to be proportional to higher orders of Δt and therefore negligible for sufficiently small Δt. Let $P_m(t)$ be the probability that exactly m customers have arrived by time t. During a subsequent small interval Δt customers arrive according to the Poisson assumption. Then at $t + \Delta t$ we can state

$$P_o(t+\Delta t) = P_o(t)(1-\lambda\Delta t)$$

$$P_m(t+\Delta t) = P_m(t)(1-\lambda\Delta t)+P_{m-1}\lambda\Delta t \qquad m\geq 1$$

and in the limit as $\Delta t \to 0$,

$$dP_o/dt = -\lambda P_o$$

$$dP_m/dt = -\lambda P_m + \lambda P_{m-1} \qquad m\geq 1$$

These are linear, first order differential equations with respect to t, and linear first order difference equations with respect to m, generally called differential difference equations. Solutions to the above equations can be obtained by sequentially solving each equation and substituting the solution into the equation for next

higher m.* The general solution, defining the probability that m customers arrive in an interval t is

$$P_m(t) = \frac{(\lambda t)^m e^{-\lambda t}}{m!} \qquad m \geq 0$$

A typical graph of this distribution function for $\lambda t=4$ is shown in Figure 5-1.

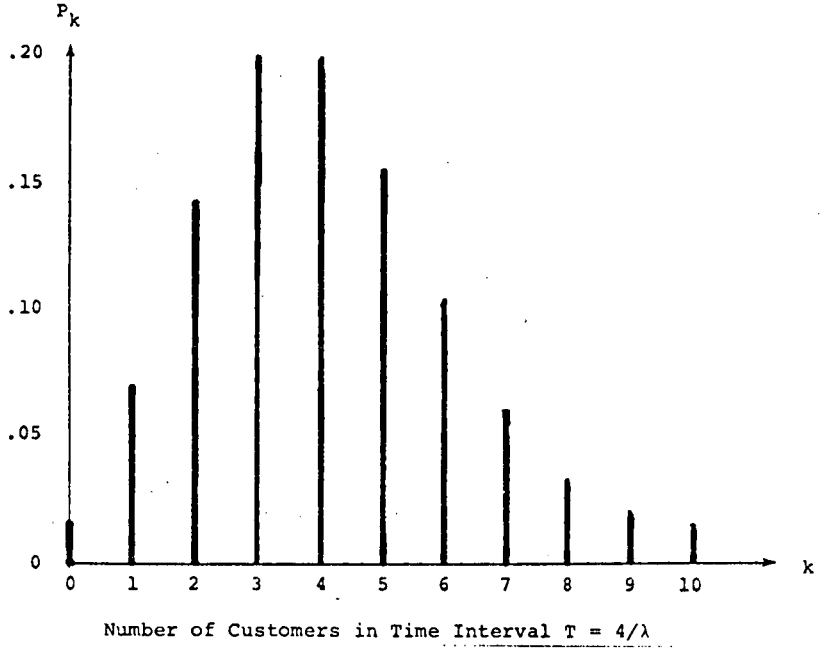


Figure 5-1 Poisson Distribution

In Erlang's model, the Poisson customer arrival assumptions are made.

Service time is the amount of time to service a customer. In Erlang's model it is a random variable governed by the same basic assumptions that governed the arrival of customers. The probability that a customer completes service in a small interval Δt is $\mu\Delta t$, where μ, the service rate, is constant.

Customers wait in line for prior customers to complete service. Using the same procedures used to develop the Poisson distribution, we obtain an expression for the probability that m customers are in the system at $t+\Delta t$, given the distribution at t.

$$P_o(t+\Delta t) = P_o(t)(1-\lambda\Delta t) + P_1(t)\ \mu\ \Delta t$$

$$P_m(t+\Delta t) = P_m(t)(1-(\lambda+\mu)\Delta t) + P_{m-1}(t)\ \lambda\Delta t + P_{m+1}(t)\mu\ \Delta t \qquad m \geq 1$$

*Another method of solution involves the use of generating functions, which are discussed in chapters 6 and 7.

yielding as $\Delta t \to 0$,

$$dP_o/dt = -\lambda P_o + \mu P_1 \qquad (5\text{-}1)$$

$$dP_m/dt = -(\lambda+\mu)P_m + \lambda P_{m-1} + \mu P_{m+1} \qquad m \geq 1$$

The solution of these equations cannot be obtained by a sequential process, since the interdependency between equations extends in both directions.

However, methods of solution, using generating functions are known and the solution is

$$P_m(t) = e^{-(\lambda+\mu)t} \left[\rho^{m/2} I_m(x) + \rho^{(m-1)/2} I_{m+1}(x) + \rho^m (1-\rho) \sum_{r=2}^{\infty} \rho^{-(m+r)/2} I_{m+r}(x) \right]$$

where

$\rho = \lambda/\mu$ (called traffic intensity)

$x = 2t\sqrt{\lambda\mu}$

$I_k(x)$ = Bessel functions of imaginary argument

Lee, in reference 2 comments, "confronted with this expression, the reader, mindful that this system is the simplest of all, will experience a feeling of alarm. This will only be increased when we point out further that whilst the solution describes the growth of the queue of number of customers present, it tells us nothing of the waiting times during the transitional period. To obtain these, we would have to consider expressions of even more complex form. If such are the difficulties of general solutions in the simplest of models, what will the solutions to more elaborate models be like? This is answered by the first working rule of queueing theory: time dependent solutions to queueing models are either unobtainable or unmanageable."

Steady state solutions to the differential equations are easy to derive and simple to use. We obtain them by setting derivatives to zero in equation 5-1. These equations become

$$(1 + \rho)P_n = P_{n+1} + \rho P_{n-1} \qquad n \geq 1$$

$$P_1 = \rho P_o$$

again using $\rho = \lambda/\mu$. ρ is a commonly used parameter, called traffic intensity, since it describes the average rate of customer arrivals normalized by the average rate that they can be serviced. To solve these equations, consider the member of n=1

$$(1 +\rho)P_1 = P_2 + \rho P_o$$

Eliminate P_1 by substitution using the relation between P_1 and P_o, yielding

$$P_2 = \rho^2 P_o$$

Repetition of the process yields

$$P_n = \rho^n P_o \qquad n \geq 1$$

P_o can be obtained by noting that

$$\sum_{n=0}^{\infty} \rho^n P_o = 1 = \frac{P_o}{1-\rho}$$

or

$$P_o = 1 - \rho$$

hence

$$P_n = \rho^n (1 - \rho)$$

is the probability distribution for the number of customers in the system in the steady state.

5-3.0 Measures of Congestion

The mean queue size is

$$L = \sum_{n=0}^{\infty} nP_n = \frac{\rho}{1-\rho}$$

and the variance is

$$V = \sum_{n=0}^{\infty} n^2 P_n - L^2 = \frac{\rho}{(1-\rho)^2}$$

The mean length of the queue grows without bound, as does the variance, as traffic intensity approaches 1. (For our purpose the queue includes the customer being serviced.)

5-3.1 Queueing Time

The next task is to derive the time that a customer would spend in the system. A customer arriving with no one ahead of him would experience a service time distribution of $\mu e^{-\mu t}$. With customers ahead of him, his total wait is the sum of the service times for him and the others ahead of him. The probability distribution for the sum of these times is best found by the Laplace transform. This is because, since the computation of the distribution of a sum of random variables involves convolution, the Laplace transform of the distribution of the sum is the product of the Laplace transforms of the distribution of the addends.

The Laplace transform of the probability distribution for the time to service one customer is

$$P(s) = \int_0^{\infty} \mu e^{-\mu t} e^{-st} dt = \frac{\mu}{\mu+s}$$

Consequently, the transform of the distribution for the sum of r customer services is

$$\left(\frac{\mu}{\mu+s}\right)^r$$

The transform of the distribution for the waiting time is obtained by weighting the wait time as a function of r, the length of the queue by multiplying by the probability of occurrence.

$$\sum_{r=0}^{\infty} (1-\rho)\rho^r \left(\frac{u}{u+s}\right)^{r+1} = \frac{u(1-\rho)}{u(1-\rho)+s}$$

But this is the Laplace transform of

$$(u-\lambda)\ e^{-(u-\lambda)t}$$

which therefore is the probability distribution of waiting times in the steady state.

The mean and variance of waiting time are given by

$$\text{Mean} = \frac{1}{u(1-\rho)}$$

$$\text{Variance} = \frac{1}{u^2(1-\rho)^2}$$

5-4.0 More Complex Models

The previous section has developed the most commonly desired measures of congestion for the simplest non-trivial queueing model, in statistical equilibrium. We will now discuss briefly extensions of the theory towards more realistic and/or more complex models.

5-4.1 Random Arrivals/General Service

The assumption of exponential service time in the Erlang model is often considered unrealistic. While arrivals can often be approximated by Poisson distribution, service times are rarely exponentially distributed.

The analysis of the mean and variance parameters for a simple single server queue with a generalized service time probability distribution has been solved, and are known as the Pollaczek-Khintchine formulae, Ref. [1]. It turns out that the mean queue statistics depend only on the first two statistical moments of the service distribution. Reference 3 contains the formulae. They are derived in reference 1.

5-4.2 Erlang - m Distributions

Another approach to the assignment of more realistic service time distributions involves the use of Erlang-m distributions. This distribution can be considered to be the probability distribution of

a servicer which consists of m substages of service each with an exponential service distribution, whose service rates are m times the overall rate.

The resulting sequence of distributions all have the same mean service time, but as m increases, the variance decreases, and the probability distribution becomes more and more concentrated about the mean value. In the limit as $m \to \infty$, the distribution becomes a unit impulse, representing constant service of length 1/u. Figure 5-2 illustrates Erlang m distributions for m=1, 2 and 10.

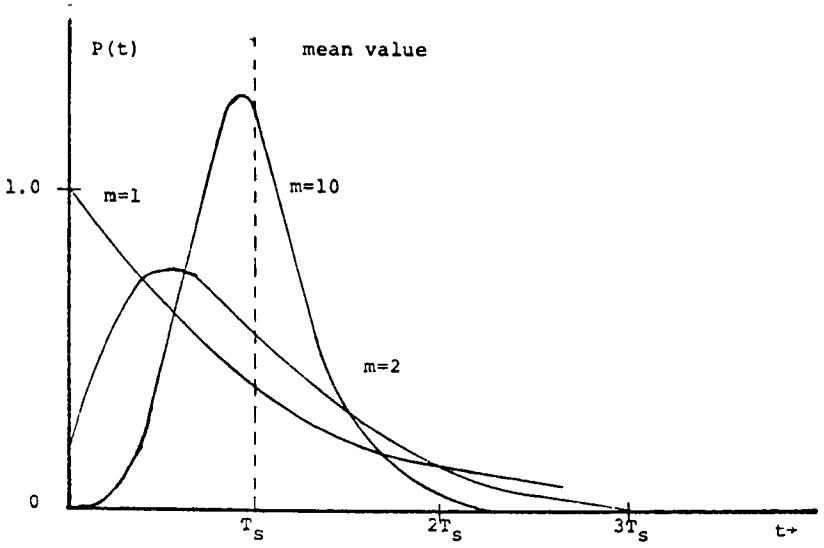


Figure 5-2 Erlang-m Probability Distribution

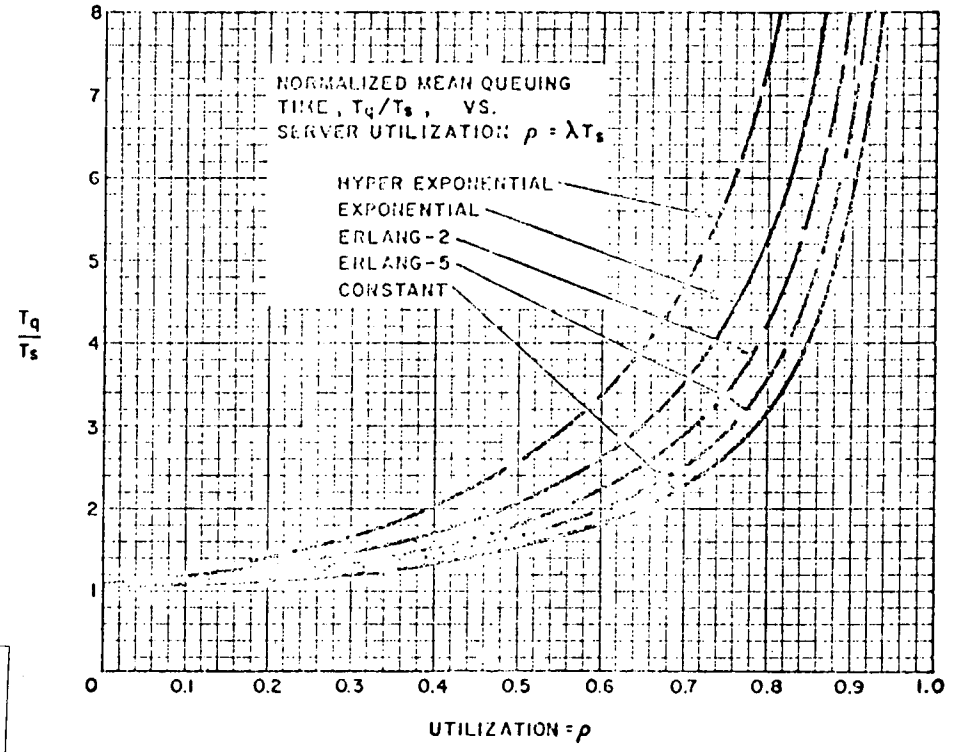
Figure 5-2 Erlang-m Distribution

Figures 5-3 and 5-4, extracted from Reference 3 summarize congestion parameters for a single server model.

5-4.3 The Effect of Queueing Disciplines

We have implied that customers are served in order of arrival, commonly called the FIFO, first-in, first-out queueing discipline.

Other queueing disciplines have been studied. It is interesting to note that the mean length of the queue and the average time spent waiting in queue are independent of the queue discipline.



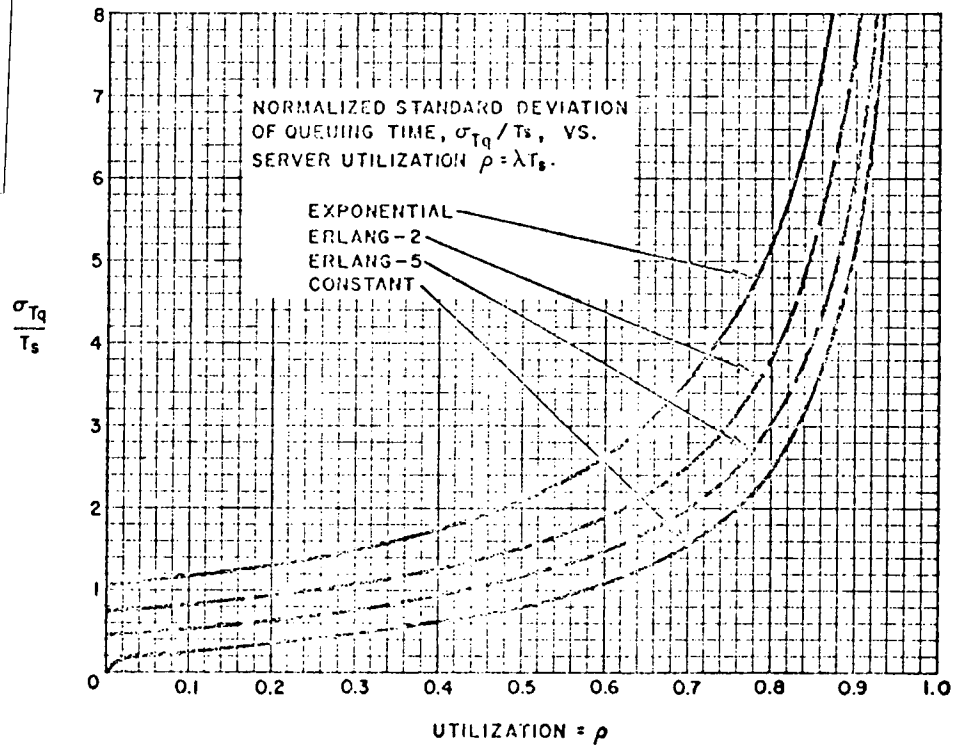
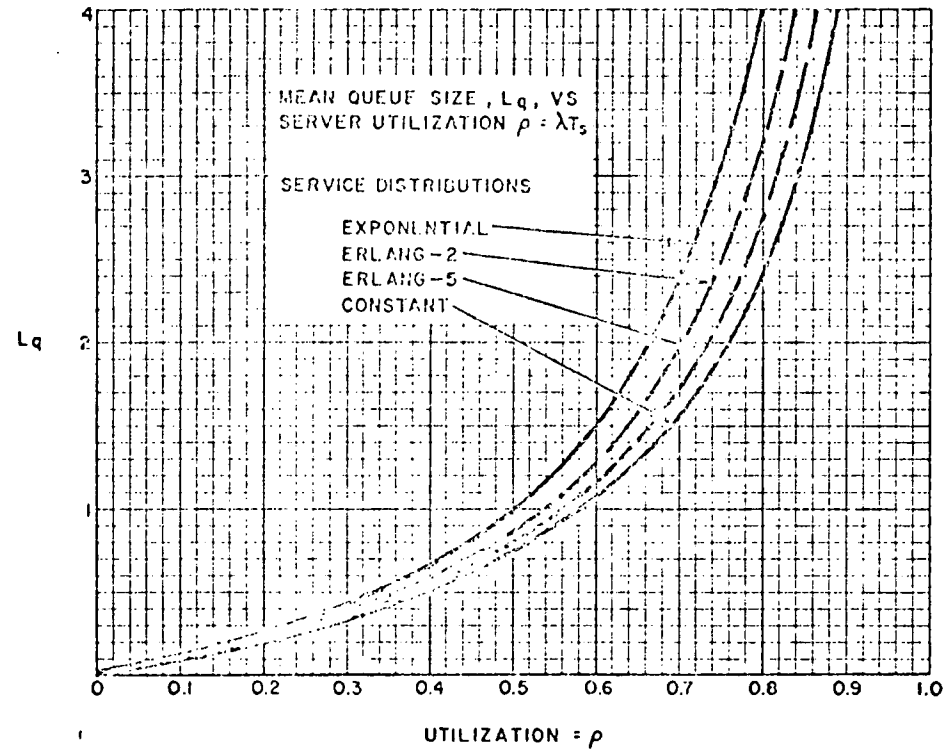


Figure 5-3 Queueing Time



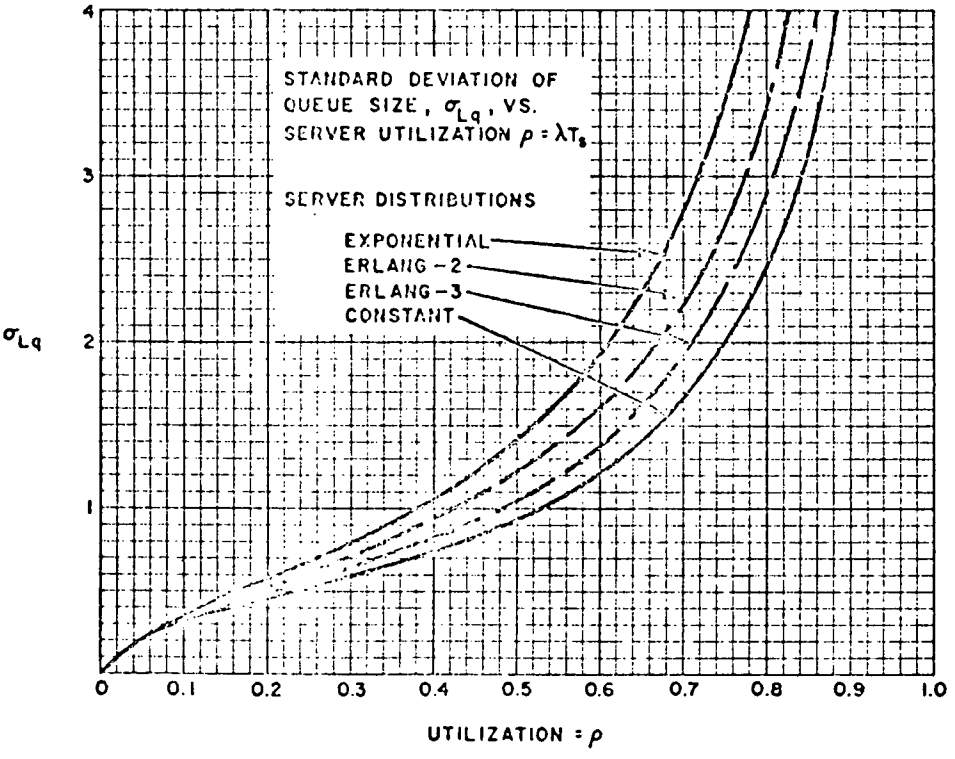


Figure 5-4 Queue Length

However, the waiting time distribution is a function of discipline, with FIFO producing the smallest variance.

5-5.0 Priority

Assignment of relative priorities to types of customers can have a dramatic effect on congestion in queues. We will discuss two simple priority concepts. In the first concept, a customer once he begins service, continues to be serviced. The next customer to be serviced is the customer with the greatest priority. If there is more than one such customer, the one who has waited longest is selected.

This is called non-preemptive priority. A second concept permits a customer of greater priority to displace the customer being serviced. The displaced customer is placed at the head of the line for his priority. This is called preemptive priority.

For non-preemptive priority, the mean waiting time before a customer with priority j begins service, where increasing j denotes decreased priority with j=1 greatest, can be derived, Ref. [1], as

$$\bar{T}w_j = \frac{\lambda \ var\ (T_s)}{2(1-u_{j-1})(1-u_j)}$$

where

λ = mean of total arrival rate over all priorities

$var(T_s)$ = variance of service rate over all priorities

$u_j = \rho_1+\rho_2+\ldots+\rho_j$, traffic intensity for j highest priorities

With preemptive priority, the queueing time is

$$\bar{T}_{qj} = \frac{1}{1-u_{j-1}} \left[\frac{1}{u_j} + \frac{\sum_{j=1}^{j} \lambda_i \ var(T_s)_j}{2(1-u_j)} \right]$$

Reference [3] uses these formulas in an example of a message processing center being fed by a random traffic stream, consisting of two types of messages. Type 1 requires a short constant process

time. Type 2 requires a much longer process time which is exponentially distributed. The actual parameters are

Type 1 message: $\lambda_1 = .5$ msg/sec

$\bar{T}_s = .2$ sec

$\bar{T}_s^2 = .2^2$ sec^2

Type 2 message: $\lambda_2 = .1$ msg/sec

$\bar{T}_s = 5$ sec

$\bar{T}_s^2 = 2\bar{T}_s^2 = 50$ sec^2

Some comparitive results are presented in Table 5-1.

	NO PRIORITY	NON-PREEMPT PRIORITY	PREEMPT PRIORITY
Type 1 msg	6.45 sec	3.0 sec	.211 sec
Type 2 msg	11.25 sec	12.0 sec	12.5 sec
OVERALL	7.25 sec	4.5 sec	2.26 sec

Table 5-1 Effect of Priority on Mean Time in System

Note that preemptive priority method greatly reduces the backlog (both delay and queue length) for the type 1 messages with relatively small effect on the delay and queue length of the longer type 2 messages.

5-6.0 Networks of Queues

Solutions to problems involving a network of interconnected queues can be solved if the job stream from queue to queue retains a Poisson distribution. In that case the individual queues can be separately analyzed. In order for the job streams to retain their Poisson character, input job streams must be Poisson, the waiting lines must be infinite (no jobs lost) and the service time must be exponentially distributed.

Under these ground rules, multiple server queues, such as Figure 5-5 have an output distribution equal to the input. Job streams that partition and merge, as in Figure 5-6 retain the Poisson character. When a partition occurs, the mean traffic rates of a partitioned path are proportional to the probability that that path is taken. On a merge, the mean rate is the sum of the merge rates.

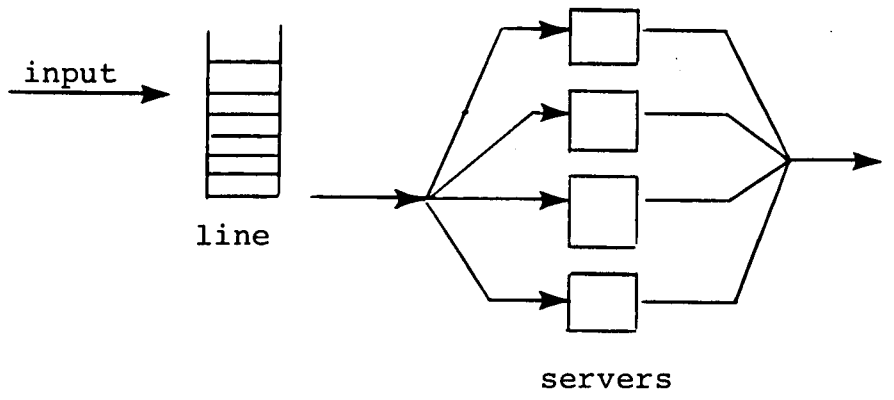


Figure 5-5 Multiple Server Queue

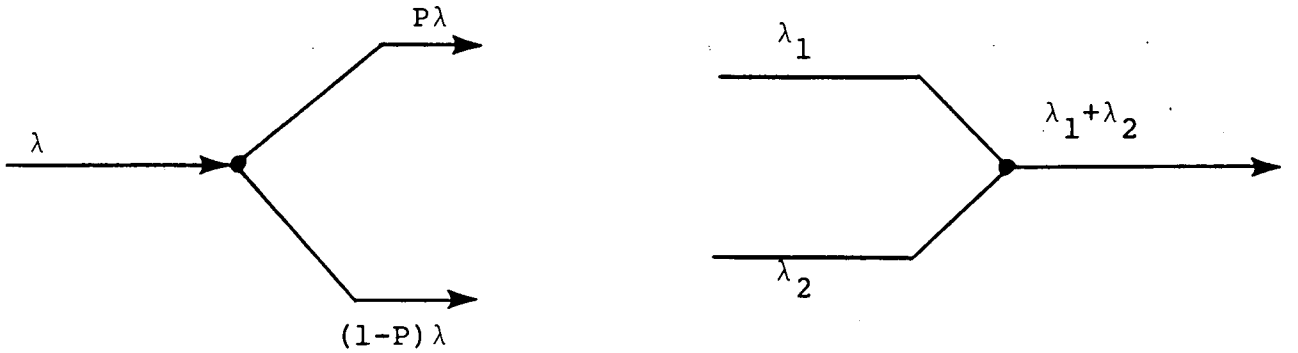


Figure 5-6 Branch & Merge Points

Under these circumstances, elements of the network can be individually analyzed and the results for the total system obtained by statistical means. Generally speaking, the calculation of the combined statistics involves Laplace transform techniques.

Networks that do not permit the assumption that each partition is a Poisson input, exponential service, infinite length queue are likely to be unsolved.

5-7.0 Finite Queues

Finite waiting lines are easy to model as a modification of Erlang's Model. For a line of maximum length L, the probability that the line has length k is

$$P_k = \rho^k \frac{1-\rho^{L+1}}{1-\rho} \qquad k \leq L$$

where ρ is the traffic intensity λ/u. Other statistical measures of congestion follow in straightforward fashion.

However, solutions to more complex networks or more realistic presumptions about the service time are not easily constructed, assuming limited queue lengths. It is often acceptable to approximate using infinite queue lengths and careful interpretation of the resulting solution.

5-8.0 Summary

Queueing theory, although new, has been rapidly exploited to yield a large body of solved models. With some experience a user can often obtain useful results by modeling his problem as a solved problem. A most useful handbook is Reference [3].

Queueing theory provides excellent insight into the frequently observed characteristics of congestion and bottlenecking in computer systems. In particular, the non-linear and divergent increase in congestion as traffic intensity approaches 1, and the even more dramatic increase in variance of congestion parameters, as a function of traffic intensity are most important. Analytical insight into the effect of priority algorithms is also a very significant product of queueing theory for the computing system analyst.

REFERENCES

1. Saaty, T.L., Elements of Queueing Theory with Applications, (McGraw-Hill, New York, 1961).

2. Lee, A.M., Applied Queueing Theory, (MacMillan, Toronto, St. Martins Press, New York, 1966).

3. IBM, Analysis of Some Queueing Models in Real Time Systems, (GF20-0007-1, Technical Publications Department, 112 East Post Road, White Plains, New York, 10601, November, 1969).

CHAPTER 6

MACROSIMULATION

6-1.0 Introduction

The concept of models is a unifying technique common to all eras and all branches of science and engineering. Simulation, a form of modeling, occupies a central role in modern engineering. Computer systems are both vehicles for and objects of simulation. Computers are used to simulate spacecraft trajectories, industrial plants, economic policy, fast food franchises, ..., and computers. The fundamental technique of modeling or simulation is to isolate and define the collection of (system of) elements relevant to the problem at hand, and to exercise these models as a learning step in the solution of the problem.

Simulation of computer systems can occur in several forms. We have chosen to describe in this report two levels of computer system simulation. Chapter 17 discusses the use of simulation as a design and verification tool for computer controlled systems. This use of simulation emphasizes detailed replication of the operation by operation sequencing of the computer, and detailed modeling of the environment that interacts with the computer logic. We have chosen to call this microsimulation of computers. This chapter discusses another application of simulation in the development of computer based systems. We will be concerned here with abstractions of computer systems, designed to expose and analyze critical design parameters. Generally speaking, these simulation techniques deliberately suppress design detail, and concentrate on broadly defined measures of system effectiveness. For this reason, we have chosen to categorize these simulation techniques as macrosimulation.

Computer simulation at this level has its basis in queueing theory, the probabalistic analysis of the interaction between users and facilities. The role of simulation is to exercise user and facility interactions whose complexity exceeds the bounds of known or feasible analytic solutions, by Monte Carlo methods.

Digital computer facilities have long exhibited the symptoms dear to the queueing analyst; namely, bottlenecks. The reader will probably have personal familiarity with situations where a data processing facility has become hopelessly inefficient due to one or a combination of bottleneck elements.

Presumably, the objective of simulation is to obtain advance notice of the performance of a computing facility at the design stage. To be successful, the simulation anticipates the way the system would work if it were built. The successful simulation designer must accomplish all of the following steps:

1. He must satisfy himself that simulation is an appropriate analytical method, and that the elements of the system and the job stream are sufficiently defined.

2. He must verify that the results of the simulation are correct, and that they are appropriate to the purpose.

3. He must explain his results and proselytize his conclusions in order to affect future events in a constructive way.

These generalizations are noted here because there seems to be a general unease among professional personnel that the technique of simulation has been often misused, and often because of neglect of the fundamentals listed above.

Macrosimulation of computer systems are discrete simulations. Such simulations are characterized by a time ordered sequence of events whose occurrence, and time of occurrence is predestined either by deterministic formulae or by the result of a pseudo-random process. The simulation consists of computing the sequence of events that characterize the movement of users through the various facilities of the system. In a computing system, users are elements of the job stream. Facilities might be processing units, working memory, data links and so forth. Generally speaking, such simulations involve macroscopic levels of description of system interactions. Thus, interactions between jobs and facilities are often defined in terms of the amount of time a facility is occupied, on the amount of core that is pre-empted in accomplishing a computing event. The level of detail to be achieved by the simulation is actually unlimited, and must be related back to the purpose of the simulation. It is both a practical and aesthetic rule to minimize the level of complexity of any simulation.

The results of the simulation are compiled by defining measures of effectiveness of the system in responding to the job stream. If there are random processes involved, a sufficient ensemble of data must be computed to obtain statistically valid results. Often the purpose is to evaluate a steady state job stream. In this case, a single run can be sufficient, provided that some means is found to verify that there is a steady state condition, and that some means is used to gather the data after a steady state has been reached.

Discrete simulations can be generalized into a collection of elements and operators. This kind of generalization permits the development of high order programming languages for discrete simulation. As an example of such a language, we will briefly discuss GPSS, a language developed by Gordon of IBM.

GPSS deals in transactions, events, facilities, storages, and queues. A transaction is generated for each element in the job stream. Events mark the movement of the transaction through the system of facilities, storages and queues. A facility is a system element that can accomodate only one transaction at a time. A storage is a system element that can accomodate many transactions, up to a specified limit, at a time. A queue is a waiting line. Gordon, Ref. [2], gives examples of these concepts as they might occur in different systems:

TYPE OF SYSTEM	TRANSACTION	FACILITY	STORAGE
Communications	Message	Switch	Trunk Lines
Transportation	Car	Toll Booth	Road
Data Processing	Record	Key Punch	Memory

6-2.0 Example

We are interested in the performance of a computing system that consists of a processor that completes service on a job in multiples of one time unit. At each time unit the job either reoccupies the processor again or is completed. The probability of branching back for recomputation is .5, independent of the number of times that the job recycles. New jobs arrive at the computing system with probability .1 at each time interval.

There is a buffer space sufficient to hold one incoming job if the processor is occupied when a new job arrives. If the buffer space is also filled, the arriving job is lost.

The intent of the simulation is to find out what percentage of jobs is lost. We will sketch out the simulation of this system using GPSS.

GPSS is a block diagram oriented language, designed to permit development of the program by an intermediate step of creating a block diagram that graphs the sequential events that mark the progress of a transaction through the facility. Our problem will be simulated by defining a storage sufficient to accomodate two transactions (two jobs), and one facility, which will be the processor. We will call the storage; core. We will create jobs by generating a transaction every 1 time unit, but we will discard nine-tenths of these jobs, on a random basis, to

account for the .1 probability that a new job will arrive at a given interval. A GPSS block diagram for this section of the simulation is shown in Figure 1.

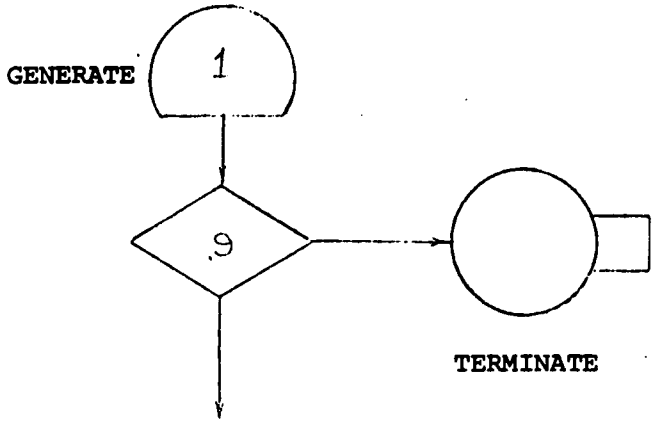


Figure 6.1 - Creating the Job Stream

Next, the transaction attempts to occupy core. If it succeeds, it attempts to occupy the processor. The declaratives enter and leave are associated with storages. An enter increments a counter associated with the capacity of the storage, and leave decrements the counter. The declaritives seize and release are used for facilities, which can accomodate only one transaction at a time.

Figure 2 is a complete block diagram. In this diagram we have used a conditional transfer test (BOTH) which attempts to pass the transaction into core, but, if that storage is full, it passes the transaction to a terminate block. Having entered core, the transaction attempts to seize the processor. If it succeeds it will, after one time unit, signified by the advance block, either release the processor and leave core, or return to occupy the processor for another time unit.

The program will maintain records of where each transaction is in the system, and when it is due to move. It proceeds by completing all events that are scheduled for a particular time and that can logically be performed. Where there is more than one transaction due to move, the program processes transactions in the order that they were generated. This order is sufficient in our problem to assure that it will execute properly, since we wish to be assured that the simulation will first process a transaction that is occupying the processor, thus possibly releasing the processor facility and opening up a sapce in core storage before it generates a new job at the otherwise simultaneous time unit interval. There is provision sion within GPSS for altering the priority of a transaction, if necessary, in order to dictate the order in which otherwise simultaneous events will occur.

The transaction will succeed in seizing the processor when it is released by the previous job.

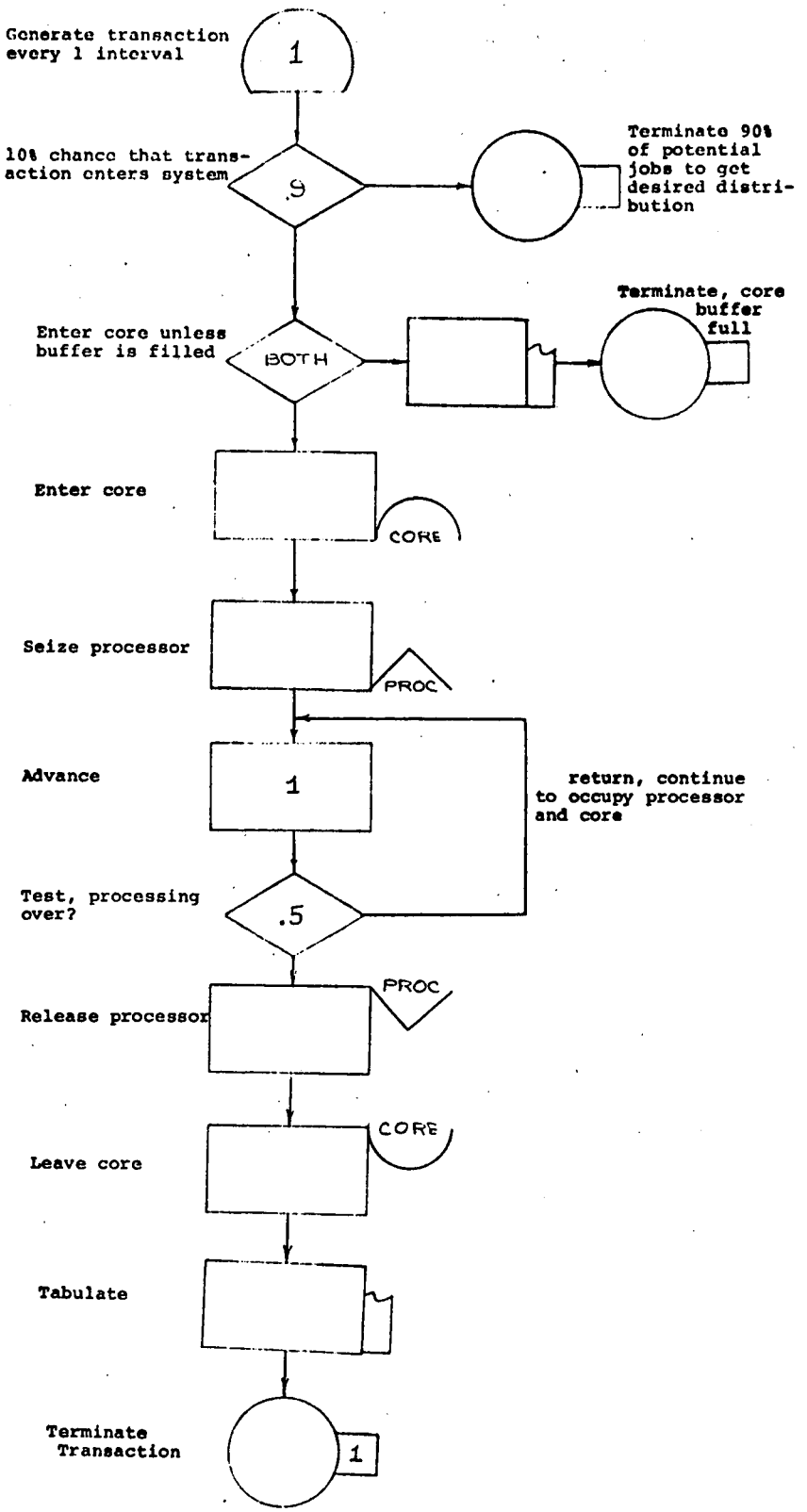


Figure 6.2 - Complete Block Diagram

The object of this simulation is to observe the number of jobs that terminate at 1 (core buffer occupied), relative to the number that terminate at 2, (job complete). To gather these statistics, we must insert tabulate blocks in the paths to these termination blocks, and create tables which output these statistics.

```
          GENERATE     1                 NEW TRANSACTION EACH INTERVAL
          TRANSFER     .9, NOJOB, JOB    NEW JOB DISTRIBUTION
JOB       TRANSFER     BOTH, NOCORE      CHECK FOR AVAIL. CORE
          ENTER        CORE              OCCUPY CORE SPACE
          SEIZE        PROC              OCCUPY PROCESSOR IF FREE
CONT      ADVANCE      1                 COMPUTE
          TRANSFER     .5, CONT, DONE    CHECK FOR FINISH
DONE      RELEASE      PROC              FINISH, RELEASE PROC
          LEAVE        CORE              FREE UP CORE
          TABULATE     DONE              TABULATE TIME IN SYSTEM
          TERMINATE    1                 COMPLETED JOBS END HERE

NOJOB     TERMINATE                      NOT A JOB

NOCORE    TABULATE     LOST              TABULATE TIME FOR LOST JOB
          TERMINATE

DONE      TABLE        M1                TIME IN SYSTEM TO FINISH
LOST      TABLE        M1                TIME IN SYSTEM TILL LOST

CORE      STORAGE      2                 NUMBER OF CORE SPACES

          START        50, NP            RUN 50 JOBS, NO PRINT
          RESET                          WIPE OUT STATISTICS
          START        1000              RUN 1000 JOBS THROUGH
```

This example is identical to example 2 in Chapter 8 and if properly coded should yield similar results. The examples were deliberately made identical so that the reader could compare the analytical approach with the method of simulation. Analytical methods, of course, are limited to problems which satisfy the assumptions necessary to derive the method of solution, whereas the construction for a simulation can accomplish any finite sequence of operations permitted by the programming language. GPSS provides many operations and procedures beyond those developed in this example.

6-3.0 Computer Simulators

There are many published examples of the use of discrete simulation languages to model and study computer systems, Refs. [1, 6, 7, 9].

There have been at least two efforts to develop computer simulators of sufficient generality that they are in effect, a specialized simulation language for simulating computer systems.

6-3.1 CSS II

CSS II, developed by IBM is such a simulator. This simulator was developed by IBM to support its own system analysis needs, and to aid in analysis of customer facility requirements.

IBM now provides CSS II as proprietary software on a rental basis. CSS II is similar in concept to GPSS but differs in one important aspect: it is not general but applies specifically to computer systems. Thus its language speaks in terms of tape units, disk files, communication lines, and terminals, and provides instructions for the modeling of programming systems.

CSS programming consists of a specification of system elements, a specification that generates job streams, and specification of the logical opreations to be performed on the job elements. Its generality is enhanced by permitting a more or less complete construction of both the system hardware configuration and the software operating system, to a level dependent on the users needs and interests.

6-3.2 IMSIM

IMSIM was developed by Systems Development Corporation for the NASA Manned Spacecraft Center. It presents a less general approach to computer simulation, in comparison to CSS, because user constructions are confined to the preparation of input tables which define the configuration of computer system elements and the job stream. The algorithms that define the software operating system cannot be modified, except for a few switch setting choices. The operating system programmed into IMSIM includes capability to simulate priority dependent multiprogrammed, and multiprocessor computing systems. IMSIM is supported only at the Manned Spacecraft Center, NASA. It is written in Modlit, a language similar in many respects to GPSS.

6-4.0 Summary

In concluding this chapter on macrosimulation, it is appropriate to quote from the IBM Program Product Description for CSS II, Ref. [11].

"There are three lessons to be learned from these case studies ... The first is that a great deal of groundwork precedes

the actual construction of a simulation model. One must know what he wishes to model. The system description does not result from the simulation but is an input to the simulation.

The second lesson is that one must have a plan of attack. Very little information is learned from a single simulation. An understanding of system operation comes after running many alternative designs under many different job loads.

The final lesson is that the effort required to carry out a simulation is not minor. This of course arises from the fact that the studies themselves require extensive analysis effort.

These lessons apply not only to CSS but to any comprehensive system design technique. Much of the work required to carry out a CSS study is work that must be done anyway, in one form or another, in defining and understanding system operation."

REFERENCES

1. Lehman, M., & Rosenfeld, J., "Performance of a Simulated Multiprogramming System", (Proc. FJCC, 1968), pp. 1431-1442.

2. Gordon, Geoffrey, System Simulation, (Prentice-Hall, Englewood Cliffs, New Jersey, 1969).

3. Gragg, D.M., "Review of Simulation Languages, Programs and Systems", (L&C Document ISD-C1-SIM-H-002, MSC, NASA, 23 June 1971).

4. Interactive Sciences Corp., GASP II A Reference Manual, Braintree, Mass.

5. Putsker, Kiriat, Simulation with GASP II, (Prentice-Hall, New Jersey, 1969).

6. Day, E.C., "Simulating the Operation of an Aerospace Computer System", (AIAA Aerospace Computer Systems Conference, 8-10 Sept. 1968, AIAA Paper 69-973), pp. 1-12.

7. Katz, J.H., "Simulation of a Multiprocessor Computer System", (Proc. SJCC, 1966), pp. 127-139.

8. MacDougall, M.H., "Computer System Simulation: An Introduction", (Computing Surveys, 2(3), Sept., 1970), pp.191-209.

9. Merikallio, R.A., and Holland, F.C., "Simulation Design of a Multiprocessing System", (Proc. FJCC, 1968), pp. 1399-1410.

10. System Development Corporation, "Information Management System Design for Future Missions, Users Manual", (Report TM-(L)-4719/001/01, Contract NAS 9-11211, NASA Manned Spacecraft Center, Houston, Texas).

11. IBM, CSS II General Information, Technical Publications Department, 1133 Westchester Ave., White Plains, New York.

CHAPTER 7

MARKOV ANALYSIS AS AN ALTERNATIVE TO MACROSIMULATION

7-1.0 Introduction

Modern computing systems show a marked tendency towards complexity in the interrelationships between job streams and computing elements. Multiple access computing, multiple processors, resource scheduling algorithms, all imply a requirement that the computing system designer understand the stochastic behavior of computer system elements as a network of job stream queues and processors.

Monte Carlo simulation is a significant and useful tool in gaining insight into these problems. However, as the system being simulated grows in complexity, these simulations tend to become expensive to run, and accuracy requires very large ensemble samples or running times. When simulation is to be used as a design tool, exploration of the effect of design parameters requires repeated simulation. Under this situation simulation can become expensive.

An alternative to simulation is available in situations where the system can be modelled as a finite state markov chain. Actually, markov chains represent a quite broad and useful class of stochastic models for computer systems. The process of creating a markovian model to study a given computer system is not presented here. It is our purpose to indicate that techniques exist for obtaining state solutions to such models by numerical analysis techniques. Two examples of Markov sequences will be presented in Chapter 8.

7-2.0 Random Sequences and the Markov Property

A random sequence is a collection of random variables indexed by a discrete valued parameter such as

$$x(o),\ x(1),\ \ldots,\ x(n) \qquad (7.1)$$

The variable x may be either continuous or discrete valued.

To describe a random sequence completely, the joint probability function

$$p[x(n), x(n-1), \ldots, x(0)] \tag{7.2}$$

of all elements in the sequence must be specified. Fortunately, most random sequences encountered in applications have the property that each succeeding probability distribution depends only on the previous distribution, or at most on a few preceeding distributions. If the probability distribution is a function only of the preceeding probability distribution, the sequence is markovian.

7-3.0 Markov Sequences and Chains

A random sequence $x(k)$, $k=0, 1, \ldots N$ is Markovian if

$$p[x(k+1)/x(k), x(k-1), \ldots x(0)] = p[x(k+1)/x(k)] \tag{7.3}$$

for all k.

If x is a continuous variable, $p[x(n)]$ is a probability density function, and the joint probability density function between any values of the ordering index can be obtained given the initial density function $p[x(0)]$ and the intervening transition density functions $p[x(k+1)/x(k)]$.

If x is a discrete variable, $p[x(n)]$ is a vector whose dimension is the number of discrete values obtainable by the state x. $p[x]$ sums to 1 over the complete set of possible values of x. In this situation, the markov sequence is usually called a markov chain, and $p[x(n)]$ can be obtained given an initial distribution $p[x(0)]$ and the intervening probability transition matrices, which define the probability associated with a transition from any value of x at k to any value of x at k+1.

7-4.0 The Generality of Markov Sequences

Although many applications of interest appear to fit with ease within the framework described above, it should be pointed out that any sequence that depends on a finite number of past values of the sequence can be make markovian by redefining the variable $x[n]$ to be a vector whose components include past values of x, for example, an appropriate expanded definition of x would be

$$x'[n] = \begin{bmatrix} x[n] \\ x[n-1] \end{bmatrix} \tag{7.4}$$

for a system where transitions are a function of the two most recent states of the variable.

7-5.0 Methods of Solution for Markov Chains

Numerical solutions to Markov chains can be obtained by explicitly performing the sequential transition from an interval for which the p[x] is known, by successive multiplication by the probability transition matrix.

$$p[x(k)] = M(k-1)M(k-2) \ldots M(n)p[x(n)] \tag{7.5}$$

If the functional relation defining the transition is non-linear, numerical solutions are still obtainable by the method described above, except, for each step, a more general functional operation is implied;

$$p[x(k+1)] = F(p[x(k)]) \tag{7.6}$$

When the probability transition matrix is independent of the ordering index, equation (7.5) becomes

$$p[x(k)] = M^{k-n} p[x(n)] \tag{7.7}$$

This "index invariant" situation also permits the application of the transform calculus of generating functions. This procedure involves the formation of generating functions which are infinite polynomials in an ordering variable z. Equation (7.8) is an example.

$$G(z) = a_0 + a_1 z + \ldots + a_n z^n + \ldots \tag{7.8}$$

The coefficients a_n are a number sequence, where the successive increasing powers of z order the sequence for increasing values of the index n.

By using generating functions, the following infinite set of matrix transition equations

$$\begin{aligned} \underline{p[x(1)]} &= \underline{M}\ \underline{p[x(0)]} \\ p[x(k)] &= M\ p[x(k-1)] \end{aligned} \tag{7.9}$$

can be reduced to a single matrix equation

$$P(z) = z\ M\ P(z) + p(0) \tag{7.10}$$

where P(z) is the (vector) generating function for the sequence of probabilities associated with the various discrete values of x(n).

M is the probability transition matrix, and the scalar multiplier z increases the power of z in each scalar generating function corresponding to the transition of the index in the sequence.

The vector p(0) is the initial value of p[x(0)]. It is the initial probability distribution for the various discrete values of the variable x. p(0) may be considered an initial condition on the transformation which picks up the effect of all previous transitions prior to the value of the index assigned to the beginning of the problem.

By rearranging equation (7.10) we obtain

$$(I - z\,M)P = p(0) \qquad (7.11)$$

which is a matrix algebraic equation defining the transform variable P in terms of the probability transition matrix M, and initial conditions p(0). This equation can be solved for the various components of the generating function; $P^1(z)$, $P^2(z)$ ... , where each generating function contains complete information on the sequence of probabilities as the markov sequence unfolds.

7-6.0 An Example

To be concrete, we turn to an example. We wish to analyze the performance of a subroutine which consists of a compute element followed by a branch which with equal probability either exits the subroutine or returns and begins the subroutine again. The flow diagram is shown in Figure (7.1).

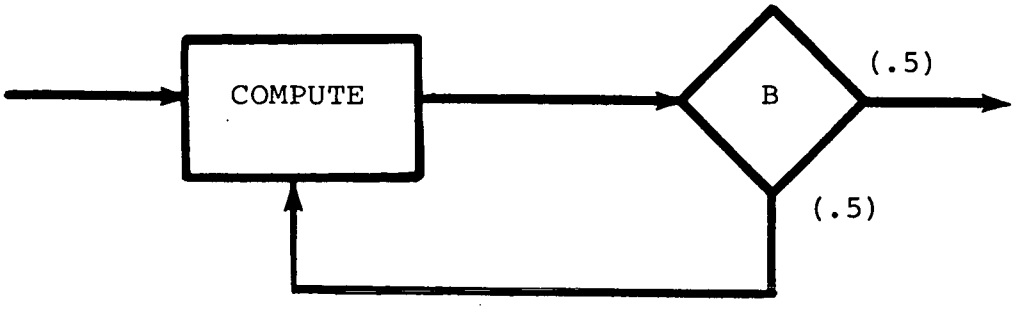


Figure 7.1 - Subroutine Example

We define a discrete variable x which has two states;

$x = 1$ subroutine is entered

$x = 2$ subroutine is finished

Associated with this variable is a (vector) probability sequence

$p^1[x(k)]$ - probability that x=1 at k^{th} interval

$p^2[x(k)]$ - probability that x=2 at k^{th} interval

Since x is either 1 or 2, $\Sigma P[x] = 1$ for all k. We define generating functions for each probability sequence

$$P^1(z) = Z\left\{p^1[x(k)]\right\} = p_0^1 + p_1^1 z + \ldots$$
$$P^2(z) = Z\left\{p^2[x(k)]\right\} = p_0^2 + p_1^2 z + \ldots \qquad (7.12)$$

The probability transition matrix for this problem is

$$M = \begin{bmatrix} .5 & 0 \\ .5 & 1 \end{bmatrix} \qquad (7.13)$$

where m_{ij} is the probability associated with a transition from the state x=j to the state x=i, in an interval.

Thus, for this problem, equation (7.10) becomes

$$\begin{bmatrix} 1 - .5z & 0 \\ -.5z & 1-z \end{bmatrix} P(z) = \begin{Bmatrix} 1 \\ 0 \end{Bmatrix} \qquad (7.14)$$

where we choose for an initial condition that

$x(0) = 1$ with certainty.

Solving this set of equations yields

$$P^1(z) = \frac{1}{1-.5z} = 1 + \frac{1}{2}z + \frac{1}{4}z^2 + \ldots$$

$$P^2(z) = \frac{.5z}{(1-.5z)(1-z)} = 0 + \frac{1}{2}z + \frac{3}{4}z^2 + \frac{7}{8}z^3 + \ldots \tag{7.15}$$

which are seen to be the ordered set of probabilities for x=1, and x=2 respectively.

The inverse transformation; getting the number sequence from the generating function is accomplished by long division, as in the above example or by noting that

$$p_n = \frac{d^n}{dz^n} P(z) \Big|_{z=0} \tag{7.16}$$

the initial value is

$$p_0 = P(z) \Big|_{z=0} \tag{7.17}$$

and the final value is

$$p_f = (1-z)\, P(z) \Big|_{z=1} \tag{7.18}$$

It is also of great value in many cases to be able to determine statistical moments of the probability sequence. This will be demonstrated in example 1 of Chapter 8.

7-7.0 Equilibrium Solutions

The transform methods discussed above yield complete solutions for finite, linear, index invariant, markov chains. However, analytic methods for many problems are too tedious to use due to the order of the algebraic equations.

One can frequently be satisfied with obtaining equilibrium solutions, which are defined by the equation

$$p = M p \tag{7.19}$$

or

$$(I - M)p = 0$$

These are terminal solutions to the markov chain since these distributions regenerate themselves on each transition. One must not presume that there is a single equilibrium solution to a process. There may be as many equilibrium solutions as there are discrete values of the random variable. A knowledge of the physical process is usually sufficient to recognize the number of possible equilibrium states and their proper interpretation.

It is fortunate that the most frequent objective of practical computer analysis projects is to find the equilibrium probability distribution of the state. Since the state consists of the value of the discrete variable x (x=1, x=2, etc.), the equilibrium probability distribution is a vector whose elements can be roughly interpreted as the probability that x (the state) will assume each of its various discrete values. Care must be taken to realize that at any given interval, the state assumes exactly one and only one value, and therefore, its associated distribution must be interpreted as a limit of either an ensemble average, or if the sequence is ergodic, as the average condition over a large interval, following sufficient elapsed intervals to eliminate transient effects due to initial conditions.

Once the equilibrium probability distribution for the state of the computing system is known, many other probabilistic measures of system sufficiency or performance are readily established as combinational distributions, expected values, variances, and so forth. Throughput rates, wasted core space, length of wait times, "busy signals" are but a few of the derived measures of performance that are computable from the equilibrium probabilities of state.

7-8.0 Comparison with Macro-Simulation

Solution of equation 19 can be obtained either analytically, as will be done in example 2 of Chapter 8 or numerically, by assuming a value for P and solving the fundamental transition equation

$$p^{k+1} = Mp^{k} \tag{7.20}$$

until

$$p^{k+1} = p^{k}$$

This procedure is an alternative to Monte Carlo simulation techniques which form ensemble or ergodic averages of solutions obtained by sequencing the state variable x from interval to interval by transitions which are based on random numbers, generated according to the probabilities defined in the transition matrix.

7-9.0 Speed of Solution

Reference [1] describes the implementation of a computer program to solve the equilibrium distribution equation. The program described was designed to efficiently solve the equation for systems with as many as 5000 discrete states. The conclusion of that paper is that for most problems of significance, solution of the markov equilibrium equation will require several orders of magnitude less computing time than solution by simulation.

7-10.0 Summary

This chapter has described a series of alternatives to simulation in the analysis of computing systems. We have shown that significant analytical tools are available for problems where the state of the computing system can be modelled as a markov chain. In particular we have shown that explicit analytic solutions are available if the chain is linear and index invariant.

Equilibrium numerical solutions can be obtained by direct use of the state transition equation. We have quoted from Reference [1] to claim that such procedures are superior in terms of computing requirements in comparison to Monte-Carlo simulation.

REFERENCES

1. Wallace, V.L., and Rosenberg, R.S., "Markovian Models and Numerical Analysis of Computer System Behavior", (Proceedings, Joint Spring Computer Conference, 1966), pp. 141-148.

2. Bryson, A.E., and Ho, Y.C., Applied Optimal Control, (Blaisdell Publishing Co., Waltham, Mass., 1969)

CHAPTER 8

DISCRETE MARKOV TECHNIQUES IN COMPUTER ANALYSIS

8-1.0 Example 1

Suppose that one has a subroutine, characterized by a compute time T, followed by a branch, which, with equal probability, either repeats the computation or exits. A flowgraph is shown in figure 8.1.

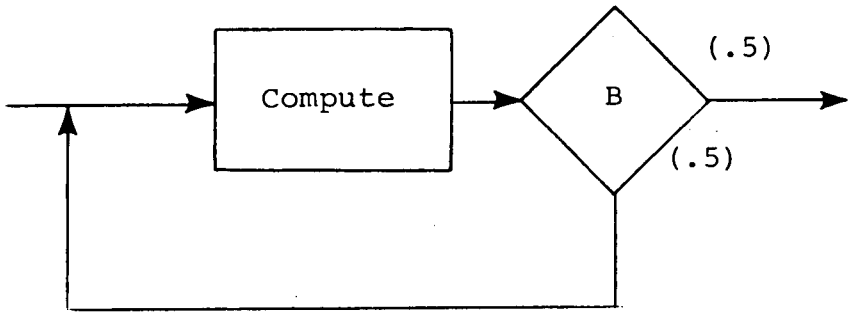


Figure 8.1 - Flowgraph for Subroutine

The minimum time for the subroutine is T. The maximum time is infinite. What is the average time?

This problem is sufficiently simple so that one can easily construct the sequence of probabilities associated with possible discrete results:

$$P(0) = 0$$
$$P(T) = 1/2 \qquad (8.1)$$
$$P(2T) = 1/4$$

The average time is

$$\bar{t} = T \sum_{0}^{\infty} kP_k \qquad (8.2)$$

where

$$P_k = P(kT) \tag{8.3}$$

Similarly, the variance is

$$\text{var} = \overline{t^2} - \bar{t}^2 = T^2 \left(\sum_{\circ}^{\infty} k^2 P_k - \left(\sum_{\circ}^{\infty} k P_k \right)^2 \right) \tag{8.4}$$

The probabilities associated with the various possible subroutine times is a countably infinite sequence of numbers. A very interesting procedure for analyzing such number sequences involves the introduction of the concept of generating functions, as we have seen in Chapter 7.

We define an ordering variable z, and create an infinite sum in terms of the number sequence and z. Thus,

$$G(z) = p_0 + p_1 z + p_2 z^2 + \ldots = \sum_{\circ}^{\infty} p_k z^k \tag{8.5}$$

is a generating function for the probability of exiting the subroutine at the various intervals.

Let $G_1(z)$ be another generating function associated with the probability sequence that the subroutine is entered at the kth interval. Since it is equally likely that the subroutine will branch back or exit, we can conclude that the function for the branch back path is also G(z), and, since the function for the first entry is equal to 1;

$$G_1(z) = 1 + G(z) \tag{8.6}$$

where z=0 corresponds to the interval that the subroutine is first entered.

Let $G_2(z)$ be the function for the arrival at the branch point. This sequence of numbers is identical to $G_1(z)$, except that one interval has elapsed. Therefore, its generating function is

$$\begin{aligned} G_2(z) &= G_1(z)\ z \\ &= z(1 + G(z)) \end{aligned} \tag{8.7}$$

Finally we close the loop by noting that

$$G(z) = .5\ G_2(z) \tag{8.8}$$

yielding

$$G(z) = .5z\ (1 + G(z))$$

$$G(z) = \frac{.5z}{1-.5z} \tag{8.9}$$

This equation is a closed form expression for the generating function. It has been derived by recognizing the recursive nature of the process which generates the sequence.

We can recover the individual values of the number sequence by explicit division, or by noting that

$$\frac{d^n}{dz^n} G(z) \Big|_{z=0} = P_n \tag{8.10}$$

Since

$$G(z) = \sum_0^\infty P_k z^k \tag{8.11}$$

the derivative with respect to z, evaluated at z=1 is

$$G'(z) \Big|_{z=1} = \sum_0^\infty kP_k \tag{8.12}$$

which is the average value of k.

The second derivative, evaluated at z=1, is

$$G''(z) \Big|_{z=1} = \sum_0^\infty P_k k(k-1) \tag{8.13}$$

Thus, the variance can be calculated from

$$\text{var} = G''(z) + G'(z) - G'(z)^2 \tag{8.14}$$

For our subroutine

$$G'(z) = \frac{.5}{1-.5z} + \frac{.25z}{(1-.5)^2} \tag{8.15}$$

yielding

$$G'(z) \Big|_{z=1} = 2 \tag{8.16}$$

Evaluating the variance according to equation (8.14) yields the following average time, and standard deviation in time for the subroutine

$$\begin{aligned} \text{avg} &= 2T \\ \text{S.D.} &= \sqrt{2}T \end{aligned} \tag{8.17}$$

The procedure used to obtain the closed form solution for the probability generating function can be constructed directly from the block diagram. The procedure is to substitute the variable z for the compute box, signifying the transition in the ordering variable, and to assign multipliers in the paths equal to the probability that the path will be taken. The block diagram becomes; Figure 8.2:

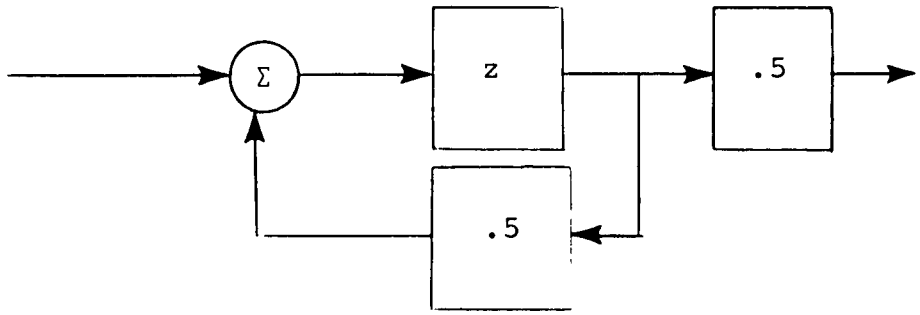


Figure 8.2 - Block Diagram for Subroutine

Those familiar with the technique of block diagram reduction, will be able to determine that this system has a "transfer function" of

$$G(z) = \frac{.5z}{1-.5z} \tag{8.18}$$

For more complicated processes, standard techniques of block diagram reduction yield the closed form expression desired. The reader will notice the similarity between the ordering variable z and the delay operator z used in the analysis of sampled data systems. They are in fact equivalent, although the most common practice defines the operator in sampled data analysis such that the ordering function is reversed, generating past values, rather than future values of the sequence for increasing powers of z, [7].

8-2.0 Example 2

Suppose, now that we are interested in a system that consists of a device (or subroutine) that services users in a manner similar to the subroutine example. However, the users of the device will wait for service if the device is busy. However, the queue is of length one, and if the place in line is filled, subsequent users are lost to the system.

We define two parameters that are sufficient to characterize the system:

1. Service rate μ; the probability that service is completed at an interval.

2. Arrival rate λ; the probability that a new user arrives at an interval.

We specify that only one user can arrive per interval. There are three possible states of this system, all mutually exclusive:

state 0; no users in the queue or being served

state 1; a user is being served

state 2; a user is being served and the place in line is filled

We can construct a transition diagram for this problem (or a block diagram for that matter), by defining the probabilities that are associated with transitions between states at an interval. The transition diagram is as follows:

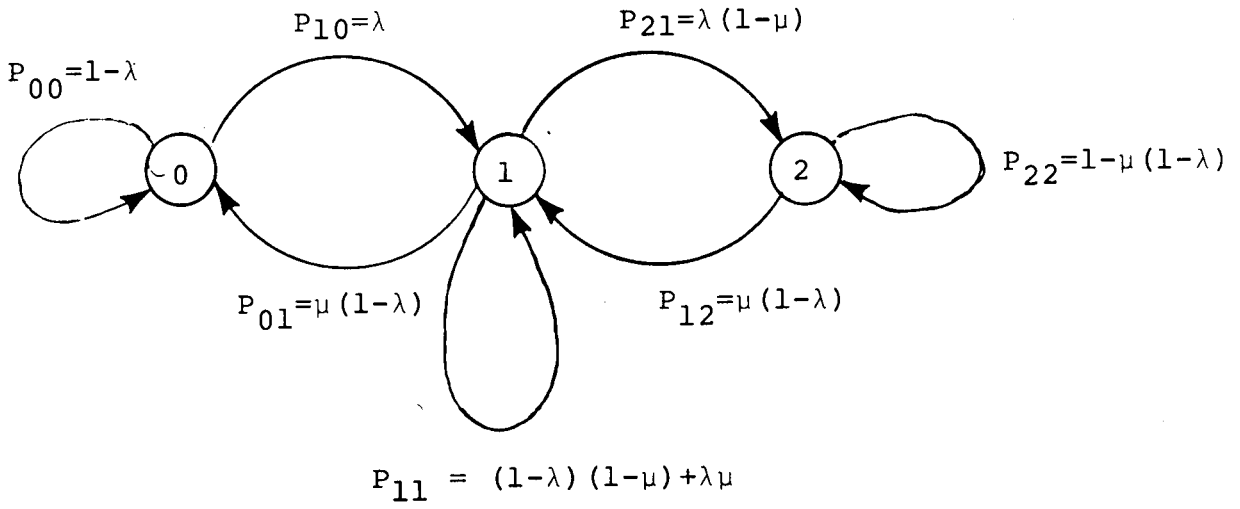


Figure 8.3 - Transition Diagram for Queueing Example

The probabilities for the transition are

P_{00} = no new users arrive

P_{10} = a user arrives

P_{01} = a user finishes service, no new users arrive

P_{11} = no new users, no finished service, or a new user and a finished service,

P_{21} = a user arrives, no finished service

P_{12} = no new users, and a finished service

P_{22} = not (no new users, and a finished service)

The unusual definition of P_{22} illustrates the use of the fact that that since some transition must occur, and since state 2 can transition either to state 1 or state 2, the sum of these probabilities must be one, hence $P_{22} = 1 - P_{21}$.

In a problem of this type, we are often interested in the steady state behavior of the system. The transient behavior can be analyzed, using the generating functions, as before, but here we will confine our interest to the condition where the system has reached a steady state. Let R be a vector whose elements are, in order, the probability that the system is in state 0, state 1, or state 2. Then, in steady state R will be constant, and the following equation must be true

$$R = PR \tag{8.19}$$

where P is the matrix whose elements were defined above, and described on the transition diagram. Rearranging this equation, we obtain

$$(I-P)R = 0 \tag{8.20}$$

The matrix (I-P) is written out below

$$I-P = \begin{bmatrix} \lambda & \mu(\lambda-1) & 0 \\ -\lambda & \mu + \lambda - 2\lambda\mu & \mu(\lambda-1) \\ 0 & \lambda(\mu-1) & \mu(1-\lambda) \end{bmatrix} \tag{8.21}$$

In order that there be a solution to equation (8.20) other than R=0, the determinant of I-P must equal zero, [4, ch. 1].

Inspection of equation (8.21) confirms that this will be so, since the sum of the elements of each column are zero, (even one column would have been sufficient). The fact that the sum of columns are zero follows from the fact that the sum of a column in P is the sum of the probability of all possible transitions from a state, which must equal 1.

Therefore, a non-trivial solution for R always exists. An aribtrary constant must be supplied by recalling that $\Sigma R=1$.

We will solve the problem for values of $\mu=.1$ (arrival probability)

$\lambda=.5$ (completed service probability)

The equation becomes

$$\begin{aligned} .1\ R_0 - .45\ R_1 &= 0 \\ -.1\ R_0 + .5\ R_1 - .45\ R_2 &= 0 \\ -.05\ R_1 + .45\ R_2 &= 0 \end{aligned} \qquad (8.22)$$

Solving these simultaneous equations, the steady state probabilities are found to be

$$\begin{aligned} R_0 \text{ (system empty)} &= .802 \\ R_1 \text{ (server busy)} &= .179 \\ R_2 \text{ (queue full)} &= .019 \end{aligned} \qquad (8.23)$$

It is interesting to note that somewhat less than one in a hundred arriving users will be turned away.

8-3.0 Comments on the Examples

The preceeding examples have introduced several concepts useful in the analysis of computing systems. The most important concept was not commented on during the development. Both of these examples involve the analysis of a markov sequence. A markov sequence exists where the transition from state to state is a random event whose probable outcome is completely defined by the previous state.

The stock market is a good example of a system which is not a markov process, since the price of stock, and all the events that happen during the day do not seem sufficient to determine closing

prices. Past experience, previous tops, limit lines, and so forth are being avidly tracked by technical traders, and other interrelationships reaching far into the past are at work and serve to make the problem "non-Markovian". Of course, at some level of complexity, any physical system can presumably be made markovian. (One description of Markov sequences, or processes, is to say that a process is Markov if the future is connected to the past only by the present.)

Furthermore, the systems are linear, yielding linear difference equations, or algebraic equations in the ordering variable.

Finally we have introduced the concept of generating functions, first invented by LaPlace. Generating functions greatly increase the power to solve problems involving number sequences, since closed form solutions can often be found by inspecting the recursion formula that generates the sequence. Having found an expression for the generating function, it is a simple matter to obtain the individual terms of the sequence. Limit values, such as initial value, or final value can also be easily obtained. When the number sequence is a complete set of probabilities, the statistical moments of the set are also easily determined as was demonstrated.

We have also shown that there is an equivalence between the concept of generating functions and the z transform functions of sampled data analysis.

8-4.0 Characteristic Functions [2]

Characteristic functions are used in probability theory. They are also sometimes referred to as moment generator functions. We will describe them briefly here in order to perceive the duality between the use of generating functions for the discrete random process in example 1, and the use of characteristic functions for continuous processes.

The characteristic function of a random variable, whose probability distribution function is $f(x)$, is

$$\phi(u) = \int_{-\infty}^{\infty} f(x) e^{jux}\, dx \qquad (8.24)$$

It will be noticed that $\phi(u)$ is actually an inverse Fourier transform of $f(x)$. It can be shown that u is real, for x real, that $\phi(u)$ exists for any $f(x)$ that is a probability distribution, and that $f(x)$ is uniquely defined by $\phi(u)$.

Now, we expand $\phi(u)$ in a MacLaurin series;

$$\phi(u) = \phi(0) + \phi'(0)\, u + \phi''(0)\, \frac{u^2}{2!} + \ldots + \phi^{(n)}(0)\, \frac{t^n}{n!} + \ldots \qquad (8.25)$$

but

$$\phi(0) = \int_{-\infty}^{\infty} f(x)dx = 1$$

$$\phi'(0) = \int_{-\infty}^{\infty} jx\, F(x)dx = j\bar{x} \qquad (8.26)$$

$$\phi''(0) = \int_{-\infty}^{\infty} j^2x^2\, f(x)dx = -\bar{x}^2$$

Thus, the moments of the probabilities distribution function f(x), can be determined from the derivatives of the characteristic function, evaluated at the origin. This is analagous to evaluating moments of the discrete distribution by evaluating the derivatives of the probability generating function at z=1.

8-5.0 Further Reading

The reader is advised to read section 1.2.10 of Reference 1 entitled "Analysis of an Algorithm" to see an example of the application of these techniques in determining the solution time to be expected of an algorithm designed to search through a set of randomly ordered numbers in order to obtain the largest number in the set.

Also, Reference 5 contains a comparative analysis of two methods of performing binary multiplication, using the analysis techniques described in this chapter.

REFERENCES

1. Knuth, D.E., Fundamental Algorithms, (Addison-Wesley Publishing Company, Reading, Mass., 1968).

2. Laning, J.H., and Battin, R.H., Random Process in Automatic Control, (McGraw-Hill, New York, 1956).

3. Saaty, T.L., Elements of Queueing Theory, (McGraw-Hill, New York, 1961).

4. Hildebrand, F.B., Methods of Applied Mathematics, (Prentice-Hall, Englewood Cliffs, New Jersey, 1961).

5. Ramamoorthy, C.V., "Discrete Markov Analysis of Computer Programs", ACM, 20th National Conference, 1965, pp. 386-392.

6. Wallace, V.L., and Rosenberg, R.S., "Markovian Models and Numerical Analysis of Computer System Behavior", Proceedings, Spring Joint Computer Conference, 1966, pp. 141-148.

7. Ragazzini, J.R., and Franklin, G.F., Sampled-Data Control Systems, (McGraw-Hill, New York, 1958).

PART III

HARDWARE TECHNIQUES

CHAPTER 9

AEROSPACE COMPUTER ARCHITECTURE

9-1.0 Introduction

In this chapter we will review the architectural properties of some recent aerospace computers. The salient features of these machines are characteristic of trends that have been evolving in aerospace computer design. Features such as stack organization, microprogramming and modularity are becoming commonplace in today's systems. A clear understanding of these features is important to the system designer if he hopes to efficiently utilize the computational power of these machines, and an excellent way of building this understanding is to study some typical computer architectures. Such a study is the purpose of this chapter.

An expanded version of this chapter covering all aerospace computers presently available would be a useful system design aid. Computer systems would be listed according to available features and physical characteristics. When the need to specify a computer arises, those systems having the desired characteristics could then easily be isolated and chosen. The industry search and system analysis necessary to compile such a book is large and beyond the scope of this contract. However, we will attempt to show the benefits of such a design aid by reviewing the features of some recent aerospace computers, such as the Burroughs D-Machine, the Navy's Advanced Avionic Digital Computer (AADC), the JPL STAR Computer, etc.

9-2.0 Structural Levels

The structure of a digital computer may be specified on several levels each having a specific category of components [1]. These levels are the following:

a) Circuit Level: the components are resistors, capacitors, transistors, etc.

b) Switching Level: the components are flip-flops, delays, logical gates.

c) Register-Transfer Level: the components are registers, control operations, transfers, data operators.

d) Program Level: the components are programs, instructions, memory cells, etc.

e) Architectural Level: the components are memory modules, processors, data buses, switches, instructions.

It is the last level, the Architectural Level, upon which we will concentrate.

9-3.0 HOL Architecture

One of the interesting trends in computer organization has been toward higher order language (HOL) architectures [2]. That is, architectures designed for efficient execution of HOL source statements. A good example of this concept is the SPLM computer [3,4] designed to directly execute SPL source code without compilation. The system employs a one pass loader to transform the source code into reverse Polish notation (intermediate code) and then load it into the machine's memory. Compilation can be eliminated because the machine does not have a traditional von Neumann architecture, but rather a stack organization. These stacks are used for dynamic storage allocation, addressing, and nesting of processes.

The SPLM requires three stacks and one pushdown list. (A stack is a last-in-first-out (LIFO) queue in which all elements are visible, while a pushdown list is a LIFO queue in which only the top element is visible.) The machine has a declaration stack to define data objects for later manipulation by procedure execution, a parameter stack to hold the actual values of parameters, and a slice control stack to control nested procedures. The pushdown list is used for expression evaluation.

Upon calling a procedure its beginning is marked in the parameter and declaration stacks. Then all parameters to be passed to the procedure are evaluated and put into the parameter stack. Next the procedure code is executed using the pushdown list, and during this execution the procedure's declarations are put into the declaration stack. When the procedure is finished, the stack pointers are returned to a position appropriate to execute the next statement following the procedure call.

The instructions needed to efficiently perform this process are rather unusual compared to a von Neumann architecture. Some of these instructions are specially designed for stack and list manipulation as well as the usual arithmetic operations. They allow efficient execution of the intermediate code via the stacks and the pushdown list.

9-4.0 Reliability

Reliable performance is a prerequisite of aerospace computers. Should a malfunction occur, immediate fault isolation and correction techniques must be employed. This philosophy has guided the development of the STAR Computer [5], perhaps the most advanced design for reliability. Its architecture is illustrated in Figure 9.1.

Reliability is achieved by modularizing the system into a set of replaceable functional units, a technique which simplifies fault isolation and unit replacement. Each unit is then backed up with several spares, and replacement is implemented by power switching. Special hardware carries out the fault detection, recovery and replacement. This topic is further treated in the chapter on failure detection methods.

9-5.0 Modularity

The Burroughs' D-Machine [6-8] is a highly modular aerospace computer. Its architecture is shown in Figure 9.2. There are four major units: interpretors, memory units, devices, and a switch interlock. The interpreters are the processing units of the computer. Each interpreter has five functional parts:

a) logic unit (LU) - performs arithmetic, shift and logical operations;

b) control unit (CU) - provides commands to LU, global commands to other interpreters, tests and sets control conditions;

c) memory control unit (MCU) - controls accesses to memory and device units and to microprogram memory;

d) microprogram memory (MPM) - contains addresses of control words;

e) nanomemory - contains control words pointed to by MPM.

These five functional parts of the interpreter are modularly organized as shown in Figure 9.3. Furthermore, additional modularity results from some of these units themselves being modularly designed. The LU is modular in 8 bit increments. Word lengths from 16 to 64 bits can be provided using the same functional unit. In addition, the MCU is expandable depending upon the addressing capability required.

The switch interlock (SWI) connects interpreters to memory units and devices. Such a connecting device allows modular expansion for incremental numbers of interpreters, memory units and devices. It also

C.3

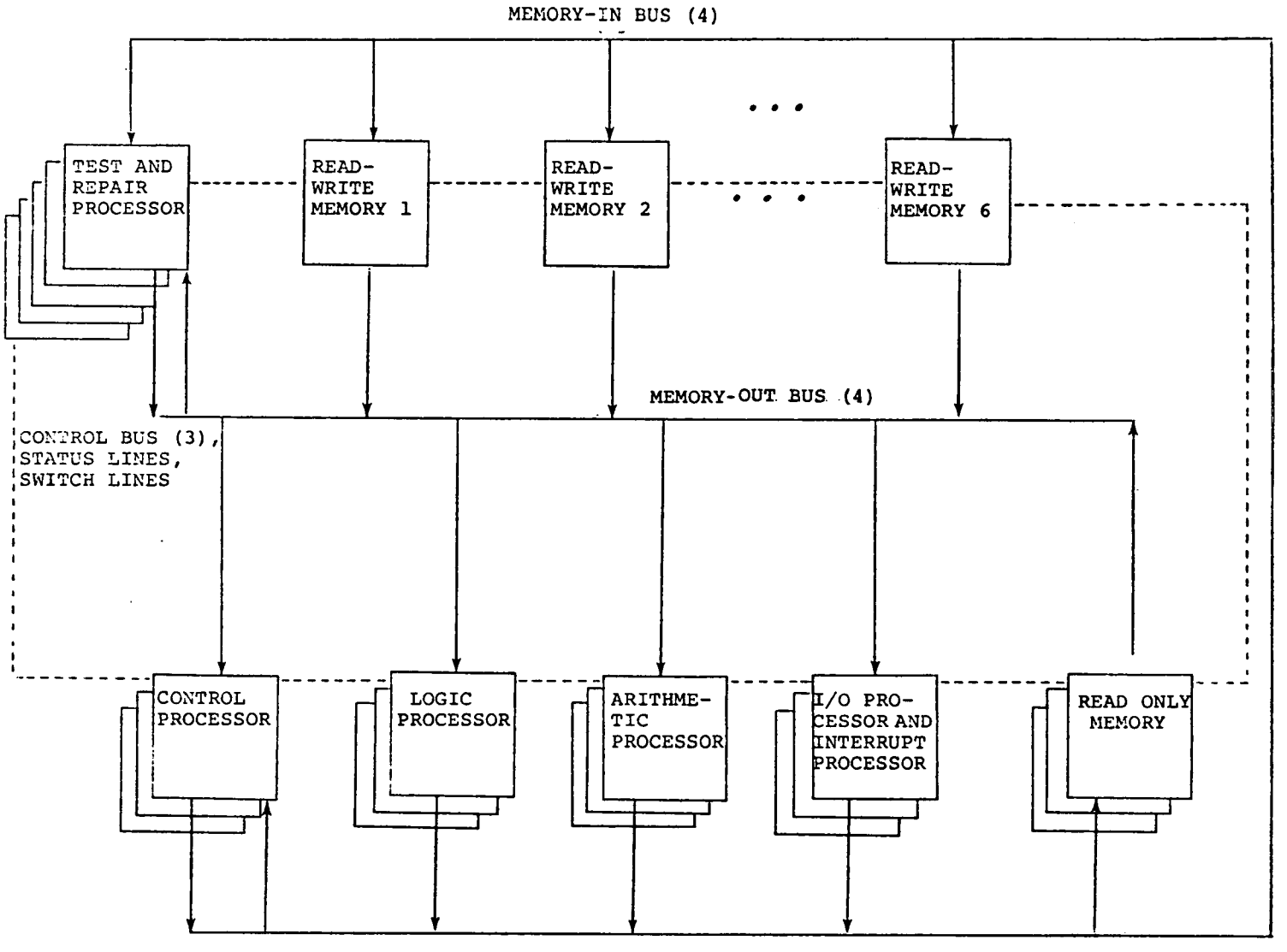


FIGURE 9.1: ARCHITECTURE OF STAR COMPUTER

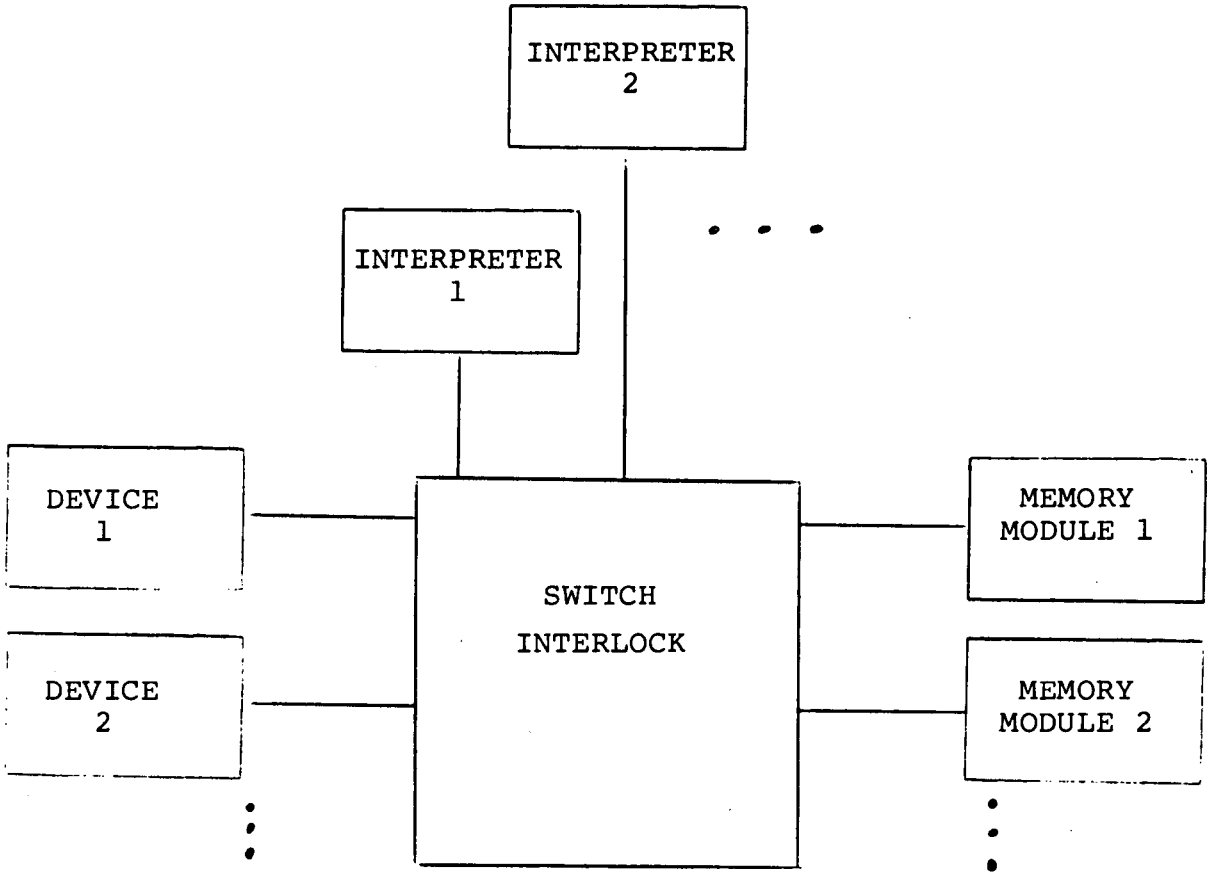


FIGURE 9.2: D-MACHINE ARCHITECTURE

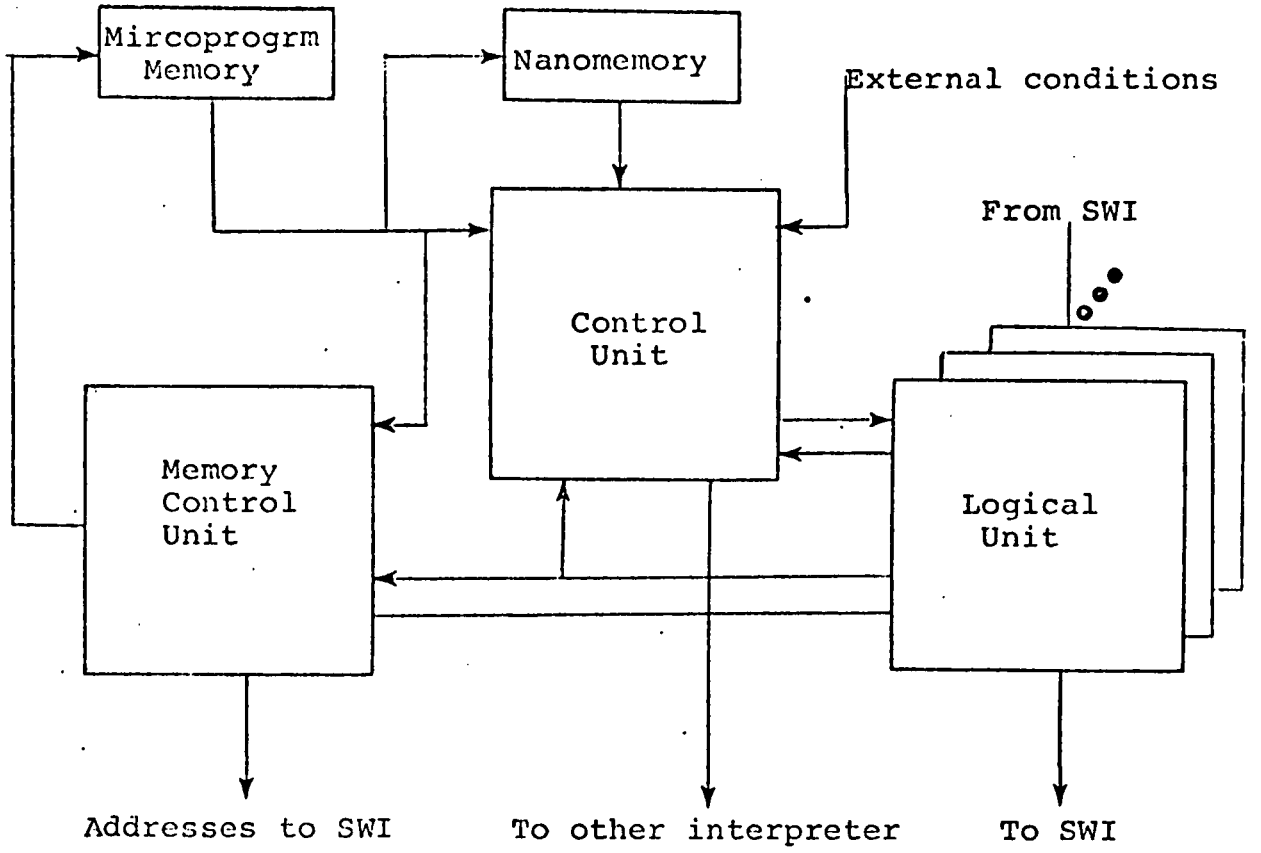


FIGURE 9.3: INTERPRETER ORGANIZATION

allows an optional amount of parallel transmission between connected units. The less complex SWI operations per clock pulse compared to the interpreter allows the SWI to be run at a higher rate. Thus, more serial operations can be introduced reducing the number of parallel paths needed and reducing complexity. This unit can be easily modified to go from full serial to an optional level of parallel operation.

The result of this modularity is a smoothly expandable system, one that can be tailored to match the complexity of the application. We will see that additional versatility results from the D-Machine's unique microprogramming organization, a topic we will treat in the next section.

9-6.0 Microprogramming

The D-Machine takes a rather unique approach to microprogramming. Each of the 56 bit words in nanomemory is a microprogram control word with each bit corresponding to a control line for the LU, CU and MCU. A nanoword is selected under control of a microword, a word in MPM, specifying the nanoword's address. Thus, the microprogram for a given machine instruction is a series of addresses in MPM with each address selecting a control word in nanomemory. If a control word is used by several instructions, it need only be stored once in nanomemory. Although its address must be repeated in MPM, a considerable bit savings still results due to this two level organization of micro-program control.

The dynamic microprogramming capabilities of the D-Machine allow great versatility in the use of this computer. Several interpreters can be provided with different microprograms, each one specialized to a particular language. This fact would enable the interpreters to have a HOL architecture via the microprogram and enable direct execution of source code for several HOLs simultaneously. A second potential of the D-Machine for dynamic microprogramming is the ability to switch the microprogram of a single interpreter. By doing so an interpreter can execute sequential processes each tailored to a particular instruction set without needing extra hardware.

The advantages of using microprogramming in an aerospace application are vast. Most importantly as mission requirements change new computers do not have to be bought; the capabilities of the original computers can be modified by new microprograms. We will not list additional applications and benefits of microprogramming here. The reader is referred to Reference [9].

9-7.0 Multiprocessor Networks

The D-Machine and the Navy's AADC [10,11] are two recently designed aerospace multiprocessors. We have already seen how several

interpreters can be connected to the D-Machine's switch interlock to make the system a multiprocessor. We will now study the architecture of the AADC, which is illustrated in Figure 9.4.

The system is being designed to meet airborne computational requirements for 1975-1985 naval missions. It is modularly organized so that it can be assembled from off-the-shelf units, and it can be configured as needed, from a single processor machine to a large multiprocessor. Each processing element (PE) contains a processor and a cache, called the task memory (TM), of from 1K to 4K words with paging and an address space of 65K words. Program modules are stored on a block oriented random access memory (BORAM) and are read into the TMs as necessary. The use of a cache decreases the number of needed accesses to the shared memory.

Sequentially organized problems are handled by the PEs. For parallel organized problems there is a programmable matrix-parallel processor, which consists of a fast Fourier processor, an associative processor, and associative memory. These devices are connected by logic controlled by the master executive control (MEC), which coordinates the computational power of this computer.

9-8.0 Highly Parallel Organization

The last trend in recent aerospace computer design that we will investigate is that of highly parallel organization. This is a topic that is a variation upon the theme of multiprocessor networks. Instead of having several processors, each executing an independent computation, these machines operate on a string or an array of data in parallel [12]. The Goodyear STARAN computer [13], an associative array processor (see Figure

The STARAN contains up to 32 arrays, where each array consists of 256 processing elements and an associative storage of 256 words of 256 bits each. Long word length are used because computing is done within a word rather than between words. Each array can perform an arithmetic or logical operation on 256 data elements simultaneously. The processing time for a given instruction is independent of the number of data elements being operated upon. In addition, using an associative organization enables search time for an array element to be reduced to one memory access. With an associative memory data is located by content not physical address searching.

Associative array processing has been applied successfully to solving problems in numerical analysis, air traffic control, pattern recognition, and radar signal processing. It is best suited to applications in which large amounts of data can be processed simultaneously.

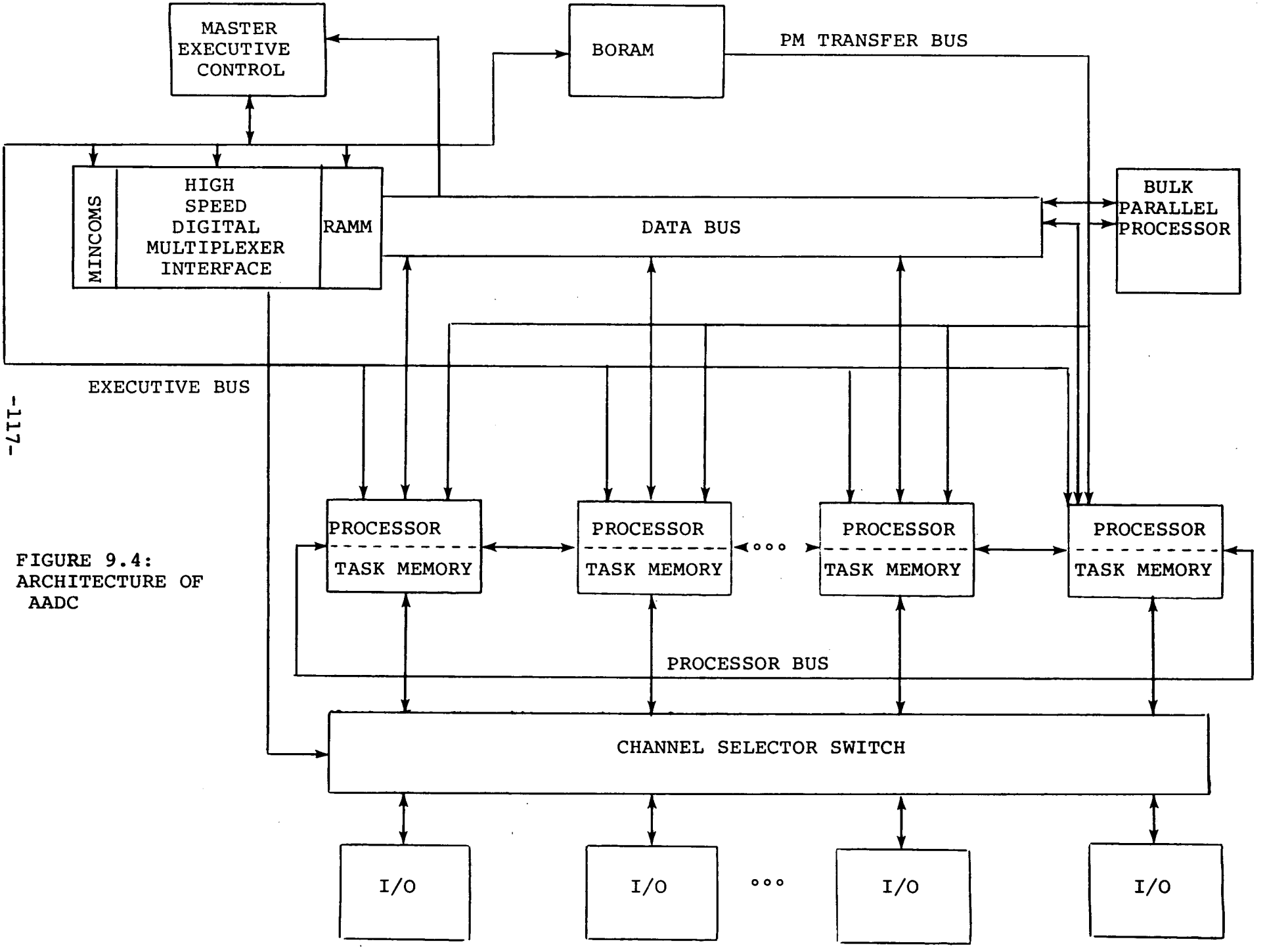


FIGURE 9.4:
ARCHITECTURE OF
AADC

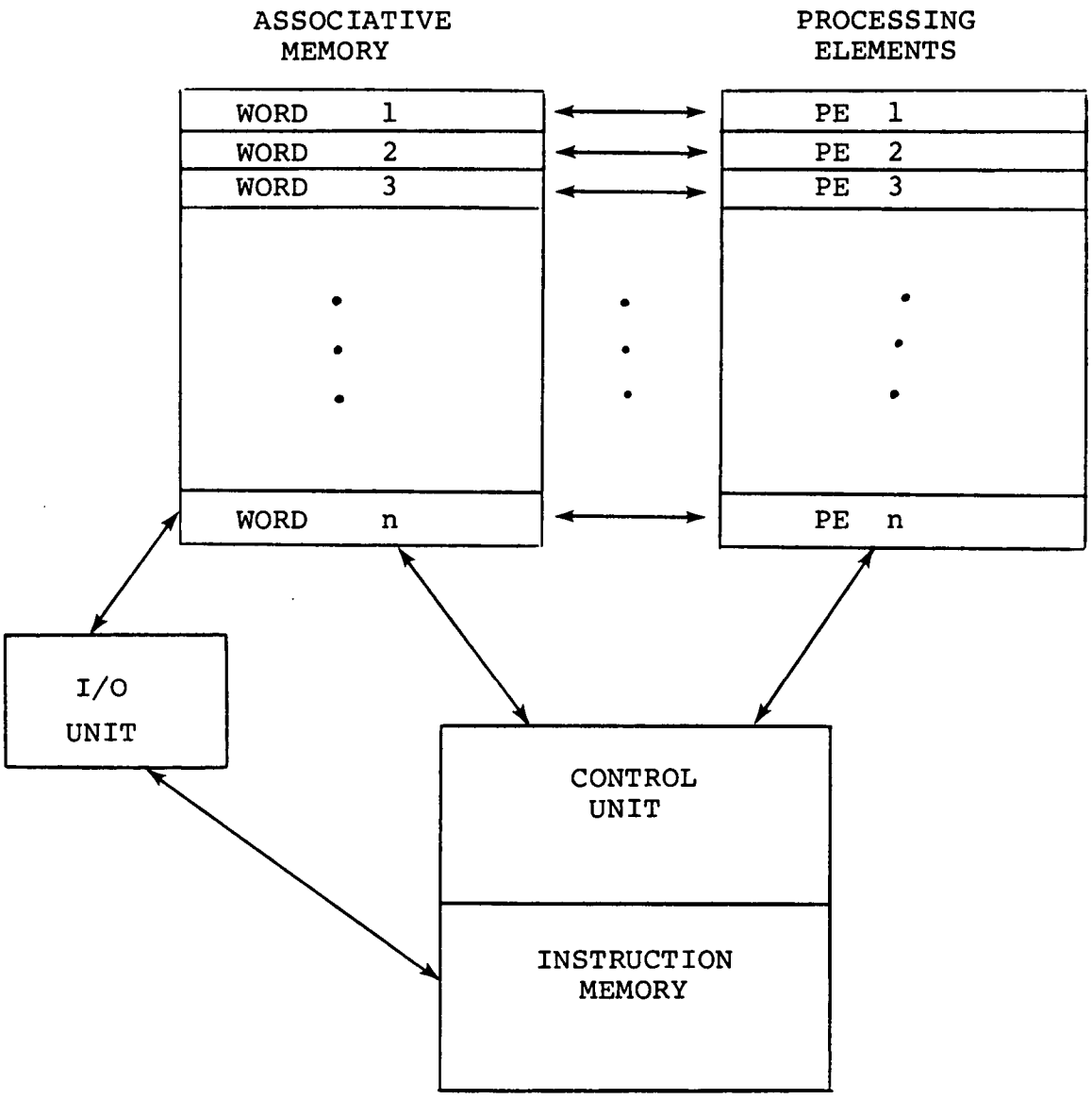


FIGURE 9.5: ORGANIZATION OF AN ASSOCIATIVE ARRAY PROCESSOR

REFERENCES

1. Bell, C. G., and Newell, A., Computer Structures: Readings and Examples, (McGraw-Hill, New York, 1971).

2. Foster, J. R., "Development of a Higher Order Language Architecture", (NAECON '71 Record), pp. 201-205.

3. Keeler, F. S., et al., Computer Architecture Study, (Information and Communication Applications, Inc., Silver Spring, Maryland, October 1970, SAMSO-TR-70-420).

4. Cirad Corp., Architectural Study for Advanced Guidance Computers, (Claremont, California, February 1971, SAMSO-TR-71-6).

5. Avizienis, A., et al., "The STAR (Self-Testing and Repairing) Computer: An Investigation of the Theory and Practice of Fault-Tolerant Computer Design", (IEEE Trans. on Comp., C-20(11), November 1971), pp. 1312-1321.

6. Burroughs Corp., Microprogramming Manual for Interpreter Based Systems, (Defense, Space and Special Systems Group, Paoli, Penn., November 1970).

7. Davis, R. L., et al., "The Building Block Approach to Multiprocessing", (Proc. SJCC, 1972), pp. 685-703.

8. Reigel, E.W., et al., "The Interpreter - A Microprogrammable Building Block System", (Proc. SJCC, 1972), pp. 705-723.

9. IEEE Trans. on Comp., Special Issue on Microprogramming, C-20(7), July 1971.

10. Entner, R.S., "The Advanced Avionic Digital Computer", Chapter 10 in Parallel Processor Systems, Technologies, and Applications, (Hobbs, L.C. et al., (eds.), Spartan Books, New York, 1970), pp. 203-214.

11. Entner, R., "Advanced Avionic Digital Computer Development Program Report Number 9", (Naval Air System, Washington, D.C., November 1, 1971, Code AIR-5333F4).

12. Bell, C. G., et al., "Effect of Technology on Near Term Computer Structures", (Computer, 5(2), March/April 1972), pp. 29-38.

13. Goodyear Aerospace Corp., "STARAN - A New Way of Thinking", (Sales Brochure, Computer Marketing Division, Akron, Ohio).

CHAPTER 10

SYSTEM PERFORMANCE MONITORING TO AID OPTIMIZATION

10-1.0 Introduction

A method of optimizing computer system design that has been successfully used [1,2], is to monitor the system while it is in operation and collect data reflecting its performance. This data then indicates how to vary system parameters to achieve better performance, or it indicates if optimal performance with respect to these parameters has already been achieved. Monitoring can also be done at the process level to enable programmers to observe the dynamic performance of their programs and how they interact with the system. Very often static measurements of a program cannot determine its dynamic performance. For example, the number of different operation codes included in a program does not indicate the number of times each type of opcode is executed when the program is run. For this reason a dynamic monitoring of performance is necessary.

The purpose of this chapter is to survey some of the ways system designers have used monitoring to aid both hardware and software system design and to review their findings. More specifically, we will present some of the issues that the system designer should be aware of when building a monitoring system. Then we will see two approaches to building such a system that one can take and present an example of each in the form of a specific implementation. Most importantly we will see how the data collected can help optimize system design, thus making performance monitoring a technique with which a system designer should be familiar.

10-2.0 Design Issues

One might ask the value of performance monitoring over micro-simulation in optimizing system design. For a large system such as MULTICS or TSS/360 having random job streams, microsimulation is

impractical, if not just about impossible, due to the size of the system. On the other hand, macrosimulation presents a statistical simulation of system performance but does not carry the simulation down to the register level. Thus, there are questions it cannot answer about the design. To observe the actual performance of a large system down to the register level dynamic monitoring is necessary. However, in designing such a monitoring system the architect must be aware of several design issues.

1. The designer must be able to monitor during normal system operation. This point is obvious, otherwise the monitoring system would be worthless.

2. The monitor should interfere as little as possible with the actual processes being measured so as not to destroy the integrity of the data.

3. It must record all occurrences of the events being measured so that no sampling uncertainty is introduced.

4. The monitoring algorithm should concentrate on gathering data and save as much analysis of this data as possible for later processing.

5. If the monitoring system is meant to be general purpose, i.e., meant to measure many aspects of system performance at the option of the designer, it should be expandable to allow easy inclusion of monitoring functions determined necessary in the future. Such a general purpose system can be a diagnostic aid to programmers.

10-3.0 Two Approaches to Monitoring

The two approaches to performance monitoring are by means of hardware and by means of software. With the former a special piece of hardware is interfaced with the system to count the occurrences of a specific event(s). If this approach is chosen, the designer must know exactly what he wishes to monitor since a hardware redesign can be costly. Schroeder [1] discusses a monitoring experiment using extra hardware to determine the optimal number of associative registers the MULTICS system should have. We will describe this work shortly.

The second approach to performance monitoring is via a programming system using as little extra hardware as possible. In this case the designer must be sure the system is general enough so that massive

reprogramming is unnecessary should monitoring requirements change. Grochow [2] designed such a system using a graphic display unit to present data and implemented it as part of MULTICS. We will also describe his work as an example of the software approach.

10-3.1 An Example of Hardware Monitoring

The MULTICS system, implemented on a GE 645 computer, allows paging and segmentation of addressable memory [3,4]. A given address reference contains a segment number s and a word number w. To translate this address into an absolute memory location a several step process is necessary, as shown in Figure 10.1.

1. The contents of the descriptor base register is added to part of s to give the address of the desired page of the descriptor segment.

2. This address and the remainder of s locate the segment descriptor word (SDW), which points to the page table for segment s.

3. Part of w is added to the SDW to give the address of the desired page table word (PTW), which points to the starting location of the desired page.

4. This address and the remainder of w determine the absolute core address desired.

If this entire process were performed every time a core reference was made, the MULTICS system would be unacceptably slow. To increase execution speed a set of m associative registers was added to the system to aid in address translation.

The associative registers contain the m most recently used SDWs and/or PTWs. Upon referring to an address a search is made of this associative memory under the assumption that most programs have sufficient reference locality to enable reuse of a small set of address translation table entries. The search key is the segment number and word number of the address being translated. The desired result is the PTW and/or SDW needed.

The value of the associative memory is obvious, and we will present quantitative results shortly. However, Schroeder wanted to determine the optimal value of m. He attached an electronic counter to the GE645 to measure four events: instruction executions, associative memory searches, no match associative memory responses, and absolute core references by the processor. During the measurement period MULTICS ran under a normal user load. In addition, measurements

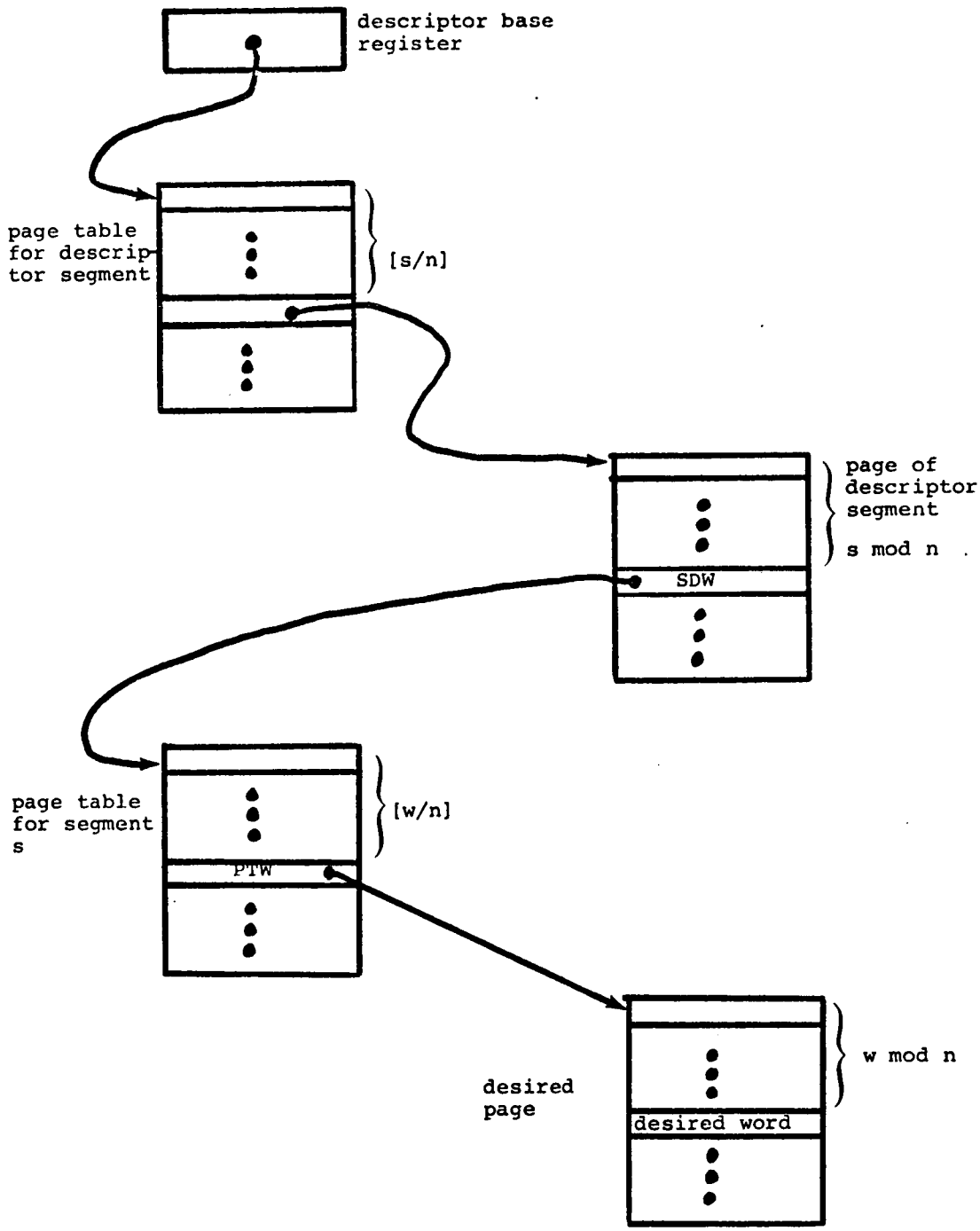


Figure 10.1: Translation of address (s,w) by MULTICS, where n = page size

were taken for m=0, 4, 8, and 16. Schroder's results are shown in Figure 10.2.

The inclusion of even 4 associative registers has a dramatic effect upon system performance. Moreover, the optimal number of registers was found to be 16. An increase beyond this number was predicted to have no significant effect upon system performance. This experiment is an excellent example of system study and optimization through hardware monitoring.

10-3.2 An Example of Software Monitoring

Grochow [2] has implemented a Graphic Display Monitoring System (GDM) as part of MULTICS, enabling a visual presentation of the dynamically changing properties of the operating system. GDM runs on a PDP-8 computer with CRT units, the display computer, which is in turn interfaced with the GE645. This is a programming system which allows users to type in their display requests on the PDP-8 using a special language for describing the desired data manipulation and display formats. Users may request predefined displays or create their own displays. In the latter case they must define format, data to be displayed, and sampling rate. This use of two computers insures as little interference with MULTICS processes as possible. The only requirement upon MULTICS is the running of a special process to transmit data as requested to the display computer.

Some typical displays for overviewing the MULTICS system are the following:

a. the major hardware modules in the configuration, e.g., number of drums, memory modules, etc.;

b. a core map which indicates system loading; and

c. the number of active processes.

Displays can also be presented on the process level. The general nature of the monitoring system allows flexibility in the data displayed. The user may provide any segment number and offset of a piece of data and get an octal and ASCII representation of the data, updated once per second.

10-4.0 Further Examples

In this section we will review two further examples of dynamic monitoring used to optimize system performance. The first example studies a time sharing system's performance, while the second studies opcode utilization on a computer.

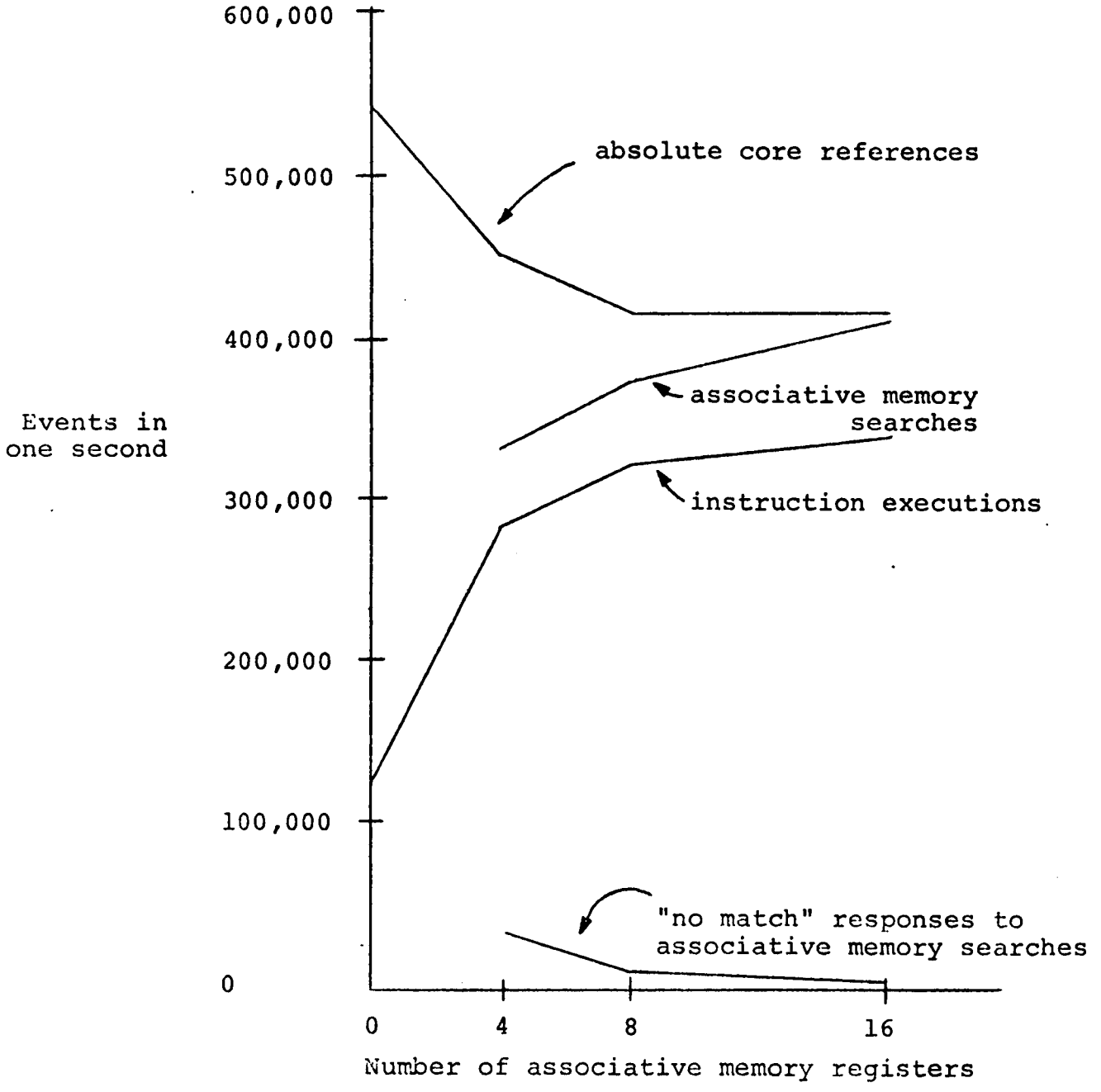


Figure 10.2 - Multics Performance with Various Associative Memory Sizes [1]

10-4.1 Time - Sharing Performance Monitoring

The University of Michigan has implemented a performance monitor on their time sharing system which executes on an IBM System/360, model 67 [5]. It is a software monitoring facility that has been used to collect a variety of data on system performance. Such basic data as interrupt processing times, resource utilization, and page swap times have been collected and studied. The result has been, as with MULTICS, that system programmers can now make informed choices of system parameters in their quest of optimizing performance. For example, the data on page swap times and delays has enabled helpful changes to the demand paging algorithm which led to a decrease in these delay times. Furthermore, as more data of this type is collected, similar improvements can be expected.

10-4.2 Opcode Utilization

Foster et al., [6], have studied opcode utilization on a CDC 3600 computer. Their goal was to determine if all opcodes are sufficiently used to justify their inclusion in the instruction repetoire. Two types of code were statically and dynamically monitored for opcode usage. The first was a set of hand coded assembly language programs. The second was object assembly code produced by Fortran, Cobol and Simscript compilers.

The static measures show that the hand coded programs have more diverse opcode usage than the compiler code. If the 142 CDC3600 instructions were limited to 64, about 2% of the hand coded instructions would not be included but all the object code would be covered. Moreover, if the instruction set were limited to 32, 10-16% of the hand code and 0-3% of the object code would have to be redone. Thus, little flexibility would be lost in somewhat limiting the available CDC3600 instructions.

The dynamic measures show little difference in opcode usage for hand code and object code. About half of the available instructions were rarely executed. This fact provides further justification to limit the available opcodes and thus, save on the cost of CPU hardware. Studies such as this are useful to the system architect who must design an instruction set which is flexible yet not too costly to implement.

10-5.0 Summary

The conclusion we wish to draw is obvious from what has been presented. Dynamic performance monitoring can provide valuable data for the system designer. The examples cited substantiate this point. It is certainly a design technique with which the informed system designer should be familiar.

REFERENCES

1. Schroeder, M.D., "Performance of the GE-645 Associative Memory While MULTICS is in Operation", ACM SIGOPS Workshop on System Performance Evaluation, Harvard University, April 1971, pp. 227-245.

2. Grochow, J.M., The Graphic Display as an Aid in the Monitoring of a Time-Shared Computer System, Project MAC, M.I.T., TR-54(Thesis), October 1968.

3. Corbato, F.M., and Vyssotsky, V.A., "Introduction and Overview of the MULTICS System", Proc. FJCC, 1965, pp. 185-196.

4. Denning, P.J., "Virtual Memory", Comp. Sur., 2(3), September 1970, pp. 153-189.

5. Pinkerton, T.B., "Performance Monitoring in a Time-Sharing System", CACM, 12(11), November 1969, pp. 608-610.

6. Foster, C.C. et al., "Measures of Op-Code Utilization", IEEE Trans. on Comp., C-20(5), May 1971, pp. 582-584.

CHAPTER 11

FAILURE DETECTION TECHNIQUES

11-1.0 Introduction

Whenever a system is in operation, a prime concern of those who are using it is its correct performance. Any error that the system commits must be caught and somehow corrected to insure the integrity of the process it is performing. Even presuming extremely careful system design and implementation, factors such as aging components can cause costly system errors. To prevent these errors from having catastrophic effects, failure detection techniques must be employed to identify malfunctioning equipment.

Failure detection techniques take many forms and may be employed at many levels in the system. At the circuit level diagnostic input sequences can be periodically fed into a circuit to check for its correct operation. The corresponding circuit output then indicates any faulty circuit performance. On a higher level error detecting and correcting codes can be employed to identify transmission errors between system modules.

Redundancy is a frequently used technique to detect errors. For example, the output of three identical subsystems is fed into a voter. Should two of the three subsystems have identical output, these subsystems are presumed to be operating correctly. The remaining subsystem is judged to be faulty and can be replaced with standby equipment. In this case we say that the system has been reconfigured. Voting is a technique that may be employed at many system levels.

In this chapter we will investigate several methods of error detection such as those mentioned above. However, when one discusses error detection, the topic of error correction also arises. We will devote part of this chapter to error correction techniques as well.

We will begin by introducing the topic of error detection from the point of view of finite state automata. Fault detection methods for these machines will lead us to the idea of designing machines with diagnosis in mind. It will also be seen that these techniques are readily applicable to digital circuits. Next we will investigate error detection and correction methods in digital circuits, such as voting, quadded logic, and error detecting codes. Other methods of correction, including reconfiguration and software rollback, will also be treated. Finally, we will look at some specific applications of these methods in computers such as STAR and SIRU.

11-2.0 Diagnosing Sequences for Finite State Automata

Finite state automata provide a convenient model for discussing sequential circuit design. Properties of these circuits become apparent when represented as automata in their state table form, and this fact provides a means of studying the diagnosability of a sequential circuit. In order to diagnose we must fully understand the state transitions of the circuit as shown by the state table. We will now present a brief introduction to finite state machines (FSM) before proceeding to diagnosing sequences. More complete treatments of these topics may be found in the literature [1,2].

Def: A finite state machine M is a 5-tuple

(S, I, O, ρ, η) where

S = a set of states $\{s_1, \ldots, s_n\}$;

I = a set of input variables $\{i_1, \ldots, i_m\}$;

O = a set of output variables $\{\sigma_1, \ldots, \sigma_p\}$;

ρ : $S \times I \rightarrow S$ is a map taking M into a new state depending upon the old state and input variable; and

η : $S \times I \rightarrow O$ is a map yielding an output variable from M depending upon the old state and input variable.

We will assume here that $I = O = \{0, 1\}$; i.e., there are two possible input and output variables, 0 and 1.

Example: We may represent a machine M_1, by Table 1. This is M_1's state table representation. Each table entry denotes the new state and output variable of M_1, for M_1 in a given state being presented

with an input variable. M_1 may also be represented in the state diagram form of Figure 11.1. Table 2 and Figure 11.2 are another example of these FSM representations for machine M_2.

Def: An FSM is strongly connected if from every state we can get to every other state.

M_1 and M_2 are strongly connected. M_3 shown in Figure 11.3 is not.

Def: An FSM is reduced if there are no equivalent states. That is, for each two states of the machine there is a finite input sequence which yields one output sequence when the machine is started in one state, and a different output sequence when it it started in the other state.

We assume for the remainder of this section that all FSMs are strongly connected and reduced. We will now begin defining the properties of some very useful input sequences for FSMs. These properties will in turn be used for diagnosis.

Def: An input sequence is a homing sequence for M if its application yields an output sequence which uniquely determines the final state of M, independent of initial state.

Example: The sequence 10 is homing sequence for M_1. The possible output sequences corresponding to this input sequence are 00, 01, 10, 11. These correspond to the final states 3, 1, 2, 1, independent of M_1's starting state. Thus, by application of 10, the final state of M_1 is determined by its output sequence. Hennie [2] gives an algorithm for determining a homing sequence which we will not present here. Let us just note that every reduced, strongly connected FSM possesses a homing sequence. However, the homing sequence provides no information about M's initial state. Thus, we make the following definition.

Def: An input sequence is a distinguishing sequence for M, an n state machine, if its application yields n different output sequences depending upon M's initial state. Thus, such a sequence enables the determination of M's initial state.

To construct a distinguishing sequence for M, combinations of input symbols must be systematically tried to eliminate the ambiguity of what M's initial state is. Figure 11.4 shows the construction of a distinguishing sequence for M_2. Each succeeding lower level of the tree represents a part of the initial state ambiguity being resolved until finally no ambiguity remains. For example, in going from level

state \ input	0	1
1	2,0	3,0
2	4,1	1,1
3	3,0	5,1
4	5,0	3,0
5	1,1	5,0

Table 1: State Table for M_1

b

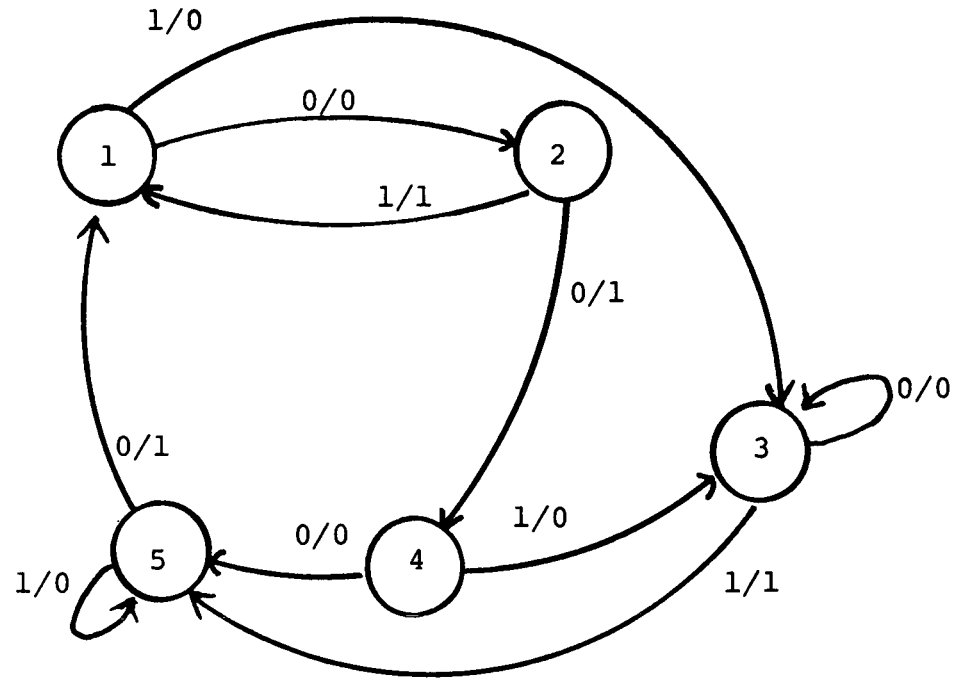


Figure 11.1: State Diagram of M_1

state \ input	0	1
1	3,0	4,0
2	1,0	3,0
3	4,1	2,0
4	3,1	1,1

Table 2: State Table for M_2

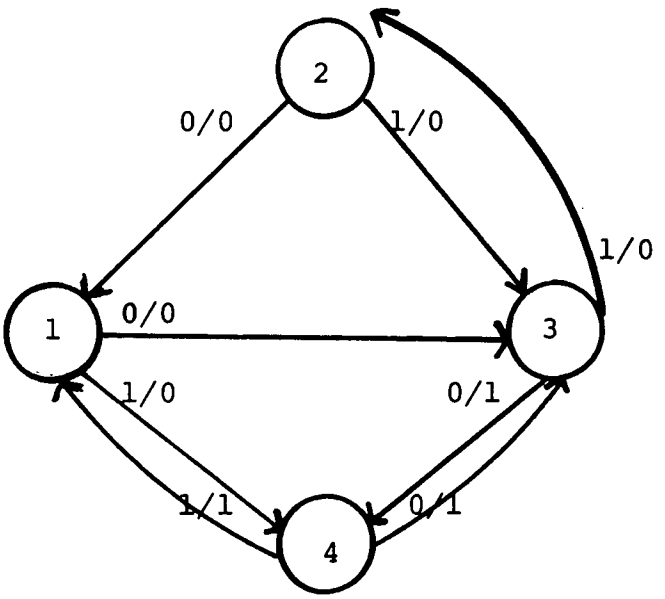


Figure 11.2: State Diagram of M_2

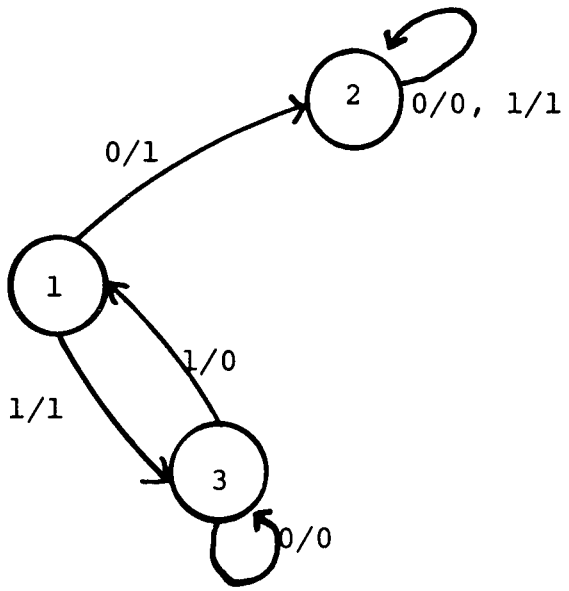


Figure 11.3: State Diagram of M_3

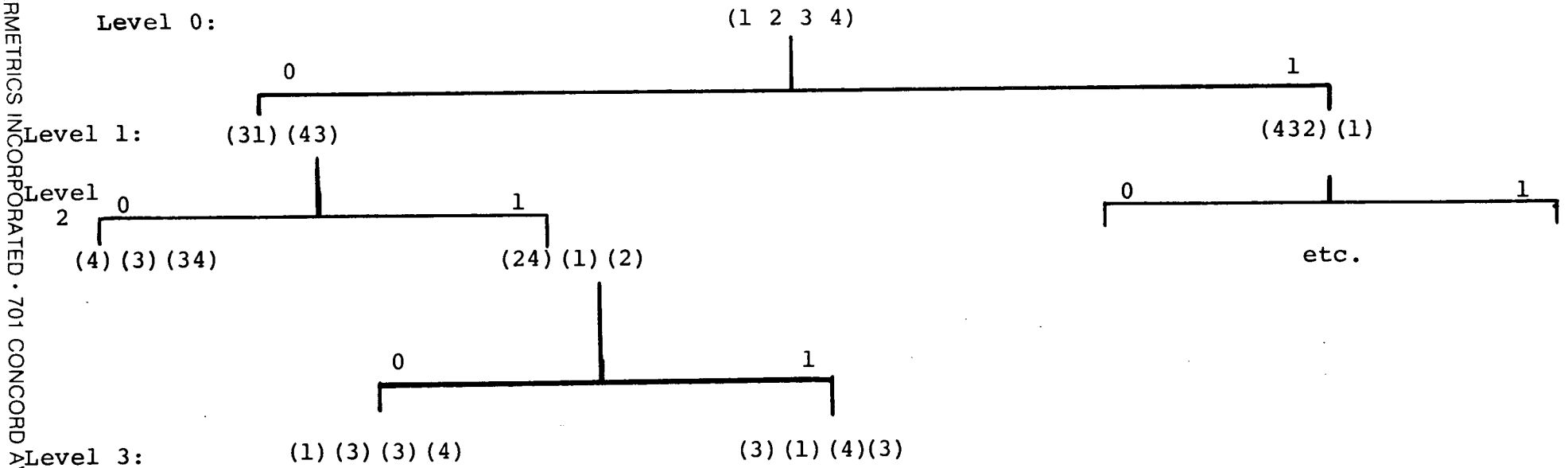


Figure 11.4: Construction of Distinguishing Sequence for M_2

0 to level 1, application of a 0 input results in 0 output if M_2 is in states 1 or 2 and 1 output if M_2 is in states 3 or 4. The new states to which M_2 goes after application of 0 or 1 are noted on level 1. Finally, at level 3, we see that the sequences 010 and 011 are distinguishing sequences for M_2 because their application to M_2 yields 4 distinct output sequences depending upon M_2's initial state. No initial state ambiguity remains at level 3.

Note that all FSMs do not possess distinguishing sequences. For example, M_1 does not, as may be verified by trying to construct a distinguishing sequence similar to that of Figure 11.4. With the concept of homing and distinguishing sequences in mind let us now proceed to diagnosis of FSMs.

Def: An input sequence for M is a diagnosing (testing) sequence if it determines whether M's state table accurately describes M's present performance.

Thus, a diagnosing sequence must check M's performance for each possible transition from each state. To do this testing M must possess a distinguishing sequence as a prerequisite to possessing a diagnosing sequence. Hennie [2] presents an algorithm for constructing a diagnosing sequence for such an FSM.

In using this algorithm we presume that any malfunction of M, for which we will use the diagnosing sequence to test, does not increase the number of M's states. Hence, we always know the number of states of M from its state table. The steps of the algorithm are the following:

a) apply a homing sequence to identify M's present state, followed by a sequence to bring M to a desired state from which the diagnosing begins·

b) apply a sequence incorporating the distinguishing sequence such that the correctly operating machine displays the response of each of its states to the distinguishing sequence;

c) apply an input sequence checking the remaining transitions unchecked in (b).

The output of M under steps (a)-(c) determines if M is operating correctly. Any incorrect output response from M indicates a malfunction.

We will now construct a diagnosing sequence for M_4 whose state table is given in Figure 11.5a. First note that 10 is a homing sequence for this machine, and 10 is also a distinguishing sequence. Figure 11.5b shows M_4's response to 10. The homing sequence allows us to bring

state \ input	0	1
1	2,1	1,0
2	3,0	1,1
3	1,0	2,0

Figure 11.5a: State Table for M_4

initial state	output	final state
1	0 0	1
2	1 0	1
3	0 1	2

Figure 11.5b: Response of M_4 to 10

M_4 to state 1 to begin diagnosis. Now assuming M_4 works correctly, application of the distinguishing sequence with initial state 1 leaves the machine in state 1. Applying 1, M_4 is then in state 2. We apply the distinguishing sequence again while M_4 is in state 2 and similarly for state 3. The input sequence for part (b) of the algorithm is

1 0 1 1 0 0 1 0 0 1 0 1, and the correct response is

0 0 0 1 0 0 0 1 0 0 0 0. Any incorrect response demonstrates a machine malfunction.

It now remains to check the reamining state transitions. To do this we must be sure that the state of M_4 before and after each transition can be uniquely specified in terms of behavior shown previously in the testing. The sequence of part (c) of the algorithm is

0 1 0 1 1 1 0 0 0 1 0 0 1 1 0 0 0 1 0 0 1 1 0 with correct response

0 0 0 0 1 1 0 0 1 1 0 0 0 0 1 0 0 0 1 0 0 1 0.

The total testing sequence uses 37 input variables for a 3 state machine. Even using heuristics to reduce the number of input variables does not yield a very practical testing procedure for a sequential circuit. However, the use of some extra hardware greatly reduces the length of the diagnosing sequence needed [3]. We will now discuss this approach to diagnosis.

11-3.0 Diagnosis Using Augmented Hardware

An n state FSM M is said to be definitely diagnosable (DD) if any input sequence of length k, where $k \leq \frac{n(n-1)}{2}$, is a distinguishing sequence for M. We will see how testing the state table will reveal if M is DD or not. In the latter case a hardware augmentation will transform this machine into a machine M´ which is DD. This transformation consists of adding output variables to M, as shown in Figure 11.6. In M´ the original operation of M is not changed, but we now have an additional output channel to aid diagnosis. Kohavi [3] has shown that every strongly connected and reduced FSM can be made DD in such a manner, whether or not the original FSM possesses a distinguishing sequence. We will see how diagnosing sequences for a DD machine need far less input variables than for a non DD machine.

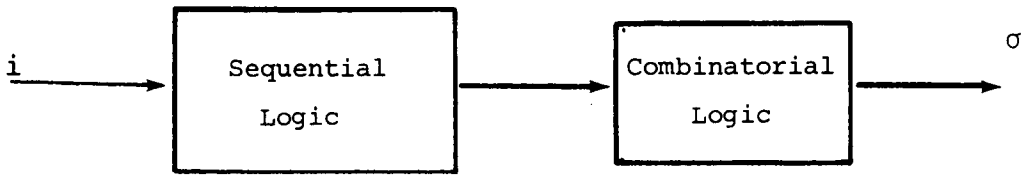


Figure 11.6a: Machine M

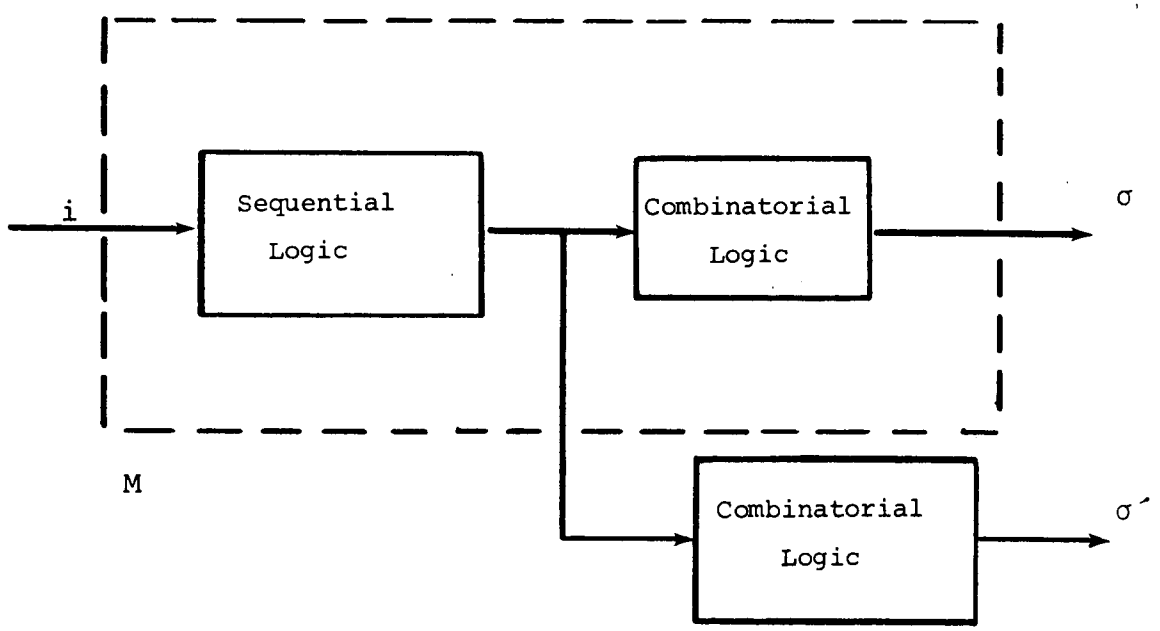


Figure 11.6b: Machine M´

To test M for the DD property we augment its state table to form a testing table. A testing table for M_5 shown in Figure 11.7, is given in table 3. The lower half of the testing table contains all possible uncertainty pairs of states and is constructed from the upper half. A row index of the lower half represents pairs of states which are distinguishable if the entries in its row are distinguishable. A circled entry represents a pair of states indistinguishable by means of an input variable equal to the column index.

From the lower part of the testing table of M we construct a testing graph as shown in Figure 11.8. To transform M into M´ we must

a) eliminate all circled entries of the testing table by assigning different outputs to the corresponding next state entries, and

b) open all testing graph loops by eliminating the smallest number of branches in the graph. The elimination is done by assigning distinct output symbols to the next state entries covered by the node to which the branch leads.

Table 4a shows step (a) applied to M_5, and table 4b shows step (b) applied. Table 4b is the state table of M_5', a DD FSM.

Kohavi [3] outlines the construction of a diagnosing sequence for a DD FSM. The algorithm is the following:

a) bring M´ into a known state, say A;

b) choose the shorter distinguishing sequence S consisting of all 0's or all 1's, (assume here all 0's);

c) apply S followed by a 0 to check M´'s first transition from A under a 0 input;

d) continue with 0 inputs as long as new transitions are being checked;

e) when more 0's don't yield new transitions apply 1 followed by S;

f) apply 0's as long as new transitions are being checked, otherwise go to step (e);

g) when (e) and (f) yield no new transitions, go to a state whose transition has not been checked, using a sequence that goes through checked transitions only;

h) repeat (e)-(g) until all transitions are checked.

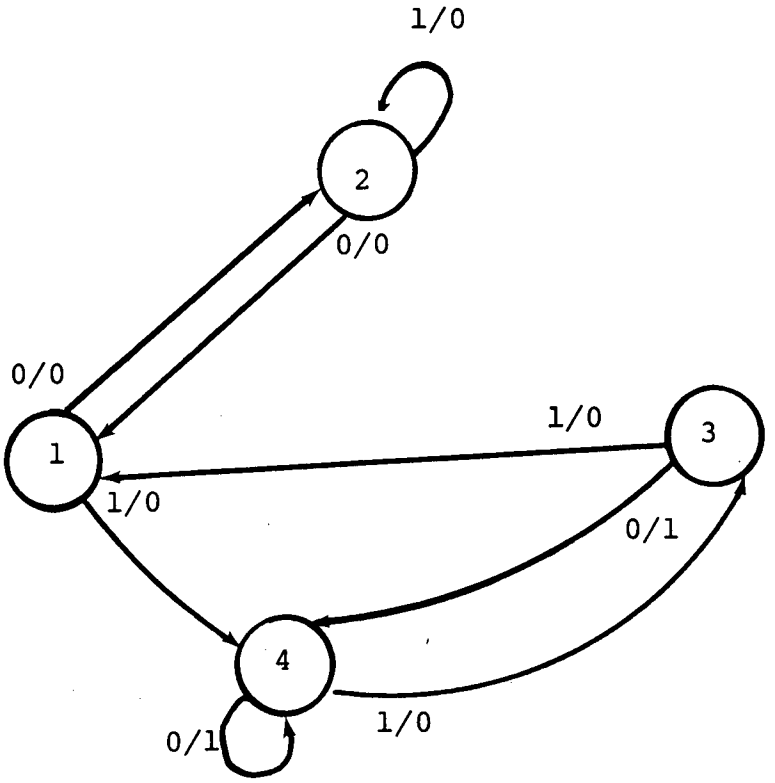


Figure 11.7: State Diagram of M_5

i/o / state	0/0	0/1	1/0	1/1
1	2	-	4	-
2	1	-	2	-
3	-	4	1	-
4	-	4	3	-
12	12	-	24	-
13	-	-	14	-
14	-	-	34	-
23	-	-	12	-
24	-	-	23	-
34	-	(44)	13	-

Table 3: Testing Table for M_5

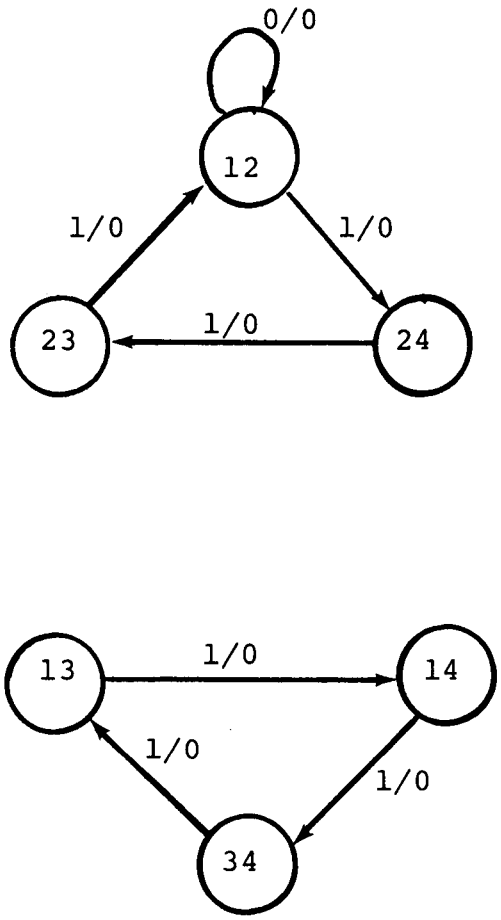


Figure 11.8: Testing Graph of M_5

input / state	0	1
1	2,0	4,0
2	1,0	2,0
3	4,10	1,0
4	4,11	3,0

Table 4a: Step (a) applied to M_5

input / state	0	1
1	2,01	4,00
2	1,00	2,00
3	4,10	1,01
4	4,11	3,01

Table 4b: State Table of M_5'

Application of this algorithm to M_5' yields the diagnosing sequence:

input: 0 0 0 1 0 1 0 0 1 0 0 1 1 0

state: 1 2 1 2 2 1 4 4 4 3 4 4 3 1 2

output: 1 0 1 0 0 0 3 3 1 2 3 1 3 1

This sequence needs only 14 symbols, as compared to the 152 symbols Hennie [4] needed to diagnose M_5. This is an 11 to 1 savings. Thus, by the addition of a small amount of hardware dramatic reductions in the length of diagnosing sequences can be realized. Hennie's method (section 2) has a bound of $2n^4(n + 1)!$, where n is the number of states of M; Kohavi's method has a bound of n^3. For $n = 8$, we have 3×10^8 vs. 512. However, either method is only useful when periodic diagnosis of the circuit is acceptable.

For a circuit with a large number of states even the second method of diagnosis becomes impractical. The system designer now faces two choices to achieve fault detection. First, he can decompose his circuit into a series-parallel array of subcircuits using methods of partition algebra [2,5]. Then the subcircuits can be checked in parallel, each having a much shorter diagnosing sequence than the original circuit. Second, he can use voting or quadded logic to detect and correct errors. However, either approach involves additional hardware costs.

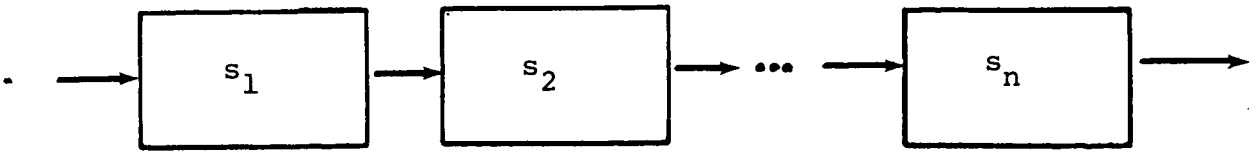
11-4.0 Additional Detection Methods

We will now investigate other methods of failure detection in digital circuits. These detection methods will ofteh involve correction of the error as part of the method.

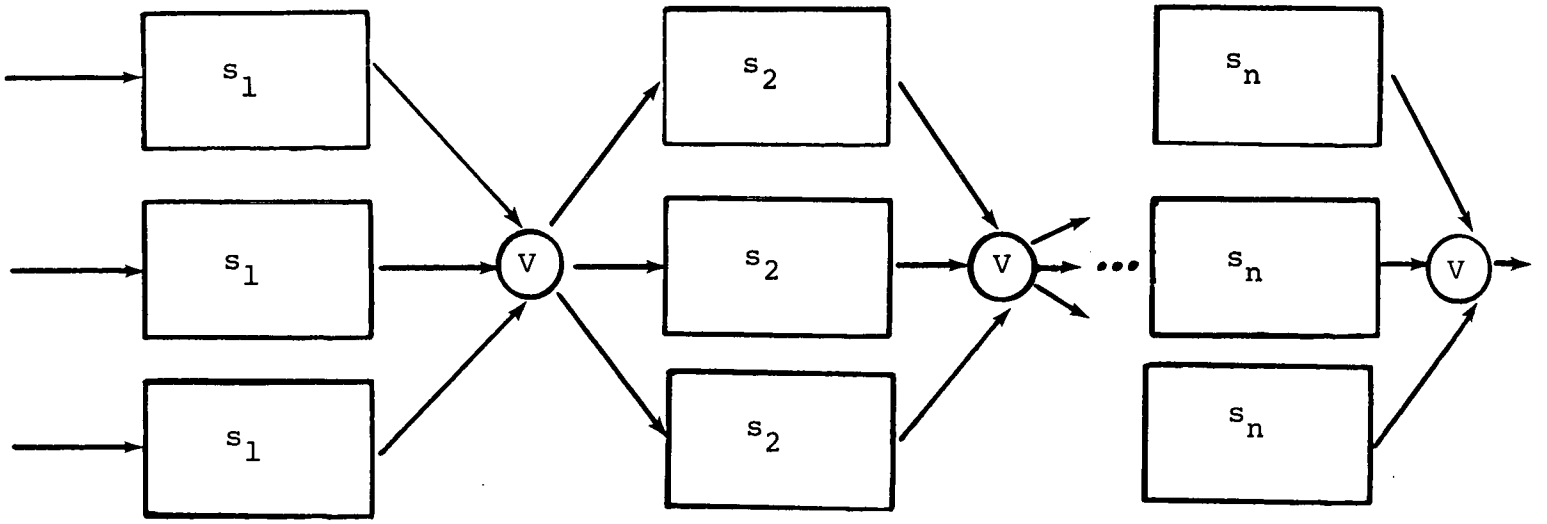
11-4.1 Voting

Voting detects errors by comparing three redundant signals. If two out of the three agree, the nonagreeing signal is judged erroneous, and the faulty equipment that generated this signal is identified. Thus, voting automatically provides correction by transmitting the correct signal as its output and automatically locates the faulty equipment. The price to be paid for using voting is the triplication of equipment and the cost of the voter.

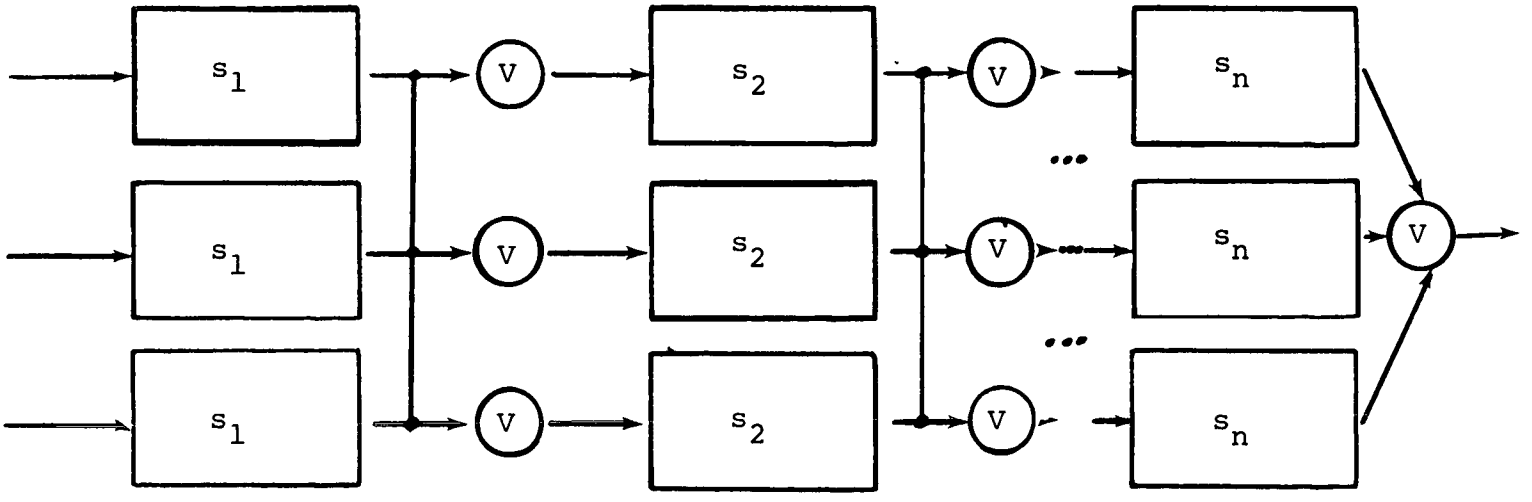
Despite this price the reliability of most networks can be improved using voting. Let us consider the three networks shown in Figure 9, where the reliability of each module in stage i is p_i and the reliability of eacn voter is v. If the number of stages



(a) nonredundant



(b) redundant with one voter per stage



(c) redundant with three voters per stage

Figure 11.9: Voter Configurations

n = 1, the reliability of the nonredundant network, N_1, is $R(N_1) = p$, and the reliability of the redundant network with one voter per stage, N_2, is $R(N_2) = v(3p^2 - 2p^3)$. For voting to be advantageous in this case we must satisfy the condition

$$p < v(3p^2 - 2p^3).$$

For $v \approx 1$ this condition reduces to $p > \frac{1}{2}$. That is, even using a very reliable voter the reliability of a module must be greater than $\frac{1}{2}$ for voting to increase the reliability of the circuit.

Let us now state the reliability equations for the three networks of Figure 9 where we now have n stages.

$$R(N_1) = \prod_{i=1}^{n} p_i$$

$$R(N_2) = v^n \prod_{i=1}^{n} (3p_i^2 - 2p_i^3), \text{ and}$$

$$R(N_3) = v(3p_1^2 - 2p_1^3) \cdot \prod_{i=2}^{n} (3p_i^2 v^2 - 2p_i^3 v^3)$$

Gurzi [6] shows that $R(N_2)$ and $R(N_3)$ are maximized when

a) $p_1 = p_2 = \ldots = p_n = p$ and

b) $\frac{dR(N)}{dp} = 0$ is observed.

Condition (b) for $R(N_2)$ yields

$$v = \frac{1}{(3-2p)p^{\alpha}}, \text{ where } \alpha = \frac{2p}{3-2p}$$

For $R(N_3)$ condition (b) becomes

$$(3-2pv)\ln(3v^2-2pv^3) + 2pv \ln p = 0$$

In both cases the optimal reliability is independent of the reliability of the nonredundant network.

11-4.1.1 Detection Level

Detection at a high system level, e.g., modular vs. circuit, has certain advantages and disadvantages. Its advantages are:

a) The cost of voters is reduced because less voters are necessary.

b) There is easier replacement of faulty equipment. An entire module detected to be in error is either replaced manually or by automatic system reconfiguration.

c) In the case of a digital computer (b) implies less complex reconfiguration software.

The disadvantages are:

a) There is no identification of where in a module the error occurred. Hence, repair is more difficult.

b) More redundant equipment must be standing by for replacement. This fact means added weight and volume in aerospace applications.

c) There can be a time lag in detecting an error. The module must communicate with its environment through the voter before an error can be detected.

The level at which to detect for system malfunctions depends upon the characteristics of the equipment, costs, reliabilities of subsystems, and the application involved. There is no readily available rule to a priori decide whether to use redundancy, and if so, where to place voters in a system. The above factors must all be weighed before deciding where to detect for malfunctions.

11-4.2 Path Sensitizing

Armstrong [7,8] developed a method of testing combinatorial circuits, called path sensitizing. An input pattern is applied to the circuit so that the output depends only on the one input lead being tested. For example, consider the network shown in Figure 11.10. Suppose we wish to make the output z directly dependent upon the input variable c. By letting a = 0, b = 0, d = 1, e = 1, the output

z equals the value of c. Systematic use of this technique allows tracing of signal flow in a network and determining the output due to specific inputs and faults present [9-11].

The disadvantage of this method is that certain errors cannot be detected when the circuit has fan-out and convergent paths. This problem has been overcome with a generalization of path sensitizing known as the D-algorithm [11,12].

11-5.0 Correction Techniques

In this section we will discuss error correction techniques that can be employed when a system error is detected. These techniques include error detecting and correcting codes, quadded logic, and software restart.

11-5.1 Error Correcting Codes [13-15]

We are all familiar with elementary types of error detecting codes. For example, the use of parity bits in a digital computer is a method of detecting a single error in a group of bits. However, there is no correction of this error possible with the use of parity bits, so we must use a more sophisticated code to correct a detected error.

The Hamming code allows single error detection and correction. Suppose a transmitted word of m bits is to be checked using a Hamming code. We require k extra bits to locate any error in one of the transmitted bits. The code bits are transmitted along with the information bits and are designed so that a single error in any one of (m + k) transmitted bits can be found and corrected.

The value of

$$k = \text{the smallest integer such that } 2^k \geq m + k + 1.$$

For example, if m = 8, k = 4.

The k check bits are placed in positions 1, 2, 4, 8, ..., 2^{k-1} of the transmitted word. Each check bit is assigned a value so that a subset of the (m + k) bits (including the check bit) has even parity. For example, the check bit in position 1 has value so that all bits in odd positions have even parity. The ckeck bit in position 2 has value so that bits in positions 2, 3, 6, 7, 10, 11, ... have even parity, etc.

When each of these subsets of bits are checked upon receiving the coded word, an odd parity denotes an error. In fact the code is constructed so that the positions of all odd parities corresponding to check bit positions denotes the position of the error. The correction procedure is merely assigning the erroneous bit its complement value.

As an example suppose m = 8. Let the word to be transmitted be 1 1 0 1 1 1 0 0. Its coded form is $k_1\ k_2\ 1\ k_4\ 1\ 0\ 1\ k_8\ 1\ 1\ 0\ 0$, where we calculate k_i as follows.

$$k_1 = k_1 \oplus k_3 \oplus k_5 \oplus k_7 \oplus k_9 \oplus k_{11} = 0$$

$$k_2 = k_2 \oplus k_3 \oplus k_6 \oplus k_7 \oplus k_{10} \oplus k_{11} = 1$$

$$k_4 = k_4 \oplus k_5 \oplus k_6 \oplus k_7 \oplus k_{12} = 0$$

$$k_8 = k_8 \oplus k_9 \oplus k_{10} \oplus k_{11} \oplus k_{12} = 0$$

Thus, the transmitted word becomes

0 1 1 0 1 0 1 0 1 1 0 0. Now suppose an error occurs in position 9 upon receiving the word. Thus bit 9 is 0. When the Hamming code is reconstructed odd parities are found for k_1 and k_8. Thus, $k_1k_2k_4k_8 = 1001 = 9$. The code gives the position of the error. Hence, we can correct by setting bit 9 to 1. Such a correction technique can easily be implemented in digital hardware.

11-5.1.1 Distance Codes

The Hamming code falls into a more general category of codes called distance codes. Some other distance codes have greater capabilities such as detecting and correcting multiple errors. Let us now define the distance d between two n bit words, p and q. Let the words be represented in binary form

$$p = p_1\ p_2 \cdots p_n \text{ and } q = q_1\ q_2 \cdots q_n.$$

Then we define $d = \sum_{j=1}^{n} (p_j - q_j)^2$.

If the collection of information that we wish to work with is encoded into a set S of words, the distance between any two distinct members of S must be at least 1. The higher the minimum value of d for any two members of S the greater the detection and correction capabilities of our encoding. We summarize these capabilities in the following table.

Minimum distance d between any two distinct members of S	Capability of Encoding
1	unique words, no detection
2	single error detection
3	single error correction, or double error detection
4	single error correction and double error detection, or triple error detection
5	double error correction, or single error correction and triple error detection, or quadruple error detection

Note that a Hamming code is distance 3. As the capabilities of the code increase (i.e., d increases), more bits must be used to encode the working set of information. This fact must be considered when deciding upon what detection and correction capabilities the code is to have.

11-5.1.2 Binary Cyclic Codes

Many codes utilize algebraic structures, such as groups and rings, for their error detection properties [14]. One such code utilizing the properties of polynomials is the binary cyclic code [13].

Def: A code C is cyclic if it has the property:

an n-tuple $\bar{a}=(a_0,a_1, \ldots, a_{n-1}) \in C$

implies $\bar{a}^{(1)} = (a_{n-1}, a_0, a_1, \ldots, a_{n-2}) \in C$ also.

We may consider the members of C to be coefficients of (n-1)th order polynomials. We define

$$\bar{a}(x) = a_0 + a_1 x + \ldots + a_{n-1} x^{n-1} \quad \text{and}$$

$$\bar{a}^{(i)}(x) = a_{n-i} + a_{n-i+1} x + \ldots + a_{n-i-1} x^{n-1} \quad .$$

Then $\bar{a}^{(i)}(x)$ is the result of dividing $x^i \bar{a}(x)$ by x^n+1. Thus,

$x^i \bar{a}(x) = \bar{q}(x)\ (x^n+1) + \bar{a}^{(i)}(x)$, where $\bar{q}(x)$ is a polynomial in x.

If our code C uses k out of n bits for information and n-k for redundancy, we have an (n,k) cyclic code. In an (n,k) cyclic code there is one unique polynomial $\bar{g}(x) \epsilon C$ of degree n-k such that every polynomial in C is a multiple of $\bar{g}(x)$ and such that every polynomial of degree n-1 or less which is a multiple of $\bar{g}(x)$ must be a member of C. Thus, every member of C can be expressed as $\bar{a}(x) = \bar{m}(x)\ \bar{g}(x)$. $\bar{m}(x)$ is of degree k. Let the coefficients of $\bar{m}(x)$ be the k information bits. Then every coded message is generated by multiplying $\bar{m}(x)$ by $\bar{g}(x)$. We call $\bar{g}(x)$ the generator polynomial of the code. The following facts apply to $\bar{g}(x)$:

a) the generator polynomial of an (n,k) cyclic code is a factor of $x^n + 1$; and

b) if $\bar{g}(x)$ is of degree n-k and is a factor of x^n+1, $\bar{g}(x)$ generates an (n,k) cyclic code.

An (n-k) stage shift register can be used to generate a (n,k) cyclic code. The k information bits $\bar{m}(k)$ are fed into the circuit and form the first k bits of the code unchanged. The circuit output then forms the next n-k bits, and this is the redundancy information (check bits). For example, $x^7+1 = (x^3 + x + 1)\ (x^4 + x^2 + x + 1)$. Thus $\bar{g}(x) = x^3+x+1$ generates a (7,4) cyclic code. This code can be generated by the circuit in Figure 11.11. It is a distance 3 code so single error correction is possible.

When the transmitted word is received, correction proceeds as follows. The received word $\bar{r}(x) = \bar{p}(x) \cdot \bar{g}(x) + \bar{s}(x)$, where $\bar{s}(x)$ is called the syndrome. If $\bar{s}(x) = 0$, $\bar{r}(x)$ is a code vector. Otherwise, $\bar{r}(x)$ is not a code vector, and an error has been detected. This decoding process can also be implemented with shift registers [13].

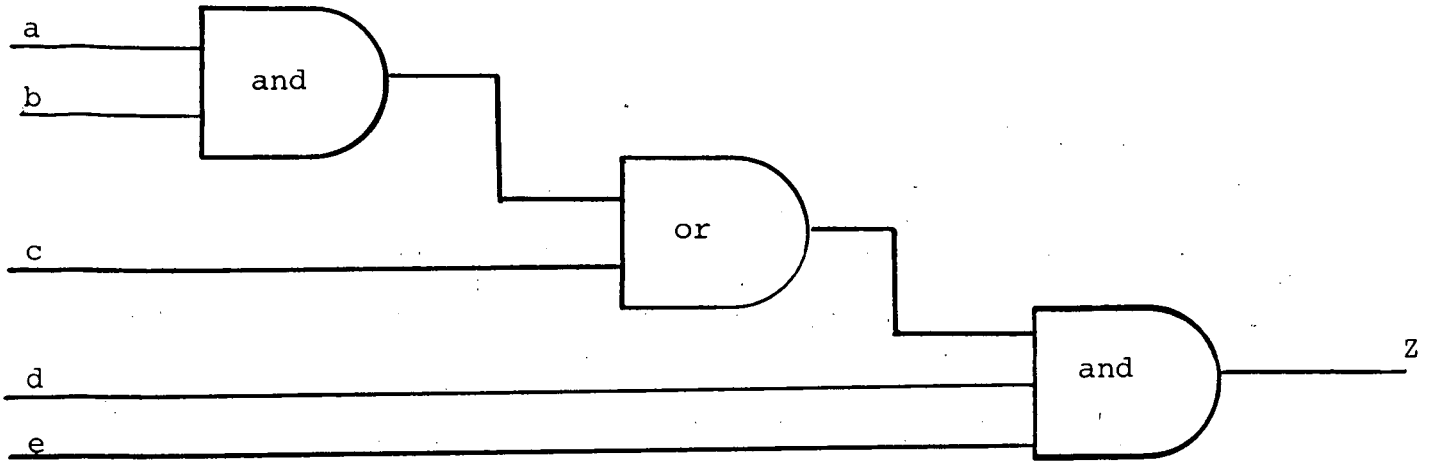


Figure 11.10: Network to Illustrate Path Sensitizing

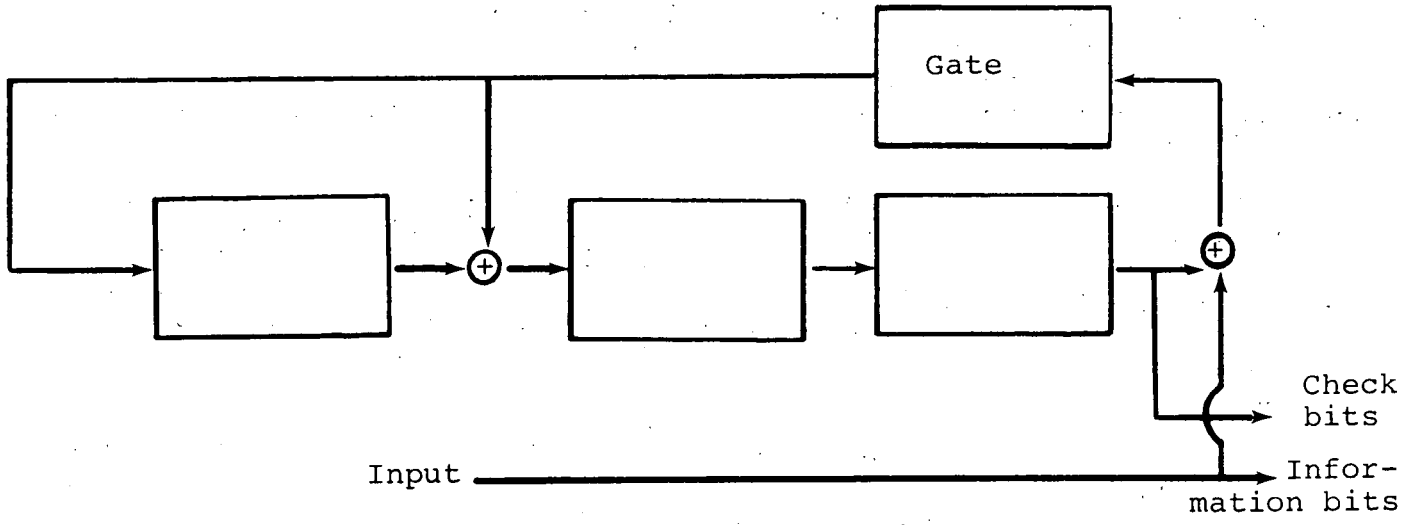


Figure 11.11: Shift Register to encode (7,4) cyclic code generated by $x^3 + x + 1$

11-5.2 Quadded Logic

Another redundancy technique to achieve correction in combinatorial circuits is quadded logic. Instead of having a voter decide if any signal is incorrect, we interconnect components in the quadded circuit so that single errors and some multiple errors are automatically corrected by the hardware and will not cause failure. The disadvantage is that four times as much hardware than the nonredundant network is necessary, and a complex interconnection of components is necessary. Jensen [16] shows an example of a quadded half adder built from NOR gates. In the nonredundant case 5 gates are needed, for quadding 20 gates are needed. However, if these gates are already very reliable, this technique reduces the probability of failure by several orders of magnitude.

11-5.3 Reconfiguration

So far we have seen several methods of detecting errors. Several active units can operate in parallel whose output is compared by a voter. A faulty unit can then be detected by the voter. On the hand, a single module need only be used employing redundancy within its generated output, i.e., error detecting codes. When the code is analyzed, an error indicates a faulty unit. In either case the (persistent) occurrence of an error requires the faulty equipment to be removed from the network.

Reconfiguration, either manually or automatically, is simply the switching out of a faulty unit and its replacement by a standby spare. A drawback of this technique is the extra cost (monitary, volume, weight) of the standby equipment needed. If a unit must be protected against 3 failures, at least 4 such units are necessary. The cost can get high, but at times it is justified for the sake of reliability.

An important question to keep in mind when using reconfiguration is how to handle time critical processes that are going on at the time of failure. Such processes impose additional complexity upon the reconfiguration algorithm. As little information as possible must be lost (hopefully none at all). How this goal is accomplished is a function of the specific application and the ingenuity of the system designer.

11-5.4 Software Restart

So far we have dealt primarily with hardware correction of system failures. We will now consider the case of software recovery from a digital computer failure. Clearly upon the occurrence of a hardware failure in a digital computer, a method must exist for restarting the software after the malfunction is corrected. If voting is used in the system, the two computers that did not fail can continue their computations after a possible system reconfiguration. In this case the

two properly operating computers do not actually need software restart since the hardware error did not destroy the integrity of their computations. However, when voting is not used, a method of software restart is necessary.

The program can be restarted by rolling it back to a known restart point in the computation. That is, a point at which the program is known to have executed correctly before the failure occurred. In many large computer facilities restart points are established by taking complete core and register dumps at fixed times. This information is then stored on some secondary storage device. Upon system failure the programs can be restarted by loading memory with this information. Such a technique has large overhead in terms of recovery time and the need for secondary storage. It can be impractical in other than large commercial applications. Furthermore, fixed restart points can be established in programs without the need for dumping the contents of core. With this method a minimal amount of restart information is associated with each restart point and is kept in core. Should a restart be necessary, all necessary restart information is readily available for the restart point used. This fact enables faster restarts. In fact, such a technique was used in the Apollo project.

11-5.4.1 Apollo Restart [17]

In the Apollo Guidance Computer (AGC) the restart process is initiated upon detection of a computer malfunction. These malfunctions include:

a) parity errors,

b) a program not checking often enough for a ready higher priority program,

c) endless one-instruction loops,

d) oscillator failure,

e) voltage failure, and

f) excessive time spent in interrupt mode.

The computer begins restart by transferring control to location 4000 in fixed memory. The mission programs are organized into five restart groups each having redundant restart pointers which are used to restart these programs. Programs that run simultaneously must be

in different restart groups. At location 4000 the restart program uses the restart pointers to restart the active programs. Each pointer contains the address of restart information in the Restart Table. During execution these pointers are advanced to successively point at the proper restart information for that phase of the program.

Alternative restart options are available in the AGC. The instruction that began the last display sequence was often considered a proper restart point. Another method was to let the next instruction to be executed be the restart point. In this case no Restart Table is necessary.

11-5.4.2 Single Instruction Restart

The M.I.T. SIRU Computer [18,19] uses a single instruction restart method. All errors are detected in the instruction in which they first occur so no errors are propagated to later instructions. Upon detection of an error the program is restarted from the instruction in which the error is detected. This recovery is transparent to the programmer.

To aid recovery and restart SIRU instructions are executed in two phases by the microprogram. During phase 1 results are computed and stored in scratchpad restart buffers. During phase 2 the results are moved from the buffers to their final destinations. Since input data is not overwritten during phase 1, restart during this phase is easy. During phase 2 restart uses the data in the restart buffers.

One of the important advantages of single instruction restart is the small amount of time necessary to restart. This fact is especially important for real time processes.

11-6.0 Additional Applications and Other Topics

11-6.1 Error Propagation

When triple redundancy and voting are used for error detection and correction, error propagation is not too serious a problem. The system has two correctly operating units after the faulty one is switched out. However, without this redundancy an error may propagate extensively through a system before it is detected. Should this condition occur, even the software restart mechanisms we have discussed can be inadequate to correct the process. In short, the sooner an error is detected, the smaller the error propagation, and the higher the probability of being able to correct for the error. Quick detection eliminates otherwise uncorrectable errors but requires carefully designed computer modules, such as arithmetic units.

11-6.1.1 Error Correction in High-Speed Arithmetic

Chien and Hong [20] present a solution to the above problem of quick error detection in computer modules, by using error correcting codes in arithmetic units. The particular problem they discuss is the correction of any single, iterative failure in a high speed multiplier. The advantages of such a technique are obvious: fast detection and correction invisible to the user, and no error propagation for a single failure.

11-6.2 Intermittent Failures

Intermittent failures pose special problems for digital systems. Because of their temporary nature isolation of faulty units becomes very difficult. During factory checks vibrational and thermal testing can locate some intermittent faults in equipment but not all are found. Although they do not solve this problem, Ball and Hardie [21] have recently performed a study on intermittent failures.

This study simulated intermittent failures in a digital computer. The results show only a small percentage of single occurrence intermittent failures being detected. However, many of these intermittent failures do not make the faulty hardware act differently from their non-failed state. To detect the failure the hardware must have the appropriate input data to make the failure visible. The probability of fault detection is monotone increasing with the duration of the failure. For a failure lasting one clock cycle the probability of detection was close to 0; for 10 computer word cycles about 1/2; for 50 computer word cycles about 1. Thus, the longer the failure, the more easily it can be detected and the faulty unit isolated.

11-6.3 The STAR Computer

The STAR Computer [22] is being built by the Jet Propulsion Lab, to satisfy all predictable requirements of a spacecraft computer used in a long mission (at least ten years) and to study fault-tollerant computing. The goal is to achieve fault tollerance for intermittent, permanent, random and catastrophic failures. To do this STAR uses coding, monitor detection, standby redundancy, redundancy with voting, and component redundancy to protect against faults and enable self-repair. (See Figure 9.1).

Specifically STAR employs the following detection and correction methods:

a) Error detecting codes for all machine words with simultaneous detection and program execution;

b) Decentralization of functional units to allow simple fault isolation;

c) Special hardware for detection, recovery and reconfiguration;

d) Correction of intermittent faults by program restart;

e) Reconfiguration by powering down faulty units and powering up replacements;

f) Monitoring circuits supplement error detecting codes to check synchronization and internal operation in functional units; and

g) A special test and repair processor protected by triple redundancy.

At present, research on this project is continuing in order to improve the above detection and correction methods in a second generation STAR computer. The results should provide new state-of-the-art concepts in fault tollerant computing.

REFERENCES

1. Gill, A., Introduction to the Theory of Finite State Machines, (McGraw-Hill, New York, 1962).

2. Hennie, F., Finite State Models for Logical Machines, (John Wiley and Sons, New York, 1968).

3. Kohavi, Z., and Lavallee, P., "Design of Sequential Machines with Fault Detection Capabilities", (IEEE Trans. on Elect. Comp., EC-16(4), August 1967), pp. 473-484.

4. Hennie, F., "Fault Detecting Experiments for Sequential Circuits", (Proc. 5th Ann. Symp. on Switching Theory and Logical Design, Princeton University, November 1964), pp. 95-110.

5. Hartmanis, J., and Stearns, R., Algebraic Structure Theory of Sequential Machines, (Prentice-Hall, New Jersey, 1966).

6. Gurzi, K.J., "Estimates for Best Placement of Voters in a Triplicated Logic Network", (IEEE Trans. on Elect. Comp., EC-14(5), October 1965), pp. 771-717.

7. Armstrong, D. B., "On Finding a Nearly Minimal Set of Fault Detection Tests for Combinatorial Logic Nets", (IEEE Trans. on Elect. Comp., EC-15, 1966), pp. 66-73.

8. McCluskey, E.J., "Test and Diagnosis Procedure for Digital Networks", (Computer, 4(1), January-February 1971), pp. 17-20.

9. Seshu, S., and Freeman, D.M., "The Diagnosis of Asynchronous Sequential Switching Networks", (IRE Trans. on Elect. Comp., 11, 1962), pp. 459-465.

10. Kautz, W.H., "Fault Testing and Diagnosis in Combinational Digital Circuits", (IEEE Trans. on Comp., EC-17(4), April 1968), pp. 352-366.

11. Bennetts, R. G., and Lewin, E.W., "Fault Diagnosis of Digital Systems - A Review", (Computer, 4(4), July-August, 1971), pp. 12-20.

12. Roth, J.P., "Diagnosis of Automata Failures: A Calculus and a Method",(IBM Journal, July 1966), pp. 278-291.

13. Lin, S., An Introduction to Error Correcting Codes, (Prentice-Hall, New Jersey, 1970).

14. Peterson, W.W., Error Correcting Codes, (M.I.T. Press, Cambridge, Mass., 1961).

15. Sellers, F.F., et al., Error Detecting Logic for Digital Computers, (McGraw-Hill, New York, 1968).

16. Jensen, P. A., "Quadded NOR Logic", (IEEE Trans. on Reliability, September 1963), pp. 22-31.

17. Copps, E. M., "Recovery from Transient Failures of the Apollo Guidance Computer", paper 68-823, AIAA Conference, Pasadena, California, August 12-14, 1968.

18. Crisp, R., et al., "SIRU - A New Inertial System Concept for Inflight Reliability and Maintainability",(M.I.T. Draper Lab., E-2407, May 1964).

19. Griggs, K.M., and Schwartz, G., The DCA Computer, (M.I.T. Draper Lab., E-2590, December 1970).

20. Chien, R.T., and Hong, S.J., "Error Correction in High-Speed Arithmetic", (IEEE Trans. on Elect. Comp., C-21(5), May 1972), pp. 433-438.

21. Ball, M., and Hardie, F., "Effects and Detection of Intermittent Failures in Digital Systems", (FJCC, 1969), pp. 329-335.

22. Avizienis, A., et al., "The STAR (Self-Testing and Repairing) Computer: An Investigation of the Thoery and Practice of Fault - Tolerant Computer Design", (IEEE Trans. on Elect. Comp., EC-20(11), November 1971), pp. 1312-1321.

CHAPTER 12

INFORMATION TRANSMISSION BY ORTHOGONAL FUNCTIONS

12-1.0 Introduction

The increasing complexity and duration of exploratory space missions has led to a demand for increased information capacity in space-to-ground communication links. The information includes a variety of data, such as voice transmitted communication, biomedical data, scientific data, and video information. Practical limitations on available transmitted power, antenna gain and receiver noise have constrained attainable signal-to-noise ratios for a given bandwidth, propagation path and wavelength. This constraint necessitates more efficient carrier modulation (or coding) techniques so that information capacity can be increased at acceptable error rates. Recently more attention has been turned to the converse problem of reducing redundancy in the raw data. By removing redundancy, a reduction results in the amount of information that needs to be transmitted. Orthogonal transform techniques are applicable to the signal processing necessary for both coding and compression. These transforms allow a reduction in communication complexity and power by relaxing the bandwidth requirements of the transmitted signals and by reducing signal degradation due to channel noise.

This chapter studies the use of orthogonal functions in transmitting information. We will present an introductory discussion of communication channels followed by a summary of past experience

in space communication. With this background material in mind a mathematical introduction to orthogonal functions and transforms is then presented enabling a detailed discussion of specific transform methods. We will concentrate on the Discrete Fourier Transform (DFT) and the Discrete Hadamard Transform (DHT). Computational algorithms for the Fast Fourier Transform (FFT) and Fast Hadamard Transform (FHT) will also be presented. As we will show, these algorithms enable fast computer processing of transmitted information. Finally, we will discuss the advantages of using orthogonal transforms to transmit information.

12-2.0 Communication Channels

Transmission of information requires a transmitter, a receiver, and a communication channel. The channel is the path over which the information is sent from the transmitter to the receiver. This information can be discrete (bits) or continuous as in the case of voice transmission. Continuous signals are often broken down by time quanta. Discrete orthogonal transforms can then be applied to improve communication efficiency. We will be primarily concerned here with discrete orthogonal transforms.

The capacity of a communication channel can be stated analytically. If each symbol transmitted over the channel contains s bits of information, then a total of 2^s different symbols can be transmitted. Suppose symbols are freely chosen from this set of 2^s members, and suppose a noiseless channel can transmit n symbols per second. Then the capacity C of the channel is given by

$$C = ns.$$

Shannon [1] generalized this result by considering the communication system to have a finite number of states, and within each state only certain symbols can be transmitted. Upon transmission the state changes to a new state depending upon the old

state and the symbol transmitted. Shannon then proved the following fundamental theorem.

Theorem: Let b_{ij}^{s} be the duration of the $s^{\underline{th}}$ symbol which is allowable in state i and leads to state j. Then the channel capacity $C = \log_2 W$, where W is the largest real root of the determinantal equation

$$\left| \sum_{s} W^{-b_{ij}^{s}} - \delta_{ij} \right| = 0, \text{ and where } \delta_{ij} = 1 \text{ if } i = j,$$

and is zero otherwise.

This theorem may be applied to the case of a telegraph. For the telegraph there are 2 states and 6 symbol durations of 2, 4, 5, 7, 8, and 10 time units τ. The determinantal equation in question is

$$\begin{vmatrix} -1 & (W^{-2} + W^{-4}) \\ (W^{-3} + W^{-6}) & (W^{-2} + W^{-4} - 1) \end{vmatrix} = 0$$

Upon reduction we get the equation

$$W^{-10} + W^{-8} + W^{-7} + W^{-5} + W^{-4} + W^{-2} = 1.$$

The largest real root W = 1.45, and hence, C = .539 bits/τ.

12-2.1 Entropy

Information is a measure of one's freedom of choice in selecting a symbol to transmit. Suppose k independent symbols can be transmitted with probabilities of choice $p_1, p_2, \ldots, p_k$. The expression for information is

$$H = \sum_{i=1}^{k} p_i \log_2 p_i .$$

H is referred to as entropy and is used in information theory as a measure of information, choice, and uncertainty. The above

equation possesses the following properties:

a) $H = 0$ if and only if all the p_i but one are 0; the one being equal to 1. Thus, the information measure is 0 because there is no freedom of choice. Only one symbol can be selected.

b) H is maximal if $p_i = \frac{1}{n}, \forall i = 1, \ldots, n$. That is, all symbols can be chosen with equal probability. (This is the most uncertain situation.)

Using the concept of entropy Shannon proved the Fundamental Theorem for a Noiseless Channel:

> Let a source have entropy H (bits per symbol) and a channel have a capacity C (bits per second). Then it is possible to encode the output of the source in such a way as to transmit at the average rate $(\frac{C}{H} - \varepsilon)$ symbols per second over the channel, where ε is arbitrarily small. It is not possible to transmit at an average rate greater than $\frac{C}{H}$.

12-2.2 Noisy Channels

In actual applications communication channels are limited by the signal to noise ration (SNR) at the receiver. Although noise has been reduced by the use of MASERs and other low noise devices, noise sources occur in nature. The Galaxy, the atmosphere, and interferring transmissions are sources of noise. The effects of interferring sources are often theoretically intractible because of their localized nature and non-ergodic signal statistics. However, the desire for increased channel bandwidth and freedom from interference tends to cause an increase in carrier frequencies. Combined galactic and atmospheric noise shows a broad minima in the 2-10 GHz range, but higher atmospheric "windows" at 30 and 80 GHz have been considered and are particularly

applicable to telemetry from re-entry bodies [2].

The effect of noise upon information received can be measured in terms of entropy. With noise the received message contains distortions and, hence, has greater uncertainty. We may represent this fact as follows. Let x represent a message transmitted through a noisy channel, and y the corresponding received signal. Then the amount of useful information transmitted despite noise effects is given by $I = H(x) - H_y(x) = H(y) - H_x(y)$,

where $H(x)$ = average amount of information transmitted

$H(y)$ = average amount of information received

$H_y(x)$ = average amount of information lost due to noise (equivocation), and

$H_x(y)$ = that part of y's entropy due to noise.

The capacity C of this noisy channel is the maximum rate at which useful information can be transmitted over the channel.

Shannon has extended his Fundamental Theorem for a Discrete Noiseless Channel to the case of a noisy channel. This theorem may be stated as follows.

THEOREM: Let a discrete channel have the capacity C and a discrete source the entropy per second H. If $H \leq C$ there exists a coding system such that the output of the source can be transmitted over the channel with an arbitrarily small frequency of errors (or an arbitrarily small equivocation). If $H > C$, it is possible to encode the source so that the equivocation is less than $H - C + \epsilon$ where ϵ is arbitrarily small. There is no method of encoding which gives equivocation less than $H - C$.

12-3.0 Communication Systems

Space communication systems of increasing complexity have been flown since the launching of the earlier satellites. The

communication links have been divided into telemetry (down-link), command (uplink), voice and TV systems due to the differing techniques applied to each class of signals. We will ignore the special needs of communication (relay) satellites here.

Early telemetry systems used various combinations of AM, FM, PFM, and PDM for transmitting analog quantities. Subsequently use of the FM-FM technique became standard, using standardized channels adopted by the Inter-Range Instrumentation Group. These IRIG channels define subcarrier frequencies, bandwidths and percentage FM deviations. Low frequency signals can be conveniently time division multiplexed (sub-commutated) onto an individual channel. Other commuted methods of analog telemetry have used sequences of PAM, PDM or PFM signals. These pulse-analog techniques have not virtually been displaced by Pulse Code Modulation. Strictly speaking PCM represents the transmission of digital information, and the preceeding analog-to-digital conversion represents a coding technique.

The transmission of digital data now has become most advantageous becuase of the ease with which modern digital computers can handle the reception and editing of such data. Furthermore, the sampling techniques allow great flexibility in mixing signals at differing sampling frequencies as appropriate, and the use of digital signals allows the introduction of redundancy where low error rates are important.

The provision of secure, low error rate transmission is particularly important in the case of command channels. Fortunately ground stations do not have the power limitations inherent in space systems, and the information rates are reasonable, although future uplink data rates may be considerably higher than those used currently. Early command systems used tone (FDM) signals, but resulted in very limited capability in which a high degree of redundancy results in great channel security.

It is instructive at this stage to review the characteristics of some modern space communication systems. Due to the integrated nature of these systems it is necessary to look at the overall system rather than at individual aspects as above.

Project Apollo uses a most sophisticated unified S-Band System. The S-band system provides for ranging, telemetry, command signals, voice and TV signals. The ground station transmits a carrier phase modulated by a pseudo noise code sequence for ranging, and two sub-carriers, one carrying voice and the other carrying up-telemetry. This uplink uses a standard code sequence for both commands and date transmission. Each code word is begun with vehicle and system address codes, and each bit is further sub-bit encoded to increase security. The spacecraft contains duplex transponders where the signal is synchronously demodulated using a phase-lock loop containing a VCO used to synthesize the transmitted carrier. This carrier is remodulated (PM) by the ranging PN sequence and subcarriers for down-telemetry (PCM-PSK) and voice (FM). In addition, a separate FM S-band transmitter can transmit TV, recorder information, or scientific data alternatively. Besides the S-band, AM is used at VHF for orbital, lunar module and extravehicular communication. The Manned Space Flight Network now used to communicate with Apollo contains sophisticated data transmission and signal formatting capabilities.

Command and telemetry systems for lunar and deep space probes have become increasingly complex. The transmission of TV pictures from Mars represents a remarkable achievement. High Rate Telemetry (HRT) using block data coding was used in this reception from Mariner VI, obviating the need for the tedious collection and processing for the standard low rate telemetry from many ground stations. HRT subcarrier signals were decoded using high rate correlators. Pioneer IX carried a convolution coding experiment (CCSP) using 100% redundancy, demonstrating a

significant improvement in error rate at great distances. Due to the flexibility of computers there has been a tendency to use programmable facilities. The Multiple Mission Telemetry System started this trend, although much special purpose hardware remains. The Multiple Mission Command System will permit the assembly and generation of complex command words of variable length, selectable subcarrier frequencies, and a choice of modulation and synchronization techniques at selectable bit rates. Post reception processing of Surveyor lunar pictures has demonstrated the potential of processing techniques.

12-4.0 Orthogonal Functions

Before discussing Fourier and Hadamard Transforms, we will present a general description of orthogonal functions and transforms [3,4]. A clear understanding of the mathematical structure of orthogonal functions will make the discussion of specific functions in later sections easier.

Let $h(x)$ and $g(x)$ be real valued, almost everywhere non-vanishing functions on the interval (a,b). We say that these two functions are orthogonal on (a,b) if

$$\int_a^b g(x)h(x)dx = 0 .$$

For example, if $g(x) = x^3$, $h(x) = 1$ and $(a,b) = (-1,1)$, $\int_{-1}^{1} x^3 \cdot 1 \, dx = 0$. Thus, these two specific functions are orthogonal on $(-1,1)$.

We may extend this definition of orthogonality to a system of functions $\{\phi_i(x),\ i=0,\ 1,\ \ldots\}$. Each $\phi_i(x)$ is real valued and almost everywhere non-vanishing on (a,b). We say that the system is orthogonal on (a,b) if

$$\int_a^b \phi_j(x)\phi_k(x)dx = M_j \, \delta_{jk} \qquad (12.1)$$

where M_j is a nonzero real number and $\delta_{jk} = 0$ if $j \neq k$ and 1 if $j = k$.

If, in addition, each $M_j = 1$, $j = 0, 1, \ldots$, the set of functions $\{\phi_i(x)\}$ is said to be orthonormal. A set of orthogonal functions $\{\phi_i(x)\}$ can always be normalized by dividing by the M_i. We then get $\left\{\frac{\phi_i(x)}{M_i}\right\}$.

An interesting property of orthogonal functions is summarized in the following remark.

Remark: A system of m orthogonal functions $\{\phi_1(x), \ldots, \phi_m(x)\}$ is linearly independent.

Proof: Suppose the system is linearly dependent. Then $\sum_{i=1}^{m} c_i\phi_i(x)=0$ without all the constants c_i being zero. For each $j \leq m$, by orthogonality

$$\int_a^b \phi_j(x) \cdot \sum_{i=1}^{m} c_i\phi_i(x)dx = c_j\int_a^b \phi_j^2(x)dx = 0$$

Therefore, each $c_j = 0$ which implies that $\{\phi_i(x)\}$ must be linearly independent.

Orthogonal functions occur quite often in nature. For example, consider the **heat conduction problem**

$$\frac{\partial\phi(x,t)}{\partial t} - k\ \frac{\partial^2\phi(x,t)}{\partial x^2} = 0$$

with boundary conditions

$$\phi(0,t) = 0 \text{ and } \phi(1,t) = 0.$$

Solution of the equation yields particular solutions

$$\phi_n(x,t) = e^{-n^2\pi^2kt} \sin n\pi x.$$

This system of equations $\{\phi_i(x,t)\}$ is orthogonal on the interval $0 < x < \pi$.

12-4.1 Orthogonalization

An orthogonal set of m functions $\{\phi_i(x)\}$ may be constructed from a linearly independent set of m functions $\{\theta_i(x)\}$ by use of the Schmidt Orthogonalization Procedure.

We let

$$\phi_i(x) = \sum_{k=0}^{i} c_{ik}\, \theta_k(x) . \qquad (12.2)$$

It remains to determine the c_{ik} so that the resulting $\phi_i(x)$ are orthogonal. To do this we use equation (12.1). Substitution of (12.2) into (12.1) yields a system of equations which enable determination of the c_{ik}. The M_j may be arbitrarily chosen in this procedure.

Example: Let the $\theta_i(x)$ be Bernoulli Polynomials:

$$\theta_0(x) = 1$$
$$\theta_1(x) = x - \frac{1}{2}$$
$$\theta_2(x) = x^2 - x + \frac{1}{6}$$
$$\theta_3(x) = x^3 - \frac{3}{2}x^2$$
$$\theta_4(x) = x^4 - 2x^3 + x^2 - \frac{1}{30}$$

Also, let $M_j = \frac{2}{(2j+1)}$. Application of the Schmidt Orthogonalization Procedure yields the orthogonal functions:

$$\phi_0(x) = 1$$
$$\phi_1(x) = x$$
$$\phi_2(x) = -(3x^2-1)$$
$$\phi_3(x) = \frac{1}{2}(5x^3-3x)$$
$$\phi_4(x) = \frac{1}{8}(35x^4-30x^2+3)$$

12-4.2 Representation by Orthogonal Functions

A given function f(x) can be expanded in terms of an orthogonal set of functions $\{\phi_i(x)\}$ in (a,b). We desire an expansion

in the form

$$f(x) = \sum_{i=0}^{\infty} a_i \phi_i(x) . \tag{12.3}$$

The a_i can be calculated to be

$$a_i = \frac{\int_a^b f(x)\phi_i(x)dx}{\int_a^b \phi_i^2(x)dx} \tag{12.4}$$

At this point we do not know that the above series truly represents $f(x)$ in (a,b), and we do not know if the series even converges in (a,b). We will say more about convergence shortly.

A set of orthonormal functions is said to be complete if for any continuous function $f(x)$ such that $\int_a^b f^2(x)dx$ is finite,

$$\lim_{m \to \infty} \int_a^b \left[f(x) - \sum_{i=0}^{m} a_i \phi_i(x) \right]^2 dx = 0 .$$

The most important property of complete sets of orthonormal functions that concerns us here is the fact that any continuous function $f(x)$ for which $\int_a^b f^2(x)dx$ is finite is completely determined by the series given by equation (12.3) with coefficients given by (12.4), where the $\phi_i(x)$ are a complete set of orthonormal functions.

An example of a complete set of functions is the Fourier series given by the functions 1, sin x, cos x, sin 2x, cos 2x, ... It can be shown [4] that this set of functions is complete on the interval $[-\pi,\pi]$. Hence, for any continuous $f(x)$ such that $\int_{-\pi}^{\pi} f^2(x)dx$ is finite,

$$s_n = a_0 + \sum_{k=1}^{n} a_k \sin kx + \sum_{k=1}^{n} b_k \cos kx$$

C 4

can be used to approximate $f(x)$ in $[-\pi,\pi]$. This fact enables the approximation of a continuous communication signal by a discrete Fourier series.

12-5.0 Fourier Transforms

A general integral transform may be expressed in the form $h(\beta) = \int_a^b g(\alpha)\ K(\alpha,\beta)d\alpha$. The quality $h(\beta)$ is the transform of $g(\alpha)$ by the kernel $K(\alpha,\beta)$. If we let $a = \infty$, $b = -\infty$, α=t (time), β = f (frequency) and $K(t,f) = e^{-2\pi ift}$, we obtain the Fourier transform for a signal $g(t)$. We have $h(f) = \int_\infty g(t)\ e^{-2\pi ift}dt$, which transforms the signal $g(t)$ into the frequency domain. Similarly $h(f)$ can be transformed back into the time domain by the inverse Fourier transform; namely

$$g(t) = \int_{-\infty}^{\infty} h(f)\ e^{2\pi ift}df.$$

The discrete Fourier transform and inverse transform are the following

$$h(j) = \frac{1}{N}\sum_{k=0}^{N-1} g(k)\ e^{-2\pi ijk/N} \qquad \text{and}$$

$$g(k) = \sum_{j=0}^{N-1} h(j)\ e^{2\pi ijk/N}$$

For a continuous signal $g(t)$ we can take a discrete Fourier transform by constructing a time series $g(k\Delta t)$. We assume the series is periodic with period T and the Fourier coefficients of $h(jf_s)$ are periodic over the sample frequency f_s. For each Δt time interval a discrete Fourier transform of $g(t)$ is calculated.

Such a discrete transform can be used for efficient information transmission as follows:

a) The signal is transformed into the frequency domain as a linear combination of the basis functions, $e^{-2\pi ijk/N}$.

b) The quantities h(j) are transmitted to the receiver.

c) The receiver takes the inverse discrete transform to reconstruct the original waveform.

The efficiency of this process is increased by a rapid computational algorithm for calculating the transforms. This algorithm is known as the Fast Fourier Transform [5,6].

12-5.1 Fast Fourier Transforms

Let $W_N = e^{2\pi i/N}$. Consider the problem of evaluating the complex Fourier series

$$h(j) = \sum_{k=0}^{N-1} A(k) \cdot W_N^{jk} \qquad (12.5)$$

where the A(k) are complex. The original paper by Cooley and Tukey [5] presents the algorithm for $N = 2^r$, where r is a positive integer. For simplicity in presentation we will choose r=3(N=8).

The integers j and k can be expressed in the form $j = 4j_2 + 2j_1 + j_0$ and $k = 4k_2 + 2k_1 + k_0$ for $j_i, k_i = 0, 1$ and $i = 0, 1, 2$. Then (12.5) can be written in the form

$$h(j) = h'(j_2,j_1,j_0) =$$

$$\sum_{k_0=0}^{1} \sum_{k_1=0}^{1} \sum_{k_2=0}^{1} A(k_2,k_1,k_0)\, W^{(4j_2+2j_1+j_0)\,(4k_2+2k_1+k_0)} \cdot \qquad (12.6)$$

Note the following identities

$$W^8 = e^{2\pi i} = 1,$$

$$W^{(4j_2+2j_1+j_0)4k_2} = W^{8(2j_2+j_1)k_2} \cdot W^{4j_0k_2} = 1 \cdot W^{4j_0k_2} = W^{4j_0k_2}$$

and

$$W^{(4j_2+2j_1+j_0)2k_1} = W^{8j_2k_1}\, W^{(2j_1+j_0)2k_1} =$$

$$1 \cdot W^{(2j_1+j_0)2k_1} = W^{(2j_1+j_0)2k_1} \quad .$$

Applying these identities to (6) yields

$$h'(j_2,j_1,j_0) = \sum_{k_0=0}^{1} \sum_{k_1=0}^{1} \sum_{k_2=0}^{1} A(k_2,k_1,k_0)\, W^{4j_0k_2} \cdot$$

$$W^{(2j_1+j_0)2k_1} \quad W^{(4j_2+2j_1+j_0)k_o} \qquad (12.7)$$

This equation can in turn be rewritten as

$$A_1(j_0,k_1,k_0) = \sum_{k_2=0}^{1} A(k_2,k_1,k_0)\, W^{4j_0k_2} \quad ,$$

$$A_2(j_0,j_1,k_0) = \sum_{k_1=0}^{1} A_1(j_0,k_1,k_0)\, W^{(2j_1+j_o)2k_1} \quad ,$$

$$A_3(j_0,j_1,j_2) = \sum_{k_0=0}^{1} A_2(j_0,j_1,k_0)\, W^{(4j_2+2j_1+j_0)k_0} \quad ,$$

and

$$h'(j_2,j_1,j_0) = A_3(j_0,j_1,j_2) \quad . \qquad (12.8)$$

These equations represent the Fast Fourier Transform Algorithm.

12-5.1.1 Computational Savings

Without the FFT algorithm, evaluation of h(j) would require 64 complex multiply and add operations. The FFT algorithm given by (12.8) shows 48 operations. However, by taking advantage of the

fact that $W^0 = -W^4$, $W^1 = -W^5$, $W^2 = -W^6$, and $W^3 = -W^7$, a reduction to 12 multiplications results.

Now consider the more general case of $N = 2^r$. The FFT algorithm reduces the computation from N^2 operations to $\frac{N}{2} \log_2 N$ complex multiplications, and the same number each of complex additions and subtractions. For N = 1024, a computational reduction of 200 to 1 is achieved.

12-5.1.2 Hardware Implementation

To further increase computational speed Fast Fourier Transforms have been implemented in special purpose computer hardware [7,8]. Bell Telephone Laboratories has built an FFT signal processing system [9]. The system costs 5 times less per hour than a general purpose computer and performs the FFT algorithm 20 times faster. Thus, a 100 to 1 cost savings results.

12-5.2 Video Information

Fourier Transforms can also be applied to the transmission of visual data. Let g(x,y) represent the amplitude of image samples over a square array of N^2 points. The discrete transform h(u,v) is given by

$$h(u,v) = \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} g(x,y)\, e^{-2\pi i(ux+vy)/N} ,$$

and the inverse transform by

$$g(x,y) = \sum_{u=0}^{N-1} \sum_{v=0}^{N-1} h(u,v)\, e^{2\pi i(ux+vy)/N} .$$

Fourier Transform image coding gives good quality image transmission using the same number of bits in the (u,v) domain as is needed in the (x,y) domain for an image by pulse code modulation. Moreover, the transform method allows bandwidth reduction and yields immunity to some channel errors.

In the (x,y) domain image energy is usually uniformly distributed. When a transform is taken, the image energy tends to be concentrated near the origin of the (u,v) domain. Thus, higher spacial frequency components have low magnitude and need not be transmitted. Bandwidth reduction results.

Avoiding certain channel errors is due to the averaging process of the transform. After the transmitted image is reconstructed, each point is a weighted sum of all points in the (x,y) domain. Hence, the overall loss of resolution due to channel errors in the (u,v) domain is often less serious than defects occurring when channel errors are introduced in the (x,y) domain.

The third attraction of using discrete Fourier transforms for image transmission is the fast computational algorithms existing for processing the data (as we have already seen).

12-6.0 Hadamard Matrices

Another method of information transmission uses the Hadamard Transform [10, 11]. This method possesses the advantages of the Fourier transform for image coding and also has a faster computational algorithm. Hadamard matrices are used to transform the image code into its frequency domain for transmission and then to reconvert the received information back into the special domain.

Before defining a Hadamard matrix let us review some basic matrix algebra definitions and notations.

Definitions:

a) The transpose of a matrix A is that matrix A^T formed by interchanging the rows and columns of A.

b) The identity matrix I is a square matrix such that the elements along the principal diagonal are all 1 and all other elements are 0.

c) A matrix A is orthogonal if $AA^T = I$.

d) A matrix A is symmetric if $A = A^T$.

We will now define the Hadamard matrix and point out its basic properties. A Hadamard matrix H is a square (n x n) matrix whose elements are +1 or -1 and such that $HH^T = nI$. Thus, except for the factor n, the order of the matrix, a Hadamard matrix is orthogonal. If H is also symmetric, $HH = nI$.

It has been proven that if a Hadamard matrix of order n exists where $n > 2$, then $n \equiv 0 \pmod 4$. However, the converse of this theorem has not been proven. The lowest order H is for n = 2, in this case

$$H = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} .$$

Furthermore, Hadamard matrices of higher order can easily be constructed from other Hadamard matrices. For example, if G is a Hadamard matrix so is

$$H = \begin{bmatrix} G & G \\ G & -G \end{bmatrix} .$$

Also, if G and H are Hadamard, then

$$H' = \begin{bmatrix} g_{11}H & \cdots & g_{1m}H \\ g_{21}H & & \vdots \\ \vdots & & \vdots \\ g_{m1}H & \cdots & g_{mm}H \end{bmatrix} \quad \text{is Hadamard.}$$

12-6.1 Hadamard Transforms

The Hadamard Transform h(u,v) of g(x,y), an n x n point image, is defined by

$$[h(u,v)] = [H(u,v)]\,[g(x,y)]\,[H(u,v)] , \qquad (12.9)$$

where H(u,v) is an nth order symmetric Hadamard matrix. The inverse property also holds since

$$[H(u,v)][h(u,v)][H(u,v)] = [H(u,v)][H(u,v)][g(x,y)][H(u,v)][H(u,v)] =$$

$$n^2\,[g(x,y)] \qquad \text{because of symmetry.}$$

$$\therefore \quad [g(x,y)] = \frac{1}{n^2}\,[H(u,v)]\,[h(u,v)]\,[H(u,v)] \ .$$

For symmetric Hadamard matrices of order 2^r, we may also write (12.9) in a more useful form; namely,

$$h(u,v) = \sum_{x=0}^{n-1} \sum_{y=0}^{n-1} g(x,y)\,(-1)^{p(x,y,u,v)} \qquad (12.10)$$

where $$p(x,y,u,v) = \sum_{k=0}^{r-1} (u_k x_k + v_k y_k) \pmod 2 \ . \qquad (12.11)$$

The terms x_k, y_k, u_k, and v_k are binary representations of x, y, u, and v. We call this representation of (12.9), given by equations (12.10) and (12.11), natural form.

12-6.2 Sequency

By sequency we mean the number of sign changes the elements of a row of a Hadamard matrix exhibit. It has been proved that one can construct a Hadamard matrix of order 2^r such that the sequency values cover every integer from 0 to $2^r - 1$.

The ordered representation of (12.9) uses the concept of sequency. We now wish the sequency of each row of H to be larger than that of the preceeding row. The series representation is again given by (12.10), but now (12.11) is replaced by

$$p(x,y,u,v) = \sum_{k=0}^{r-1} [d_k(u)x_k + d_k(v)y_k] \ , \qquad (12.12)$$

where

$$d_o(u) = u_{r-1}$$

$$d_1(u) = u_{r-1} + u_{r-2}$$

$$d_2(u) = u_{r-2} + u_{r-3}$$

$$\vdots$$

$$d_{r-1}(u) = u_1 + u_0$$

A computational algorithm will now be presented for this ordered representation of the Hadamard transform.

12-6.3 Fast Hadamard Transform

This algorithm requires expressing (12.10) as two one dimensional transforms. These are

$$h(u,y) = \sum_{x=0}^{n-1} g(x,y)\ (-1)^{\nu} \quad , \tag{12.13}$$

where $\nu = \sum_{k=0}^{r-1} d_k(u)x_k$,

and

$$h(u,v) = \sum_{y=0}^{n-1} h(u,y)\ (-1)^{\eta} \quad , \tag{12.14}$$

where $\eta = \sum_{k=0}^{r-1} d_k(v)y_k$.

The FHT computes (12.13) followed by (12.14), so it suffices to illustrate the algorithm for a one dimensional transform. All of the operations involved in taking the transform are done in a selected order. Suppose 2^n data points are involved as shown in Figure 12.1. The operations in taking the transform follow a sieving sequence with intermediate results saved. We first compute h(0) by computing the sum of all data points as shown in Figure 12.1. This is the S sequence. Then we compute h(1) using the intermediate results of level 1. To calculate h(1) we use a 1 sequence and subtract the lower node from the upper node of a pair of nodes at level 1 to get a result at level 0. Next we compute h(2) with a 2 sequence using the intermediate results of level 2. At level

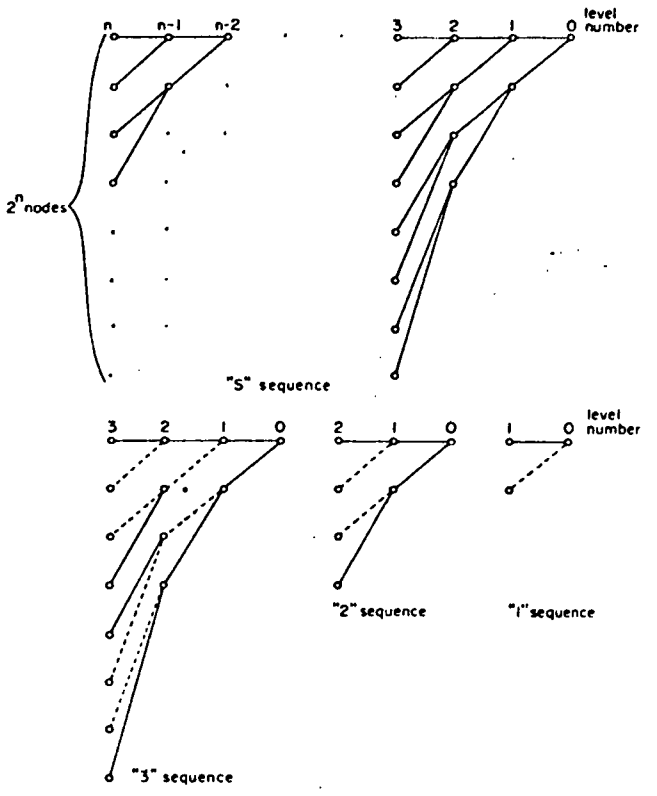


Figure 12.1: Hadamard Transform Computational Sequence [10].

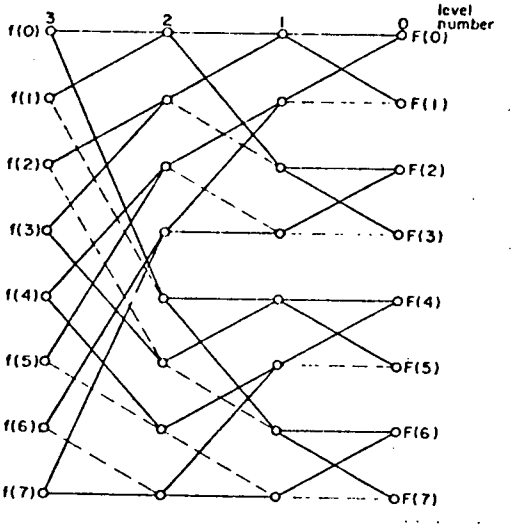


Figure 12.2: Computation of One Dimensional, Third Order Hadamard Matrix [10].

2, pairs are subtracted from one another to produce level 1's results. These are then added together. This procedure is followed for the remaining sequences; i.e., 3, 4, ..., n.

An example of the algorithm for $2^n = 8$ is given in Figure 12.2. In this example (F(0), F(1), ..., F(7)) are computed using the sequences (S, 1, 2, 1, 3, 1, 2, 1), respectively. A total of $8 \log_2 8 = 24$ operations are needed. This is a considerable savings compared with the $8^2 = 64$ operations needed by the brute force calculation.

12-6.3.1 Computational Savings

The fast execution time of this algorithm results from a judicious matrix decomposition and the storage of intermediate results. The brute force computation of a one dimensional Hadamard transform requires a total of n^2 additions and subtractions. The above algorithm reduces this number to $n \log_2 n$.

Pratt [10] gives an interesting comparison of Fourier and Hadamard transforms. Both were programmed on a TRW-530 digital computer. The image to be transformed had n = 256. The computation time for the Fourier transform was 20 minutes versus 3 minutes for the Hadamard transform. A reason for the difference is that Hadamard transform only requires real adds and subtracts, whereas the Fourier transform requires complex multiplies, adds and subtracts.

12-7.0 Advantages and Disadvantages of Orthogonal Transforms

As we have seen, taking discrete transforms of continuous signals introduces errors. The time interval over which the transform is taken must be sufficiently large to allow the desired low frequency signal components to be included. Moreover, we are approximating a continuous signal with a finite series of orthogonal functions. However, this fact can be an advantage in eliminating noise effects as we will discuss in a moment.

Aliasing must also be avoided in using a discrete transform. Aliasing refers to a situation in which high frequency components of a time function impersonate low frequency components when the sampling rate is too low. To completely avoid aliasing a sampling frequency twice as high as the highest frequency component present must be used.

Some of the factors which influence the efficiency and performance of the transform encoding system are the quantization scheme, the amount of bandwidth compression realized, and the effects of channel noise.

12-7.1 Bandwidth Compression

After taking the transform of the signal, it is possible to send the coefficients to the receiver as though they were the original signal. However, in many applications it is known that the high frequency components of the signal are really not important pieces of information. This implies that one could omit the coefficients of the high frequency contributions and send out only the coefficients for the low frequency portions. Two advantages are then realized. First, some bandwidth compression has been realized since the receiver can reconstruct the essential features of the signal on the basis of fewer bits received from the transmitter. Here we are relying on the fact that the signal's low frequency components have more power than the high frequency components. (There is one case where transform encoding gives no improvement; namely, when the signal has the same power at all frequencies; i.e., when it is white noise.) The second advantage to neglecting the high frequency components of the transform during transmission is that one effectively filters out any high frequency noise that had been imposed upon the signal before transformation. The answer to the question of how many coefficients should be transmitted is determined primarily by the allowable distortion in reconstructing the signal at the receiver.

12-7.2 Quantization

There are many ways to quantize the coefficients. For

example, one might consider the process of block quantization [12] in which the coefficient vector is first transformed to a set of uncorrelated random variables and then quantized. The bits which have been allocated to transmit these coefficients are distributed so that they minimize the mean square error in reconstructing the coefficients at the receiver. In the original paper by Huang and Schultheiss, a noiseless channel was assumed.

Another approach can be taken by realizing that since the transforms are taken from a block of data T seconds long, each coefficient in the transform now becomes a discrete random process with samples occurring every T seconds. With the problem case in this form one can consider all the various time domain quantization techniques and optimal quantization techniques that have previously been considered for time series along. Examples include predictive quantization and predictive-comparison data compression [13]. (We will not discuss these topics here.)

These optimal quantizing methods and others depend on a priori knowledge of the probability distribution of the coefficient vector. If good information about this distribution is not available, or if it can be expected to change drastically, then one must consider adaptive quantization schemes to ensure efficient operation of the transform encoding system.

12-7.3 Channel Noise

One of the advantages that has been cited for transform coding has been that it is less susceptible to channel noise for video data [10]. The argument is that the effect of a coefficient error becomes spread out over the entire scene. The human process of perceiving such a picture tends to filter out the errors. For example, an error in the coefficient of the d.c. level of the picture would change only the apparent brightness and would not greatly effect the detail of the picture. However, an error made in transmitting the coefficient for a signal waveform can drastically change its basic characteristics. There is no

redundancy in the received signal which can be used to eliminate these transmission errors. This susceptibility of bandwidth compression techniques to channel errors has been well documented [14]. In summary then, when considering the effect of channel noise on orthogonal transform encoding systems one must specify the type of signal that is being processed and the use for which it is intended.

REFERENCES

1. Shannon, C., and Weaver, W., The Mathematical Theory of Communication, (Univ. of Illinois Press, Urbana, Ill., 1949).

2. Filipowsky, R.F., and Muchldorf, E.I., Space Communications Systems, (Prentice-Hall, New Jersey, 1965).

3. Harmuth, H., Transmission of Information by Orthogonal Functions, 2nd Printing, (Springer-Verlag, New York, 1970).

4. Weinberger, H., A First Course in Partial Differential Equations, (Blaisdell, Waltham, Mass., 1965).

5. Cooley, J.W., and Tukey, J.W., "An Algorithm for the Machine Calculation of Complex Fourier Series", (Math. Comput., 19,April 1965), pp. 297-301.

6. Bergland, G.D., "A Guided Tour of the Fast Fourier Transform", (IEEE Spectrum, 6(7), July 1969), pp. 41-52.

7. Bergland, G.D., "Fast Fourier Transform Hardware Implementations - An Overview", (IEEE Trans. Audio Electroacoust., AU-17, June 1969), pp. 104-108.

8. Bergland, G.D. "A Parallel Implementation of the Fast Fourier Transform Algorithm", (IEEE Trans. on Comp., C-21(14), April 1972), pp. 366-370.

9. Klahn, R. et al., "The Time Saver: FFT Hardware", (Electronics, June 24, 1968), pp. 92-97.

10. Pratt, W.K., et al., "Hadamard Transform Image Coding", (Proc. IEEE, 57(1), January 1969), pp. 58-68.

11. Comsat Laboratories, Orthogonal Transform Feasibility Study, (Clarksburg, Maryland, November 1971, prepared under contract NAS 9-11240).

12. Huang, J., and Schultheiss, P., "Block Quantization of Correlated Guassian Random Variables", (IEEE Trans. Comm. System, September 1962), pp. 289-296.

13. Curry, R.E., Estimation and Control with Quantized Measurements, (M.I.T. Press, Cambridge, Mass., 1970).

14. Davisson, L.D. "The Theoretical Analysis of Data Compression Systems", (Proc. IEEE, 56(2), February 1968), pp. 176-186.

CHAPTER 13

SAMPLED DATA ANALYSIS

13-1.0 Introduction

This chapter describes some of the methods one might use when presented with the need to use a digital computer to measure a continuous variable, and then based on some desired end result, to generate an output signal which is linearly related to the measured signal. The methods described here are directly analogous to methods commonly used in continuous signal control system analysis.

Since the digital computer is by its nature not a continuous device, the introduction of the computer will involve 1) a sampling process, 2) discrete operations within the digital computer, and 3) the smoothing of the computer output before it is re-introduced into the system.

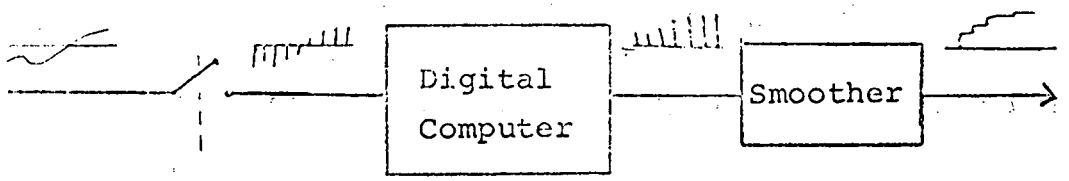


Figure 13-1
Components of a Digital Control Element

13-2.0 The Sampling Process

Digital computers are discrete processors in two senses. The process of encoding data into a number system of finite word length introduces magnitude quantization, which results in propagation of errors due to round-off. The other form of discontinuity involves sampling of data at discrete intervals, and working with the sampled data as an acceptable representation of the continuous data. We will discuss the sampling process here.

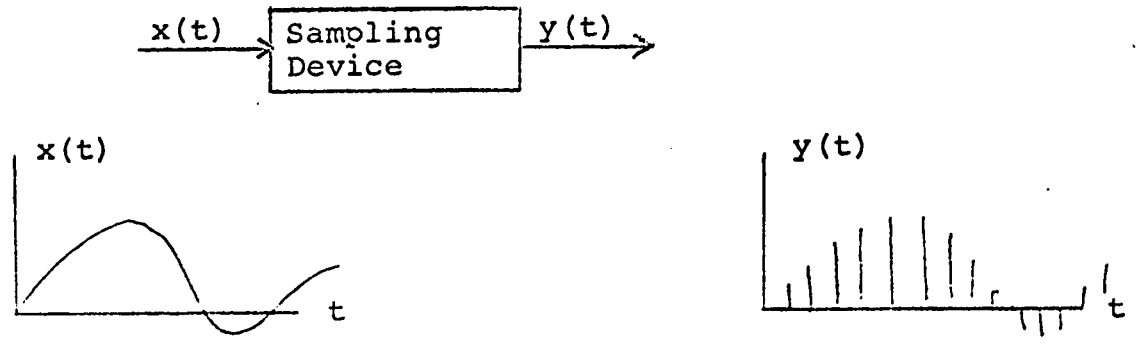


Figure 13-2

Waveforms of the Input and Output of a Sampling Device

By treating a sampling device as a modulator, which multiplies the input signal by an infinite series of impulses, one can analyze the effect of sampling by comparing the frequency spectrum of the sampled signal with the spectrum of the original signal. The simplest way to study the spectra is to assume that the original signal contains only one frequency component, for example

$$x(t) = v \cos(\omega_0 t + \theta_0) \qquad (13\text{-}1)$$

$$= \frac{v\, e^{j\theta_0}}{2}\, e^{j\omega_0 t} + \frac{v\, e^{-j\theta_0}}{2}\, e^{-j\omega_0 t} \qquad (13\text{-}2)$$

The amplitude of the frequency spectrum of the time function x(t) is

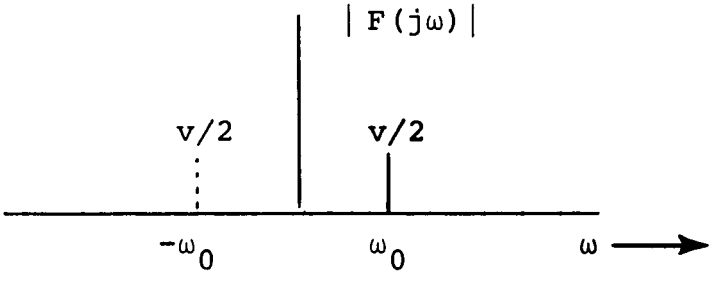


Figure 13-3

The Amplitude of the Fourier Transform of x(t)

The frequency spectrum of the time function y(t) (the sampled version of x(t)) turns out to have the following character.

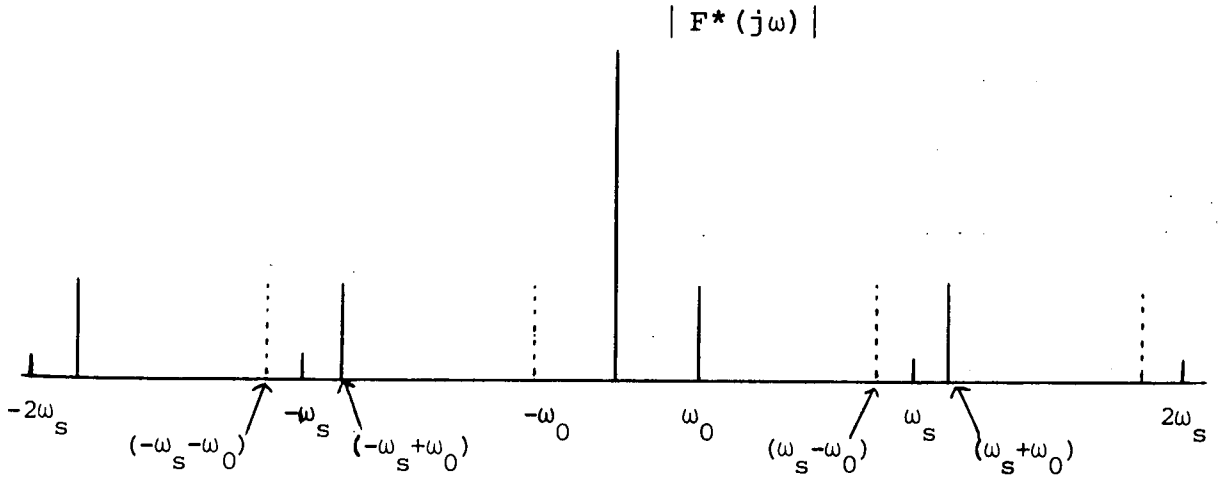


Figure 13-4

The Amplitude of the Fourier Transform of y(t)

where $\omega_s = 2\pi f_s$ (The sampling frequency)

and $T = \frac{1}{f_s}$

Note that sampling frequency is in radians/sec. defined by having one sample per cycle.

We see that spectrum analysis indicates that the sampling introduces "complementary" spectral components at multiples of the sampling frequency. If the spectrum of the input were continuous, the spectrum of the sampled signal would look like this

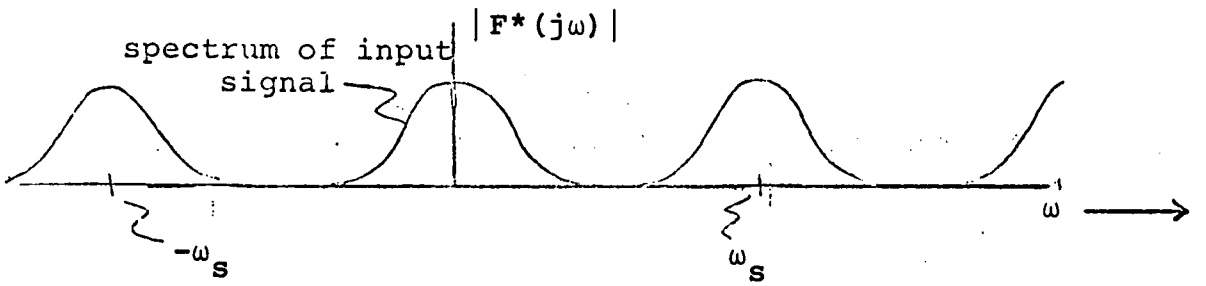


Figure 13-5

The Spectrum of a Sampled Signal

13-3.0 Reconstruction of the Unmodulated Signal

The continuous signal can be reconstructed from the samples under certain circumstances:

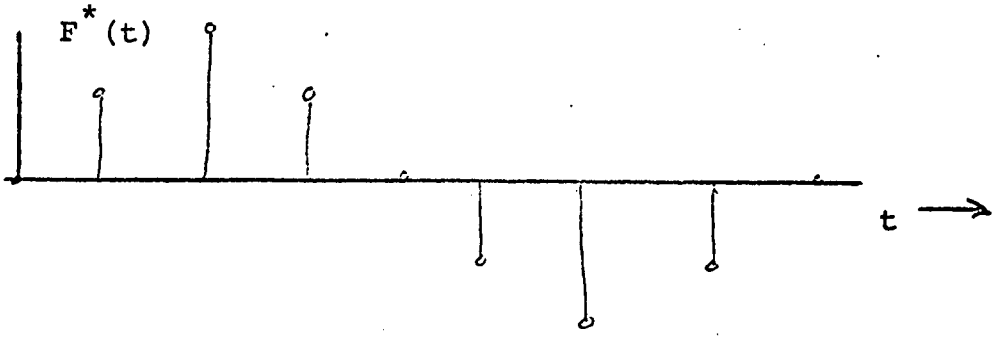


Figure 13-6

A Sampled Signal

The sampled function above could be filled in by holding the last sampled value between sample intervals, or by fitting more complex extrapolations based on past values.

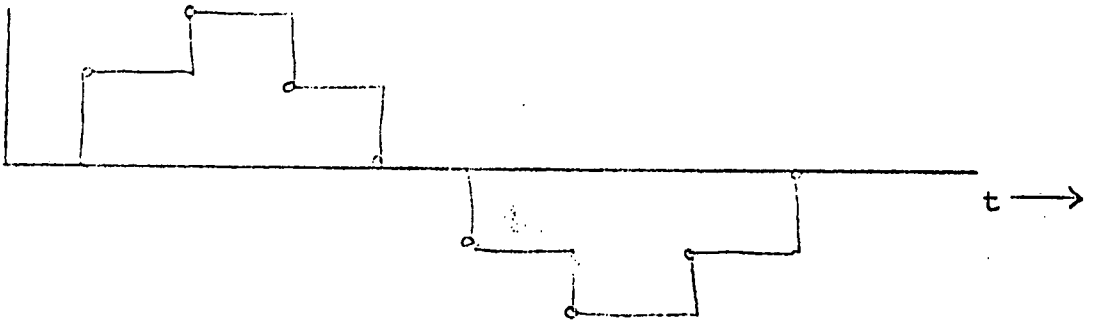


Figure 13-7
A Zero'th Order Hold Reconstruction

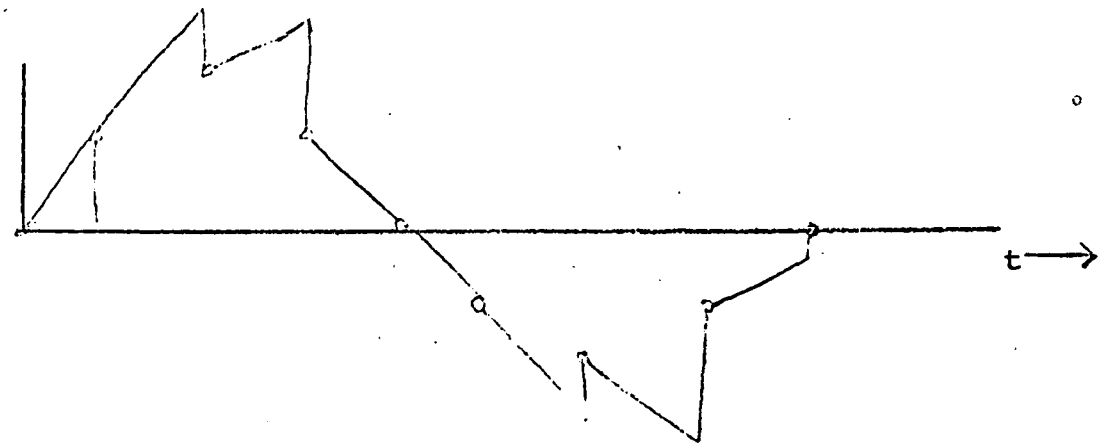


Figure 13-8
A First Order Hold Reconstruction

One can in principle at least, use all the past data and get a pretty smooth fit:

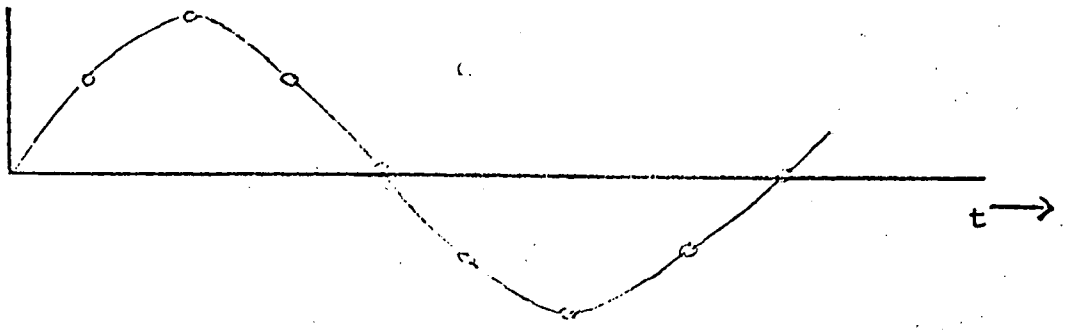


Figure 13-9

A Smooth Fit

However, there is no reason (yet) to believe that the actual input to the sampler wasn't

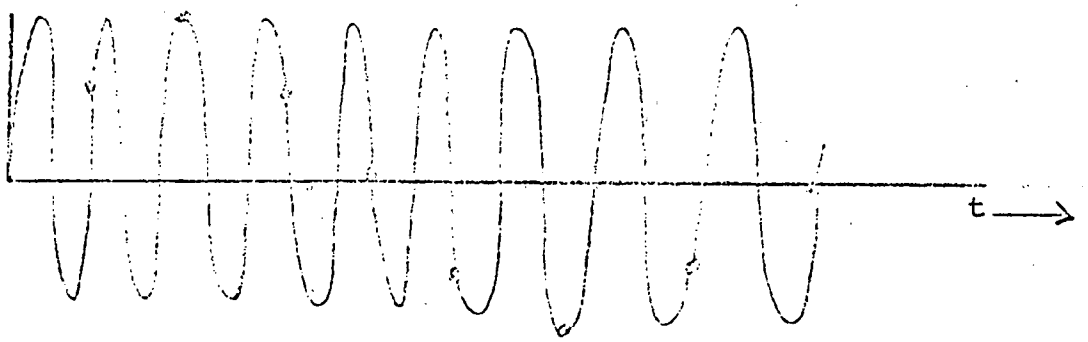


Figure 13-10

Another Signal that Could Produce the Same Set of Samples

13-4.0 Basic Theorem of Sampling

This dilemma leads to a basic theorem which must not be forgotten.

> Theorem: If you can be certain that the signal to be sampled contains no frequency components greater than one-half the sampling frequency, then the samples are sufficient to reconstruct the signal.

Looking back at the example that we have been following, we can see that the sampling theorem seems to apply. It is necessary to know a-priori that high frequency components do not exist in order to construct the plausible sine wave of Figure 13-9.

In terms of the frequency spectrum, we can see that, if the sampling theorem rule is violated, there is overlap of the complementary spectrum into the desire spectrum. There is no way that the original function can be reconstructed.

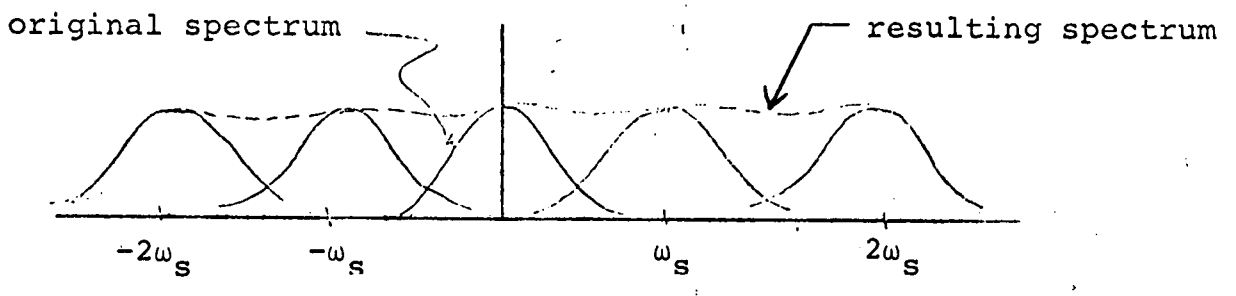


Figure 13-11

Effect of Overlap on the Frequency Spectrum

13-5.0 Noise Into a Sampler

When dealing with a situation where there is both signal and noise, it is necessary to either eliminate the high frequency noise before sampling the signal, or to sample at a frequency at least twice the highest frequency component of the noise.

As an example of the consequence of not observing the twice frequency rule, figure 13-12 illustrates the consequence of passing a signal with high frequency noise through a sampler, and then reconstructing the original signal. Note that the high frequency

noise has been "folded", or "aliased" into low frequency noise. Note that a low pass filter on the reconstructed signal will do no good.

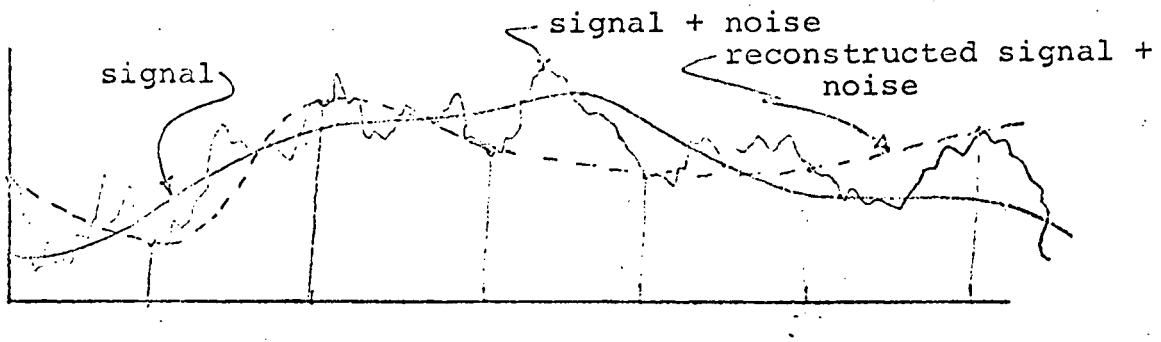


Figure 13-12

Aliasing of High Frequency Noise

It is an important design rule to insure that high frequency information or noise does not get sampled.

13-6.0 Analysis of Linear Sampled Data Systems

Figure 13-13 introduces elements of a computer controlled system.

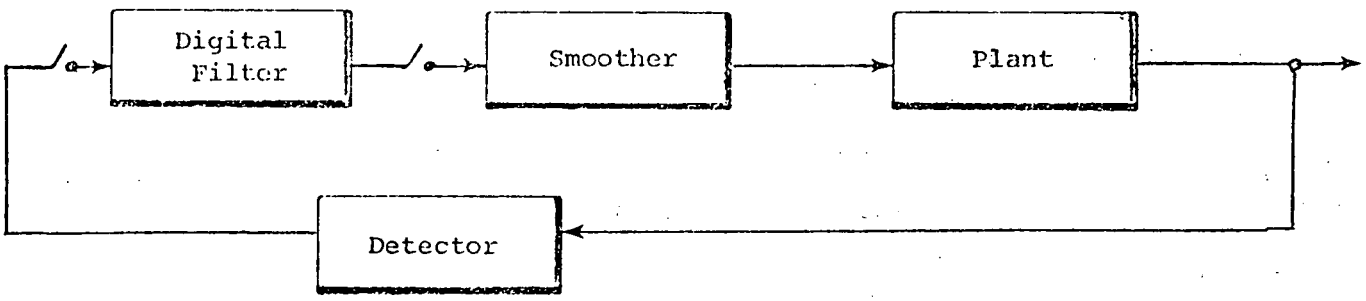


Figure 13-13

A Digitally Controlled System

The sampler is a linear element since, if for any additive component of input signal x(t), the sampler output is x*(t), then, if x(t) is modified by any scale factor, A, the sampler output will

contain the component Ax*(t). Therefore, introduction of a sampler does not of itself introduce non-linearity.

13-6.1 Difference Equations, z and w Transform. Linear continuous systems are characterized by sets of linear differential equations

$$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_0 = b_n y^n + b_{n-1} y^{n-1} + \ldots + b_0 \qquad (13\text{-}3)$$

where the superscript indicates the order of differentiation of the variables x and y.

In discrete systems, the analogous equations are linear difference equations, since the information in the system consists of samples at present and past sample instants. Equation 13-4 is a linear difference equation

$$a_n x_{-n} + a_{n-1} x_{-(n-1)} + \ldots + a_0 = b_n y_{-n} + b_{n-1} y_{-(n-1)} + \ldots + b_0 \qquad (13\text{-}4)$$

where the subscript on the variable defines the "staleness" of the sample, in number of sample intervals. For example, x_{-n} is the value for x taken n intervals previous to the present sample instant.

If the coefficients of differential equations can be presumed non-time varying, the operational mathematics of Laplace transform theory becomes useful. The advantage obtained is that one can work with algebraic equations in the Laplace operator s, and relatively simpler arithmetical procedures can be substituted. (For example, multiplication replaces convolution.) For the continuous system, the operational equation is

$$(a_n s^n + a_{n-1} s^{n-1} + \ldots + a_0) X(s) = (b_n s^n + \ldots + b_0) Y(s) \qquad (13\text{-}5)$$

Presume for the moment that an equivalent operational math exists for difference equations. Let the operator z be used instead of s. Then, for the difference equation we write

$$(a_n z^{-n} + a_{n-1} z^{-(n-1)} + \ldots + a_0) X(z) = (b_{-n} z^{-n} + \ldots + b_0) Y(z) \qquad (13\text{-}6)$$

The nth previous sample of x, which we denoted as x_{-n}, can be related to the present value of the continuous signal by a Taylor

series expansion, assuming differentiability, as

$$x_{-n} = x + x'(-nT) + \frac{1}{2!} x''(-nT)^2 + \ldots \tag{13-7}$$

and its Laplace transform would be

$$X_{-n}(s) = (1 + (-nT)s + \frac{1}{2!} (-nT)^2 s^2 + \ldots)X(s) \tag{13-8}$$

$$X_{-n}(s) = e^{-nTs} X(s)$$

so the Laplace transform of the difference equation becomes

$$(a_n e^{-nTs} + a_{n-1} e^{-(n-1)Ts} + \ldots + a_0)X(s) = (b_n e^{-nTs} + \ldots + b_0)Y(s) \tag{13-9}$$

Comparison of equations 13-6 and 13-9 indicate that the presumed operator z exists and is related to the Laplace operation s by

$$z = e^{sT} \tag{13-10}$$

13-6.2 Conditions for Stability of the Equations. The imaginary axis of the complex plane defines the boundary of stability for roots of the characteristic equation in Laplace transform theory. Equation 13-10 rewritten as

$$z = e^{\sigma T} e^{j\omega T} \tag{13-11}$$

illustrates that the unit circle $|z| = 1$, is the equivalent boundary and that the left half plane in s is mapped entirely inside the unit circle.

13-6.3 w Transforms. The z transform converts a transcendental function in s (involving terms in e^{sT}) into polynomial functions in z. However, the process maps the left half plane into an infinitely multiple valued surface all within the area of the unit circle. The w transform remaps a single layer of this unit circle surface back into an entire left half plane. The w transform, or bilinear transform, is defined by

$$z = \frac{1 + w}{1 - w} \tag{13-12}$$

or

$$w = \frac{z - 1}{z + 1} \tag{13-13}$$

In this manner, a variable is created which satisfies all the requirements of the well-known Bode analysis.

There are several design techniques which take advantage of the similarity between the s domain and the w domain, Ref. [7,11].

13-7.0 Sampled Continuous Systems

Digital control requires analysis of combinations of differential equations (defining the plant and its associated elements) and difference equations (defining the computer filter).

Figure 13-14 is a rearrangement of the continuous portion of the control system of Figure 13-13 with the (sampled) computer output as the input signal, and the (sampled) computer input as the response.

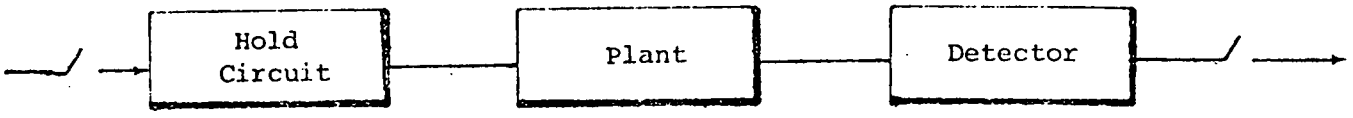


Figure 13-14

Continuous Portion of the System

The process of obtaining the Laplace transform of these elements including the effects of the sampler is a matter of extreme practical difficulty, if the continuous elements are not trivial. This difficulty arises because a mathematical description of the sampler operation involves multiplication of the signal by a series of unit impulses. The Laplace transform of a product of signals involves complex convolution.

As a practical matter, the process involves breaking down the algebraic expression for the Laplace transform of the continuous system into a partial fraction expansion (parallel) form, evaluating the z transform of each parallel term separately, and then reconstituting the results.

Figure 13-15 illustrates the partial fraction expansion form of a Laplace transform function.

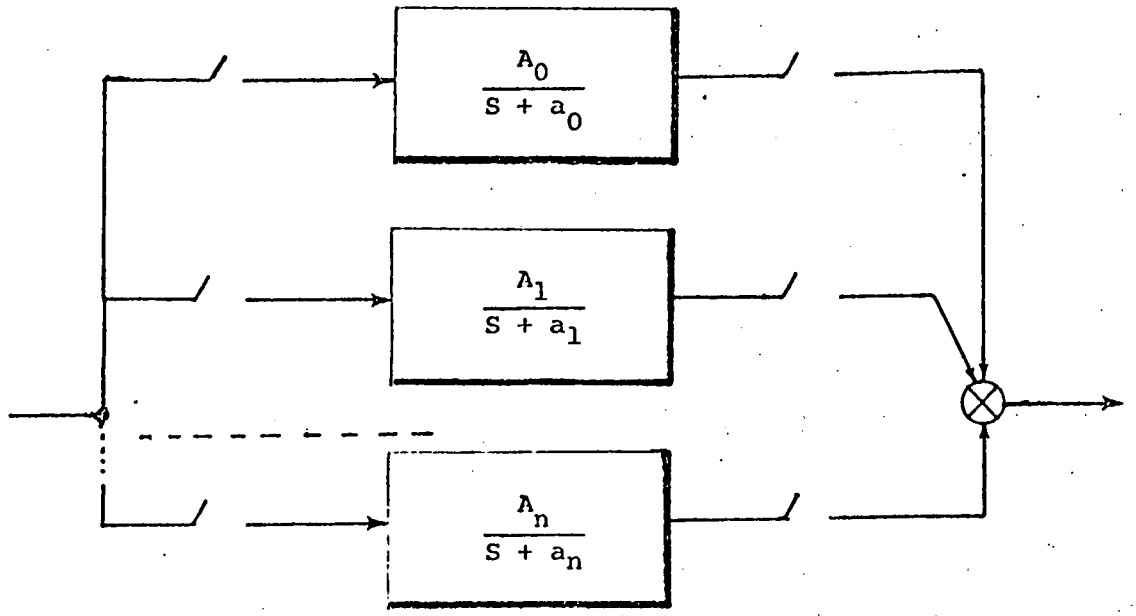


Figure 13-15

Partial Fraction Form

In this manner, one creates a complete algebraic model in the operator z for the elements in the system, as in Figure 13-16, with appropriate specifications for stability, speed of response and so forth. It is the task of the designer to create the compensation D(z) appropriate to the system characteristics and requirements.

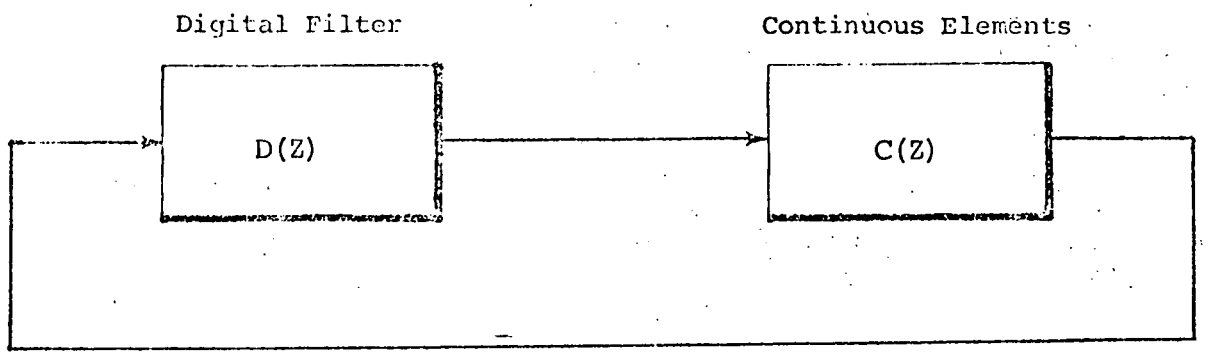


Figure 13-16

Reduction to a Discrete System

13-7.1 An Example - Ref. [12]. The Pershing Missile digital autopilot is a system which receives attitude commands from a steering loop, and, based on measured vehicle inertial attitude, θ, computes vane deflection commands, δ, to control the vehicle to the programmed trajectory. The transfer function relating measured vehicle to vane deflection commands for a particular period of its ascent trajectory can be approximated as

$$\frac{\theta}{\delta}(S) = \frac{7.86(S+22\pm 86.12j)(S+183.1)(S-186.1)}{(S+3.53)(S-3.53)(S^2+2(.008)\ (65.8)S+65.8^2)(S^2+2(.0075)231.8S+231.8^2)} \qquad (13\text{-}14)$$

This equation includes the effect of the first two bending modes as well as the high frequency rigid body motion.

A Bode plot of this dynamical system; log (amplitude) versus log (frequency) is included as Figure 13-17.

The block diagram of the closed loop system is shown in Figure 13-18.

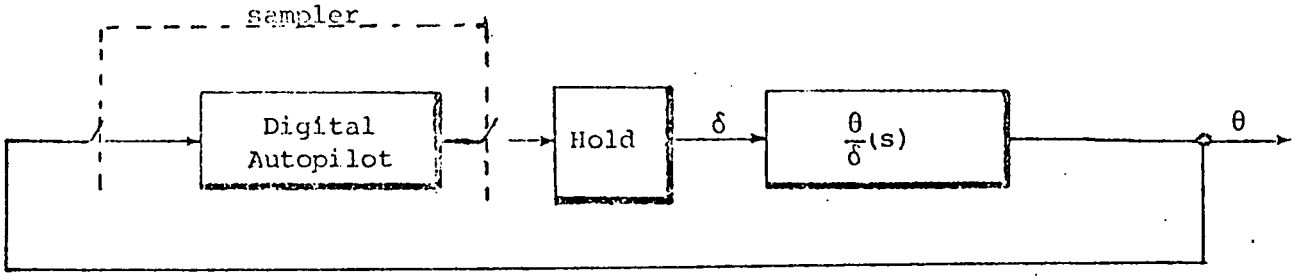


Figure 13-18
Pershing Autopilot

The z transform and w transform of the difference equation which result from applying samplers to the continuous system (for a sampling period of 16 x 10^{-3} seconds) are listed in Table 13-1.

The w plane Bode plot for the system is shown in Figure 13-19, ($w = u + jv$). It will be noticed that the functions are similarly shaped for small values of ω and v. Figure 13-20 is a Nichols chart (log amplitude versus phase) for the sampled data system with the frequency v as the parameter.

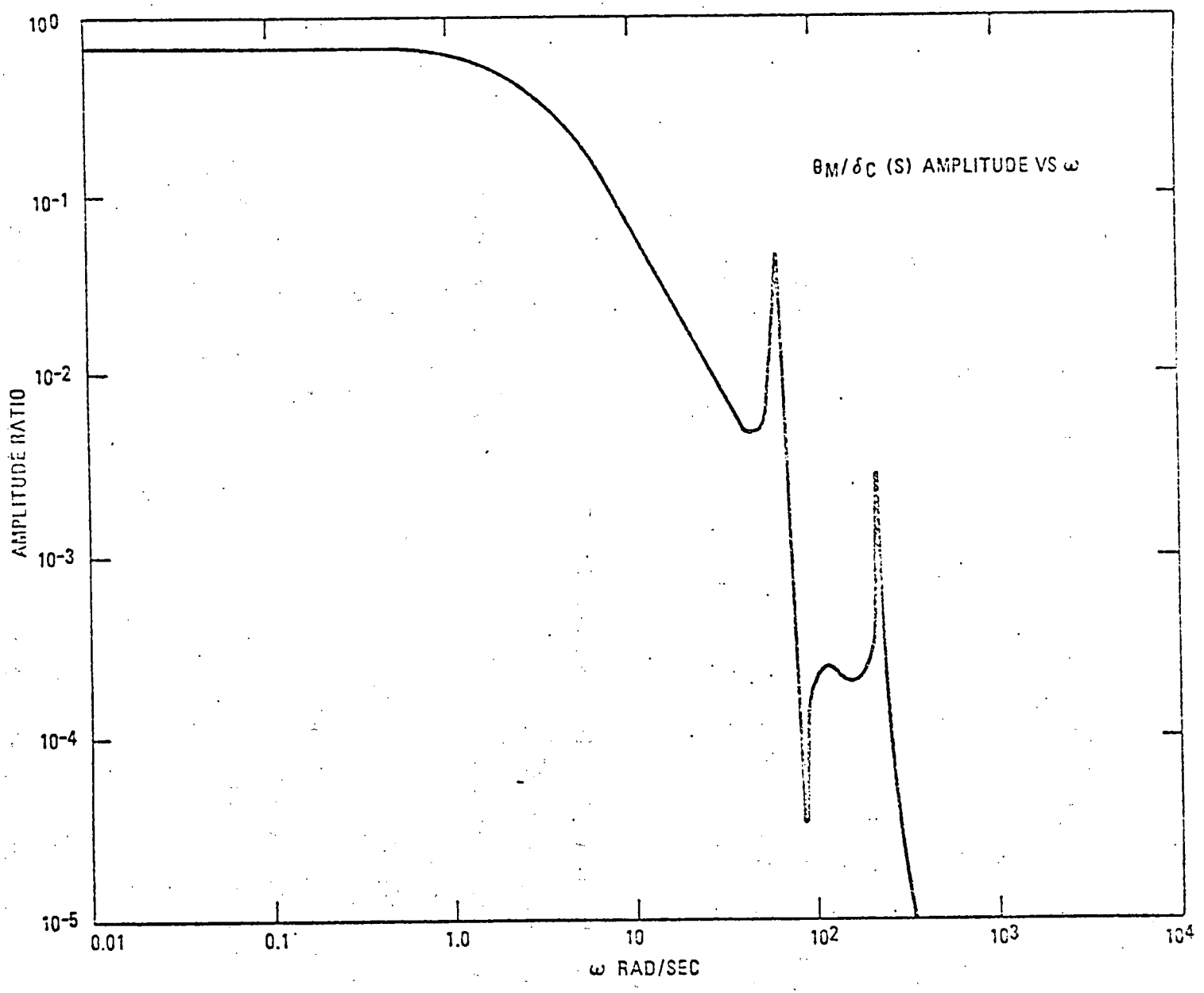


Figure 13-17 - Continuous Systems

F(z)

Sample Period 16×10^{-3} secs (62.5/sec)

GAIN = 3.662×10^{-5}

Denominator Factors:	Numerator Factors:
$z - 1.058$	$z + .03373$
$z - .9451$	$z - .05375$
$z - .2009$	$z + .3371$
$z - .4903 \pm .8619j$	$z - 16.976$
$z + .8203 \pm .5226j$	$z + 1.180 \pm .6384j$
$z + .02098 \pm .08678j$	$z - .1892 \pm .9657j$

F(w)

Sample Period 16×10^{-3} secs. (62.5/sec)

GAIN = -1.091×10^{-4}

Denominator Factors:	Numerator Factors
$w - .02818$	$w + 1.07$
$w + .02822$	$w + .898$
$w + .6654$	$w + 2.017$
$w + .00565 \pm .5816j$	$w - .889$
$w + .1768 \pm 3.422$	$w - 1.00$
$w + 1.027 \pm .1797$	$w - 1.817 \pm 2.90j$
	$w + .0135 \pm .823j$

Table 13-1: F(z), F(w) at 62.5 samples/sec.

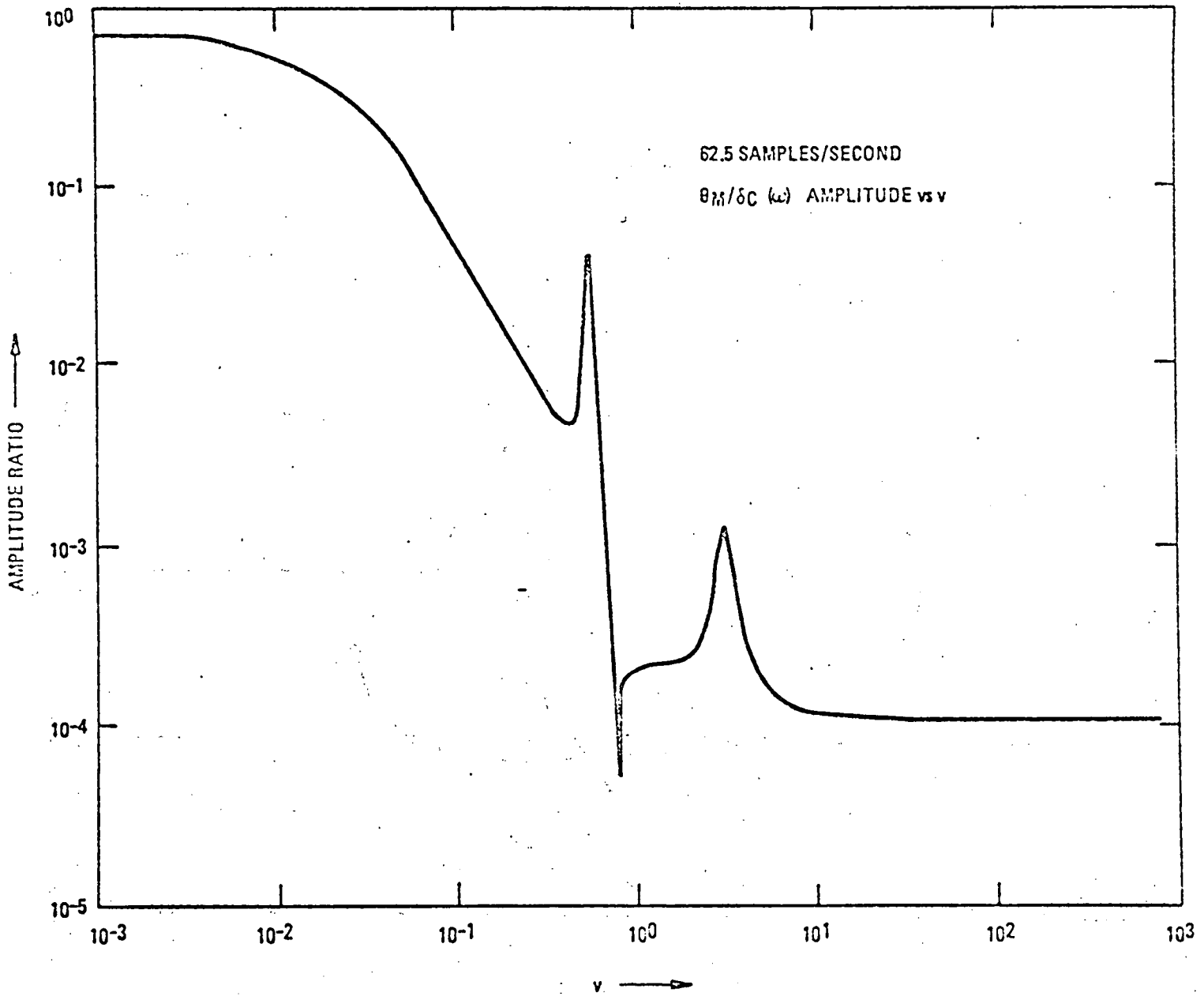


Figure 13-19 - Bode Plot, Descrete System

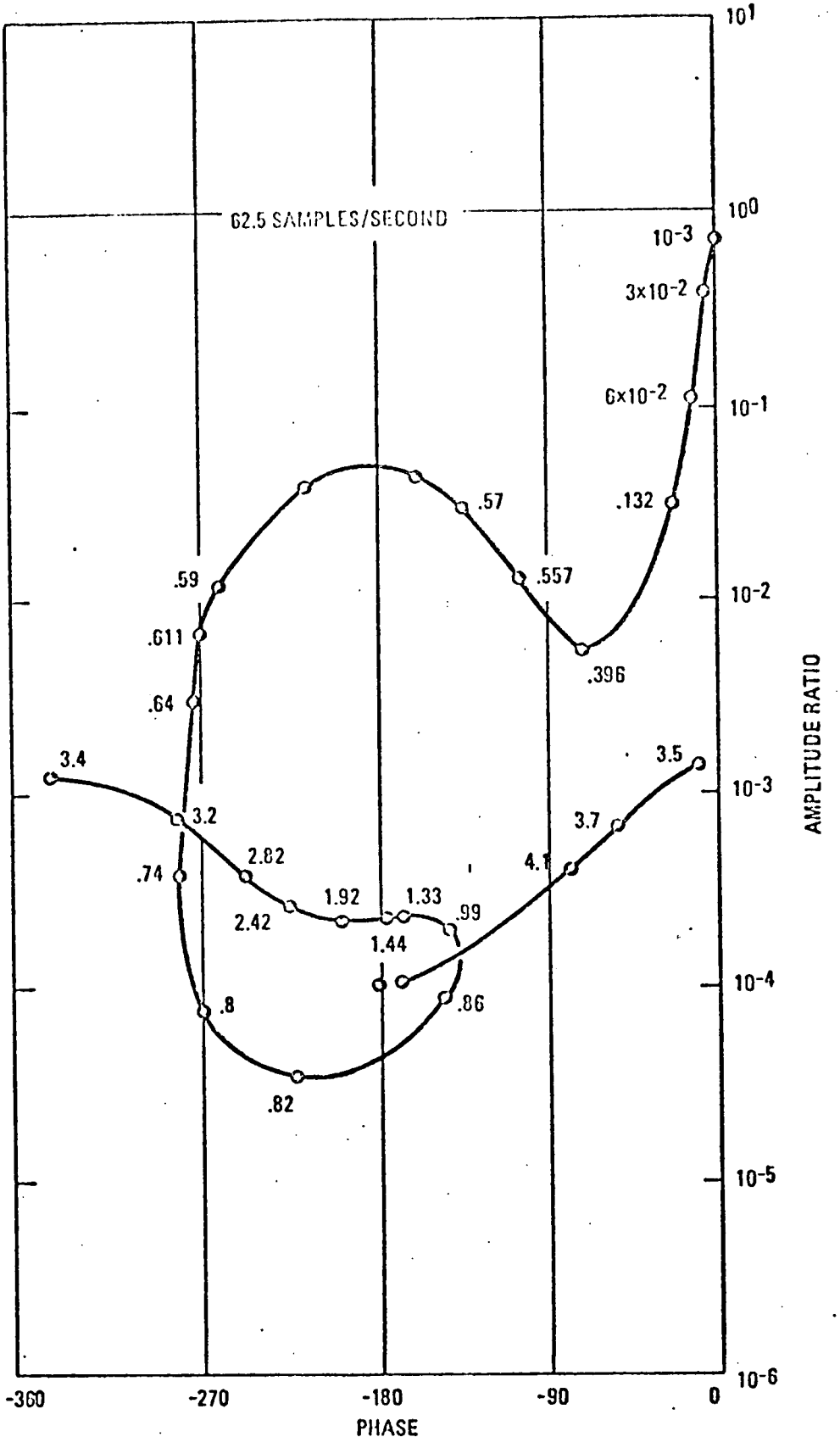


Figure 13-20
Nichols Chart Discrete System

13-7.2 Design of the Compensation. An examination of Figure 13-20 indicates various requirements for the digital filter. First the cross over frequency must be somewhere in the range of .03<v<.132 since for frequencies lower than the frequency of the aerodynamic divergence, the gain versus phase function collapses to a point. Above v = .132 the first bending mode causes too rapid and undependable fluctuation to permit reliable compensation. Another requirement involves shifting the phase of the first bending mode loop away from the 180° point by additional lead at high frequencies. By a trial and error process, a compensation function to be implemented in the digital autopilot is defined to be

$$D(w) = \frac{-13\ (w + .03)(2 + .5)}{w^2 + 2(.5)(.225)w + .225^2} \qquad (13\text{-}15)$$

A plot of the total open loop function including the compensator is shown in Figure 13-21. The gain term, -15.41, is not included in this figure.

The inverse bilinear transform yields the equivalent compensation in the z domain.

$$D(z) = \frac{-15.41\ (z^2 - 1.2751z + .31391)}{(z^2 - 1.4518z + .61646)} \qquad (13\text{-}16)$$

A digital filter that implements this compensation is shown in Figure 13-22.

13-8.0 Design Tools, Ref. [13]

We have shown how linear dynamical systems which contain sampler elements and/or time delays can be analyzed using z-transform theory. However, the numerical operations involved are tedious and the results are often extremely sensitive to lack of numerical precision. For these reasons, Intermetrics has developed a sequence of computer programs which support the analysis of sampled data systems.

These programs are designed for use on a time-shared computing facility. Data resulting from one operation frequently becomes input data for a subsequent operation. The user can either transfer the data using certain file handling operations or he can transfer the data by hand using the teletypewriter.

INTERMETRICS INCORPORATED • 701 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

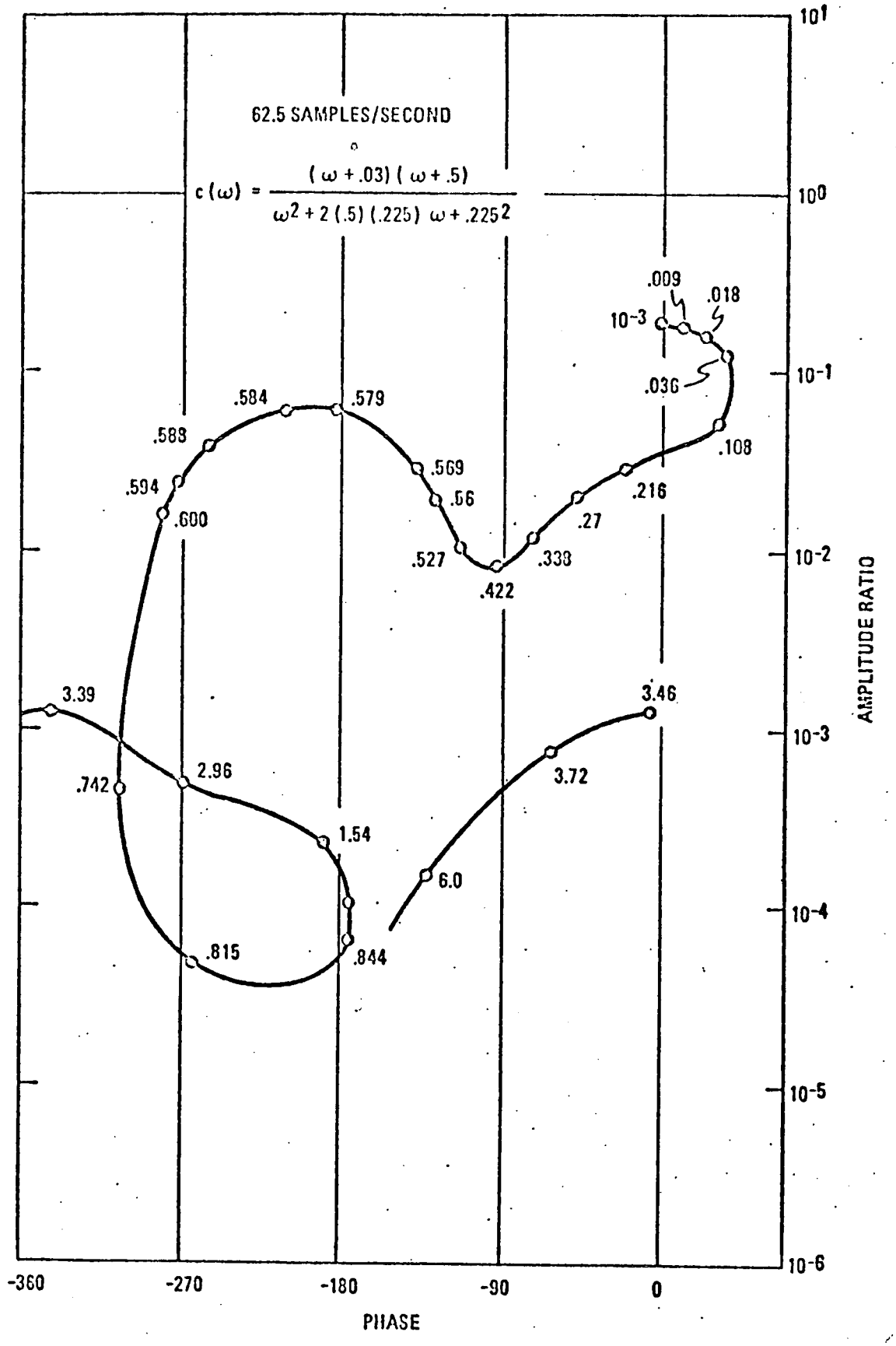


Figure 13-21
Nichols Chart Discrete System

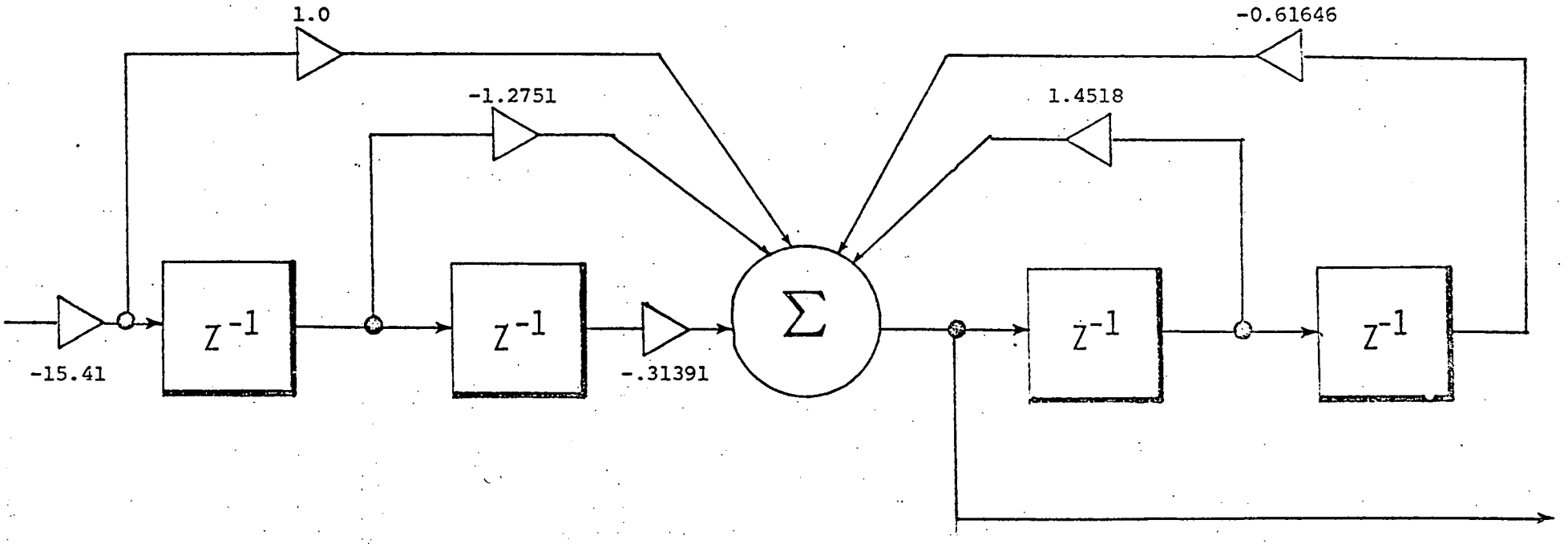


Figure 13-22 - Filter Implementation

As an example of the use of these programs, consider that a user wishes to design compensation for a digital computer used in a feedback control application. He has obtained the Laplace transform of the response at the input to the computer to an impulse at the output of the computer. Using these programs he can:

1) Calculate the z-transform of the system, accounting for the effect of synchronized samplers at the input and output; he can include the effects of a hold circuit at the computer output sampler.

2) Calculate the w-transform of the system.

3) Calculate frequency response data (amplitude and phase) versus increasing imaginary values of the complex variable w.

4) Using Bode and Nichols techniques, synthesize compensation and re-analyze the resulting system.

5) Transform the compensation function from the w-plane to the z-plane in order to determine difference equation coefficients.

6) Obtain transient response time histories of the closed loop system.

In addition to the above described sequence, the user can obtain partial fraction expansions, factor polynomials, multiply polynomials, and combine factors into polynomials. These programs are designed to accomodate on the order of 50th-order polynomials. Repeated roots are accommodated to an arbitrary degree. Some of these programs are based on programs developed at the MIT Instrumentation (Draper) Laboratory, Ref. [14].

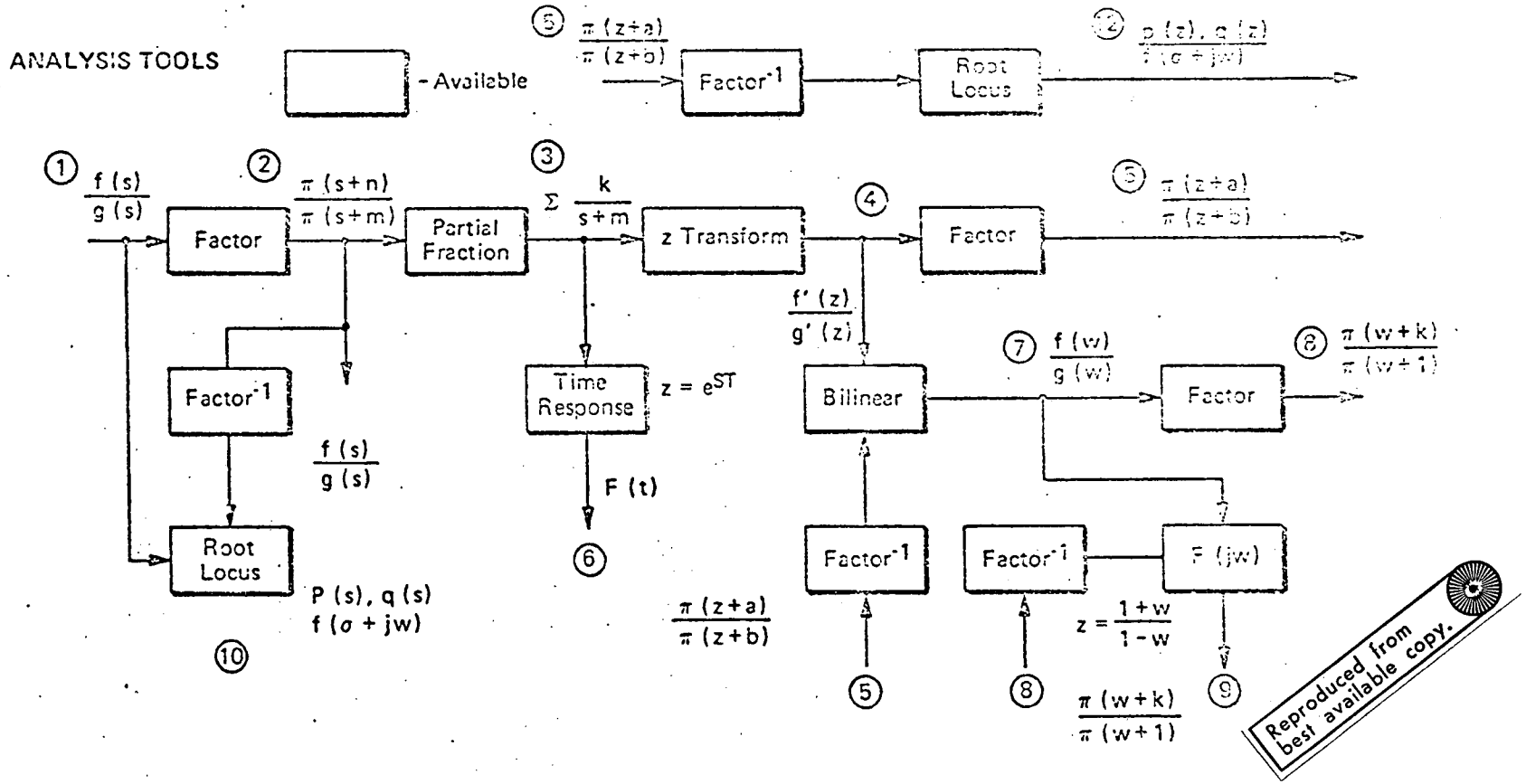
Figure 13-23 is a diagram of the various operational processes automated by these programs. In addition to those shown in the diagram, a power spectral density analysis program is also part of the package.

13-9.0 Practical Considerations in Realization of Discrete Filters.

13-9.1 Stability and Coefficient Accuracy

While the methods described in section 13-6.0 and illustrated in section 13-7.0 provide a basis for implementing a discrete filter, practitioners have been bothered by certain numerical difficulties.

As an historical example, consider the development of the digital autopilot for the Apollo command and service module. The steps in



1. Ratio of polynomials in s
2. Ratio of factors in s
3. Partial fraction expansion of f(s)
4. Ratio of polynomials in z
5. Ratio of factors in z
6. Time response L^{-1} (F(s))
7. Ratio of polynomials in w
8. Ratio of factors in w
9. F (jw) vs w (AR & Phase)
10. Roots of 1 + kG(s) for varying k
11. Roots of 1 + kG(z) for varying k

Figure 13-23
Operational Processes

its design were as follows, Ref. [15]:

A compensation filter was designed. The design utilized the w-plane to obtain the desired system stability and response. When this function was implemented on the 15 bit Apollo computer, considerable difficulty was experienced. This difficulty was only ultimately resolved by careful attention to selection of methods of rounding the coefficients and by selecting an insensitive form for the sequential solution of the algorithm. This experience is not untypical.

More recently Kaiser, Ref. [18], has been able to show in a fairly general way that the coefficient accuracy requirements for realization of digital filters increase proportionally to the order n of the filter, and that increasing the sampling rate also increases the requirements for increased word length in coefficient storage. These results confirm the practical experience of the author.

13-9.2 Canonical Forms

The implementation of a given digital filter can take several forms, all equivalent under the assumption of no quantization or round-off in the implementation. The form used in Figure 13-22 will be called the direct form, since it is directly related to a generalized form of the linear difference equation such as equation 13-4. However, this form can be shown to involve a significant excess of storage elements, and further it often exhibits poor numerical properties in the presence of finite numerical accuracy, Ref. [17].

The following technique, Ref. [10], leads to a less complex form of filter implementation. We start with the z domain equation for the transfer function, input/output relationship.

$$y(z) = X(z)\ H(z) \tag{13-17}$$

or

$$Y(z) = X(z)\ \frac{\sum_{i=0}^{r} L_i z^{-i}}{\sum_{i=0}^{m} K_i z^{-i}} \tag{13-18}$$

and define an intermediate operator W(z), such that

$$W(z) = \frac{X(z)}{\sum_{i=0}^{m} K_i z^{-i}}$$

$$Y(z) = W(z) \sum_{i=0}^{r} L_i z^{-i}$$

we derive from these the following simultaneous difference equations

$$x(nT) = x(nT) - \sum_{i=1}^{m} K_i w(nT - iT)$$

$$y(nT) = \sum_{i=0}^{r} L_i w(nT - iT)$$

This yields the structure illustrated in Figure 13-24. This form is often called the canonic form of the filter, although many other arrangements can be constructed with the same-number of storage elements. However, this is the minimum number of storage elements.

In addition to this form, Jackson, Ref. [19] and others define the cascade form, related to a factored version of the transfer function, a parallel form, related to the partial fraction expansion version of the transfer function, and combinations of these.

It turns out that these forms, while equivalent in an idealized analysis, have significantly different properties in the presence of quantized signals and finite coefficient accuracy. Apollo experience bears this out.

13-9.3 Noise Analysis

Noise analysis for discrete filters is not as straightforward as it is for continuous systems. The sampling process produces aliasing which means that a complex spectrum of side bands of noise (and signal) are created by the sampler. As a part of the Pershing Digital Autopilot design contract, Ref. [12] a loop noise analysis including the effects of the sampler and the autopilot difference equations was performed. The method was automated, and has become part of the design tool package described previously. As an example of this procedure, Figure 13-25 is an exhibit of the noise power spectral density at the Pershing vane actuator due to quantization at the inertial measurement unit attitude transducer.

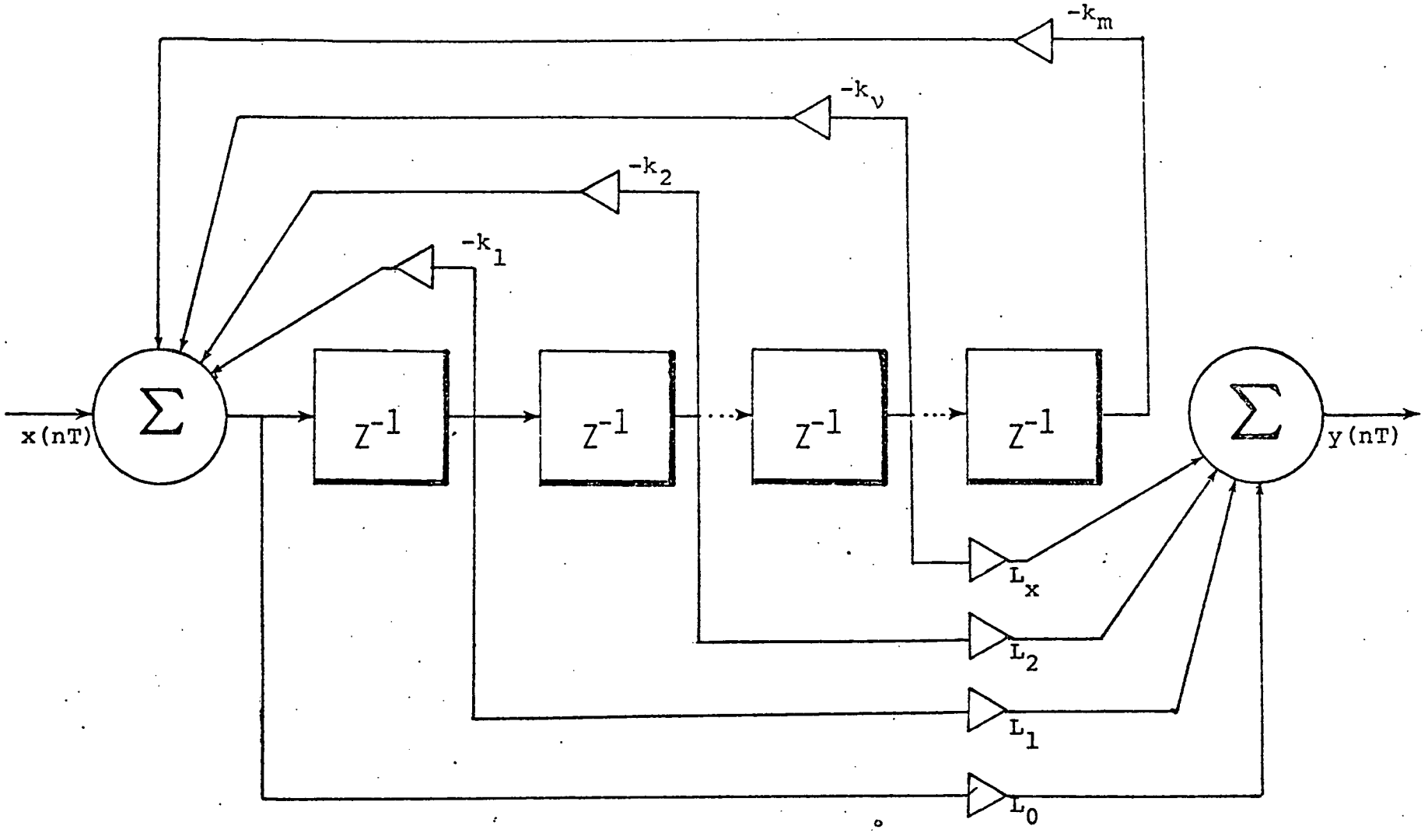


Figure 13-24 - Canonic Form

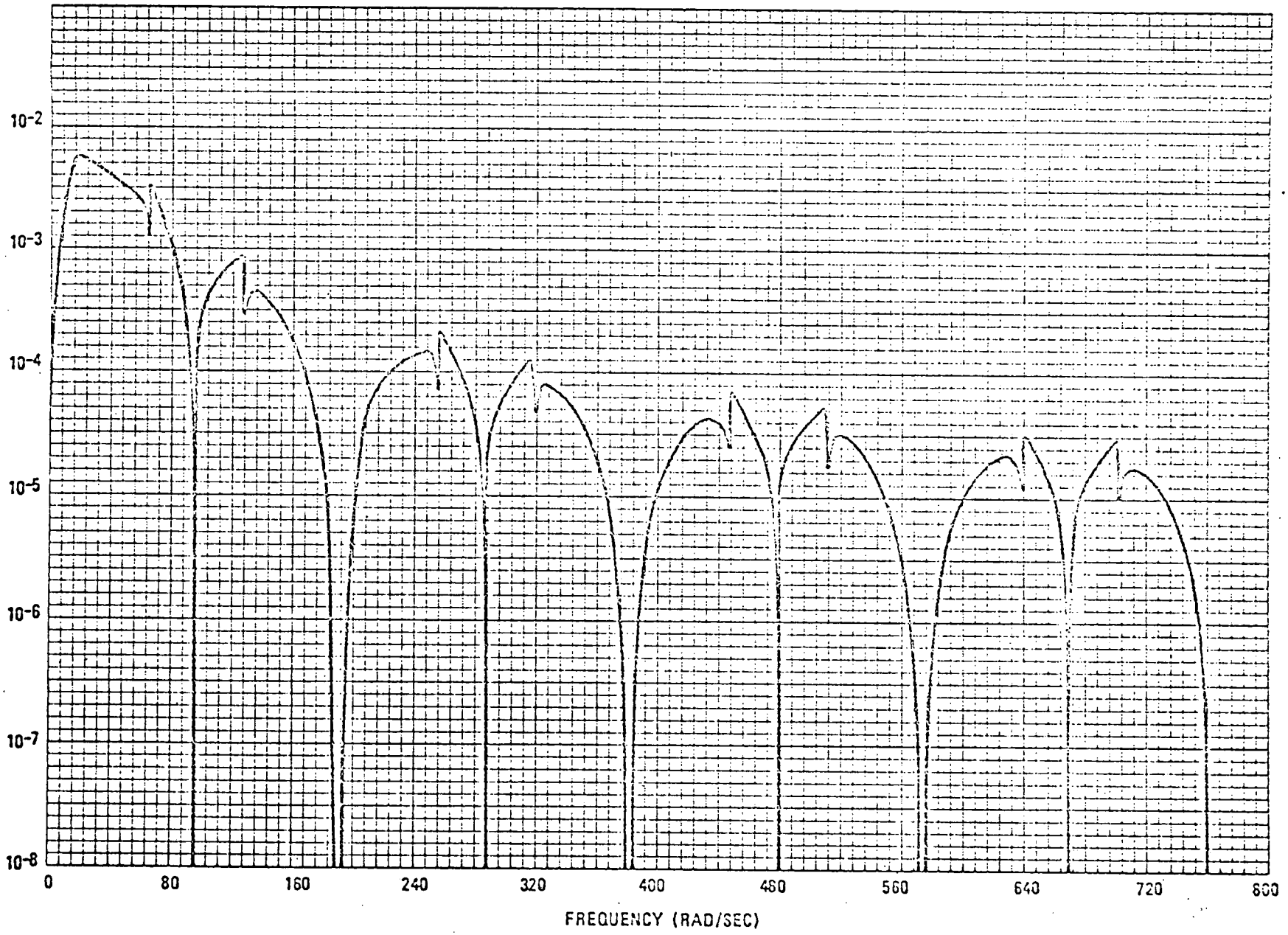


Figure 13-25 - Pickoff Noise Input 30.5 samples/sec

INTERMETRICS INCORPORATED • 701 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

13-10.0 Summary

This chapter has briefly described an approach to the analysis of sampled data control problems. The method of approach might be termed the classical approach because it is an adaptation of the spectral analysis methods developed for continuous control system analysis.

We have also emphasized a physical motivation for the sampling theorem, which is fundamental to successful reconstruction of information from discrete samples.

REFERENCES

1. Bryson, A.E., and Yu-Chi Ho, Applied Optimal Control, Blaisdell, Waltham, Mass., 1969.

2. Joseph, P.D., and Tou, J.T., "On Linear Control Theory", AIEE Trans., Applications and Industry, Vol. 80, September 1961, pp. 193-196.

3. Widnall, W.S., Applications of Optimal Control Theory to Computer Controller Design , the M.I.T. Press, Cambridge, Mass., 1968.

4. Luenberger, D.G., "Observing the State of a Linear System", IEEE Trans. on Military Electronics, April 1964, pp. 74-80.

5. Cox, K.J., "A Case Study of the Apollo Lunar Module Digital Autopilot", IEEE Case Studies in System Control, 69-C41, AC, August 1969, Boulder, Colorado.

6. Widnall, W.S., "The Minimum Time Thrust-Vector Control Law in the Apollo Lunar Module Autopilot", Automatica, Vol. 6, Pergamon Press, 1970, pp. 661-672.

7. Ragazzini, John R., and Franklin, Gene F., Sampled-Data Control Systems, McGraw-Hill, New York, 1958.

8. Tou, J.T., Digital and Sampled-Data Control Systems, McGraw-Hill. New York, 1959.

9. Kuo, Benjamin C., Analysis Synthesis of Sampled-Data Control Systems, Prentice-Hall, Englewood Cliffs, N.J, 1970.

10. Gold, Bernard, and Rader, Digital Processing of Signals , McGraw-Hill. New York, 1969.

11. Kaiser, J.F., "Design Methods for Sampled Data Filters", Proceedings, First Allerton Conference on Circuit and System Theory, November, 1963.

12. Copps, E.M., et al., Analysis and Design of a Digital Autopilot , Final Report, contract DAAH01-71-C-0056, (unclassified), Intermetrics, Inc., Cambridge, Mass., 1971.

13. Copps, E.M., et al., Users Guide to Feedback Control Programs , Intermetrics, Inc., Cambridge, Mass., 1971.

14. Fraser, D.C., A Sequence of Computer Programs Useful in the Analysis of Feedback Control Systems , E-1800, C. S. Draper Laboratory, M.I.T., Cambridge, Mass., May 1965.

15. Wiseman, R.L., Round-off Techniques for a Digitalized System, T-497, C.S. Draper Laboratory, M.I.T., Cambridge, 1968.

16. Whitman, C.L., The Implementation of Digital Filters in Computers of Small Word Length, T-443, C.S. Draper Lab., M.I.T., Cambridge, Mass, Feb. 1966.

17. Martin, F.H., "Programming and Bench Testing of the Digital Filter for the CSM Autopilot", SGA Memo 26-25, C.S. Draper Lab., M.I.T., October 1965.

18. Kaiser, J.F., "Some Practical Considerations in the Realization of Digital Filters", Bell Telephone Labs, Inc., Murray Hill, N.J.

19. Jackson, L.B., et al., "An Approach to the Implementation of Digital Filters", IEEE Trans on Audio and Electracoustics, Vol. AU-16, No. 3, September 1968.

PART IV

SOFTWARE TECHNIQUES

CHAPTER 14

DATA ORGANIZATION AND HANDLING

14-1.0 Introduction

Data handling is one of the prime purposes of a computing system. Thus, careful consideration should be given to data organization when designing the architecture of the system and the sets of algorithms to be executed on it. Judicious data organization and handling can lead to the simplification of many computational algorithms.

The purpose of this chapter is to examine the nature of data and data organization within a computing system. We will survey techniques developed to handle large quantities of data, such as table look-up techniques and sorting techniques. We will then take a more generalized look at data set organization by examining the data management structure of OS/360. Finally, we will discuss higher order languages, such as SNOBOL, developed for data handling.

14-2.0 Data Types

Data may be defined as anything that can be represented by symbols [1,2]. To a physicist, data can be represented by particle paths in a cloud chamber, or to a chemist by the specific gravity of his newly synthesized compound. However, within a computing system all data are represented by sets of bits. The organization and meaning assigned to these bits determine the type of data that the bits represent. This data is modified, altered, and manipulated by the computer while it is executing a process. The different types of data reflect the variety of processes a computer is called upon to execute.

We will consider four categories of data, which we will call data types. These are:

a.) String. String data are collections of concatenated symbols. Mathematically speaking, string data are elements of a free monoid taken over a finite alphabet.

b.) Boolean. Boolean data are sets of "true" and "false" values.

c.) Pointer. Pointer data consists of directed graphs.

d.) Numeric. Numeric data are sets of numbers.

Each data type is a partial description of the data that fall within its category. To properly interpret the bits representing the data, a more complete description of the data must be given (either explicitly or implicitly). Examples of the descriptive items that must be known in processing these data are the following:

a.) For String Data:

String length, fixed or variable
Left or right justification
Code used; e.g., EBCDIC, ASCII

b.) For Boolean Data:

The code for each truth value
Field Length

c.) For Pointer Data:

Machine address of item
Item number in table
Code for null pointer

d.) For Numeric Data:

Code; e.g., digit or radix
Sign treatment; e.g., 2's complement
Scale, fixed or floating
Rounded or truncated calculation
Numeric limits on values
Precision
Aligned or packed data

14-3.0 Data Organization

With many processes large quantities of data must be handled. It then becomes necessary to organize the data so that data handling techniques can be incorporated in the algorithms. The organization decided upon for the data is a factor in determining how efficiently these algorithms will execute. For example, when multiplying matrices, the organization of the columns and rows can greatly simplify the coding of the algorithm.

The two most widely used structures for organizing data are lists and trees [3-5]. A list is an ordered linear sequence of zero or more elements. The order between two adjacent list elements can be implicitly or explicitly given. In the former case a simple linear list is merely a set of data elements in contiguous main memory locations. Two adjacent elements are ordered by virture of the fact that they are adjacent to each other with the index of an element in the list either monotonically increasing or decreasing with memory address.

An explicit ordering between list elements is found in the case of a threaded list. Each element of a threaded list contains two items, a piece of data and a pointer to the next list element. An example of a threaded list is shown in Figure 14.1. In this example the list is threaded in both directions (doubly threaded). That is, each data element contains two pointers, one to the previous element and one to the next element in the list.

Each of these list structures has advantages and disadvantages which must be considered when deciding upon which structure to use in organizing a collection of data. A simple linear list has the advantage of fast retrieval for a particular element due to the use of contiguous memory locations. However, expansion or contraction of a linear list can be very time consuming. On the other hand, a threaded list allows easy additions and deletion of elements. These processes are often merely an exercise in pointer manipulation. The disadvantage of a threaded list is that to retrieve element i it is often necessary to follow the chain of pointers for the first (i - 1) elements. For large i, this is a costly process.

A tree is a nonlinear organization of data. Elements of a tree consist of nodes (pieces of data) and branches (pointers to nodes). An example of a tree structure is shown in Figure 14.2 Knuth [3] gives the following recursive definition of a tree: "A tree T is a finite set of one or more nodes such that

a.) there is one specially designated node called the root of the tree, root (T); and

b.) The remaining nodes (excluding the root) are partitioned into $m \geq 0$ disjoint sets $T_1, \ldots, T_m$

Figure 14.1: Threaded List Structure

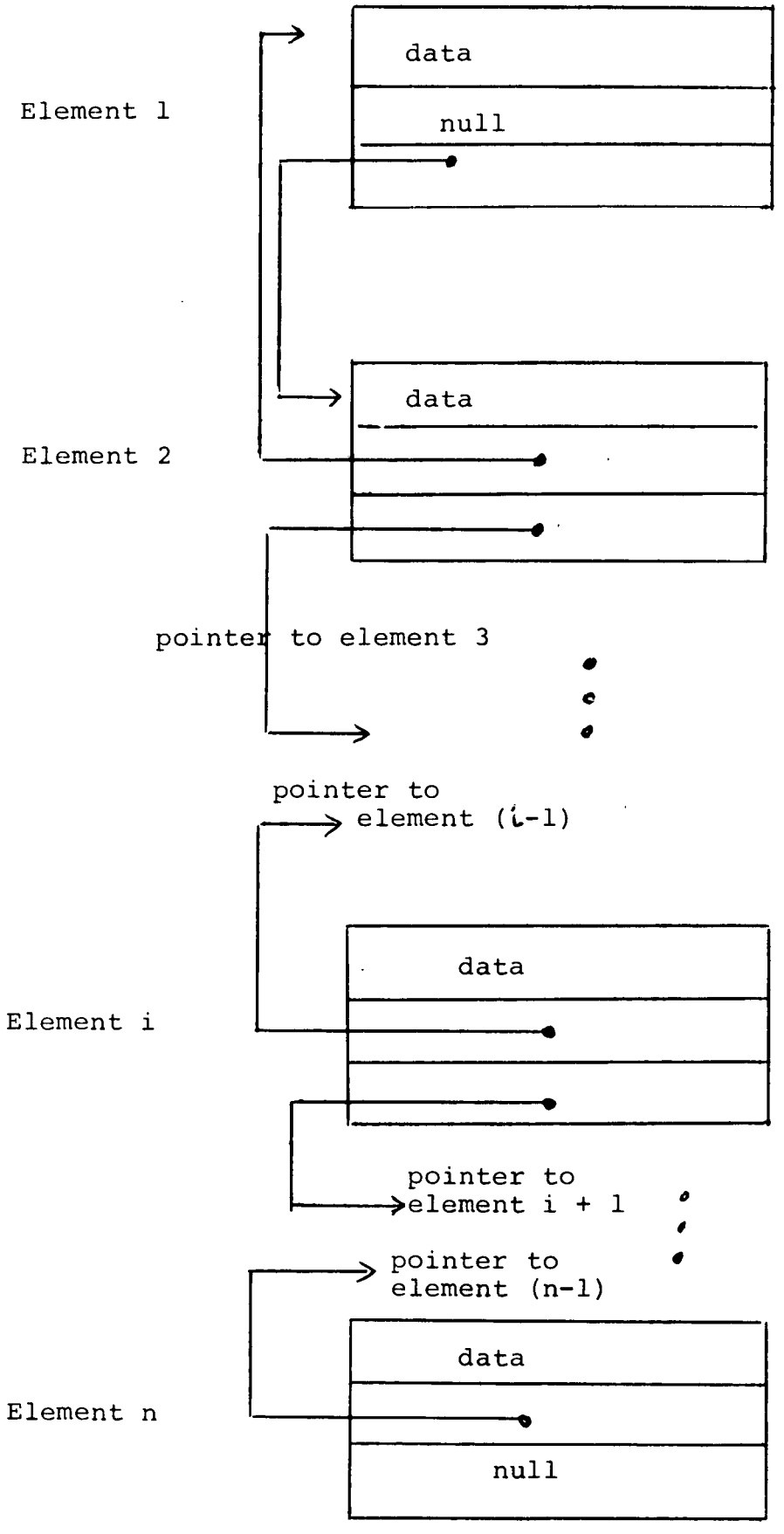
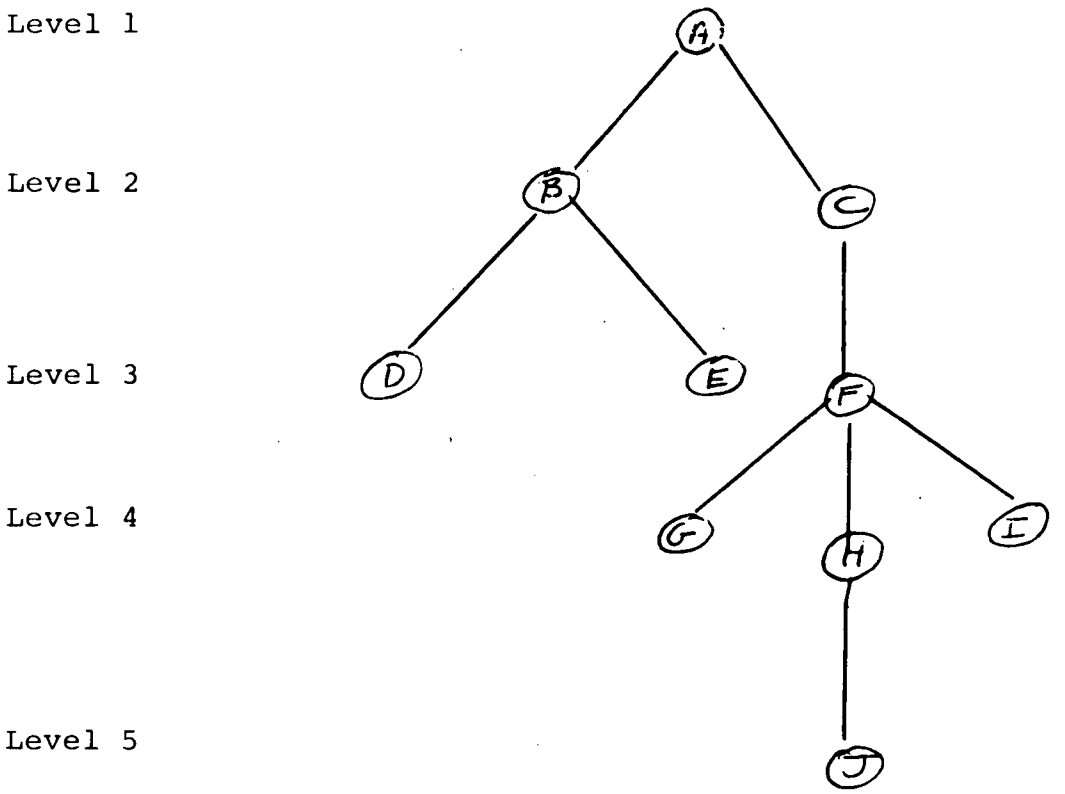


Figure 14.2: Example of a Tree Structure



and each of these sets in turn is a tree. The trees T_1, ..., T_m, are called subtrees of the root."

Trees are a necessary form of data organization in programming systems where a linear ordering cannot be imposed upon the data. For example, in list processing computer languages, such as LISP 1.5 [6], functions can be recursively defined leading to nonlinear structuring among pieces of data. The LISP Compiler will represent data as tree structures within a LISP program.

In general , when designing algorithms to handle large quantities of data, one must pay attention to the organization imposed upon the data elements. Each organization has advantages and disadvantages that will be reflected in the complexity of the algorithms processing these data.

14-4.0 Data Handling

Once a structure has been imposed upon a collection of data, a significant system design problem arises when this data must be searched for a particular item. We saw in the last section how the data organization chosen influences the complexity of such a search. In this section we will investigate two search algorithms for simple linear lists and point out the conditions under which each manifests its advantages and disadvantages. Then after pursuing this topic we will discuss data sorting algorithms and see how imposing an inherent order upon the data can speed up a search.

14-4.1 Searching

The simplest search technique for a simple linear list is a linear search. In such a search the list is examined sequentially from the first element until the desired element is found. Such a search is very efficient for short lists; e.g., 20 elements, and assumes no order among list elements other than the inherent order of successive items. However, for longer lists a technique more systematic than the linear search is possible. This technique, the exponential search, presumes each data item has a key word associated with it and that these key words are lexicographically ordered.

The exponential search algorithm begins with a comparison of the key word of the list's middle item with the key word of the desired item to be found. If the key words agree, this is the desired data item, and the search is ended. If not, either

the top or bottom half of the list is eliminated from the search, since the keyword of the data to be found falls outside its range of keywords. This process is repeated for the remaining half of the list and continues until the desired data is found or until the list is exhausted, in which case the data is not in the list.

An analytic comparison of these two searching methods shows some interesting results. Let the time for the maximum number of probes necessary to linearly (exponentially) search a list be given by T_L (T_E). Then $T_L = A \cdot N$, and $T_E = B \cdot \log_2 (N)$, where N = the number of items in the list. A and B are constants associated with each probe of the list. Hence, we expect B to be considerably larger than A. A plot of these two equations is shown in Figure 14.3. From this plot of these two equations short lists the linear search is more efficient. The crossovers point occurrs for N=50 - 100 items using a computer like the IBM System/360 [7]. For other machines this number varies depending upon the hardware available.

Other more complex search algorithms are described in a recent paper by Price [8]. Knowing which algorithms to choose depends upon the particular application, and often optimization is achieved by a mixture of techniques. In general, algorithms more complicated than a linear search are justified when the programs that use them execute often and work with large lists of data.

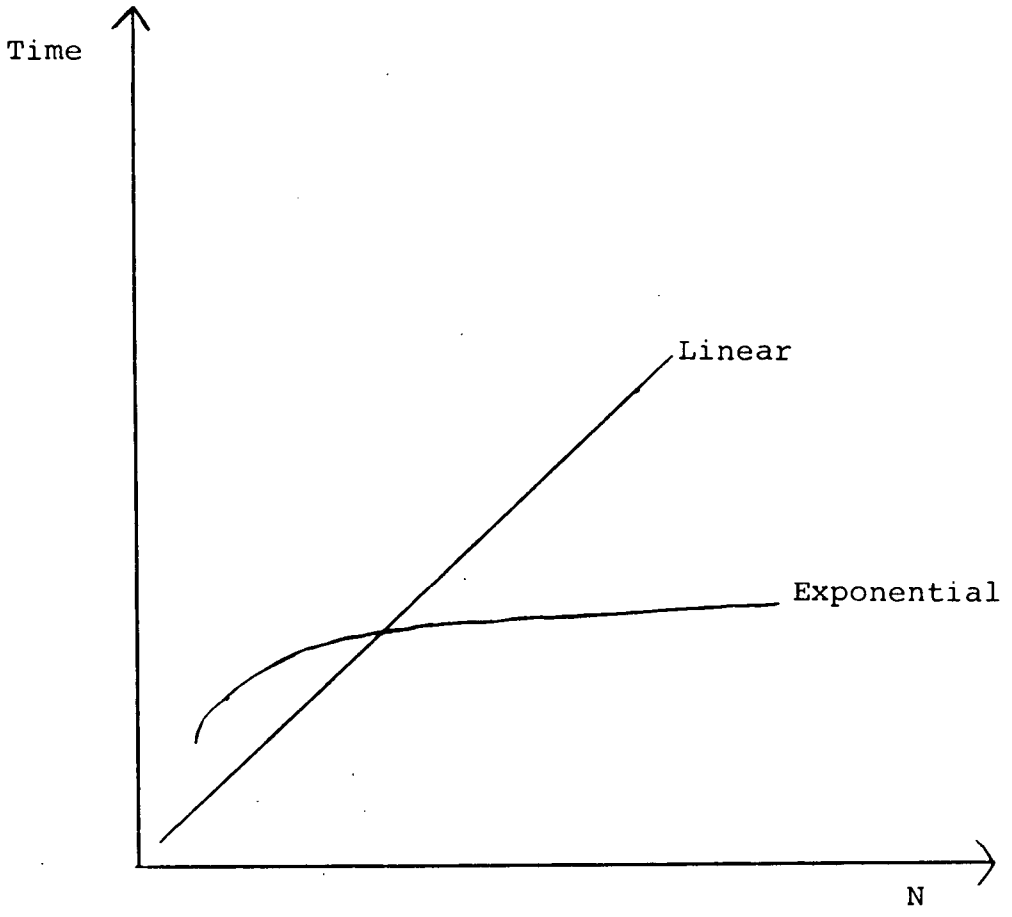
14-4.2 Sorting

As we have seen, ordering the elements of a long list (e.g., lexicographically) can lead to very improved search times. However, lists are not often generated in an ordered way. Thus, we will investigate several methods of obtaining order in a list by sorting its elements. A complete bibliography on sorting can be found in a recent paper by Martin [9].

14-4.2.1 Interchange Sort

The interchange sort is by far the simplest sort to program. It merely takes adjacent pairs of elements in the list and interchanges them to put them in order. This is a time consuming sort since several passes through the list are usually necessary. Although the algorithm can be optimized, a time roughly proportional to N^2 is necessary for the sort, where N = the number of elements in the list.

Figure 14.3: Comparison of Search Times



The execution time of an interchange sort decreases with any natural order in the data, but it is constant with respect to the distribution of magnitudes. It is interesting to note that almost no extra working storage is necessary for this sort.

14-4.2.2 Shell Sort

The Shell sort is a type of interchange sort where the items compared and interchanged are not adjacent but separated by some distance d. On the initial pass through the list $d = \frac{N}{2}$, and on each subsequent pass $d = \frac{(d+1)}{2}$, until d = 1. Figure 14.4 shows an example of an interchange sort.

The time required to perform a Shell sort is roughly proportional to $N \cdot (\log(N))^2$. Hence, it is faster than an interchange sort. The increased speed is due to the order that early passes put into the list by which later passes benefit. This sort also requires almost no extra working storage to execute.

14-4.2.3 Radix Sort

Unlike the two previous sorts the radix sort requires extra working storage for execution but has the advantage of decreased execution time. It examines the least significant digit of the keyword and assigns the data element to an area of working memory dependent only on the value of the digit. Then all items are merged in order into a list. The process continues with the next to least significant digit, etc., until no digits are left. An example of this algorithm is shown in Figure 14.5.

The execution time of a radix sort is roughly proportional to $N \cdot \log_r(M)$ where r = radix of the number system and M = keyword size. The amount of extra working storage needed is $N \cdot r$. The distribution of the magnitudes of the data affects execution time, but any natural order in the data does not.

14-4.2.4 Conclusions

Each of the sorts described above has inherent advantages and disadvantages. As in the case of searching, the type of sort used in a programming system is very dependent upon the amount and nature of the given data that it must sort. Donovan [7] concludes that for short lists interchange sorts seem best and for long lists Shell sorts. However, should speed be more important than working storage, radix sorts can replace Shell sorts.

Figure 14.4: Example of a Shell Sort

Original Order	After Pass 1	After Pass 2	After Pass 3	After Pass 4
	d=5	d=3	d=2	d=1
07	07	07	01	01
42	22	11	09	07
01	01	01	07	09
17	09	09	11	11
11	11	18	18	17
24	24	17	17	18
22	42	25	24	22
18	18	22	22	24
09	17	24	25	25
25	25	42	42	42

Figure 14.5: Example of a Radix Sort

Original List	Pass 1	List After Pass 1	Pass 2	List After Pass 2	Pass 3	Ordered List
891	0) 700	700	0) 700,501,207	700	0)	141
248	1) 891,191,341,501,141	891	1) 417	501	1) 141,191	191
191	2)	191	2)	207	2) 207,248	207
207	3)	341	3)	417	3) 341	248
341	4)	501	4) 341,141,248	341	4) 417	341
501	5)	141	5)	141	5) 501	417
677	6) 976	976	6)	248	6) 677	501
417	7) 207,677,417	207	7) 976,677	976	7) 700	677
141	8) 248	677	8)	677	8) 891	700
976	9)	417	9) 891,191	891	9) 976	891
700		248		191		976

14-5.0 Data Set Organization

Another aspect of system design in the area of data organization is the use of secondary memory devices to store large files of data. Clearly in most large computing systems the use of secondary memory is economically necessitated for storing the large quantities of data that must be handled. Using devices such as tape drives to store the data forces a sequential organization upon the data sets. However, when random access devices, such as disks and drums, are used, there is more latitude in the method of organizing data sets.

It is these methods of data set organization that we will now explore. The examples we will present are taken from OS/360 [10,11].

14-5.1 Sequential Organization

The records within a sequentially organized data set are consecutive and placed in physical, not logical sequence. Given the location of a particular record, the location of the next record is determined by physical position in the data set.

14-5.2 Partitioned Organization

Partitioned data sets contain independent sets of sequentially organized data, called members. Each member is identified in the directory of the partitioned data set along with its starting address. In OS/360 partitioned data sets are used to store program modules and, hence, are often called libraries. Figure 14.6 shows an example of such a data set.

The advantages of a partitioned organization over a sequential organization in maintaining collections of data sets are:

a.) members can be easily retrieved,

b.) members can be easily added or deleted, and

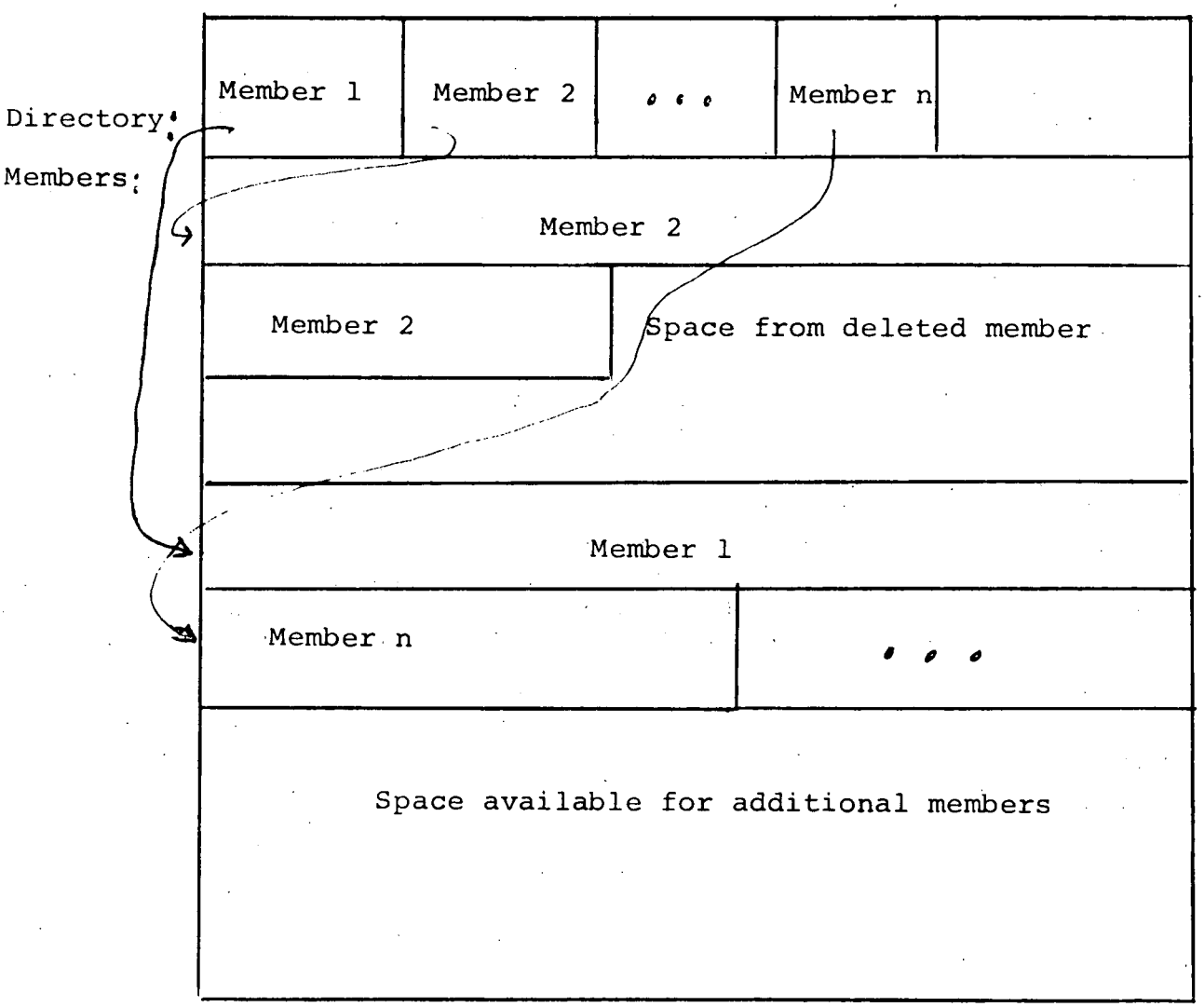
c.) deletion is merely a matter of removing the member's name from the directory.

The main disadvantage of partitioned data sets is the increased complexity of the operating system necessary to support partitioned organization and directory maintenance.

14-5.3 Indexed Sequential Organization

When data sets are stored in an indexed sequential organization, their records are arranged in a collative sequence depending upon a keyword which is part of every record. The location of each record

Figure 14.6: Example of a Partitioned Data Set



depends upon the value of the keyword. The system maintains indices of these keywords containing the location of certain recrods. These indices enable a mixture of direct and sequential access to records.

The main advantages of indexed sequential organization are:

1.) records can be inserted without rewriting the entire data set;

2.) retrieving a record is merely a matter of specifying the record's keyword; and

3.) there is convenient access to any given record without first reading the preceeding records or without maintaining a directory.

The main disadvantage is the same as with partitioned organization, the increased complexity of the operating system.

14-5.4 Direct Organization

Within this organization the records of a data set are read and written directly with the storage device address specified by the programmer. These records can be organized in any manner. No indices or directories are maintained by the system.

14-5.5 Volume Structure

A direct access volume will normally contain several data sets of differing organizations. OS/360 maintains a volume table of contents which contains a list of each data set on the volume and its address. A record of available space on the volume is also maintained.

When an executing program requests the use of a data set, the operating system finds the data set location through the table of contents. The programmer is relieved from the burden of remembering the address. In addition, when a new data set is being allocated, the operating system again helps the programmer by searching for available space. It does so by examining the volume table of contents to see if there is enough available space on the given volume. After the necessary space is found, the data set is given an organization specified by the programmer and processing can continue filling the data set with records.

14-5.6 Conclusions

Most aerospace computing systems will not need the level of generality allowed by OS/360 in data set handling. When the needs of the system are evaluated, one or two data set organizations can be settled upon. These organizations will depend upon the amount and the nature of the data to be processed. The obvious advantage of reducing the number of data set organizations used is in simplifying the operating system. A less complex operating system costs less to build and maintain.

14-6.0 Special Data Handling Languages

The value of using a higher order programming language in a computing system has been recognized [12]. In fact, special higher order languages have been developed with capabilities for creating and maintaining large complex data sets [13-15]. The question of whether to implement these languages on a proposed aerospace computing system deserves consideration. Their use can relieve programmers of the responsibility of developing algorithms to manipulate large data files. For example, on the space station the programs collecting data for the stellar and solar astronomy experiments must work with millions of bits of data per day. A language, such as SNOBOL or FILLIP, would allow scientists to easily generate data manipulation algorithms and devote more time to the analysis of their data.

14-6.1 SNOBOL

SNOBOL is a string manipulation language developed by the Bell Telephone Laboratories. The basic data structure is a data string, and the language's three basic operations are:

1.) the creation of strings,

2.) the examination of the contents of strings, and

3.) the alteration of strings depending upon their contents.

14-6.1.1 Examples of String Manipulation within SNOBOL

1.) String Formation. A string named QUOTE is formed with the following statement: QUOTE = "IS THIS A DAGGER I SEE BEFORE ME:".

2.) Pattern Matching. The contents of QUOTE can be examined for a given substring. By coding QUOTE "DAGGER", the contents of QUOTE is scanned for the substring "DAGGER".

3.) Pattern Replacement. The substring "ME" in QUOTE can be replaced by the substring "US" by coding QUOTE "ME" = "US".

In addition to these operations, provisions for conditional string operations also exist within the language.

These examples demonstrate that the capabilities built into SNOBOL yield a programming system that can be very useful in a computer that processes large files of string data.

14-6.2 FILLIP

FILLIP, developed by the MIT Draper Laboratory, is a language capable of manipulating more complex data organizations than SNOBOL. In fact, programmers can design the format of their own list structures in this language. Each of these structures can hold more than eight million bytes of data. The language consists of sets of commands specifying list structures and manipulation operations.

The list structures are basically pointer data and, in fact, can represent programs or data. In the former case they are structured by an assembler and are identical in size and organization. Moreover, programmers may treat these lists as data, enabling programs to modify themselves during execution. In the latter case the list structures are formatted by the programmer with each block containing up to fourteen fields. The number of fields and the type of data in each field is programmer defined. The ability to combine various data types in a block or various block types in a list yields the high flexibility of the language.

FILLIP commands can be very simple, such as combining data elements or reading input, or they can be very complex, such as searching a tree. In general, each command consists of an operator and one or more operands. The combination of flexible data structures and powerful commands makes FILLIP an interesting language to consider when building a computing system.

14-6.3 Extensible Languages

FILLIP falls into the category of extensible programming languages; i.e., languages containing facilities for self-extension. This category provides a contrast to languages, such as PL/1, containing all facilities for all users. The extensible approach has several advantages:

1.) a variety of facilities can be provided at lower overhead in terms of compiler size and speed; and

2.) programmers can define their own data structures unconstrained by the fixed type of data structures provided by nonextensible languages.

Cheatham [16] provides another example of an extensible language, BASEL. This language provides the additional ability of extending the meaning of existing operators and defining new ones. For example, the concatenation of string data with the square root of an integer can be defined as an operator. This ability enables programmers to more efficiently manipulate the wide class of data structures that they can define.

To the best of our knowledge no aerospace compiler has yet been developed with extensible definition facilities. When developing an aerospace computer system, the need for such a compiler must be evaluated in terms of the need to handle large varieties of data organizations.

C.5

REFERENCES

1. Chapin, N., "A Deeper Look at Data", (Proc ACM National Conf., 1968), pp. 631-638.

2. Mealy, G., "Another Look at Data", (Proc FJCC, 1967), pp. 525-534.

3. Knuth, D., The Art of Computer Programming, Vol. I: Fundamental Algorithms, (Addison-Wesley, Reading, Mass., 1968).

4. Weizenbaum, J., "Symmetric List Processor", (CACM, 6(9), September, 1963), pp. 524-544.

5. Madnick, S., "String Processing Techniques", (CACM, 10(7), July, 1967), pp. 420-424.

6. McCarthy, J., et al, The Lisp 1.5 Programmers Manual, (MIT Press, Cambridge, Mass., 1963).

7. Donovan, J., "Notes for Course 6.251, Digital Computer Programming Systems", (MIT, Dept. of Electrical Engineering, 1969).

8. Price, C.E., "Table Lookup Techniques", (Comp. Sur., 3(2), June, 1971), pp. 49-65.

9. Martin, W.A., "Sorting", (Comp. Sur., 3(4), December, 1971), pp. 147-174 .

10. IBM Corp. IBM System/360 Operating System Concepts and Facilities, (#C28-6535-5, New York, July, 1969).

11. IBM Corp., IBM System/360 Operating System Supervisor and Data Management Services, (#GC28-6646-2, New York, November, 1968).

12. Saponaro, J., Advanced Software Techniques for Data Management Systems, Vol I: Study of Software Aspects of the Phase B Space Shuttle Avionics System, (Intermetrics Inc., Cambridge, Mass., February, 1972), prepared under contract NAS 9-11778.

13. MIT Draper Laboratory, Users Guide to FILLIP, (Cambridge, Mass., 1969).

14. Farber, D.J., et al, "SNOBOL, A String Manipulation Language", (JACM, 11(2), January, 1964), pp. 21-30.

15. Sammet, J., Programming Languages: History and Fundamentals, (Prentice-Hall, Inc., New Jersey, 1969).

16. Cheatham, T., et al, "On the Basis for ELF - An Extensible Language Facility", (Proc FJCC, 1968), pp. 937-948.

CHAPTER 15

THE ROLE OF HIGHER ORDER LANGUAGE PROGRAMMING IN AEROSPACE COMPUTERS

15-1.0 Introduction

The use of a higher order programming language (HOL) is currently increasing in the development of advanced aerospace software systems. There are increasing recommendations for a HOL over the more typical machine language approach because of the expected benefits of lowered software production costs, and improved management control during long term maintenance, which are traditional problems associated with any large aerospace software effort. The principal criticisms of the HOL approach that still remain are based upon the inefficiencies in code generation with its increased memory requirements, the increased execution time introduced by the HOL compiler, and the lack of experience in utilizing this approach in previous aerospace applications. Although considerable interest has been demonstrated by the Air Force and other governmental agencies in supporting the design and development of higher order languages for programming aerospace computers, there has been, to date, no wide spread application of them in actual practice.

We believe that a general purpose procedure oriented higher order programming language should be used in the development of advanced flight software. It will be a significant step toward a more orderly and controlled software production effort, toward a useful analytical tool for the designer, and toward a convenient straightforward technqiue for the programmer. Furthermore, it will be an essential ingredient in the effective production of highly reliable flight software.

The aerospace software industry as well as other governmental agencies are devoting a great deal of attention to the development of common higher order programming languages for use in such applications. In the 1973 to 1980 time frame of the Space Shuttle, the use of programming languages will most likely become commonplace for aerospace computers of that generation just as they are with the large third generation ground based computer systems of today. Consequently, they should be included in the planning of a major space project of the 70's such as the Shuttle.

We recognize that a HOL approach may not be applicable or cost effectively applied to all aerospace computer systems, particularly small dedicated systems. However, the size and complexity of some flight software posed for the future appears to be of sufficient magnitude to effectively apply the use of a HOL.

In this chapter we will determine the role that higher level compiler languages should have in programming the flight computers for future space missions. This chapter discusses those features of the language compiler which will aid in structuring and verifying software. Those areas traditionally difficult to code in a HOL, such as system programming, are discussed, as well as the role and interaction of other special languages; e.g., the crew language and the checkout language.

15-2.0 Languages Needed for Advanced Space Flights

A distinction must be made between the classes of languages needed. For purposes of this chapter, only those languages utilized within the onboard data management computer system are considered. There are, of course, others which will be used in conjunction with other facilities involved with the space flight; e.g., those for test and ground checkout operations, simulation facilities, and other computer operations. An aim of this study was to distinguish between those languages used to control and operate the computer system onboard the vehicle, and those used to develop the software for the onboard computer system. Both are referred to as languages but will be distinguished as the crew languages and the software development language.

15-2.1 Role of the Crew Language

Pilots or other crew members will require a language to communicate with the computer system. They must be able to insert information, control the flow of processing, and receive information from the computer. This language will be referred to as the crew language (CL). The CL will depend to a great extent on the capabilities of the display and control software system. A CRT type display system with an alphanumeric keyboard input is most likely for the avionics system. The syntactic structure of the CL can range from simple numeric function control, as was used in Apollo, to English language statement commands entered through the alpha keyboard. Alphanumeric and graphical outputs will be used for communication from the computer to the crew.

The Apollo Guidance Computer display and control system transmitted commands and requests with a limited vocabulary of 99 nouns and 99 verbs. To command the computer the astronaut depressed the verb (operator) key followed by two decimal digits, and then the noun (operand) key also followed by two decimal digits. Then when the function key was depressed the computer began to take action on the request. For example, verb 16 noun 20 meant display and monitor spacecraft attitude. Verb 16 meant

"display and monitor" (continuously update), and noun 20 identified what to display; in this case, spacecraft attitude. Moreover, major mission programs were selected by verb 37 with a program number identified by the noun.

This type of crew language has a disadvantage in that the operator must learn the coded list of nouns and verbs and the operational procedures associated with using them. However, once learned, it is very efficient.

English language commands for a CL would consist of a finite set of keywords and elements defined with syntactic properties which would be entered by the crew. They would be decoded and translated by display and control software in the flight computer. For example, an on-line control language is defined as part of the breadboard fault tolerant data management systen at NASA, Houston. It is used in conjunction with a checkout and data management language and allows the systems operator to exercise manual control over the system while it is operating. It includes English language text entered through a keyboard which enables it to initiate, control and display information while the software is executing. Typical commands are DISPLAY, LOAD, and CALL.

Other general purpose languages of this type have been designed for the control and operation of software: a) executive job control languages such as the OS 360 operator language and CRBE; b) information retrieval languages such as ADAM and AESOP; and c) test editing languages such as DATATEXT. Although these languages have been tailored for specific needs they contain some basic features needed in a CL such as the ability to retrieve and manipulate data and displays. These languages are of course more flexible, but they are slower to use, and rapid crew interaction with the computer during critical flight phases is needed.

15-2.2 Crew Language Requirements

A summary of typical requirements for a command language is presented below. The ultimate structure capabilities in the on-line command language is a significant factor in the design of the total system and is only presented here to indicate the type of capabilities that are expected.

For the purpose of most avionics systems, pilot commands should be entered from a display and control device, and then decoded and executed on-line. The language should not be compiled by the computer system but rather interpreted as on-line commands before the appropriate action is taken. When the English language statements or numeric coded functions are used, the language should provide the following functional capabilities. It must provide the crew with capability to:

a) select and control software functions for all phases of the mission;

b) control and configure avionics equipment;

c) request display of pertinent mission and trajectory information;

d) enter data pertinent to the mission programs;

e) control system priorities and options;

f) initiate and control checkout of subsystems.

15-2.3 Role of the Software Development Language

As previously stated, it is recommended that a general purpose, procedure oriented, higher order programming language be used in developing the flight computer software. The role of this language will be primarily for the preparation of code for all software in the flight computer. It is also expected that the language can be used for developing other related non-flight software, particularly in the areas of mission planning and design analysis. This fact will facilitate standardization and communication among organizations working on the project.

Associated with the HOL will be a compiler with a machine independent syntax analyzer and machine dependent code generators for several computers including the flight computer, development computer and others as applicable. The requirements for such a programming language have been derived and are documented [1]. The language should be capable of supporting the programming of all flight software applications: navigation, guidance, control, data management, onboard checkout systems monitoring, communications, displays and controls.

3.0 Justification for Using a Higher Order Programming Language

In the past, manned space flight computers have been special purpose machines performing tasks, principally for guidance and control. The computer was provided with a restricted instruction set, small working memories, no secondary storage capability, and established interfaces to a limited number of output devices. For the most part, programming was accomplished in basic machine language.

The architecture of aerospace computers is now maturing to a close functional similarity to ground based computers. General registers, modular word lengths, and larger memories are already in evidence. Years of initial programming and making programming changes are becoming more important as these computers assume multipurpose use. The use of higher order programming languages which had practically no utilization in the aerospace community in the past, can now be reasonably considered. The lowering of costs associated with memory and hardware in the aerospace computers has changed tradeoff factors. In addition, the increased computational tasks required in the manned space environment have required use of larger, more general purpose computer systems and corresponding software to support them.

Flight computer software developments will certainly continue to suffer schedule pressures. In spite of careful planning, the software effort will often be disrupted by additional requirements to perform functions that were inadequately specified at the outset.

Programming languages have been effectively used in large scale ground based military systems. There are a number of standard arguments in favor of using a higher order language approach.

a) Ease of Communication with the program

   1) The program becomes self-documenting, and therefore reduces the cost of and need for separate documentation at different levels of management (e.g., mission definition, analysis, program specification).

   2) In any large project, the problems of maintainability are aggravated by the inevitable turnover of personnel. Not only must different people be able to maintain the program, but they must also be able to easily modify, add, and redesign sections of the software.

b) The HOL is chosen because it is oriented to the problem being solved and uses languages more natural to the programmer. The concise formulation of the problem is therefore enabled. This leads to:

   1) fewer errors due to conceptual difficulties and different ways of stating a problem;

   2) shortened program design and development time.

c) The programmers need be less concerned with the following traditional machine features and problems:

   1) scaling and precision problems,

   2) base register allocations,

   3) general register considerations,

   4) initialization problems, particularly in loops,

   5) data protection.

d) The HOL aids program transferability from one machine to another. It eases debugging and reduces checkout problems due to problem oriented modularity and separation from hardware.

e) Carey and Sturm [2] present some interesting facts concerning the costs of existing space software and the projected cost savings of a compiler for aerospace programming. In particular they are concerned with the compiler. The following information is extracted from the above reference to indicate the software cost for aerospace missions.

1) The cost of software for manned space missions is two to four times the hardware cost.

2) The Apollo Saturn V's Instrument Unit software was produced at a rate of 2.5 instructions per man-day.

3) As much as 1-2 months was needed to make a 500-1000 instruction change in the Titan III computer.

4) Software checkout is very expensive and not perfect. A single error in a 2000 instruction space program might require 50-100 validation runs on a simulated ground-based machine. Extrapolation to a 25,000 insturction program indicates 1000 to 1200 runs.

5) Typically 100 instructions in new unvalidated machine code written by a senior programmer may contain 3-8 errors. Carey and Sturm estimate up to 70% of these errors can be avoided by the use of a compiler.

6) By hand, machine code typically is produced at a rate of 270-350 instructions per man-month. With a compiler, 500-540 instructions per man-month are possible.

7) Writing a JOVIAL compiler for an IBM 4 Pi computer would cost between $300,000 and $500,000.

15-3.1 Higher Order Programming Language Experience

In the past several years there has been an effort to develop higher order procedure oriented programming languages for use in spaceborne software development efforts. Among those specifically aimed at spaceborne programming are SPL (Space Programming Language) [3] developed by the Air Force under the sponsorship of the Space and Missile Systems Organization (SAMSO); CLASP (Computer Language for Aeronautics and Space Programming) [4] developed under contract to NASA Electronics Research Center, Cambridge, and the HAL language developed by Intermetrics, Inc. under contract to NASA MSC, Houston [5].

Other military agencies have similar efforts to develop such programming languages. The Army has funded a survey to determine the most appropriate procedure oriented language for its TAC-FIRE system and selected a subset of PL/1 designated as TACPOL for the job [6]. The Navy utilizes a programming language termed CMS/2 for the development of software for shipboard and airborne applications. In addition, the Navy is pursuing development of an advanced programming language based on CMS/2 for the advanced avionics digital computer system. This language, designated CMS/3, will be a problem oriented language which will express avionic missions and requirements in terms which are pertinent to a commanding officer.

CLASP and SPL MK2 are primarily directed at small fixed point aerospace computers. Heavy emphasis is placed on code optimization, scaling operations, and limited data manipulation. SPL MK4 and HAL encompass more general purpose features applicable to the wide variety of aerospace programming tasks.

15-4.0 Single Compiler Approach

We believe that a single compiler should be used for generating code for the flight computer. To assist in software verification all code generated for the flight computer should be subjected to standardized automatic checking within the compiler. The system specification, design, documentation, and verification are all built around the unified idea: the HOL.

It is reasonable to assume that the statements provided within applicable programming languages do provide most of the capabilities necessary for the flight application. If, however, a separate special purpose language is necessary, the proposed solution is to express source language statements in the general purpose higher order language. For example, a checkout language becomes a special application which is "grafted" onto the general language at a higher level. It appears as a collection of procedures and subroutines to the compiler.

This approach, however, does not necessarily bar the use of other languages. Rather, it forces others to link at either a high level, by producing outputs which are the source languages for general purpose programming languages, or at a low level, by accepting the standardized operating procedures and conventions established for the general purpose programming language. Moreover, it recognizes that there may be a need for programs to be prepared using statements tailored to a specific application. At a high level such applications are subsystem checkout or hardware interfacing; at a low level, systems programming. However, each set of statements is directed into the single compiler system to facilitate standardization and commonality of checks which are performed on the software during compilation. This standardization, not unlike that experienced by other industries, will help to produce a higher quality, more reliable software product.

Other options are also available to extend the general purpose programming language to meet these needs. For example, through macros the language can be extended to incorporate special features for certain problem applications.

15-4.1 Systems Programming

Generally there is a small section of coding which is difficult to accomplish in the higher order language. This involves machine dependent coding, such as I/O, address constants, machine registers. Usually, the basic machine language is used for these functions, but more recently system implementation languages have come into usage [7,8]. The justification for the special treatment is based upon:

a) the need for efficiency, and

b) the need to get at special registers, I/O channels, and absolute memory locations.

While the efficiency question is often nothing but a hollow fear, there is no doubt that at some point the coding must come to grips with the actual machine that it will run on. However, the number of places where an I/O channel needs to be directly addressed is normally minimal. I/O requests should generally be funnelled through well-defined localized areas in controlled subroutines. In any case, the need to do system programming and machine dependent operations is recognized.

On the other hand, the need for a system language could be minimized or totally eliminated if the computer were designed to go with the language and to execute its constructs directly and efficiently. It is then unnecessary to operate in a "lower level" language since there are no machine dependent features outside the scope of the language. Additionally, all application programs written in the higher language are executed far more efficiently both in terms of the speed and especially the core size they require. Burroughs has been structuring its computers to higher order languages for many years. When a machine is constructed in this fashion, it is easy to efficiently accomplish system programming tasks. Burroughs writes its operating system (ESP), its scheduler, and all its compilers in extended ALGOL, the language its computer is designed around. In fact, the computer does not have an assembly language. Since the computer is designed around a higher order language, there are no addressable special registers to be dealt with by the programmers. There are special registers, of course, but they are automatically updated by the hardware using higher order language instructions. In addition, the computer is stack oriented, which makes it easier for a higher order language compiler to generate efficiently executed code for it.

15-4.1.1 Approach to Systems Programming

If a currently off-the-shelf computer is selected for flight application, then some degree of machine dependent coding will be required. There are two ways that this might be accomplished. The first approach is to extend the scope of the higher order language to include more low level features even though they might be machine specific. However, the ultimate in direct, hands-on, programmer control is the capability to switch from compiler code into direct or in-line machine language. There are several drawbacks to this approach.

1) This kind of capability jeopardizes program integrity. Once address constants, pointers, and register manipulations are available to the programmer, the possibilities for creating errors is significant. The entire structure that was so carefully contrived within the compiler to ensure program standardization and reliability can easily be circumvented. The introduction of such hazardous programming practices can hardly enhance program reliability.

2) Readability and understandability goals can be jeopardized when obscure machine dependent code appears with the listing. In-line basic assembly language code is particularly unfathomable and obfuscates the meaning of entire sections. These are fundamental reasons for using higher order languages.

Neither of the above two drawbacks would be of so much concern if their use could be confined to areas where it was essential. However, even if sensible groundrules for their use and control were established, it is a virtual certainty that nearly every programmer will advance persuasive arguments as to why his task is special and needs to use machine language coding to produce highly tuned efficient code.

Another approach is to keep all low level language capability, such as options for direct machine code, out of the general purpose language. When the need arises for a task or procedure to be programmed that cannot be accomplished in the regular language, it is assigned to a special implementation group that programs it in another language, usually the assembly language for the specific flight computer. These experts tailor the code so that it is compatible with the higher order language environment that exists in the running computer, and conforms to the accepted standards and conventions, while meeting its functional specifications. Thus, the usage of this powerful but hazardous capability is isolated and controlled. Applications programmers must either accomplish coding in the higher order language, or else it is developed by a special group after interfaces and specifications have been negotiated and defined. This seems superficially to be attractive but has two drawbacks, besides the obvious one of dependence on "experts".

1) It isolates the low level activity to machine language subroutines which are not readily visible or easily understood even when located.

2) It is still quite possible for the programmer to engage in a great deal of "trickery". He can, for example, call an assembly language subroutine that returns a variable purported to an integer but which is actually a memory address value computed in the subroutine. It is then arithmetically manipulated and used as an index in fetching other data. The achieved effect is a program that superficially accomplishes one thing, but when examined closely, is doing something entirely different. This sort of "trickery" is commonplace in Fortran usage of assembly language coding.

The proposed solution is basically to define a selected subset of the programming language with added features to improve its deficiencies. The proposal is that there be established a special language to accomplish low level and machine dependent tasks. But rather than use the completely separate assembly language, it is proposed that this low level language be incorporated and integrated into the main language compiler as a restricted subset of the language. That is, those given access rights to the "lower level" language can use the special statements and data types, and also freely intermix these with the higher level language statements. All are compiled together so that standard interfacing and data type checking is performed by the compiler. This effectively prohibits the "trickery" of (2) above. In addition, it is possible to intermingle both types of language statement when it is natural to do so. This removes the restriction of the forced and sometimes artificial dichotomy objected to in (1).

This approach should yield a program listing that is more readable and understandable even when computer specific. Applications programmers are in general, prohibited by the compiler from using these low level, relatively unsafe statements. Their use is granted to a select few who have the authority of the project manager. For the purpose of ease in use, it is recommended that these lower level language routines be available not only as callable procedures and subroutines but also as in-line parameterized macros, or the equivalent. This provides a convenient method for using commonly required low level functions in a carefully controlled manner.

The intent of this somewhat cumbersome and laborious process should be made perfectly clear. It is not the intent to put obstacles in the paths of applications programmers or to thwart their efforts to get the job done. It is proposed only as an additional technique to assist in the production and maintenance of quality flight software of high integrity and high reliability. This goal is accomplished by insisting on conformance to a highly structured and controlled environment. These constraints are not meant to hamper the programming effort but to place sensible limitations and bounds so that the overall result is of uniform high quality.

15-5.0 Advantages of the HOL and Compiler to Software Modularity

The benefits derived from modularizing the static software structure and the automatic checking features offered by the compiler will be a significant contribution to high quality software. This section discusses some of the advantages which result from using the HOL and compiler.

15-5.1 Apollo Experience

In a sense, the primary Apollo computational facility was concentrated in a centralized data management system - the Apollo Guidance Computer (AGC). This single computer was responsible for guidance (i.e., steering), automatic control, navigation, I/O processing, (e.g., radar, IMU, optics, engines, keyboard, etc.), hardware compensation (e.g., for gyro and accelerometer inaccuracies), and a set of miscellaneous tasks including self-check, system test (onboard and preflight), crew communications, status monitoring, and up- and down-link telemetry.

15-5.2 Software Modularity

Future aerospace data management tasks promise to be more extensive and complex than that of Apollo. In addition, the reconfiguration logic associated with high reliability systems presents software challenges not previously encountered. In order to accommodate all programs in a single computer, or substantial portions in distributed computers, it is imperative that systems be introduced which effectively isolate programs from one another except at controlled and visible interfaces. This isolation should prevent the unrestricted access of common data and the arbitrary transfer of control to any location in the instruction logic.

Software techniques now exist which allow many programs, designed to do various related and unrelated functions, to be written and incorporated in a single computer without conflict. The apprenhension that the future flight DMS might be a bigger and more complicated Apollo-type effort with even more erasable conflicts and control interferences is relieved by the introduction of effective software modularity through language and compiler. The following features have been incorporated in the HAL compiler and provide significant capabilities toward handling a large, complex, cooperative programming effort.

15-5.2.1 Independent Compilation and the Compool

Figure 15.1 illustrates a suggested program organization. The individual numbered programs represent independently compilable units. Thus, for example, Program #1 might be rendezvous navigation, Program #2- autopilots, Program #3 - environmental system monitoring. Independent compilation permits divergent groups to contribute to the whole and yet progress at varied paces with measures of local management control.

The communication between programs is provided through a common data pool (compool). The compool is a centrally defined and centrally maintained group of definitions. Variable names and location labels in the compool are potentially known to all programs and, in fact, provide the only means of communication between programs.

The computers many tasks can be apportioned into programs which are managerially or functionally convenient. Information interfaces among programs then become visible at the compool level and can be monitored with respect to definition and usage by a central authority.

Note that, except for the necessity of communication among programs, the complete separation (or isolation) of programs within a single computer is commonplace today in a time-sharing environment. That is, to each programmer the machine appears to be a dedicated facility, and the probability of his conflicting with another user is remote.

15-5.2.2 Blocks Structure (Name Scope)

Figure 15.2 defines the nested structure of name scope. For the purposes here, tasks, procedures, and functions may be considered as subroutines (or blocks). Thus, names defined in the compool are potentially known in every program. Names defined at the program-level are potentially known within all included (or nested) sub-routines and so on. For the region in which a name is known any particular name can be declared again in an inner block. Then its scope becomes all the nested blocks within this block. An example may help to illustrate these principles (see Figure 15.3).

Two desirable effects of the scope rules are:

1) Common data must be declared only once at the highest level. This contributes to more direct management control and better visibility.

2) Local variables may be defined within inner blocks and remain unaffected by outisde definitions. For example, a programmer declaring X in procedure CHARLIE (Figure 15.3) need not fear that any other program will overwrite his quantity. That is, this particular X is not addressable from outside this block. In fact, the X in GRAB (Figure 15.3) must refer to different memory cells.

A name scope or block-oriented language means that many programs and subsections of programs (i.e., subroutines) can "live" in the same computer, isolated, and unaware of each other. They are incapable of writing-over or otherwise interfering with variables or locations that are not mutually defined.

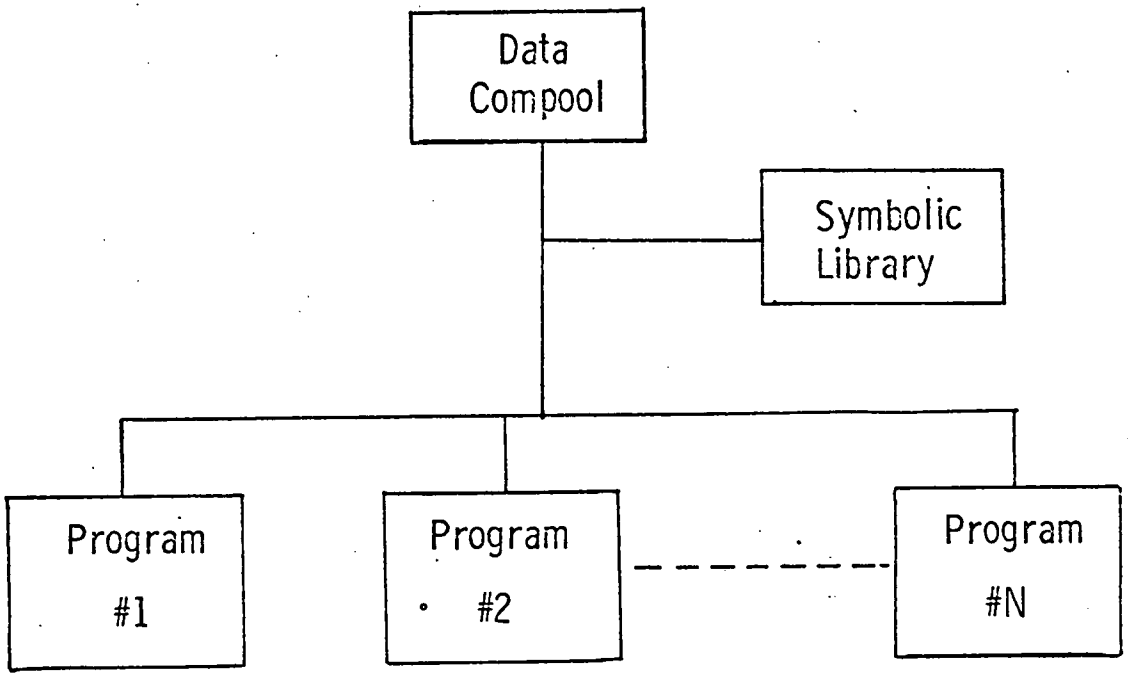
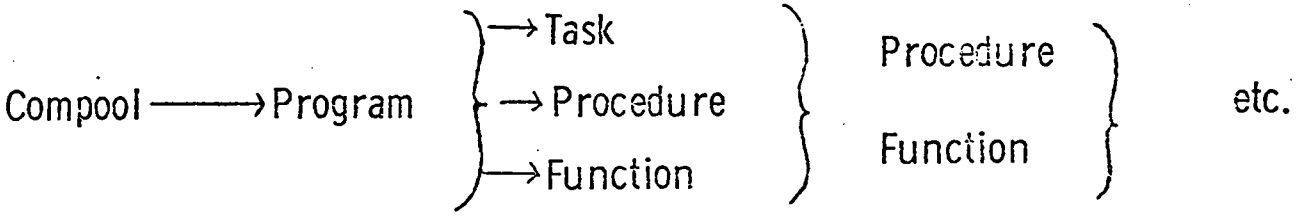


Figure 15.1: Program Organization

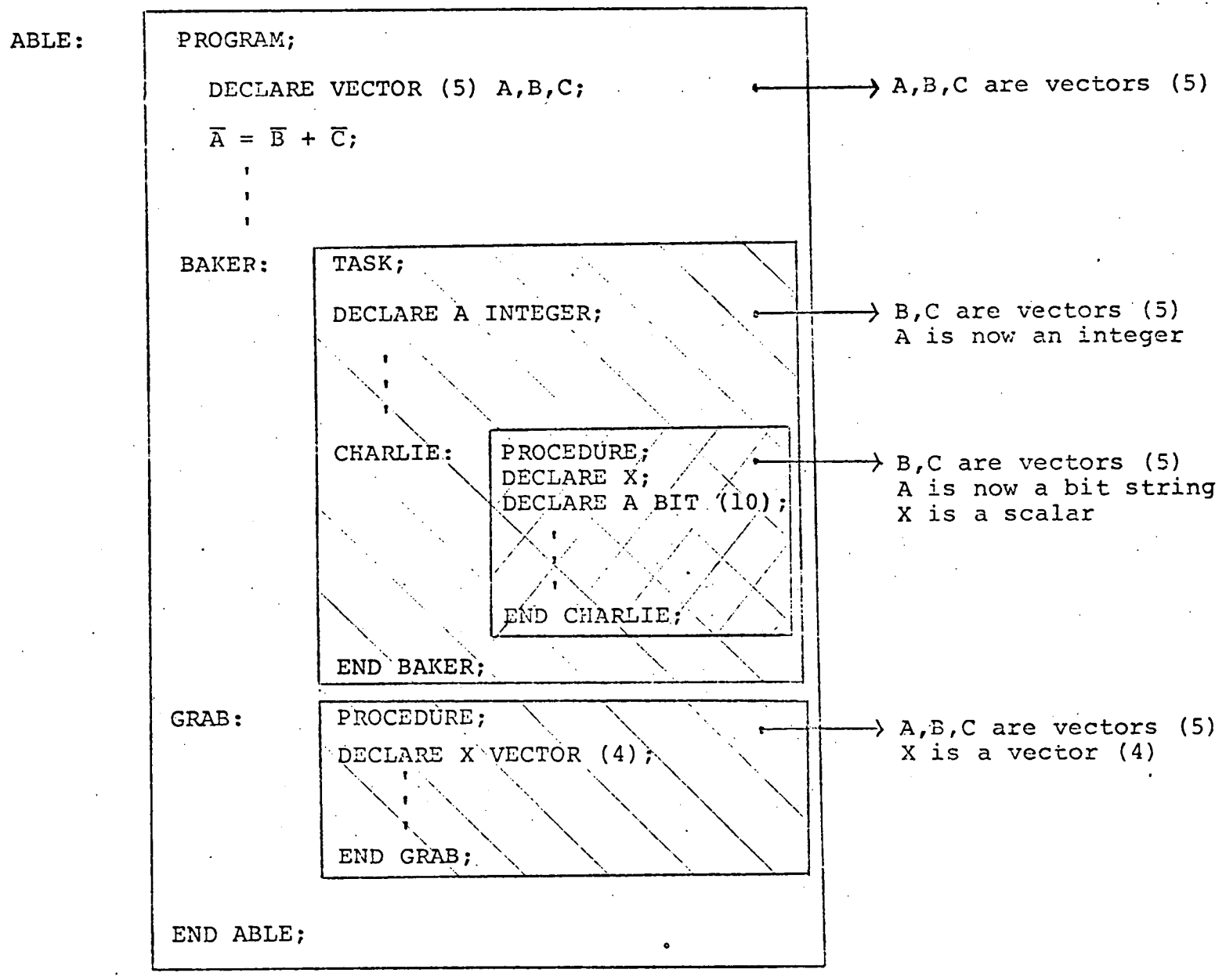
- Scope is the region in which a name is recognized.
- Scopes are defined from the outermost block toward the inner; i.e.,



- Names defined in an inner block are never recognized in an outer block. Inner blocks effectively isolate locally defined variables.

Figure 15.2: Scope of Names

Figure 15.3: Example of Name Scope



15-5.2.3 Control of Shared Data

The erasable memory conflict, along with restart and scaling problems, provided most of the Apollo software anomalies. To illustrate the problems, in a general way, that can arise because of sharing data, consider the examples shown in Figure 15.4.

In both examples TASK B interrupts TASK A during the execution of a statement. The interruption may be caused by a hardware or software interrupt or by a "job swap" based on priority. In either case, the interruption of TASK A causes a conflict in common data usage.

The approach taken, in HAL, towards solving these problems is to confine the read and write accesses of shared variables to identified update blocks. The compiler assigns a locking control variable to each shared variable. The value of "lock" is examined at run-time and only consistent (i.e., safe) accesses are permitted. (See Figure 15.5).

The use of an update block is not a simple solution to the data sharing problem and presumes a sophisticated compiler; and yet the goal is worth the effort. The problem of sharing common data in a real-time flight environment always exists. The Apollo solution was to attempt to arrange memory so that conflicts did not occur. This proved to be a time-consuming process at best, requiring extensive verification with inconclusive results.

For future space flights data sharing will be a necessity regardless of avionics configuration. A unified approach through a compiler, as outlined above, will permit safe operation in multiprogram and even multiprocessor environments.

15-5.2.4 Access Rights

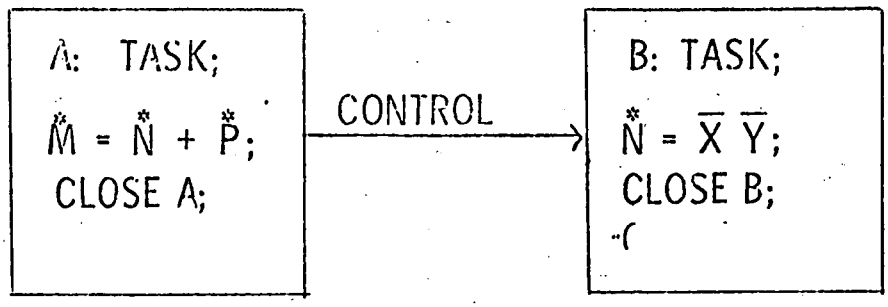
The sharing of compool variables among several programs may be restricted and controlled by the issuance of access rights. These rights are attached to the data declarations within the compool. Each program is identified by number and permitted to access only those variables which have been declared with corresponding identification numbers. An illegal reference to a compool variable will prevent successful compilation of the program. For example, access rights might be employed to allow only those programs comprising guidance and control to address compool variables associated with main and reaction control jet engine performance.

15-5.2.5 Automatic Checking

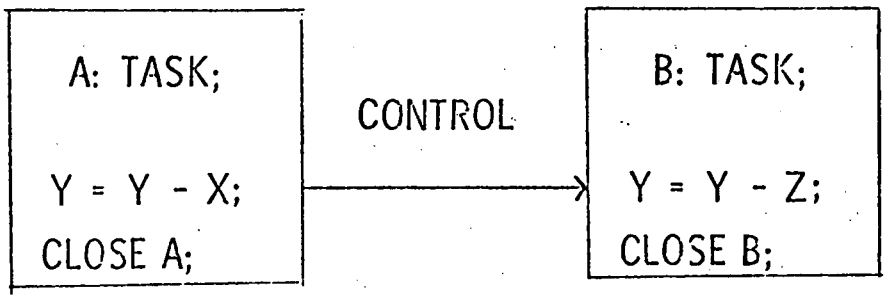
Besides being expressive and enforcing programmer conventions, additional major advantages of a compiler language are the ability to perform extensive checking at compile time and the opportunity to structure and modularize programs. Compile time checking can verify that subroutines are called with proper data; that dimensions (i.e., the units) of variables and constants are consistent; and that array

- PROBLEM IS THE CONFLICT OVER UTILIZATION OF COMMON DATA ELEMENTS BY EXECUTING TASKS.

EXAMPLE 1: READ AND WRITE CONFLICTS



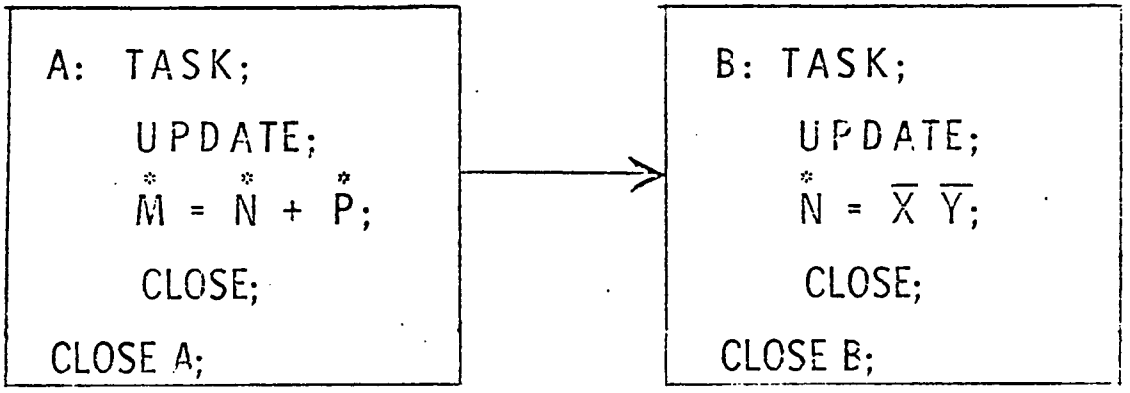
EXAMPLE 2: UPDATE CONFLICTS



NOTES

1. B "INTERRUPTS" A IN BOTH CASES
2. #1 TASK A RESUMES USING OLD AND NEW VALUES FOR N
3. #2 TASK A RESUMES "CLOBBERING" THE VALUE FOR Y SET BY TASK B

Figure 15.4: Background in Problems of Controlled Shared Data



IF N IS DESIGNATED AS A SHARED VARIABLE THEN:

1. UPDATE IN A CAUSES N TO BE 'READ-LOCKED'
2. UPDATE IN B CAUSES N TO BE 'WRITE-LOCKED' AND A COPY MADE OF ALL 'WRITE-LOCKED' VARIABLES
3. CLOSE IN B WILL NOT ALLOW UPDATE OF N IF VARIABLE IS STILL 'READ-LOCKED' (TASK B WILL STALL)
4. POTENTIAL CONFLICT IS AVOIDED

Figure 15.5: Use of Update Block to Avoid Data Conflicts

variables (vectors, matrices, etc.) are not referenced out of range. In addition, the compiler can perform other static cross-checks on the intent of the programmer.

15-5.3 Additional Advantages of the HOL Approach

15-5.3.1 Management

The technical management of the software development faces problems of visibility and control. Design changes, short production times, and pressing operational schedules would demand flexibility in software design and organization. Clearly, an overall management and control plan is required which will define the procedures for developing software design requirements, interface specifications, documentation requirements, testing requirements, change procedures and organizational responsibility. Presumably, a higher order language should provide features which support the software production environment in general. It would be self-documenting to a maximum extent, provide ease in program modification, and provide mechanisms for enforcement of management rules and programmer conventions.

15-5.3.2 An Improvement in Communications

In this context communications are meant to include requirements, specifications, descriptions, all forms of documentation, methods of configuration and change control, management visibility and technical exchanges (written and oral) that must occur among engineers, analysts and programmers. Traditionally, the engineer designs and expresses his algorithms using conventional mathematics, or perhaps Fortran-like statements, and the programmer translates these into his language, usually a basic assembly language appropriate to the particular computer. The programmer must then explain his efforts by using other media, e.g., detailed functional charts, user-guides, or other apparently helpful devices. Unfortunately, in many projects the coding language has isolated the programmers from everyone else associated with the effort. The programmer becomes too busy to learn the physics and objectives of the mission and is too busy to explain to others how the code works. He, therefore, is forced to assume an increasing share of the total responsibility. Small indispensable groups of experts direct and shape the code and become the overworked "authorities".

A properly designed higher order language could be a useful analytical tool for the designer and a convenient, straightforward technique for the programmer. The specific format of the language should promote the ability to read, write, and understand the language quickly and easily, and to document results in a clear and unambiguous manner.

15-5.3.3 Prevention of Errors by Readability of Code

A higher order language can be instrumental in preventing errors. The truth of this assertion can be seen simply by comparing the probability of error when using assembler versus compiler coding techniques. A programmer using a higher order compiler language can express his problem in a problem-oriented manner. For example:

$$\overline{W} = \overset{*}{M}\ \overline{V}$$

indicates that the product of the matrix M and the vector V be assigned to the vector W. The programmer does not have to express how he wants the machine to handle his statement; e.g., where the variables are in memory, what base or index registers to use, what basic machine instructions to employ, or how to set up and call an assembly language subroutine. The single statement above will generate many assembly language instructions automatically. If these had to be hand-coded, the probability of programmer error would greatly increase. It also is an aid to visual inspection or "eyeballing" of the code for correctness. The HOL enforces standard code and discourages a "handcrafting" that invariably leads to subtle errors.

15-5.4 Summary

In considering methods of implementing aerospace software, hardware and software techniques are available to insure program modularity. For a centralized avionics configuration this means that effective isolation can be insured among programs performing different functions and that the interferences and potential memory conflicts of Apollo need not occur. For a decentralized system the enforced hardware separation of the several functional computers adds a measure of safety in that, assuredly, a program operating in one cannot cause a memory conflict with a program operating in another. However, even in this case, the computational load in a single computer; e.g., guidance and control, can be sizable (perhaps 40% of the total) and modular programming techniques and aids should be utilized. Once these techniques and aids have been provided, it makes little difference from a programming point of view whether the total software is centralized or decentralized.

15-6.0 Checkout Languages

Several higher level languages have recently been developed for purposes of system checkout. Examples of these are GOAL and ATLAS. These languages have, however, been primarily directed at integrated ground checkout and subsystem test operations.

It is reasonable to assume that checkout software for the flight computer can be developed using the general purpose HOL. It can be operated and controlled interactively by the crew using the crew language as any other flight software. The crew language may require a special subset to accommodate all memory options and

control functions required to perform checkout and maintenance. The capability to select diagnostic and subsystem checkout programs and to control their options must be provided.

In most aerospace software environments software will be assembled and loaded prior to the flight. No on-line compiling of software and program generation is assumed. Accordingly checkout software must be constructed to allow modes or crew options for accommodating the variety of fault isolation and diagnostic requests.

In the event that the general purpose HOL cannot be extended to satisfy the needs of this type of software, the single compiler approach discussed in Section 15-4.0 would be very useful.

15-7.0 HOL Compiler Implementation

15-7.1 Compiler Problem

The chief complaint about higher order languages has been that HOL compilers are inefficient generators of machine language code, in terms of both quantity of code and execution time. Secondary factors are that compiler design is a very significant effort if it has to be considered in line with the operational software task and that the indirect and unclear relationship between a program written in the HOL and the resulting machine code impedes the correction of program errors discovered at the machine language level. The reason for the compiled code's stigma of inefficiency is that compiler systems have not evolved with the conservation of machine resources as a primary design criterion but have concentrated on isolating the programmer from having to worry about the machine characteristics. Since it is difficult, perhaps even impossible, to serve both the programmer and the machine interfaces equally well within the mechanism of a single translation, the tendency has always been to incur object code inefficiencies rather than decrease the programming effectiveness.

It should be noted, however, that with the continual decrease in hardware costs and corresponding increases in cost for software, the conservation of memory may no longer be the prime objection to a HOL and compiler. Certainly, if the software is sized with higher order language considerations initially, and if a secondary memory system is used for loading mission phase programs to lessen the impact of operating memory size, the software cost savings of the HOL approach may well exceed the increased hardware costs.

The penalty of an increased memory capacity, however, will always be considered when the use of a HOL is contemplated. A completely written compiler can be almost as efficient as an average programmer. The M.I.T. experience with PL/1 on MULTICS has demonstrated this [8]. But compared to the highly efficient machine code customarily produced (at considerable cost) for military aerospace computers, a compiler may be less economical.

Since compiler efficiency is still an important consideration, the purpose of this section is to describe some possible approaches to improve compiler efficiency. Higher order language machines, interpreters, microprogramming, and high speed memories are all approaches that aid in achieving more efficient code generation if it is required.

15-7.2 Approaches to Efficient Code Generation

An approach that circumvents the drawbacks of compilers is the construction of special higher order language machines that decode and execute the HOL operations directly within the logic of the hardware. Although a number of these has been reported in the literature [9-12], it is not a widely applied principle.

There is another approach that appears to solve a number of the previously identified problems and whose drawbacks show promise of being eventually diminished by current trends in computer hardware design. It involves the establishment of the program in a coded form intermediate between the HOL and machine language. The translation from the HOL to the intermediate form is accomplished off-line in an operation that can be made much simpler, faster, and cheaper than the traditional compilation of machine code from the HOL. The translation of the intermediate form into machine operations is done at execution time in an interpretive fashion. This concept appears to offer the following benefits.

1) For a given application the computer memory requirements can be made less by up to a factor of two compared with the direct translation compiler approach.

2) It allows the choice of HOL to be uncoupled to a great extent from the problem of satisfying the machine characteristics, and it is unaffected by consideration of machine to machine transferability.

3) The intermediate form of code provides a very convenient, visible "stepping stone" between the machine and the HOL, which would greatly assist the problems of debugging.

4) Current trends of computer design offer the possibilities of higher performance using this approach than can be obtained by hand-crafted assembly language programming and offer a reduction in the amount of machine-dependent coding that is required whenever a new computer is being considered.

15-7.2.1 The Concept of an Intermediate Language

It is feasible to formulate a medium which lies intermediate between the problem and the machine, which enables a concise enough description of the problem's characteristics, and yet accommodates sufficiently to be limited word format and instruction repertoire of the computer. Such a medium would possess a high information content

and would be storable in the computer's memory. The basic concept, however, is the translation of the operational program (expressed in a language highly appropriate to the problem it seeks to solve), into a compact intermediate form (or language) which, when stored in the computer memory, maximizes the density of the information.

For the condensed information of the intermediate language to perform any operation, its basic instructions must be decoded and executed by some mechanism within the computer. This process involves a number of logical operations which will consume a certain amount of time. For an individual instruction, it need not be changed (unless, of course, the instruction is modified).

15-7.2.2 Characteristics of Compact Form

The structure and notation of the compact form of the program must be defined in a formal code or language. A basic set of more elementary instructions can always be derived to mechanize all the basic HOL statements [9]. The proposed intermediate language (IML) will be based on this set of elementary instructions. The processing of the HOL into the IML becomes a more direct, less complicated, faster operation than compilation into machine code. This is attributable in part to the fact that a good deal of the decoding task is done at execution time, relieving the translator of some of the burden. Furthermore, since the translation is less difficult, it becomes natural to contemplate fairly sophisticated and universal HOLs like HAL, SPL, or PL/1 for the application programming.

15-7.3 Implementation Factors

An important constraint in the design of the IML is the method of decoding and execution by the machine. The more concise and compacted the language, the higher becomes the potential economy in memory. However, the full impact of its advantages will be realized when the current trends in microprogramming achieve operational status. The IML design must remain cognizant of this trend. Experience with and acceptance of the language today will then constitute a firm foundation which will provide continuity into future programming.

15-7.3.1 Software Interpreter

The majority of today's aerospace computers possess a fixed internal logic which defines their basic operating modes. The IML program would exist in memory in encoded form produced by the machine section of the HOL-to-IML translator. The decoding and execution of the individual instructions of the IML program must be performed by the standard instruction set of the computer under the direction of an interpreting routine written in the assembly language of the machine. Instruction by instruction software interpreters have been used in aerospace applications for the purpose of storage efficiency [13],

but they are more usually employed in commercial applications where their ability to decode and execute individual statements can be used to advantage in on-line programming and debugging.

The usual complaint against a software interpreter, which is well earned, is that because it repetitively performs the redundant operations of decoding and dispatching for each statement, it is considerably slower than the object code of a compiler, which is analyzed and translated prior to execution. However, the example of interpretive programming in the case of the Apollo Guidance Computer demonstrates that the penalty is quite acceptable. An equivalent instruction, for example a double precision add, was 20-30 times slower in the interpretive mode than in the machine language. Although the computer with its 12 μs cycle time was ten times slower than a typical small machine of today, interpretive routines were used to implement guidance and control loops with periods of less than 1 second. The use of the interpreter enabled 50% more interpretive programs to be accommodated in the memory than if a pure assembly language approach had been taken.

With the higher performance computers available today, it should be possible to do at least as well; and with a more sophisticated interpretive language than was used for Apollo, a much higher ration of IML to assembly language programming should be achievable. With this level of performance less than half as much memory is needed to contain a HOL program translated into the interpretively executed IML than one in machine code generated by a regular 25% inefficient compiler. The cost savings come with all the advantages of comprehensive HOL programming.

15-7.3.2 Hardware Implementation and Use of Microprogramming

The use of special logic circuitry within a computer to assist the interpretation of a higher order problem-oriented language has been reported in the literature. Some of these attempts have mechanized subsets of Fortran directly with specially designed logical hardware [9,11]. Other more promising approaches have applied the concepts of microprogramming. A very relevant example is reported by Weber [10], in which a machine independent interpretive language is decoded by microprogramming on a modified IBM 360/30. The original programming is done in a higher order language, and a relatively short compiler generates an intermediate text or middle language for storage in the machine. The string language interpreter reduces storage, and the microprogramming feature allows special instructions which actually improve the run time over standard assembly language techniques.

It is true that microprogramming brings with it its own problems of language and design. However, a microprogram instruction is generally more powerful than a basic machine instruction. The microprogrammer is given greater scope to optimize the sequence of operations required to decode and execute an IML statement. Once it is set up, the microprogram storage resides in a read-only memory, which is generally capable of higher speeds than main read-write memory. We do not suggest that the technique of microprogramming is without characteristic problems of its own, but for the short fixed logical sequences associated

with decoding a set of IML instructions, it offers a higher efficiency than the software approach and is far more flexible than advanced logic.

15-7.3.3 An Interesting Example

Although the following description of an aerospace programming application is not an example of HOL usage, its significance lies in its conscious attempt to economize on memory requirements. Since this is the central objective of the concept described in this section, and because of the relationship of the techniques, the application will be briefly considered here.

The example in question is the software interpreter [13] used in the Apollo command and lunar module computers: the CMC and LGC. The computer is a 36,000 word 16-bit machine with a 12 microsecond memory cycle time. The requirements placed upon the onboard computer grew with the development of the total program. For the lunar landing mission, Apollo 11, each computer had less than a hundred or so unused memory registers. The coded interpreter implemented 127 double precision arithmetic, vector and matrix operations, and many trigonometric functions. Yet it took less than 1600 16-bit registers of computer memory. The command module program used approximately 16,000 interpretive instruction registers.

If this effort had been done in basic assembly language, it may be presumed that instead of all in-line coding, a number of subroutines would have been written to conserve storage. Some 75% of the interpreter would have to remain as basic language subroutines; i.e., 1200 words. This represents a saving of 400 words. Of the 16,000 words about one/half are instructions and one/half are addresses. The assembly language approach would retain the addresses and would require, on the average, about two instructions for every one interpretive instruction. The net result is that without an interpreter the Apollo computer would have required approximately 8,000 additional words of memory to accomplish the job. This represents a saving of 33% over efficient assembly code.

REFERENCES

1. Intermetrics, Inc., Requirements Analysis for a Manned Spacecraft Programming Language and Compiler, MSC 01845, April 1970.

2. Carey, L., and Sturm, A.A., "Space Software: At the Crossroads", Space/Aeronautics, December 1968.

3. Preliminary Functional Design of the SPLM as Developed and Procured by the Air Force, RFQ F04 701-71-A-0145.

4. Lickly, D.J., Clasp Critique, (Intermetrics, Inc., Cambridge, Mass., April, 1970).

5. Intermetrics, Inc., The Programming Language - HAL, June 1971, MSC - 01846, Cambridge, Mass.

6. Hess, H., and Martin, C., "TACPOL - A Tactical C&C Subset of PL/1", Datamation, 16(4), April 1970, pp. 151-157.

7. Graham, R.M., "Use of High Level Languages for Systems Programming", MIT Project MAC Tech. Memo 13, September 1970.

8. Corbato, F.J., "PL/1 As a Tool for System Programming", MIT Project MAC Memo M378, July 1968.

9. Kerner, H., and Gellman, L., "Memory Reduction Through Higher Level Language Hardware", AIAA Paper No. 69-693, Aerospace Computer Systems Conference, Los Angeles, California, September 8-10, 1969.

10. Weber, H., "A Microprogrammed Implementation of EULER on the IBM System 360 Model 30", CACM, 10(9), September 1967, pp. 549-58.

11. Baskow, T.K., Sasson, A., and Kronfeld, A., "System Design of a FORTRAN Machine", IEEE Trans. Elec. Comp., ED-16(4), August 1967, pp. 485-99.

12. Melbourne, A.H., and Pugmire, J.M., "A Small Computer for the Direct Processing of FORTRAN Statements", Computer Journal, 8(1), April 1965, pp. 24-27.

13. Muntz, C., "Users Guide to the Block II AGC/LGC Interpreter", R-489, MIT Draper Laboratory, Cambridge, Mass., April 1965.

CHAPTER 16

THE DESIGN OF AN ADVANCED AEROSPACE EXECUTIVE SYSTEM

16-1.0 Introduction

As aerospace computers play an ever increasing role in space exploration, an understanding of software systems becomes more and more important for the system designer. At the heart of an aerospace software system is the executive program which controls the execution of the application software on the flight computer.

Describing the architecture of an executive system consists of more than an explanation of how the various parts of the executive software work. It also consists of an explanation of how these parts dynamically interact with each other to extend the power of the host machine. Furthermore, the hardware structure of this machine plays an additional role in executive system design since particular hardware features, such as I/O channel structure, influence the software design. In a sense we may consider the machine together with its executive software to be the full executive system that enables application programs to be executed.

The executive system is responsible for the control of all computing tasks in the real time software environment. Thus, the fundamental features of an aerospace executive system must be based on the requirements of its environment and the application software it controls. Ideally, it should be efficiently tailored to meet the design objectives and operating environment of the total system. In particular it must manage the allocation and utilization of all resources of the system including processor, memory, data bus system, secondary memory, timers, and all other devices connected to the computer. The executive system must be organized such that it simply and efficiently allocates system resources to the computing tasks and provides sufficient general services to application programs to enable them to achieve mission requirements.

In order to make the system flexible, it must be structured such that the executive modules are either self-contained or utilize a standardized set of subroutines. It must be possible to make alterations to these modules without jeopardizing the rest of the executive functions.

In order to make the system simple, it is necessary to prevent application programs, regardless of their complexity, from directly performing system control functions. This fact limits the number of checks and balances necessary in order to assure full system reliability. This does not mean that application programs are denied use of hardware facilities, but rather that the control of such facilities is restricted to one responsible module.

Since the system must support applications which will have real-time inputs and outputs, it will have to be oriented toward being able to guarantee response within some predictable time constraints and yet not be performing supervisory tasks so frequently as to constrict throughput rates, a problem encountered in many highly interactive systems.

This chapter presents a description of the architecture of an aerospace executive system designed for an advanced space mission. It is assumed that the computer on which the executive is executed is a simplex (single processor) system with a multiprogrammed task stream. We will place particular emphasis here upon the design issues which must be resolved when designing an executive system (e.g., how to allocate dynamic memory) and upon identifying the fundamental parts of the executive. A more complete functional description (to the flowchart level) of an executive system designed for Space Shuttle application may be found in [1].

16-2.0 Design Criteria

We may now ask ourselves what overall criteria should we adopt in designing an aerospace executive system. The primary objective or goal usually adopted by most executive system designers is the achievement of an "efficient" executive where efficiency is some measure of throughput. Efficiency may be defined by either the fraction of executive overhead time spent doing nonproductive work or in terms of response time. However, efficiency becomes less important a factor when it leads to a complex design resulting in complex testing and verification of software. Ideally, flight software should not only be tailored to meet operational mission requirements but should be structured to enhance software verification and flexibility to adjust to changing needs. Therefore, the following design criteria are important in evaluating structure. The designer must be able

a) To provide an executive system which will control and allocate resources of the system to satisfy

operational mission requirements (i.e., one that does the job).

b) To establish an executive organization which facilitates verification of application software and reliability of code.

c) To structure an executive enabling flexibility and modularity in incorporation of application software changes over long term maintenance periods.

d) To define simple and well defined application program interfaces to the executive system. It should be structured as a virtual machine to the applications programmer.

e) To develop an executive structure which is both simple and efficient but consistent with other objectives.

16-3.0 General Description of the Executive System

Most aerospace software systems are driven by a real time interrupt around which the software is organized. Similarly we assume our executive is driven by a minor cycle real time interrupt every n msec (where we leave n unspecified). A fixed number of minor cycles constitute a major cycle. Upon occurrence of a minor cycle interrupt the cyclic sequencer is executed.

The cyclic sequencer is an executive task which performs all functions that are characterized by precise timing specifications. It commands all I/O operations done on a periodic basis, supervises execution of all computations to be run on a periodic basis, updates core memory with input received in the last minor cycle, and monitors the status of avionics subsystems. Upon termination of the cyclic sequencer, the dispatcher is called to select a non-cyclic (background) task for execution.

The dispatcher is at the heart of the executive system. It is this executive function that selects tasks for execution on a priority basis. When a task terminates, it returns to the dispatcher, which calls a terminator routine to insure the release of all system resources held by the task.

While an application task is executing, it may request another task to be scheduled for execution by calling the scheduler. Scheduling can be done unconditionally, or on a time basis, or on the occurrence of an event. A function of the scheduler is to put this new task in a state ready for execution. It does so by calling the resource allocator to give the task any resources it may need. Should a resource be unavailable, the task must wait for scheduling until this resource is freed. At this time, the resource can then be assigned to the task, and the task is then ready for execution. It competes for CPU time on

a priority basis with all other tasks in a similar ready state. The dispatcher will choose the highest priority task that is ready for execution and assign the CPU to this task. A task will continue executing until it ends, or until it voluntarily releases the CPU, or until a system event occurs necessitating the CPU being assigned to another task.

At any time during its execution, a task may request I/O operations to be done and may request its own execution be halted until these I/O operations are completed. It is one of the functions of the executive to supervise and schedule all I/O operations. In addition, the executive must supervise error recovery functions. Should a hardware or software error occur, the executive must provide the capability of running a specific recovery routine depending upon the type of error. A system reconfiguration routine might then have to be executed if a piece of hardware is judged faulty. The faulty equipment will then be switched out, and the system will continue operation using standby equipment.

The executive software to perform all the above functions will be organized in modular fashion. We will now identify the necessary modules.

16-3.1 Identification of Executive Program Modules

a) Cyclic sequencer: performs all services done on a minor cycle basis.

b) Scheduler: puts previously inactive task or waiting task in a status ready for execution.

c) Dispatcher: assigns CPU to a task ready for execution.

d) Resource allocator: assigns system resources to tasks.

e) I/O supervisor: dispatches all I/O requests to channels.

f) Machine check supervisor: diagnostic routines executed when hardware error is detected.

g) Reconfiguration routines: brings up standby equipment when active unit is judged faulty.

h) Timer routine: sets hardware timer and signals events based upon elapsed times.

i) Program check supervisor: provides recovery from detectable software errors, such as division by zero.

j) Supervisor service routines: provide supervisor services for application programs; e.g., enable a task to await an event or to free an assigned resource.

16-3.2 Executive Operating Environment

In most aerospace applications the executive is not presented with a random stream of tasks queued upon secondary storage as is OS/360. Instead there is a fixed set of tasks organized on a mission phase basis. Within a particular phase, task throughput is maximized.

In extended and complex missions requiring a large amount of software a mass memory unit (MMU) such as a drum can be used to store program modules until they are needed. Then if core memory must be overlaid with new program modules, they are loaded from secondary storage at the beginning of a new mission phase in order to minimize the use of the mass memory unit. Moreover, since the modules loaded will be known preflight, their loading addressed and relocation constants will be determined at compile time. In other words, fully dynamic loading and binding of program modules need not be supported by the executive. This minimal use of the MMU presents a fixed program environment for the executive system.

16-4.0 Definitions

We must now define some of the terminology which we will use for the remainder of this chapter. So far we have used the word "task" relying for understanding upon our intuitive sense of what a software task is. We will now formally define a task as an executive unit of work which competes for system resources. A task is created dynamically upon execution of the executive's scheduling function. A task is identified and defined by a unique task control block. A task control block (TCB) is a table containing all pertinent control information for a task used by the executive for task management. The TCB is created by the scheduler when it attempts to bring a currently unscheduled program module into the system. Each TCB contains a pointer to a program module into the which the task executes.

A program module is code executable by the system. Program modules are started by the executive and return control to the executive END function upon completion. A program module may be associated with more than one task.

A task may be in one of four task states at any time.

a) Active: The task has been allocated the CPU and is executing.

b) Ready: The task has been assigned all its resources and is ready for execution. It only awaits the CPU.

c) Wait: The task is awaiting the occurrence of some event or events in the system. Such an event may be the release of a resource, an elapsed time, or an I/O interruption.

d) Inactive: The task is not presently known to the scheduler. However, its program module is present in core storage or on a mass memory unit. (Strictly speaking, an inactive task is merely a program module and not a task. A program module is made a task at schedule time, when its TCB is created.)

Our concept of the states of a task is analogous to the MULTICS concept of the states of a process [2,3]. A state transition diagram is shown in Figure 16.1.

16-4.1 Executive Queues

The executive queues are lists used by the executive to associate and control tasks of a similar condition. Task control blocks are linked into lists corresponding to a particular executive queue. A task can only exist in one queue at any instant of time. One possible executive system organization allows four major executive queues:

a) Ready queue: The ready queue is a threaded list whose elements are the TCBs of the tasks ready for execution. These TCBs are organized on a priority basis with the TCBs corresponding to the highest priority tasks occurring at the beginning of the list. An entry is established by the scheduler in the ready queue when a task is brought to the ready state.

b) Wait queue: The wait queue is a threaded list whose elements are the TCBs of the task waiting for the occurrence of some event or events. When all these events or some allowable combination of them have been completed, the task can be put on the ready queue.

c) Time queue: The time queue is a subqueue of the wait queue. The tasks on the time queue are awaiting the occurrence of a timed event. At some multiple of a minor cycle time interval, the executive examines the tasks on this queue to determine if they can be made ready at the present time. If so, those that can are placed on the ready queue.

d) I/O queue: The I/O queue is a subqueue of the wait queue. The tasks on the I/O queue are awaiting the completion of some I/O operation. When the I/O operation completes, a task awaiting it in this queue can now be placed on the ready queue.

16-4.2 Common Data Pool

The COMPOOL is an area of operating memory permanently assigned to data variables shared by tasks. All communication between tasks is done through the compool. Data assigned in the compool remains in

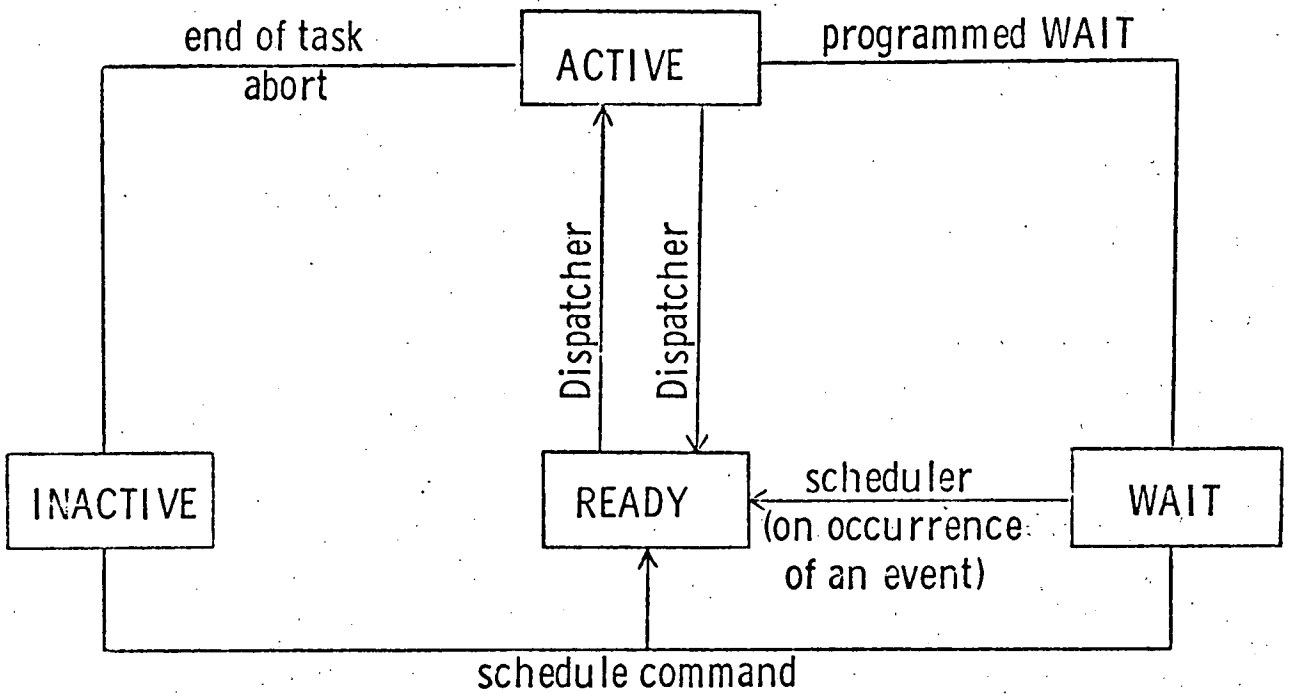


Figure 16.1: Task State Transition Diagram

the system subsequent to a task completion. It is statically assigned as opposed to the dynamic memory assigned to a task for working storage. The compool can be organized in two parts: a mission compool and a phase related compool. The data assigned in the mission portion of the compool is permanently resident. Data assigned to the phase dependent portion of the compool exists only during that phase of the mission. It is overlaid with other phase data during subsequent mission phases. When a mission phase is initiated, the phase is loaded from the secondary memory, and the phase dependent compool is initialized. Data which is to be retained subsequent to a task completion must be included in the compool. All accesses to data in the compool must be coordinated for the application task by the executive system. The executive prevents conflicts in the use of this data by tasks. The SECURE, RELEASE and COPY executive functions can be provided for compool interaction and are discussed in [1].

16-4.3 I/O Request Block

The I/O request block (IORB) is a table of all pertinent control information for the I/O channel to execute an I/O operation.

16-5.0 General Discussion of Executive Design Issues

Now having presented a general description of how the executive operates and having defined some fundamental concepts, we can begin discussion of several executive design issues. These are problems that most designers must face when designing an advanced aerospace executive system. In fact, they are fundamental in nature to most multiprogrammed systems.

16-5.1 Interrupt Handling and Task Dispatching

The interruption of a running program in response to an external signal was introduced into the computer technology to serve two purposes:

a) to provide rapid response time to asynchronous events; and

b) to eliminate the necessity of polling (and its overhead) to discover whether an awaited event has yet occurred.

In single-processor systems, particularly dedicated systems where most or all of the computation is devoted to a single application, the introduction of interrupt-mode computation raises the hazards associated with multiprocessing. At arbitrary times, an interruption can introduce what appears to be a parallel task which is at least conceivably capable of disrupting the progress of the interrupted

task by altering its variables. Thus, methods for masking or inhibiting interruptions were added, and the nature of the functions allowed in interrupt-mode was restricted. Properly and thoroughly applied, these fixes allowed programs to perform properly, although no truly thorough method has been found of proving that the system was actually properly programmed.

There exist therefore, two relevant negative aspects of interruptions: timing response uncertainties, and potential data disruption and conflict. Both can be minimized by causing interrupts to schedule tasks whenever possible, as opposed to performing them. This provision reduces the multiplicity of possible timing situations, since job swapping occurs only at specified intervals.

Accordingly, it is considered desirable to utilize hardware interrupts such that tasks are scheduled and the interrupted task is rapidly resumed. The primary consideration becomes when to dispatch a higher priority task resulting from an interrupt, such that respond time requirements can be satisfied.

When an active task is dispatched into the wait state, another higher priority task is dispatched (made active) from the ready queue. When else does the executive dispatch? The following summarizes various approaches considered.

a) If the executive dispatches at no other time, system response time to high priority tasks cannot be guaranteed since long duration tasks would execute to their end. This appears unacceptable in most missions having task priority levels unless all lengthy tasks were broken down into separate, sufficiently short, independent tasks.

b) The executive can dispatch whenever a task of higher priority than the active task is scheduled. In this case, interruption of the active task will occur at a random point in the coding and a higher priority task given the CPU. This uncertainty can lead to a program verification problem due to its random nature and non-repeatability.

c) Alternatively, a programmer can inhibit dispatching at dangerous points in his program. Tasks of higher priority would be dispatched when permitted. However, this method does not completely solve the verification problem or prevent a higher priority task being delayed from execution for an unacceptable amount of time. By introducing an onboard "watchdog" timer, it is possible to guarantee a maximum time in which dispatching is inhibited. If a programmer exceeds this maximum time in inhibiting dispatching, the CPU is taken from his program. However, the dispatch will now occur at a random point.

d) Another approach is to require the application task to be organized into short segments in which the dispatcher is requested at the end of each segment. If these segments were fixed at short intervals, it would enable system response time to be maintained.

Furthermore, the segment organization of a lengthy program provides visible and controllable evidence to the programmer of the possible points that alternate control paths can occur. Conversely, he is assured that once the segment begins it is non-interruptable until it ends other than by the executive servicing of an interrupt of the task placing itself into a wait state. Similar arguments could be used for the previous approach.

An attractive method involves a modification to approach (d). First, high priority cyclic tasks, operating in a synchronous mode in the foreground, will always be dispatched at the occurrence of the clock interrupt. All other tasks will only be dispatched at the segment points. This will guarantee response time where it is needed and loosen the requirement for segment operating limits.

Secondly, the establishment of segments for lengthy programs can be aided by an assembler or compiler. Given that a procedure oriented higher order language is used for application programming, it can often suggest segment points and make them visible to the programmer. Tentative examples of compiler based segment points are:

a) on all forward GO TO statements;

b) entry or exit from a block;

c) maximum time allowed in a segment exceeded.

The programmer must have a compiler override capability.

16-5.2 Resource Allocation

A resource may be defined as a facility of a computing system that can be temporarily assigned to tasks to enable them to perform their computations. Examples of resources pertinent to aerospace software are core storage, shared data, and data sets on mass memory units. Resource allocation is that function of a computer's operating system that assigns resources, when possible, to the tasks requesting them. In a multiprogrammed system, several tasks can request the exclusive use of a single resource. Since only one task at a given time can be granted its request, the others must wait until these resources are freed. Care must be exercised in resource allocation to minimize the number of transitions of a task from the active to the wait state and to avoid allocation conflicts.

To be specific, several conflicts can result from inefficient resource allocation. These are:

a) deadlock,

b) memory fragmentation,

c) priority conflict.

We will define each of these conditions in the following paragraphs.

16-5.2.1 Deadlock

Deadlock is a condition in which two (or more) tasks are each waiting for a resource held by the other before either can proceed. Neither task can release the resource it holds, so neither can be taken out of the wait state. For example, suppose task A holds resource R1 and needs R2, but task B holds R2 and needs R1. Since neither task can release its resource, neither can proceed and deadlock results.

Deadlock detection algorithms can be included in an operating system to enable the task performing resource allocation to recognize potentially hazardous situations, and hence, to avoid them. This topic has been discussed extensively by several authors [4-8]. However, such an algorithm can cost a high overhead in execution time. The aerospace executive should have an alternate way of avoiding deadlock.

Deadlock is the result of incremental resource allocation. That is, it is the result of tasks requesting resources sequentially during execution. By avoiding incremental allocation we can avoid deadlock without costly detection algorithms.

16-5.2.2 Memory Fragmentation

Memory fragmentation is a condition in which a task cannot be granted its request for a large block of contiguous core because all available core for dynamic allocation is in small noncontiguous blocks.

When this situation arises in a large ground based computing system having a large secondary memory, part of the contents of core are rolled out temporarily to create a large enough contiguous area of main memory to satisfy dynamic allocation requests. However, in the flight computer we seek to minimize the use of any MMU because of its inherent complexity. Thus, most data will be maintained in main memory so that programs can operate at maximum speed. Programs and data are only reloaded into the operating memory at low frequency during the mission, such as at the start of a new mission phase.

16-5.2.3 Priority Conflict

Finally, an allocation conflict can arise when a low priority task holds a resource that a high priority task requests. Often the resource cannot be released by the former task as in the case of temporary work areas of core storage. Unfortunately, the high priority task must now be placed in the wait state until the low priority task can safely release the resource. The result of this situation is a degradation in the system's response time for high priority computations. For a sufficiently large degradation the effects upon the overall mission can be very serious.

Each of these hazardous situations must be avoided in designing resource allocation algorithms for the flight computer. The following section will present methods of avoiding these problems.

16-5.3 Allocation of Specific Resources

In our discussion we will consider three categories of resources for which provisions must be made. These are:

a) dynamic memory allocation,

b) common data sharing, and

c) data set management

16-5.3.1 Dynamic Memory Allocation

Dynamic memory allocation occurs when the executive temporarily assigns blocks of core storage to a task requesting this resource. This core is returned to the dynamic core pool either by the task during its execution or by the executive at the end of the task. To avoid deadlock we require that all core requests of a task be satisfied when the task is placed in the ready state. That is, to avoid incremental allocation a task makes all core requests known to the executive via its TCB at schedule time. If the request can be satisfied, the task can be placed in the ready state provided it is not awaiting the allocation of any other resource. If not, the task is placed in the wait state, awaiting the release of a sufficient amount of dynamic core to satisfy its needs. When this core becomes available, the task can be placed in the ready state. Eventually when the task becomes active, it has all the core it will ever need and will not have to be placed back in the wait state during execution for lack of this resource. Hence, deadlock cannot occur because of a conflict in dynamic core allocation.

Although we have avoided deadlock fairly easily, the problem of memory fragmentation is not as readily solved. The reason for

this increased difficulty is that several alternative methods of avoiding this problem are available to us, and the specific method chosen depends upon the computational requirements of the mission application programs. So far these requirements are not known in any detail. Hence, we will examine four methods of memory allocation and determine which is optimal with respect to our present knowledge of the program requirements.

16-5.3.1.1 Fully Static This method would avoid dynamic storage allocation by permanently assigning to each task all the core storage it needs for the duration of the mission. Memory conflicts are obviously avoided.

If the total amount of core so assigned is small; e.g., 1K bytes, then avoiding the problems of dynamic storage allocation is advantageous since the executive design will be simpler. However, the amount of core needed is likely to be higher than our 1K example above, so that the extra cost in the amount of memory needed for static allocation becomes uneconomical.

This is not to say that no task should have its work areas permanently assigned. For example, a computation executed every minor cycle will utilize its work area for a large percentage of every major cycle. In this case it could be economical to statically assign this task's work area to it. However, for the large amount of tasks run on a less frequent basis the percentage of a major cycle that they utilize their work area is small. Hence, static storage allocation cannot be the only method of storage allocation considered.

Note that any task having a static work area allocation is by its very nature non-reentrant.

16-5.3.1.2 Fully Dynamic A frequently used method of dynamic storage allocation in large scale computing systems is to allow all tasks to compete with each other for all available core. A task can request a block of any size provided it does not exceed the amount of core available. If this block is available, it will be allocated to the task [9].

This disadvantage of fully dynamic allocation is that it does not solve the problem of memory fragmentation.

16-5.3.1.3 Semi-dynamic Let dynamic core be divided into blocks of several specific sizes; e.g., 50 bytes, 100 bytes, .5K bytes and 1K bytes. Tasks which request core must be structured so that their request conforms to one of these sizes. Although this method imposes a restriction upon the tasks, the problem of memory fragmentation is now solved.

There still remains the problem of low priority tasks holding core and preventing high priority tasks from executing. The problem can be partially solved by allowing several blocks of each size in dynamic core. This will reduce the probability of all blocks of a given size being simultaneously allocated. However, the number of blocks of each size cannot be too large since this would be as uneconomical as static memory allocation. Program requirements will of course determine how many blocks and what sizes to allow.

16-5.3.1.4 Priority Subpool Allocation Dynamic core will be divided into sections called subpools, one corresponding to each possible task priority level. A task requesting core will then receive its allocation only from the subpool corresponding to its priority level. Within a subpool core can be allocated on a fully dynamic or semi-dynamic basis.

If the fully dynamic method were used, fragmentation would occur within each subpool. To avoid this problem we will use semi-dynamic memory allocation (as explained above) within subpools. Each subpool will have several blocks of core of several different sizes. A task is then allocated a block of its requested size when it is placed on the ready queue.

Should a task request a block of core that is unavailable within its subpool because of existing allocations, a block from a lower priority level can be used for allocation. This will prevent a high priority task from having to wait for the release of core while low priority tasks can be scheduled. In addition, tasks of the highest priority will not have to share their subpool with any other tasks. These tasks will have the least interference from other tasks in competing for core.

The sizes of the blocks and the number of each size are determined by the number of tasks and their requirements at the given priority level. Once this algorithm has been implemented size and quantity parameters can be varied for optimization. This method of dynamic core allocation seems to be the most advantageous.

16-5.3.2 Common Data Sharing

In any multiprogramming system a resource allocation problem arises when data in core memory can be simultaneously used by two (or more) tasks. If two tasks only want to read the data, no conflict exists. However, if one of the tasks wants to update before the other has finished reading, a conflict arises.

To illustrate this, consider the examples shown in Figure 16.2. In both examples TASK B interrupts TASK A during the execution of a statement. In Example 1, presume that the interruption occurred

while the matrix $\overset{*}{N}$ was being read. When TASK A resumes, the computation of $\overset{*}{M}$ will continue using some "old" $\overset{*}{N}$ data and the "new" $\overset{*}{N}$ data assigned in TASK B. In order to prevent this conflict, initiation of TASK B would have to be stalled until the reading of $\overset{*}{N}$ in TASK A is completed.

In Example 2, presume that the interruption occurs just after the current value of Y is loaded into the accumulator. When TASK A resumes, the "old" value of Y (i.e., not reflecting the update of Y in TASK B) is restored into the accumulator, X is subtracted and the result assigned to Y. In order to prevent this conflict, the initiation of TASK B would have to be stalled until the value of Y is updated in TASK A.

These examples illustrate the fact that accesses to shared data must be controlled to prevent conflicts. One possible way of doing this is by preventing task dispatching at critical times. This method is too restrictive however, especially for high priority tasks needing fast system response. We will investigate alternative approaches to this problem.

a) OS/360 uses the ENQ and DEQ macros to grant tasks access rights to shared data. ENQ will grant a task access rights as long as no other task is using the data. In the latter case, the task requesting access rights is put in the wait state, awaiting the release of this data (DEQ). Upon this release, the next task enqueued for access rights is taken out of the wait state and allowed to proceed. For two tasks that only want to read shared data, this method imposes a needless wait for one while the other has the data enqueued.

b) A second approach to avoid common data sharing conflicts is to use UPDATE blocks as is done in the HAL compiler [10,11]. An UPDATE block is a group of statements within a program providing a controlled environment for the reading and writing of shared data variables. Upon entry into the UPDATE block, read or write locks are established around parts of the compool containing the variables to be referenced. There need not be an individual lock for each variable nor should there be only one lock around the entire compool. How the compool is organized can be decided at a later time depending upon the programs to be executed and their requirements.

   Should a part of the compool needed by a task be unavailable for locking, the task is placed in the wait state. Any other parts of the compool it has locked are now unlocked so that they can be used by nonwaiting tasks. The requesting task can be placed in the ready state when the scheduler determines that all parts of the compool requested now can be allocated to this task. At this time read or write locks are established around these parts of the compool.

Three types of locks can be established: read, write, and writing. We say that unlocked data is in state 0 and locked data can be in states 1-3 corresponding to the three types of locks respectively.

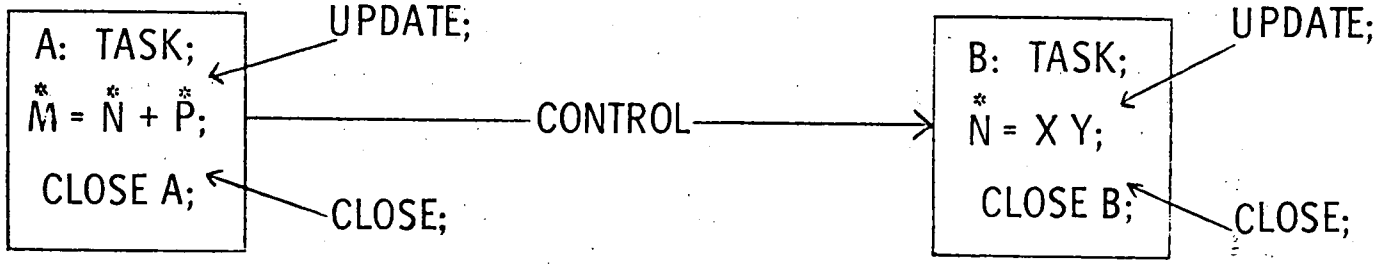
A read lock will enable another task that also wishes to read lock this data to do so. If a write lock is established around a piece of data, a copy of the data is made for the updating task. Upon closing the UPDATE block, the compool is updated as long as no other locks exist around the data to be updated. No writing locks can be put on a given part of the compool, until any read locks already there are removed by all tasks reading this data. If the locks exist, the updating task must wait until the locks are removed.

Consider the first example above and suppose that the statements in question (in TASKS A and B) were enclosed within UPDATE blocks. In TASK A a read-lock is established for N, because it will be read only. After the interruption, a write-lock is established for $\hat{N}$, and TASK B proceeds toward completion using copy-data for $\hat{N}$ rather than active data. At the end of the update block in TASK B, the process stalls because of the read-lock imposed in TASK A. As a result, TASK A is allowed to continue with consistent "old" $\hat{N}$ data. After completion of TASK A, a copy-cycle is effected in TASK B, and $\hat{N}$ is updated. All conflicts are eliminated. A table of compool state transitions follows.

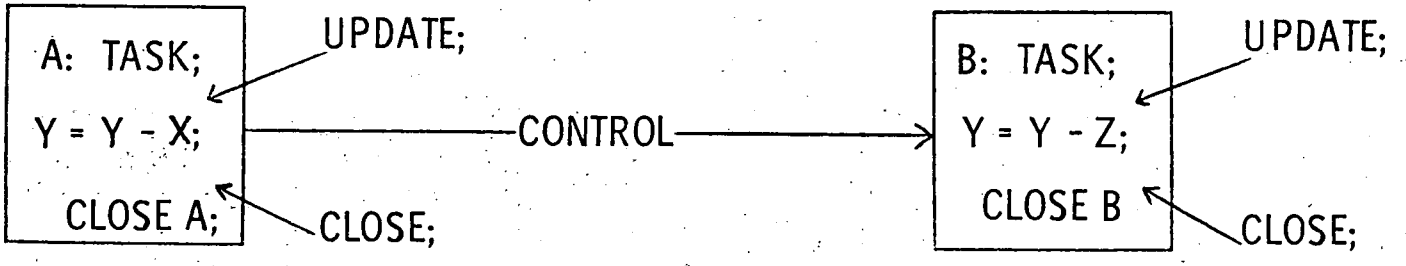
Present State / Desired State	Free	Read Locked	Write Locked	Writing Locked
Free	O.K.	O.K.	not applic-able	O.K.
Read Locked	O.K.	O.K.	O.K.	Wait
Write Locked	O.K.	O.K.	Wait	Wait
Writing Locked	not applic-able	Wait	O.K.	not applic-able

Figure 16.2: Control of Shared Data

EXAMPLE 1: READ AND WRITE CONFLICTS



EXAMPLE 2: UPDATE CONFLICTS



NOTES:
1. B "INTERRUPTS" A IN BOTH CASES
2. #1 TASK A RESUMES USING OLD AND NEW VALUES FOR N*
3. #2 TASK RESUMES "CLOBBERING" THE VALUE FOR Y SET BY TASK B

To prevent any task from locking a part of the compool any longer than necessary, no I/O statements and no programmed WAIT statements will be allowed in an UPDATE block. This requirement will prevent a high priority task from having to wait for long time intervals while a lower priority task has data locked.

To economize on the amount of core needed for the compool, part of the compool can be overlaid on transitions to different mission phases. If two tasks that are only executed during a particular mission phase use part of the compool, it is needless to keep this part of the compool in core as long as no other task in another phase will ever again use the data. In this case as new program modules are read into core during a mission phase transition, this part of the compool can be overlaid.

16-5.3.3 Data Set Management

Data set management is heavily dependent upon the type of mass storage unit used. If a tape drive is used, very little data management capability will be necessary. However, if a random access unit is used, more extensive data management facilities will be necessary.

In this chapter we will assume a random access unit; e.g., a disk drive. The data management system need not be as general purpose as in the System/360, for example. However, it must be designed to meet the needs of the aerospace mission. One of the criteria used in designing this part of the executive is the desirability of minimizing use of the random access unit during the mission. The major anticipated uses of the storage unit are to record flight data, to update the programs in core memory on a per mission phase basis, and to retrieve display skeletons for the visual display application programs. More frequent use of the mass storage unit is probably necessary.

There will be two classes of data sets on the random access storage unit, read only and read/write. The former category may be read at any time by any number of tasks without conflict. The latter category, however, can cause access conflicts, and hence, some protection mechanism is necessary.

A directory of each data set on the storage unit and its characteristics will be maintained in core memory. The data set directory entry for a read/write data set will identify only one program module with writing access rights. This program module must not be reentrant. Whenever a task requests to write upon a data set, the I/O supervisor will check to see if the data set is indeed read/write, and if the requesting task has access rights. Since only one task can update a given read/write data set, no write

conflicts are possible.

A task may also request to read a read/write data set. For example, data recorded in a former mission phase may be important to an executing task. In this case, the I/O supervisor will honor the read request. However, the software must be structured so that the requesting task is not reading part of the data that is presently being updated. The I/O supervisor can check for this fact; or each task that wishes to read a read/write data set can be made responsible for knowing the integrity of the data it receives.

16-6.0 Features of the Executive System

We now turn our attention to specific features needed by an aerospace software system and in particular the executive. These features are aimed toward a complex mission such as the Space Shuttle or manned planetary exploration. In less complex missions some of these features might not be needed. However, we will present as many as are relevant for our discussion here.

16-6.1 Directories

Several directories are usually necessary to enable the executive to efficiently manage the system's resources and manage task execution. Consistent with our view of minimizing the use of the MMU, we assume these directories are in main memory. Three typical directories are the following.

16-6.1.1 The Program Module Directory

The program module directory (PMD) is a list of all program modules known to the system; i.e., all program modules both in operating memory and secondary storage. Each entry consists of three full words and has the format shown in Figure 16.3a. It contains the following information:

a) program module identification,

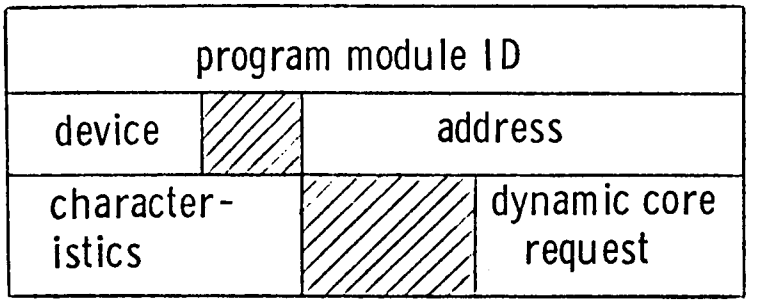
b) where the module is resident,

c) address of module,

d) module characteristics, such as reentrant,

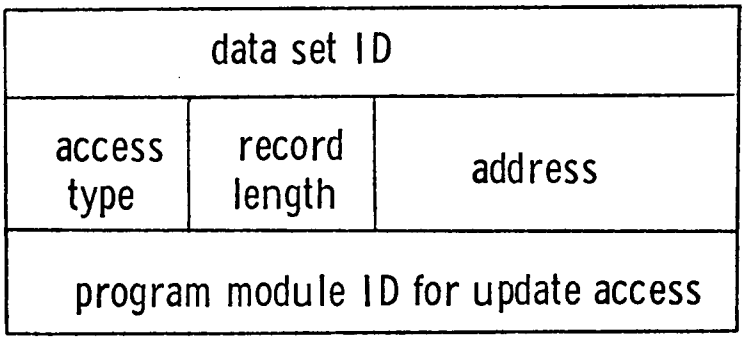
e) dynamic core needs.

1. Program Module Directory (PMD)



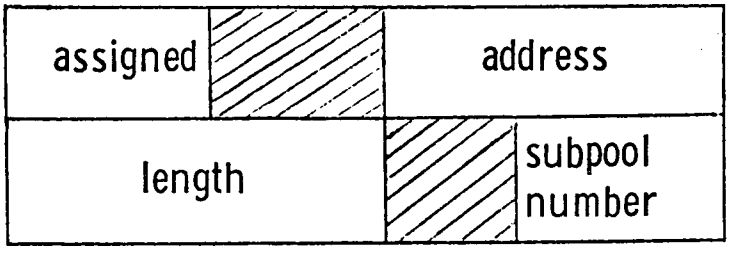
(a)

2. Data Set Directory (DSD)



(b)

3. Dynamic Core Directory (DCD)



(c)

Figure 16.3: System Directory Elements

This directory is updated when the contents of core change or new program modules are added to the system preflight. Its major purpose is to enable the scheduler to locate a program module and to provide enough information to construct a TCB.

16-6.1.2 The Data Set Directory

The data set directory (DSD) is a list of all data sets residing on the MMU. A data set may be an executable program module or a collection of flight data. An entry in this directory is three full words containing a data set identification word, MMU starting address, logical record length, and data set characteristics (i.e., read only or read/write). In addition, if this data set can be updated, the program module with update rights will be identified in the DSD entry. This information is illustrated in Figure 16.3b.

The DSD enables the I/O supervisor to locate data sets on the MMU for I/O operations.

16-6.1.3 The Dynamic Core Directory

The dynamic core directory (DCD) is a list of all blocks of core that can be dynamically assigned to a task. Each entry is two full words containing the address of the block, its byte length, its subpool number, and an assignment bit. The format is given in Figure 16.3c. The DCD enables the executive to dynamically assign core to tasks at schedule time.

16-6.2 Subroutine Linkage

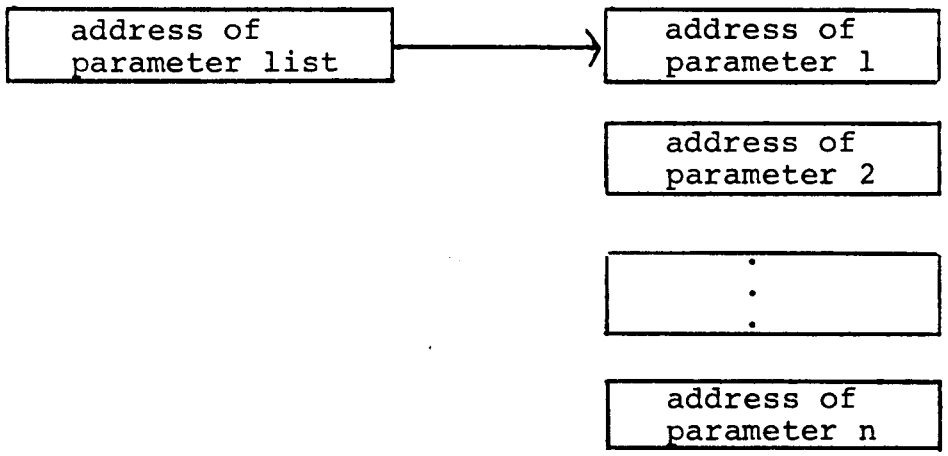
In order to standardize the way program modules are structured and to avoid conflicts in parameter passing, register usage, and register saving, a method of program module initialization and linkage must be developed. The particular method chosen depends heavily upon the hardware features of the flight computer. However, some generalizations are applicable. Usually upon entering a program module the contents of the general registers must be saved so that they can be restored upon task termination. These registers can be stored in an area of core called the save area. The task is now free to perform its computation. Then upon completion of its computation, a task terminates by restoring the content of the general registers it had saved upon entry and branching to a return address.

16-6.3 Common Subroutines

In addition to a task being able to schedule another task, a task may execute a common subroutine. A subroutine is a piece of

coding which may be used by several tasks without itself becoming a task. A common subroutine must be reentrant or serially reusable. In the former case the calling task supplies working memory for the subroutine. In the latter case, the subroutine must supply control for preventing multiple simultaneous executions. A software generated event can be used by the subroutine as a semaphore to insure only one user at a time [12]. Examples of common subroutines are square root, trigonometric functions, and vector/matrix functions.

The calling task may pass parameters to a common subroutine by providing a pointer. This pointer will contain the address of a list of pointers, each pointing to one of the parameters, as illustrated below.



The subroutine may now read each of the passed parameters and return a computed value in one of them. The general registers can be used to pass the parameter pointer and dynamic core pointer to the common subroutine.

16-6.4 Task Priority Levels

In the flight computer there will usually be several task priority levels. Let us here assume there are six priority levels, 0-5, with 0 being the highest priority. Priorities 3,4, and 5 are used by application tasks.

Priority 2 is reserved for any application task while it is executing an UPDATE block. That is, if a task of priority 3,4, or 5 is executing an UPDATE block, the task's priority is raised to 2 until the updating of common memory is completed. It then returns

to its previous value. Thus in effect, we are limiting dispatching of priority 3-5 tasks while another task executes an UPDATE block. By the nature of the system there will be at most one task at priority 2 at any given time. This places restrictions on the use of an update block in that a task cannot enter the wait state voluntarily under any conditions. It must enter the block, complete the updating of common memory, and exit the block. The high priority cyclic sequencer is allowed to interrupt an update block.

Priority 1 is only used by the cyclic sequencer. It is given priority over any application task because of the time dependent nature of its execution. Should the cyclic sequencer be unable to lock part of the compool, the task at priority 2 is executed until it closes its UPDATE block. Now the cyclic sequencer can lock its required data without interference. The use of priority 2 is specifically designed to enable the cyclic sequencer to execute with the lease possible wait due to shared data unavailability.

If a response time equal to a minor cycle is insufficient to handle critical mission functions, a special priority level could be included in the executive system. Priority 0 can be reserved for acyclic tasks that must immediately be executed for the safety of the mission. These tasks are time constrained and must execute in less than 0.5 msec. This rule is enforced by a timer in the hardware. Moreover, priority 0 tasks may not use dynamic core or use the compool since by their very nature no wait in their execution can be tolerated.

Examples of priority 0 tasks are computations that must be done during a critical maneuver, engine burn or cutoff, etc. Should one of these tasks require more than 0.5 msec to execute, it may change its priority to 3 or lower during its execution. Should there be no higher priority task scheduled, it will continue execution at this lower priority. Otherwise, it must wait for the CPU. In this way critical functions can immediately be given 0.5 msec of CPU time without seriously interfering with the executive's cyclic functions that must be performed every minor cycle.

Including priority 0 in this executive system would require hardware interfaces to the computer. There would have to be a method of generating an immediate external interrupt in the CPU from the subsystem or device sending the interrupt condition.

16-6.5 Assignment of Core Memory

Operating memory can be organized as follows: the lower core addresses are assigned to the executive, as shown in Figure 16.4. The first locations contain system registers, such

as the timer. The next block of core contains the executive's program modules, followed by the executive work area. Within this latter area the executive's queues, directories and tables are resident.

There are several types of queues present in this area, such as TCB queues and IORB queues. Since each type of control block is a fixed size, the executive can maintain threaded lists of unused blocks of core storage, each element of which contains enough core for allocation as one of the types of control blocks, respectively. Thus, when a task requires a control block, the executive can remove an element from the appropriate queue of unused blocks and assign this block to the task to be formatted into a control block. Similarly, when the executive determines a task is finished with a control block, that core that the control block occupied is then returned to the appropriate queue of unused blocks for later allocation.

Sufficient space must be allowed this part of core to hold the maximum number of control blocks that will ever be needed by application tasks at any given time. Should space be unavailable, this is an error condition since more tasks are in the system than its resources can accommodate.

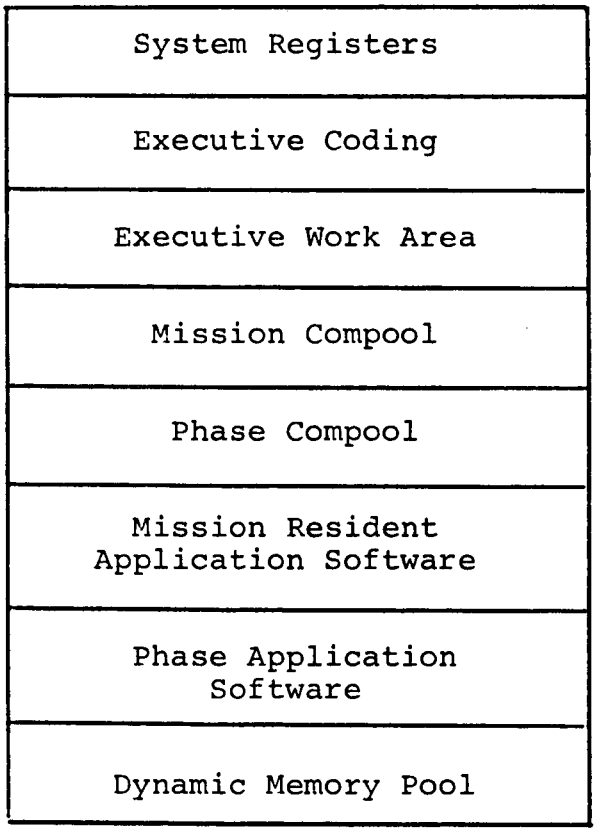


Figure 16.4: Structure of Operating Memory

The compool will immediately follow the executive work area and be divided into a mission portion, which is resident in main memory throughout the flight, and a phase portion, which is overlaid when a mission phase transition occurs. A similar feature exists with the application software which follows next in memory. The mission resident part comes first, followed by the phase dependent part. Finally, there is the dynamic memory pool.

16-6.6 Events

An event is an occurrence of significance to the system. There are a fixed number of events established for the system identified in an event directory. There are several categories of events recognized by the executive. These are for example time events, I/O completion events, release of shared data, and release of dynamic memory. If other external interrupts are used in the system they may also be categorized as an event. The final category of events include those which are controlled via application software and used for task synchronization.

There are two types of events within this last category: latched and unlatched. A latched event has associated with it a binary state either on or off. Latched events may be signalled on (posted) or signalled off (deposted) under application software control via the executive. A latched event maintains its current state until changed via signal command. An important use of latched events is to record the occurrence of an event within the system so that if a task later wishes to use the occurrence of the event as a criterion for performing a function, it can do so without having lost all record of the events occurrence. An unlatched event is only signalled on. It is signalled off immediately after processing by the executive. In a sense an unlatched event is a pulsed event analogous to a hardware interrupt.

An event control block (ECB) contains the current status of an event. It is dynamically created by the executive when a task is placed in the wait state. All events have system scope. When the anticipated event occurs, bit 0 of the ECB is set to 1 to record the event for the executive. See Figure 16.5 for the format of an ECB. The ECB contains a bit to denote if the task is awaiting the event, a bit to denote if the event is completed, and two threaded list pointers.

16-6.6.1 Event Handling

Within this software system there is a close relationship

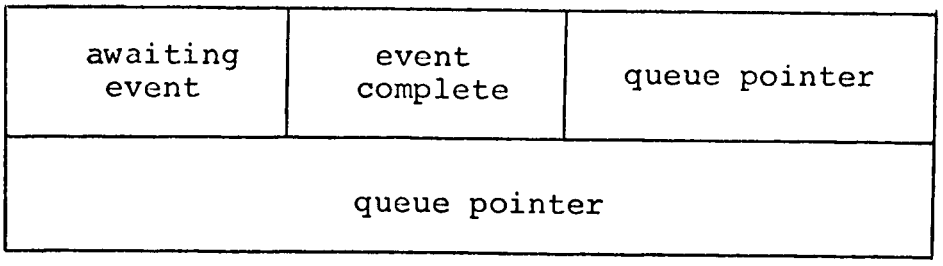


Figure 16.5: - Format of Event Control Block

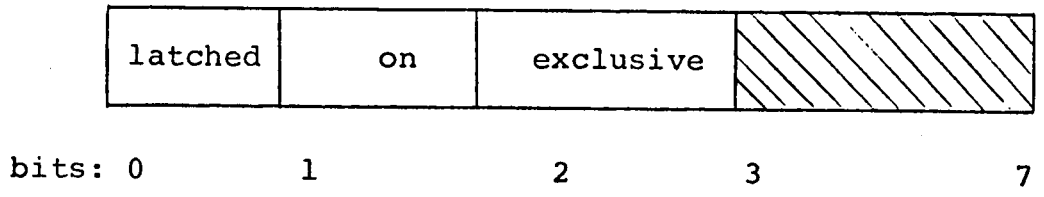


Figure 16.7: - Format of Event Descriptor Byte

between task management and event handling. Tasks that are placed in the wait state remain there until the anticipated events that they are awaiting occur. Then the event handling facilities of the executive call upon the scheduler to place these tasks in the ready state.

Tasks can be placed in the wait state in two ways. First, a task can voluntarily request the executive to place its TCB on the wait queue until some anticipated event or events occur. Second, when the scheduler attempts to place a task in the ready state, the unavailability of a resource on the nonoccurrence of some event(s) causes the task to wait until the resource is freed or the event(s) occurs.

A TCB in the wait queue is associated with a threaded list of ECBs, each corresponding to an event whose occurrence the task awaits. In addition, each event has an associated event list which contains pointers to all ECBs of tasks awaiting the occurrence of the event. Thus, when an event occurs, each ECB pointed to by the event list can be posted; i.e., record the fact that the event occurred. An illustration of this control structure is given in Figure 16.6.

After the event occurs, the scheduler is called. Its function is to determine if any task awaiting this event can be placed on the ready queue. The criterion for this decision is whether or not all (or some acceptable combination) of the events a task is awaiting have occurred. If so, the task is placed in the ready state by having its TCB moved to the ready queue and having its ECBs deleted. In addition, the scheduler can now delete the event list associated with the event. Tasks can perform a function based upon the occurrence of a single event or upon the occurrence of some combination of several events.

Each task awaiting a non-software controlled event in one of the first four categories can only await this one event and not some combination of events. However, software controlled events contain a predefined number of distinct events which may be used individually or in combinations by tasks. Events are not dynamically created by the system. Hence, software generated signals must correspond to events defined at system generation time. Each software generated event contains an event descriptor byte, containing the characteristics and state of the event. Figure 16.7 shows the format of the byte. Bit 0 describes the event as latched or unlatched; bit 1 records whether the event is on or off; and bit 2 describes the event as exclusive or non-exclusive, a distinction we will presently explain.

Within the class of unlatched events we will choose a subset to be exclusive events. An important use of exclusive

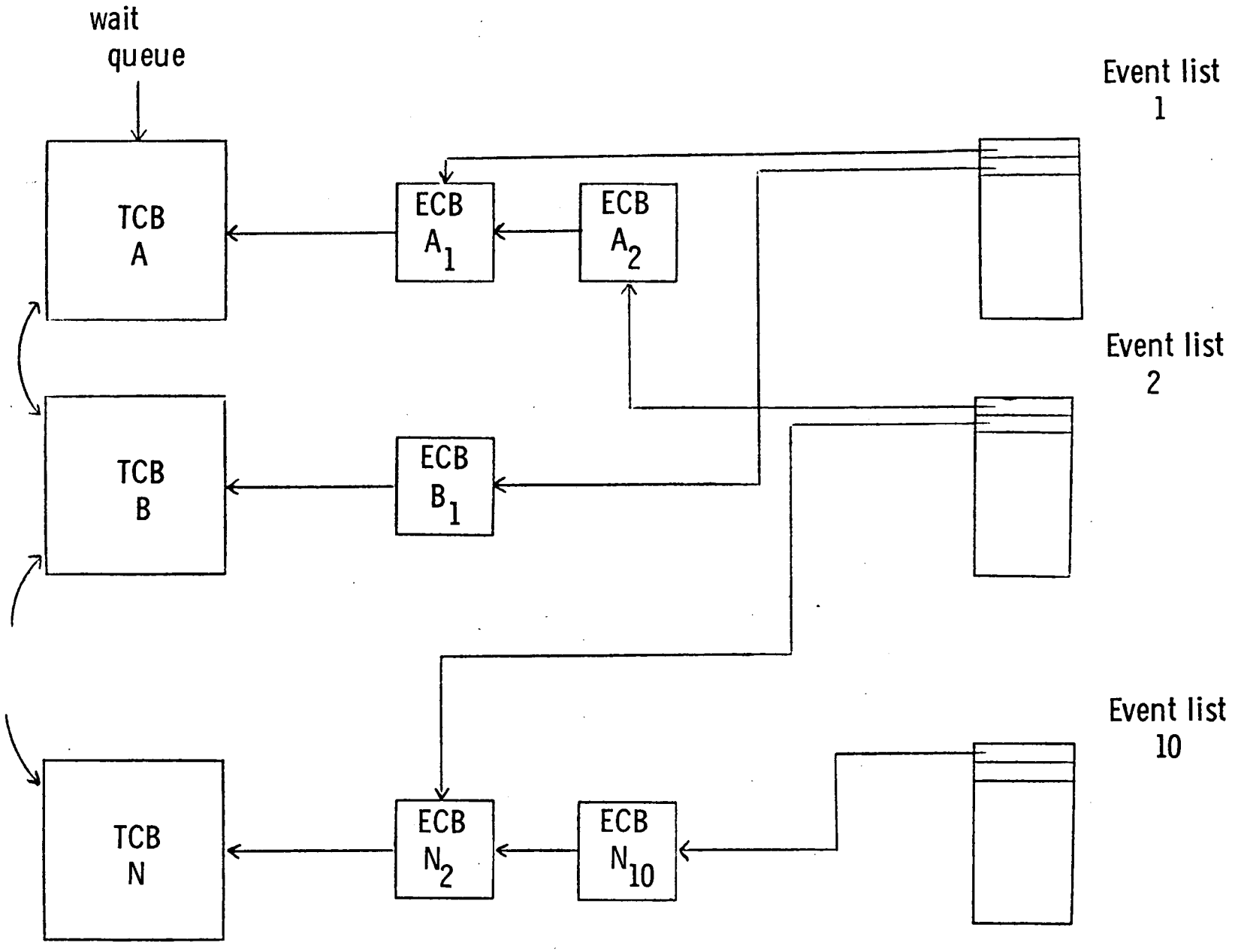


Figure 16.6: - Event Handling

events is to exclude tasks from use of some serially reusable resource. When an exclusive event is signalled on, only the highest priority task awaiting the event is placed in the ready state. All other tasks awaiting the event remain on the wait queue. When the highest priority task is made ready, the event is then signalled off by the scheduler to be sure no other application task can interfere with the exclusion process. This use of exclusive events is analogous to Dijkstra's concept of semaphores [12].

Note: it is the duty of the programmer to know if the events he is using in his tasks are being used by any other tasks. Without being sure of this fact, tasks can unintendedly interfere with each other's execution and destroy the integrity of their computations.

Also note: in the actual implementation of this executive system, some categories of events will be immediately serviced by the executive upon occurrence of the event, and hence, a record of the event's occurrence will be unnecessary. These events will therefore not need ECBs in their functional implementation. These events include release of dynamic memory and unlocking parts of the compool.

16-6.7 I/O Scheduling

If we assume a data bus system, the hardware should be mechanized in a way which allows bus operations to continue independently of the CPU once an I/O command is issued to the bus control unit. This means that the processor is only allocated to the I/O function during an I/O channel command and should be reallocated to the computation job stream upon completion of the command. The design question for the software I/O control will be how to schedule the I/O operation: should it be decoupled from the executive program control and maintain its own separate I/O queue, or should it be inserted as an integral part of a fixed sequence? For example, if I/O were operated each minor cycle it would output data from the previous cycle, and input data which is to be processed during the following cycle. With this concept, however, the I/O must be predetermined and fixed, with constraints similar to those for fixed scheduling of computational jobs. Input and output then occurs each cycle, whether it is needed or not. This approach will cause excess data to be put on the bus, reducing its effective bandwidth and its capability for expanded performance. On the other hand, scheduling I/O as a priority queue based on demand, has many features in common with scheduling jobs (e.g., priority, timing, conflicts, etc.). An effect of the I/O queue on the system is that several jobs may be in a suspended state awaiting I/O completions. Methods are available to avoid such delays, for example, buffering for data in and out and issuing commands only via a queue. The I/O algorithms used must combine the best

features of synchronous and asynchronous control.

16-6.8 Configuration Management

The topic of configuration management is very extensive, covering many aspects of computer and system design. An adequate discussion of this topic in relation to a space mission must treat the areas of power on initialization, mission phase initialization, error recovery, swtiching between simplex and redundant modes of operation, and system synchronization. Such a discussion goes beyond the intended introductory yet comprehensive nature of this chapter. The interested reader is referred to [1].

REFERENCES

1. Pepe, J., Advanced Software Techniques for Data Management Systems, Vol. II: Space Shuttle Flight Executive System - Functional Design, (Intermetrics, Inc., Cambridge, Mass., February, 1972, prepared under contract NAS 9-11778.)

2. Denning, P., "Resources Allocation in Multiprocess Computer Systems", Ph.D. Thesis, M.I.T., May 1968.

3. Vyssotsky, V., et al., "Structure of the MULTICS Supervisor", Proc. FJCC, 1965, pp. 203-212.

4. Coffman, E., et al., "System Deadlocks", Comp. Surveys, 3(2), June 1971, pp. 67-78.

5. Habermann, A.N., "Prevention of System Deadlocks", CACM, 12(7), July 1969, pp. 373-377.

6. Holt, R.C., "Comments on Prevention of System Deadlocks", CACM, 14(1), January 1971, pp. 36-38.

7. Murphy, J.E., "Resource Allocation with Interlock Detection in a Multi-task System", Proc. FJCC, 1968, pp. 1169-1176.

8. Coffman, E., et al., "Deadlock Problems in Computer Systems", Proc. Conf. sponsored by Software World, U. Sheffield, April 1970, pp. 41-48.

9. IBM Corporation, "OS/360 Supervisor and Data Management Services", IBM No. GC28-6646.

10. Intermetrics, Inc., Development of an MSC Language and Compiler, Cambridge, Mass., June 1971, prepared under Contract NAS 9-10542.

11. Intermetrics, Inc., The Programming Language HAL - A Specification, Cambridge, Mass., June, 1971, prepared under Contract NAS 9-10542, MSC Document # MSC-01846.

12. Dijkstra, E., "Structure of THE Multiprogramming System", CACM,11(5), May 1968, pp. 341-346.

C.6

CHAPTER 17

MICROSIMULATION IN SYSTEM DESIGN

17-1.0 Introduction

One of the more important techniques that the system designer should be able to use in his work is simulation. Simulation has proved to be an indispensible aid in designing digital computers and their associated software. In particular in the aerospace industry the experience of Intermetrics personnel with the Apollo and Poseidon guidance computers has shown this statement to be true. However, the proper use of simulation requires the system designer to understand the different forms simulation can take, their capabilities and advantages. On the other hand, he must also know when the simulator is being overused; i.e., when easier design verification would result from the use of different techniques.

Simulation of a system on a high (macro) level provides a heuristic model of system performance. A common heuristic is to use statistical estimates of events occurring within the system. For example, simulating the performance of an executive system could include statistical estimates of the job stream, including job execution times, arrival rates and the use of system resources. These estimates then allow the simulator to calculate performance figures on the operation of the executive. We have already dealt with macrosimulation in Chapter 6. We are now primarily concerned with microsimulation; i.e., the bit by bit interpretive simulation of a digital computer's operation.

We will develop the topic of microsimulation by discussing desirable simulator organizations and features. This discussion will be augmented with Apollo examples. We will then see how the simulator can be integrated into the total system design effort and how it can aid the development and verification of the system software. Simulation can also be an optimization aid in microprogram design, a topic we will also treat. Finally, we will list the advantages of using microsimulation in designing a system and look at some specific systems, such as Apollo and Poseidon.

17-2.0 Features of the Microsimulator

Microsimulation of a digital computer, which we will call the object computer, takes a program written for this computer and executes every instruction and control function as the object computer would in an interpretive fashion. The simulation is usually not carried down to the logic gate level but only to the level at which information is visible in the object computer. This usually means to the special register level, e.g., program instruction counter, timers, accumulators, index registers, etc. This bit by bit simulation is at the heart of microsimulation. However, simulation is also a tool in system design. Hence, most simulators contain diagnostic software aids for debugging object computer programs. For aerospace applications environmental models of the vehicle are also included enabling full flight, closed loop mission simulations. We will now investiate some of these features more deeply.

17-2.1 User Options

Simulators usually provide several categories of options that the user can invoke during a run. These options can be divided into initialization requests and diagnostic aids. They are provided to help the user set up his simulation and to help him debug his programs conveniently. Each option that is requested will usually require extra simulator overhead in terms of execution time and the output generated. Hence, they should be used judiciously.

The Apollo simulator [1] provides an excellent example of the options a simulator should provide for its users. There are three classes of special requests. The first two are primarily initialization and the third, diagnostic.

A. The first class exercises control on the simulation from beginning to end. It includes specification of the program to be simulated, starting location, initial time and time limit for simulation, action to be performed upon entering a single-instruction endless loop, whether to allow execution of instructions in data memory, and descriptions of event types whose occurrence is to be noted in the simulation output listing.

B. Class 2 includes those requests that initialize or modify the content of specific locations in the simulated program or data areas. These changes are done prior to the beginning of simulation. Thus, a program need not be reassembled for it to be simulated several times each with different initial conditions.

C. Class 3 requests are triggered when a specified location is accessed as an instruction or data during simulation or when a specified simulation time is reached. These requests may also be specified under combinations of conditions which must be satisfied before the request can be executed. Examples of Class 3 requests are stopping simulation normally or in abort mode, listing the current value of the simulator clock, listing the content of memory, and turning the execution trace on or off. When an environment is part of the simulation, this class of requests can include environment action or printout requests.

Beyond what was included in the Apollo simulator, other simulator diagnostics, which we will now describe, can be very useful to the system designer.

17-2.1.1 Stress Testing

The stress testing special request can be provided in a simulator to help determine if combinations of application programs will exceed their combined time budgets under the executed conditions of operation. This request reduces the speed of the object computer. If a group of application programs is run in a simulation with a computer whose speed is, say, 75% of the real computer capability, successful operation may be interpreted to mean that no more than three-fourths of the computer capacity has been absorbed. This special request can thus be used to "diminish" the capability of the computer until a point is reached where timing requirements are not satisfied. This level then is a guide to the amount of computer capability still available for other software.

17-2.1.2 The Coroner Request

A "coroner" special request can be implemented in a simulator for post-mortem diagnosis. The request causes storage of information from each simulated instruction in a circular buffer of size n. If the run abnormally terminates, a list of the last n instructions simulated is produced. This list is a valuable aid in determining the reason for the abnormal termination. However, the overhead associated with this request requires it only be used when its cost is outweighed by the enhancement of debugging efficiency.

17-2.1.3 An Alternative Approach to Diagnostics

The Deep Submergence System (DSS) computer was simulated at the MIT Instrumentation Lab using a different approach to diagnostic aids. The triggering of a diagnostic was the same as in Apollo, namely, by accessing a given location, etc. The difference is in how

the diagnostics were implemented. In Apollo they were part of the simulator coding, whereas in the DSS system triggering a diagnostic transferred control to a user specified routine which was then executed. The interface was achieved through a special language allowing users to specify actions to be taken or to call routines of other languages. This user-simulator interaction provided a type of extensible simulator system.

As with all powerful systems the extensibility incurred increased costs. One of these costs was the design and implementation of an interface language. Another was the additional computer time needed to assemble and link-edit diagnostic routines prior to simulator runs.

17-2.2 The Environment

In simulating aerospace computers the simulated software must often interact with the spacecraft and its environment. For example, navigation programs must receive accelerometer input before they can correctly update vehicle position and velocity. To enable simulation of these programs an environmental model must be provided for the simulator. A high degree of similarity must be maintained between the real and modeled environments so that the simulated computer can be subjected to computational loads and dynamic situations closely approximating the conditions of the actual mission.

The environment for Apollo consisted of modeling spacecraft dynamics, engines, optics, astronaut interactions, atmospheric and gravity effect, motions of celestial bodies, etc. See Figure 17.1. When, during simulation, the computer simulator encountered I/O instructions, a program, called the communicator, would be invoked to determine whether immediate interaction with the environment was necessary. If calling the environment was necessary, all interaction between the computer simulator and environment was through the communicator. The communicator effectively decoupled the computer simulator and environment and and reduced the frequency of environment updates.

In using the Apollo simulator the programmer was provided with environment options. These include:

a) selection of environment programs and files required to simulate the specific test flight condition;

b) initialization of the environment to desired conditions; and

c) specification of relavent interaction between Apollo software and environment to be recorded.

To help set up initial conditions for Apollo simulations, a special program was provided. Users normally ran this program with a convenient reference system to specify the initial position and velocity of the

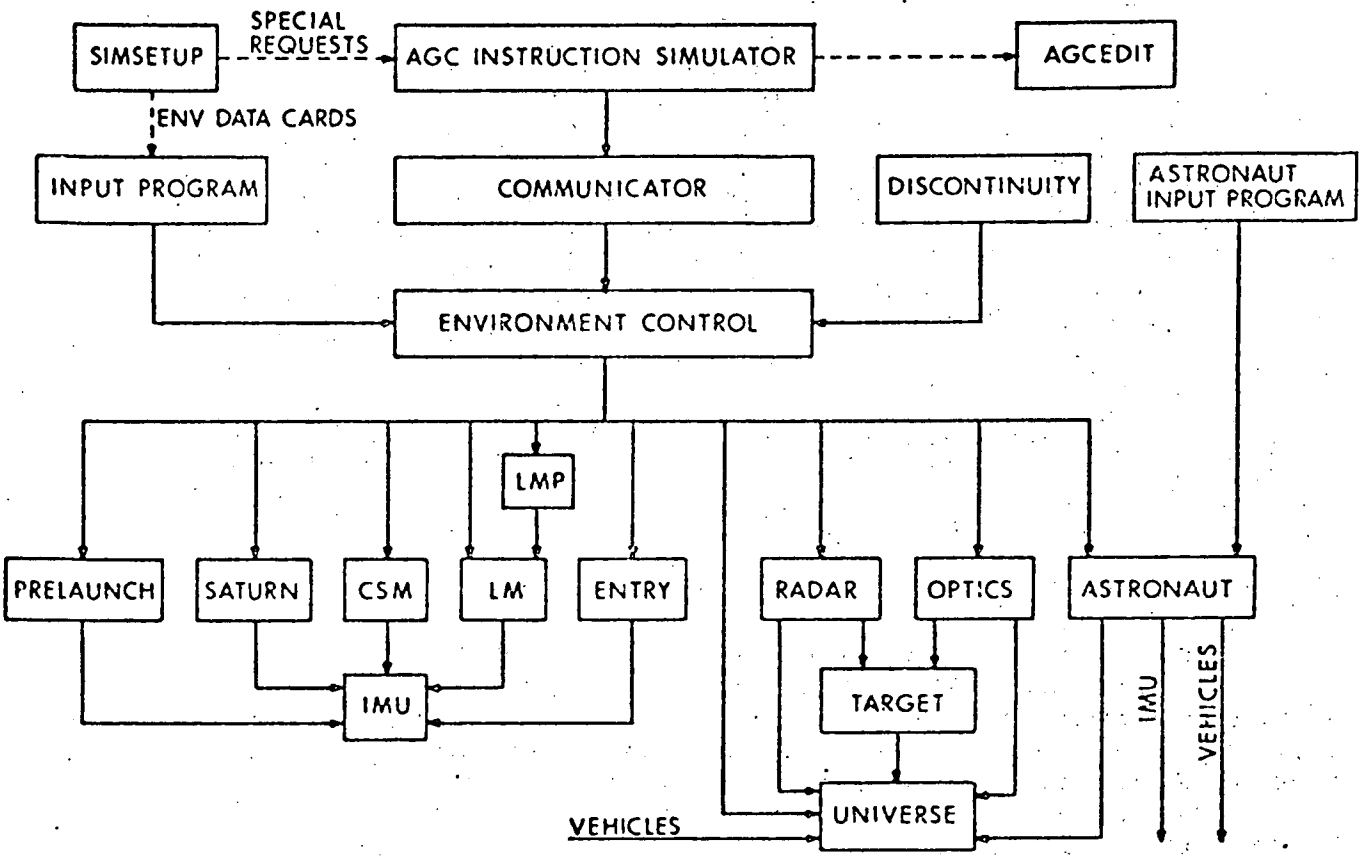


Figure 17.1: The Apollo Digital Simulator

vehicle and obtained from it an input deck of cards for the simulator. Setup of simulations thus became less burdensome.

17-2.3 Rollback

A useful feature to be used in microsimulation is rollback [2]. Long missions such as Apollo require simulation time on the order of hours. Should the host computer (on which the simulation is being executed) malfunction, the simulation will abnormally terminate. Upon restart one does not want to go back and duplicate the execution of this simulation from the beginning of flight. By establishing rollback points in the simulation this problem is avoided. At rollback times complete core and register dumps are taken, and this information is put on a secondary storage device. Then upon system failure the simulation can be restarted at the last rollback point by loading memory with this stored information. The overhead associated with rollback is well justified with long simulations, such as Apollo. However, to prevent this overhead from becoming too high the system designer must decide upon a judicious criterion for establishing rollback points. That is, he must tradeoff the cost of frequently storing rollback information with the savings in not having to resimulate a large part of the flight.

17-2.4 Supporting Software [3,4]

The simulation facility usually needs other software besides the microsimulator. Assemblers and linkage editors make up part of this support software. The assembler allows programs to be written for the object computer in a symbolic language. It translates these source modules into machine language for execution. The output of the assembler is the translated object module along with the symbol table, relocating dictionary (RLD), and external symbol dictionary (ESD) that are required by the simulator and linkage editor. The assembler provides several capabilities:

a) the symbolic expression of source programs;

b) pseudo-operations for storage allocation, instruction counter control, listing control, etc.;

c) macro and conditional assembly features; and

d) organization of source programs into modules.

These capabilities, such as the macro facility, can be made as sophisticated as necessary for the particular application. However, increased capability requires a larger implementation effort. For example, the ability to separately assemble source modules requires a linkage editor to organize the respective object modules into a single load module ready for execution. The linkage editor performs the following functions:

a) allocate storage in the object computer's memory for each object module and load these modules into place;

b) resolve the object modules' external symbol references using the ESDs;

c) resolve any relocatable addresses using the RLDs; and

d) selection of members from a library of standard object modules specified to be included in the linkage.

When the linkage editor has performed all these functions, the resulting load module is ready for simulation.

17-3.0 Design Issues and Structure of the Microsimulator

We have already raised one design issue in the last section by showing two alternative approaches to implementing diagnostic aids. Other design issues are the use of higher order languages (HOL) in implementing the simulator and the modular organization of the simulator. Before getting into these topics which relate to physical organization let us briefly look at how the simulation process can be logically organized.

17-3.1 Logical Structure

The simulation process can be organized into four factors as shown in Figure 17.2 . These are:

a) the user (USER),

b) the simulator itself (SIMULATOR),

c) the computer being simulated (COMPUTER), and

d) the simulation output (OUTPUT).

It is the logical interaction among these factors that we wish to explore.

The USER is primarily interested in the COMPUTER. His interest in the SIMULATOR appears only because it helps him control and examine the COMPUTER. Figure 17.3 shows the paths of logical interaction between the four factors as they appear to the USER. (The actual physical flow of interaction is shown in Figure 17.4).

The control path labeled A in the two figures provides the USER with the capability of specifying the load module to be simulated, start-location and initial SIMULATOR clock setting, the maximum allowable SIMULATOR clock setting (to assure run termination), the configuration of the COMPUTER (levels of redundancy, numbers of spares, initial fault states, etc.), information relative to automatic reconfiguration, illegal instruction detection, execution of instructions in read/write memory, etc.

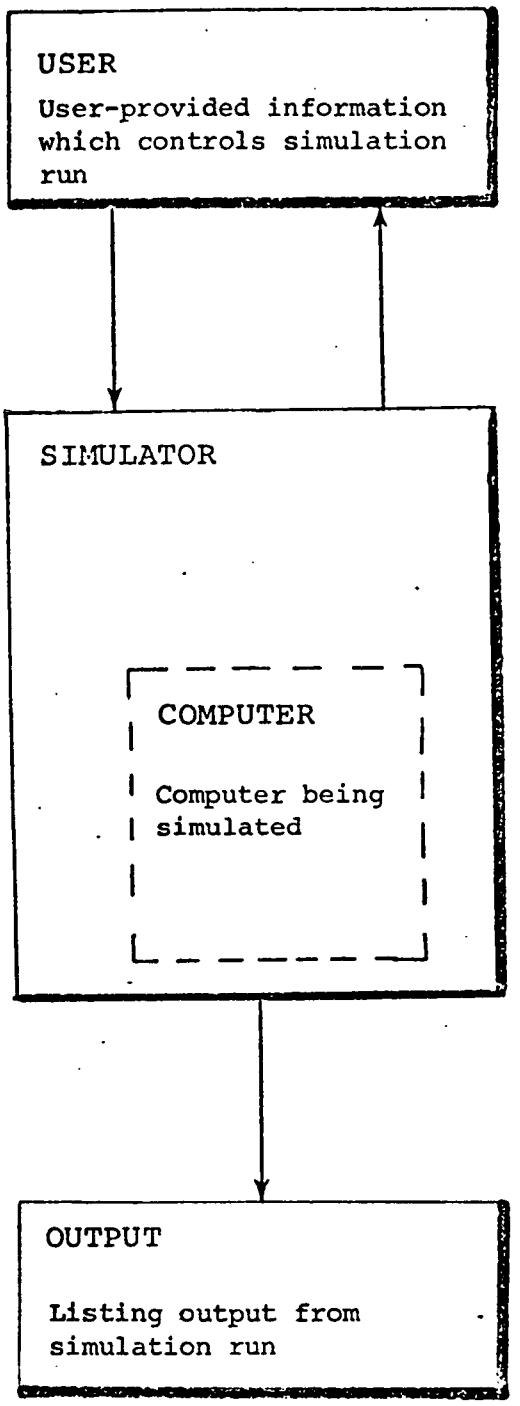


Figure 17.2: Basic Simulator: Input, Simulator, Output

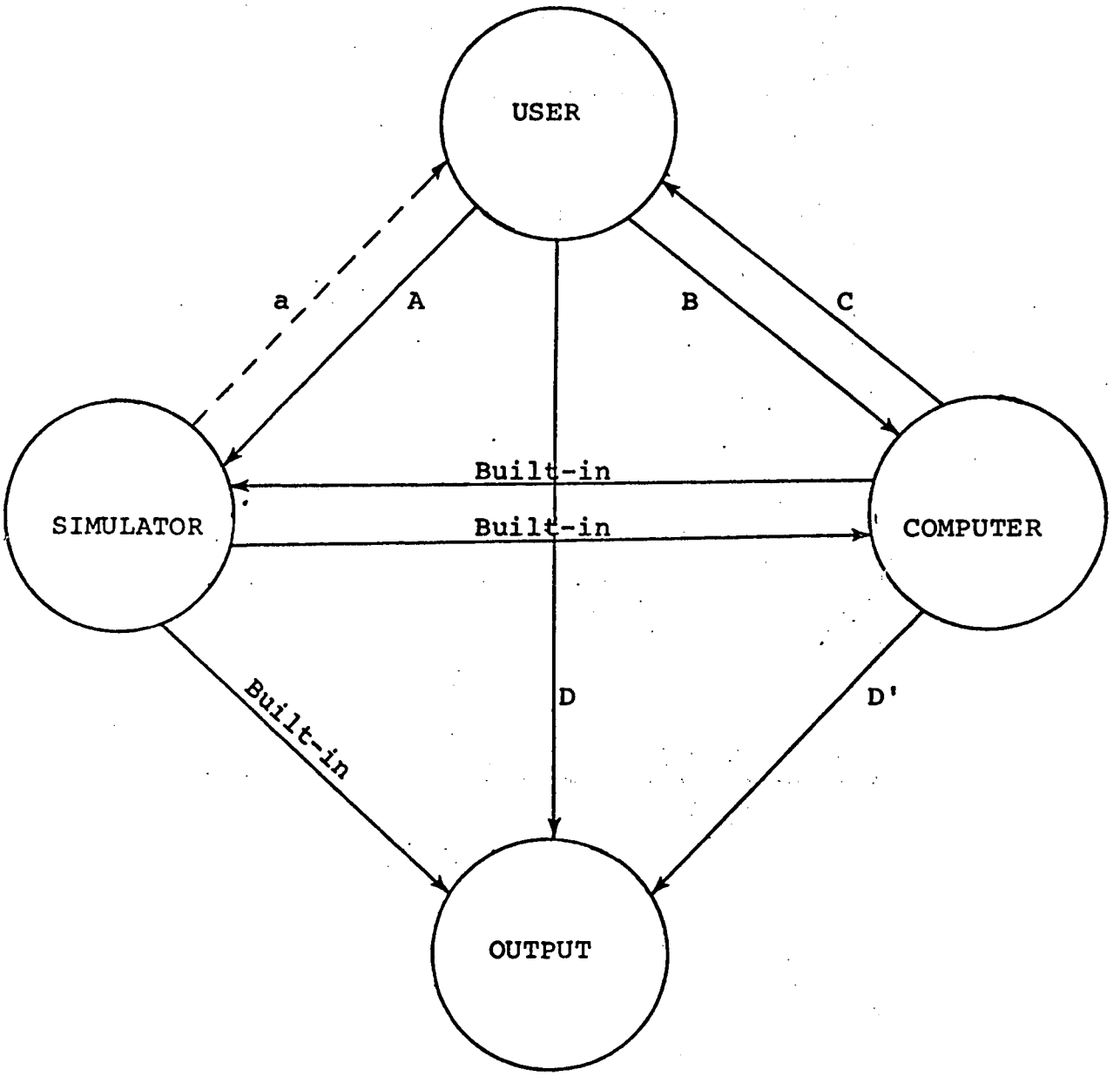


Figure 17.3: Simulator Logical Partitions

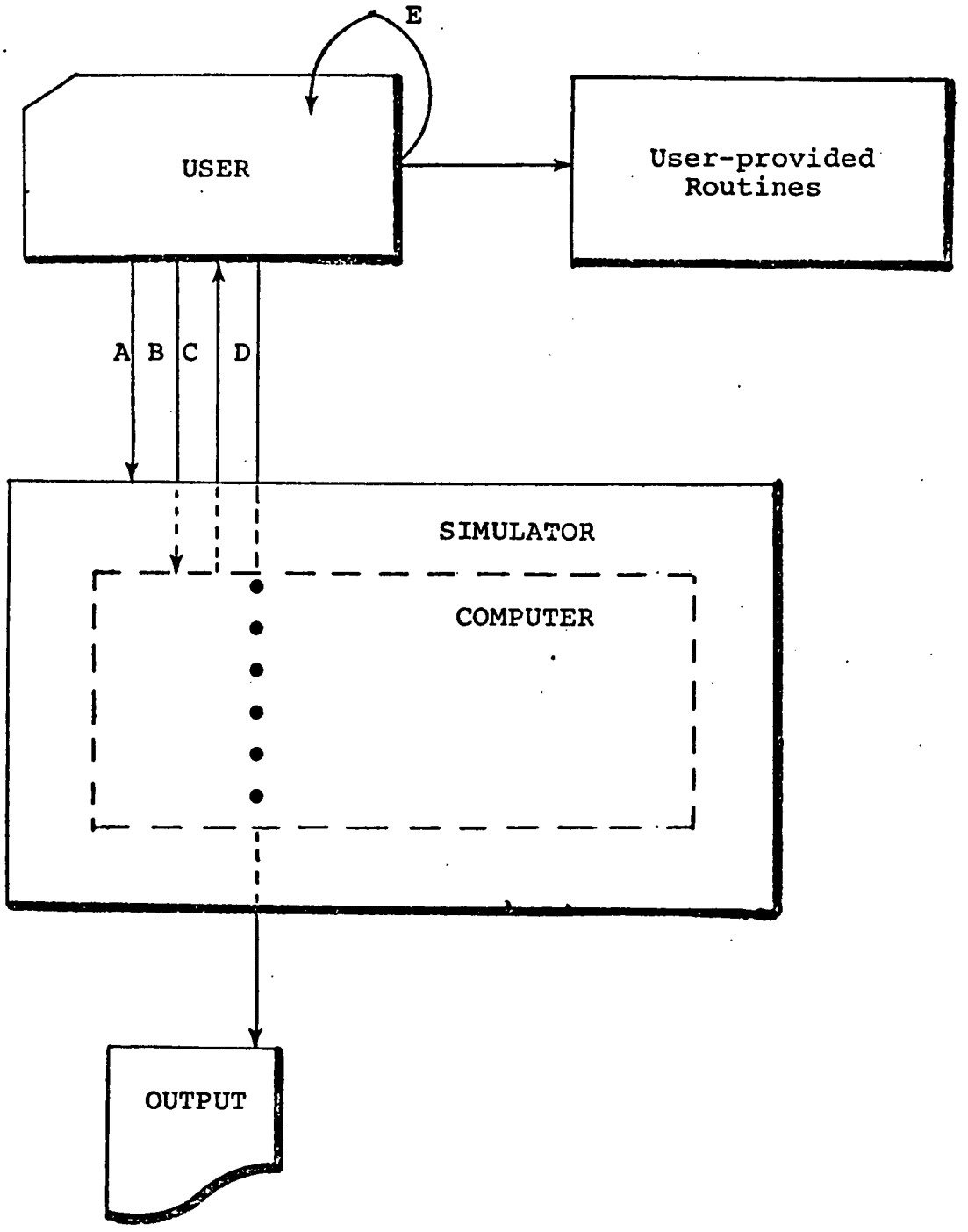


Figure 17.4: Simulator Physical Control Flow

The primary control which the USER specifies follows path B. By this path, and the return path C, the USER will be capable of ordering entry to routines which he provides, upon the occurrence of events or situations he specifies. The trigger-directives can include time conditions, location reference (instruction or operand access), and state changes (I/O, interrupt, hardware error detection signals, etc.). Once his routines have been entered as a consequence of a trigger directive, the USER is capable of accessing all locations, registers, states, and conditions in the COMPUTER, and modifying them as he sees fit. Through an interface language the USER may implement actions based upon conditions of almost arbitrary complexity, by simply programming the testing of these conditions in his routines.

Control paths D and D' provide information for OUTPUT, such as trace, flow-trace (output produced by branches only), interrupt-occurrences, faults, or output directly from the USER.

Information is not required on path a since the USER only interacts with the COMPUTER once the run starts and needs no interaction with the SIMULATOR. The fact that the SIMULATOR is actually doing all the work, including that of disguising the COMPUTER's physical absence, is only important to the designer, not the USER. Figure 4 shows that the COMPUTER is actually implemented within the SIMULATOR, and that the control paths to it actually interact via the SIMULATOR.

Path E of Figure 4 represents the closed-loop dynamic flow capability which the USER can exercise within his interface-language routines. These routines may in turn call routines prepared in other languages to perform further processing. Using external routines via this path allows the convenient addition of a data-recording capability to the system to allow post-run processing and the addition of almost any conceivable environmental model. The means by which this is achieved simply consists of providing library decks which specify the appropriate triggers and responses, and which perform calls to the associated environment in accordance with the environment interface. The kernel of the simulator needs no modification when a new or modified environment is attached. Thus, the required maintenance of the simulator is greatly reduced in scope.

17-3.2 Coding in a Higher Order Language

As we previously mentioned one of the questions that the system designer must face is whether to use a higher order language to code the simulator. Coding the simulator in a HOL offers advantages and disadvantages. It is especially advantageous in aiding the transferability of the simulator from one host computer to another. Should

an application require the simulator to be implemented on two different host computers, using a HOL minimizes the amount of recoding that must be done when transferring to the second machine.

A second advantage is the efficiency in producing the simulator coding that using a HOL can offer. MULTICS [5] experience has shown that a HOL can be very efficient in coding a large system. However, with a simulator as with most large systems there will be parts not well suited for coding in a HOL. Normally these are pieces of coding that must be executed very often. Any inefficient code here would lead to a needless waste in execution time. For example, coding the instruction execution parts of the simulator in assembly language would result in large execution time improvements over HOL coding. When the Apollo Block II simulator was coded for the Honeywell 1800, an AGC instruction, such as add, included the following steps:

a) AGC instruction fetch

b) Increment AGC instruction counter

c) Check for need to service an interrupt

d) Check for presence of diagnostic request

e) Interpret the op-code (which had variable length)

f) Interpret the operand address, checking for possible reference to AGC registers having special characteristics

g) Fetch the operand

h) Perform the AGC add (ones-complement, with end-around carry and overflow signal)

i) Increment the simulator clock

j) Check whether the next time-specified event was now due for processing.

Careful design and assembly language coding enabled this sequence to normally be executed in ten H1800 instructions. However, when using assembly language, the designer must remember that it leads to a more complex transferability problem. Thus, we see the HOL approach can offer advantages and disadvantages depending upon the part of the simulator in question.

17-3.3 Modularity

The use of modularity is another design issue that can be resolved to good advantage in building the simulator. By organizing the digital computer simulator and environmental model in different program modules, the former can interface with several models of varying complexity. Very often a programmer does not need a sophisticated environment in testing parts of his programs. Using a simple environment whenever possible reduces the high execution times a complex model would incur.

This reduction in turn, leads to a less expensive software production cost. Then when verification demands a full environment, modularity allows a complex model to be interfaced with the computer simulator as easily as a simple model. The additional advantages of modularity are:

a) It is easier to use existing modules in the simulator without having to duplicate coding. For example, system output writers can be interfaced with the simulator.

b) It is easier to expand the simulator. Building a more complex environment or computer simulator can be done independently of the other modules.

c) It allows utilization of structured programming techniques. A topic we will discuss later.

17-4.0 The Software Design Process

The microsimulator is an important system design aid. By properly integrating it into the system design effort it can aid the development and verification of the software system. We may divide the system design into four phases [6] as shown in Figure 17.5.

A. The software requirements are generated in Phase 1. These requirements include analytic formulations, interface specifications, and expected system performance results. An environmental model of the subsystems, vehicle characteristics, etc., can demonstrate the analytic concepts and generate the results to be expected.

B. During Phase 2 the systems and application programs are designed, coded, and tested. These programs are locally tested using the environmental simulator.

C. The purpose of Phase 3 is to show the software meets the performance criteria of the software requirements generated in Phase 1. This is done by interpretive microsimulation of the airborne computer with an environmental model. Phase 3 is a feedback verification process in that any poor software performance with respect to the environment can lead to a modification of the Phase 1 and Phase 2 performance. Unsatisfactory performance can be due to a faulty original design, misinterpreted software requirements, and errors in coding. When Phase 3 is satisfactorily completed, the software requirements will have been met, the design concepts validated, and confidence established in the software implementation.

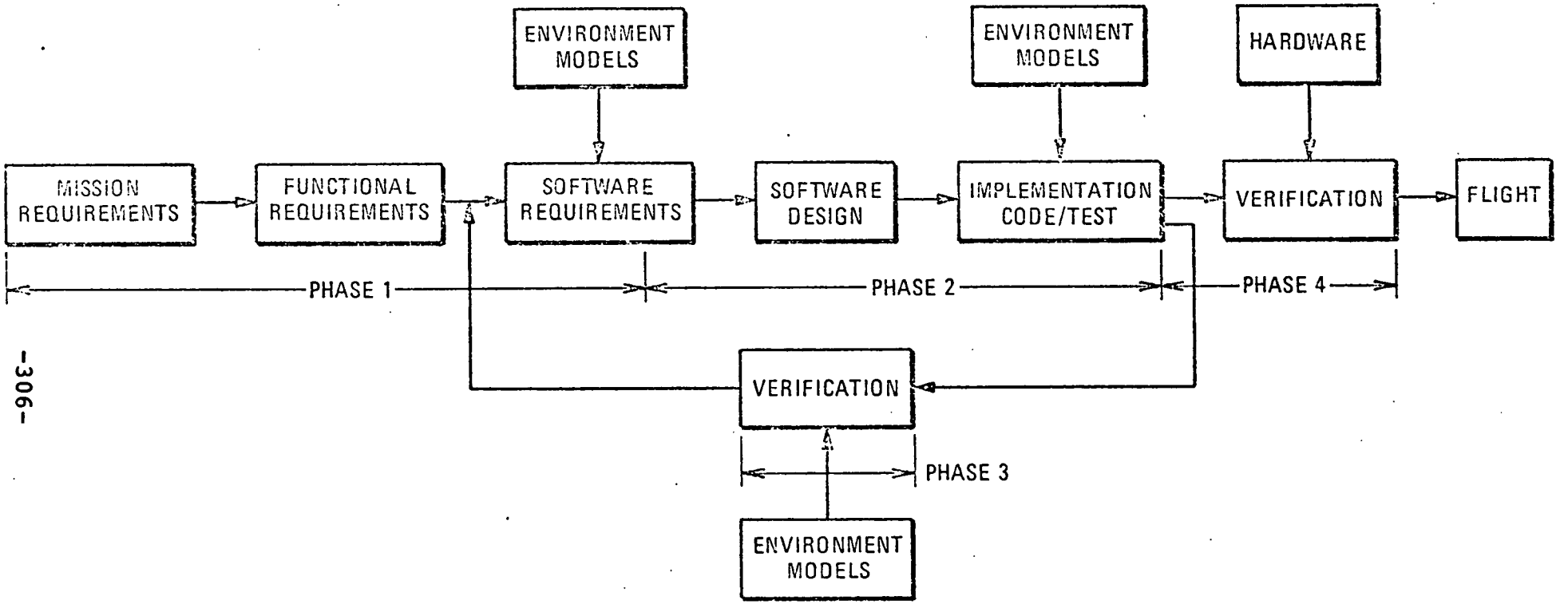


Figure 17.5: Software Development Phases

D. Phase 4 is a hardware verification phase. During this phase any hardware interface problems are corrected, and the performance of the entire system is demonstrated. We use real not simulated equipment, including flight computer, subsystems and interfaces. The software is taken from the realm of simulation to the realm of real equipment where design flaws can no longer be masked by model presumptions and inaccuracies.

   This phase helps establish the hardware/software integrity of the entire system and its ability to perform the real mission.

17-4.1 Overuse of Simulation

A word of caution at this point is in order. The software design process uses microsimulation for verification, but only as part of a well structured process. Before simulation is used, the analytical concepts are generated, and the software structured and coded. The proper emphasis on Phases 1 and 2 can prevent excessive use of the simulator. For example, Dijkstra [7] has taken a constructive approach to program correctness. Programs can be described in successive stages (top down structuring). At first a program can be specified in its most abstract form. Then successively more concepts used at one level can be refined in going to a lower level of design. The last level (the program module) leaves only the machine interface to be explained.

Such a systematic constructive approach to structuring software in Phase 2 can lead to a manifest reduction in what needs to be verified by microsimulation in Phase 3. In short, the system designer should not rely on microsimulation as his sole means of system verification.

17-5.0 Factors Influencing Simulation Speed

The speed of the microsimulator is an important concern of those defining system verification procedures. This speed is influenced by several factors.

A. We have already seen that the complexity of the environment is an important factor determinating simulator speed. During lunar landing simulations 75% of the Apollo simulation time was consumed in updating the environment.

B. The number of instructions executed on the host computer for each flight computer instruction is another factor. This figure often depends upon the architectural differences between the two computers in terms of instruction set, word length, etc. The average is 10-25 instructions [6].

C. The speed of the host computer and flight computer also influence simulator speed. However, this ratio of speeds is only important during a duty cycle when the simulator is executing flight compter instructions. During idle periods of the flight computer, the simulation can be advanced through these periods to improve overall speed. This capability of advancing simulation is an important feature that should be part of the simulator.

D. The number of diagnostic aids that the user requests has a heavy effect upon simulation speed. Full instruction traces, for example, can cause an order of magnitude increase in simulation time.

17-6.0 Simulation and Microprogramming

Traditionally, digital computers have been designed with fixed instruction sets regardless of their intended use. The recent use of microprogramming [8,9] in digital computer design now enables building machines whose instruction sets can be tailored to a specific application. The instructions chosen can be optimized in terms of economic coding and quick execution of the intended algorithms.

The system designer can use microsimulation as an aid in this optimization process. To do this the computer simulation must be carried down to the microinstruction level. Combinations of microinstructions can then be simulated to indicate to the designers which instruction organizations will be the most powerful. Kleir and Ramamoorthy [10] have recently presented a survey of optimization strategies.

One possible approach is to optimize by designing an instruction set for the direct execution of a higher order language. Foster [11] indicates that a 30% reduction in memory and a 25% reduction in execution time is possible with this approach as opposed to conventional design. However, an increase in needed circuitry is also likely.

17-7.0 Advantages of Microsimulation

One may reasonably ask the advantages of using a microsimulator for software development rather than using an actual object computer. There are several advantages to the former approach.

a) Code for the object computer need not be altered with debugging software. The simulator provides

diagnostics which can be invoked by the unmodified flight code.

b) By coding the simulator in a reentrant fashion it can support multiple, simultaneous users. This fact increases the simulation facility's throughput.

c) It allows simultaneous hardware/software implementation of a specified system design.

d) The microsimulator allows stress testing for software evaluation. Reducing the speed of the object computer or the size of its memory can easily be done in a microsimulator.

e) Diagnosing software is easier with a simulator. Functions such as the coroner and interpretive traces reduce the debugging time needed by a programmer.

17-8.0 The Poseidon System

The Poseidon Guidance Computer serves as an example of a system whose overall design was aided by microsimulation [12]. Once the computer design and mission algorithms were specified, a microsimulator was built concurrent with hardware implementation of the computer and programming of the algorithms. Frequent contact and cooperation among the hardware and software designers enabled optimization of the overall design. On several occasions programmers discovered ways of implementing the Poseidon algorithms that enabled hardware savings in terms of not needing special registers which otherwise would have been included in the computer. These improved algorithms were then simulated to check their validity. On the other hand, hardware designers could help programmers by specially tailoring Poseidon instructions to their needs since this was a microprogrammed machine.

Once the algorithms were completely coded they were tested with closed loop mission simulations complete with environment. See Figures 17.6 and 17.7. The simulations presented a wide range of flight conditions so that the formulation or the algorithms and their coding was shown to be highly reliable. The success of this approach to system design is demonstrated by the accurate and efficient performance of the Poseidon Guidance System.

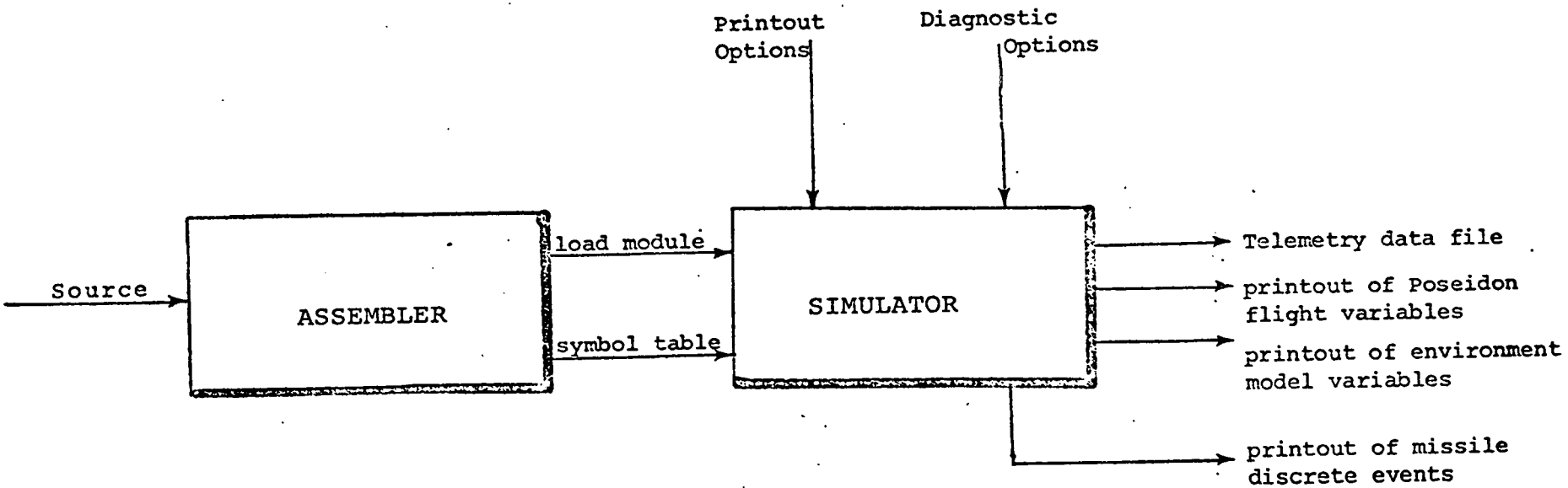


Figure 17.6: Poseidon Assembler and Simulator Configuration

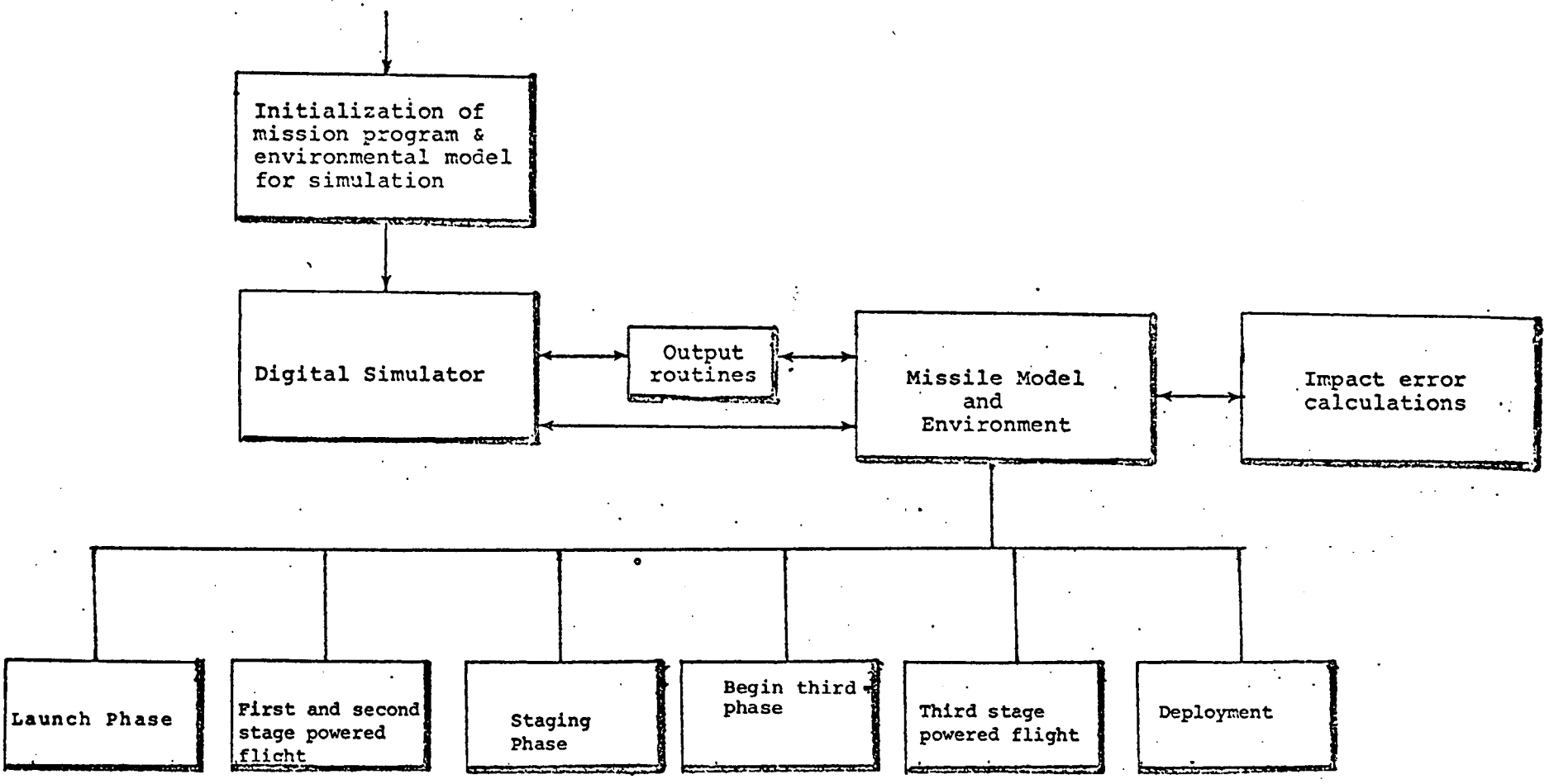


Figure 17.7: Poseidon Simulator Configuration

REFERENCES

1. Mimno, P., Digital Simulation Manual,(M.I.T. Draper Lab., Cambridge, Massachusetts, R-599, January 1968).

2. Chandy, K.M., and Ramamoorthy, C. V., "Rollback and Recovery Strategies for Computer Programs", (IEEE Trans. on Comp., C-21(6), June 1972) pp. 546-555.

3. Barron, D.W., Assemblers and Loaders, (American Elsevier, Inc., New York, 1969).

4. Donovan, J., System Programming, (Prentice Hall, New Jersey, 1972).

5. Graham, R.M.,"Use of High Level Languages for Systems Programming", (M.I.T. Project MAC Technical Memorandum 13, September 1970, AD-711-965).

6. Saponaro, J., Advanced Software Techniques for Data Management Systems, Vol. I: Study of Software Aspects of the Phase B Space Shuttle Avionics System, (Intermetrics, Inc., Cambridge, Mass., February 1972, Technical Report #12-72, prepared under Contract NAS 9-11778).

7. Dijkstra, E.W., "Structured Programming", in Software Engineering Techniques,(NATO Science Committee, Rome, Italy, October 1969) pp. 84-88.

8. Husson, S.S., Microprogramming: Principles and Practices, (Prentice Hall, New Jersey, 1971).

9. Davies, P.M., "Readings in Microprogramming", (IBM Sys. J., 11(1), 1972), pp. 16-40.

10. Kleir, R. L., and Ramamoorthy, C.V., "Optimization Strategies for Microprograms", (IEEE Trans. on Comp., C-20(7), July 1971), pp. 783-794.

11. Foster, J.R., "Development of a Higher Order Language Architecture", (NAECON '71 Record), pp. 201-205.

12. Pepe, J., The Structure of the PGC Simulator Program, (M.I.T Draper Lab., E-2435, August 1969).

CHAPTER 18

BENCHMARK PROGRAMS AS AN AID

TO COMPUTER PERFORMANCE EVALUATION

18-1.0 Introduction

Several methods and techniques have been used by the computer industry to evaluate and measure the performance of computer systems. Among these methods are: cycle time instruction mixes, kernal problems, benchmarks, synthetic programs, simulation and performance monitoring. No computer evaluation technique devised to date can guarantee selection of the one computer system which best represents the particular application requirements. There are some techniques that can be useful as part of the candidate system selection process.

Certainly, factors other than performance are of prime importance and usually dominate the selection procedure. These include credibility of the manufacture, experience with the computer in similar applications, off-the-shelf availability, undesirable features, space qualification costs, general technology and costs. All of these factors are part of the total selection process. The usual approach is to evaluate each of these factors, including performance for all candidate machines, and then to weigh each of the factors as to importance to the application and select the computer based on total evaluation. Computer performance therefore is only one of the factors and usually turns out to be between 5% - 15% of the total selection algorithm with factors of cost and credibility of the manufacture dominating the selection.

It is the purpose of this chapter to present a review of the methods used to evaluate system performance in selecting a computer and to discuss their advantages and limitations. Since aerospace computers and their environments are different than commercial installations, a higher language benchmark approach is described which may have use in comparing performance for large data management system applications such as the space shuttle or space station program.

Evaluation technique	Purpose of Evaluation					
	Selection evaluation (system exists elsewhere)		Performance projection (system does not yet exist)		Performance monitoring (system in operation)	
	New Hardware	New Software	Design New Hardware	Design new Software	Reconfigure Hardware	Change Software
—: Technique not applicable 1: Has been used but is inadequate 2: Provides some assistance but is insufficient; should be used in conjunction with other techniques 3: Satisfactory						
Timings	1	—	1	—	—	—
Mixes	1	—	1	—	—	—
Kernels	2	1	2	1	—	1
Models	2	1	2	1	2	—
Benchmarks	3	3	—	2	2	2
Synthetic Programs	3	3	2	2	2	2
Simulation	3	3	3	3	3	3
Monitor (hardware & software)	2	2	2	2	3	3

TABLE 1: SUITABILITY OF TECHNIQUES TO PURPOSES OF PERFORMANCE EVALUATION

18-2.0 Review of Computer Performance Evaluation Techniques

This section presents a review of the techniques which have been used in the past for measuring performance. Lucas [1] has categorized the various performance evaluation techniques as presented in Table 1 as to their applicability for evaluation of both existing and non-existing systems and for both software and hardware. This chart is based on a wide application of the systems, primarily commercial as opposed to special aerospace data management systems, and provides some general guidance for the usefulness of the various techniques. The rest of this section presents a description and discussion of these techniques with some illustrative examples.

18-2.1 Cycle and Add Time Comparisons

The simplest and earliest used computer performance evaluation technique was based on a comparison of cycle time and/or add time. The shorter the memory access time, cycle time, and add time, the higher performance rating for the computer.

This technique is still used very often as a gross measure of computer performance particularly in the military and aerospace computer industry. However, it is obviously not sufficient as a fine comparison of performance for several reasons:

a) the organization of the machine is ignored; (e.g., byte vs. word); varying word sizes are ignored;

b) special features, such as asynchronous properties of the architecture are ignored (e.g., instruction look-ahead);

c) it ignores differences in the address structure of the machine (e.g., single or double addresses).

For a modern machine no one instruction can be considered as an adequate evaluation of the hardware, and consequently any use of this approach should be limited to only "gross" level comparisons.

18-2.2 Instruction Mixes

A more refined application of add time comparison technique is the instruction mix. This is a selected frequency distribution of instruction classes on types which best represent the system applications or job stream. The performance P of a computer is then taken as

$$P = \sum_{r=1}^{v} N_r \, t_r$$

TABLE II - The Gibson III Mix

INSTRUCTIONS	UNIT WEIGHT
Move one word from main memory to the accumulator. Fixed Point	7
Floating Point	7
Store Accumulator	7
Move 500 words located contiguously in main memory to 500 other contiguous main memory locations. Average of.	3
Move 500 words located randomly in main memory to 500 contiguous main memory locations. Average of.	2
Conditional Branch Result Zero	6.5
Result Negative	6.5
Compare two words and set Indicator	
Fixed	3
Floating	3
Unconditional Branch	1
Fixed Point Mulitply(word) A x B→C. all in memory	6
Fixed Point Divide(word) A/B→C. all in memory	2
Fixed Point Add(word) A + B→C. all in memory	7
Shift one character(6 bits)	4.6
Logical AND or OR	1.7
Address Modification-Indexing	15.0
Address Modification-Indexing and Indirect Addressing	4.0
Address Modification-Indirect Addressing	19.0
Floating Point Add(word) A + B→C. all in memory	5.1
Floating Point Subtract(word) A - B→C. all in memory	5.1
Floating Point Multiply(word) A x B→C. all im memory	5.1
Floating Point Divide(word) A/B→C. all in memory	3.2
	135.1

TABLE III - A Commercial Mix

Compare	1 character	9
	2 characters	5
	3 characters	7
	6 characters	1
	10 characters	1
	12 characters	3
More	1 character	1
	10 characters	1
	60 characters	2
Branch	Taken	15
	not taken	13
Add	3 characters	1
Indexing		40

where t_r is the time taken to perform instruction type r, and N_r is the number of type r instructions to be performed.

Perhaps one of the best known of mixes is the Gibson mix; the origin of which is not known. There are at least three different types of Gibson mixes which have been readily used by equipment manufacturers for computer performance evaluation. The Gibson III mix illustrated in Table II has 24 categories or instruction types and represents a comprehensive list. Most mixes have only six to eight types of instruction requiring instructions to be averaged to obtain t_r.

Other instruction mixes available have been reported by Knight [2,3] and Arbuckle [4]; and a typical Commercial Mix is illustrated in Table III.

As a final instruction mix example, Table IV defines a real time radar tracking and missile control system. Table V provides a comparison of several aerospace computers using this instruction mix.

TABLE IV - REAL TIME RADAR TRACKING & MISSILE CONTROL MIX

INSTRUCTION TYPE	PERCENTAGE WEIGHT
1. Load acc with contects of memory location	25%
2. Store content of acc in memory location	14%
3. Shift	8%
4. a) Transfer unconditionally	10%
b) Transfer negative	1%
c) Transfer acc zero	1%
5. Exclusive OR operation Register to Register	7%
6. Move Register to Memory under control of bit mach	5%
7. Add	8%
8. Subtract	1%
9. Other	20%

TABLE V - COMPARISON OF AEROSPACE COMPUTER AVERAGE MIX TABLE3

COMPUTER SYSTEM	INSTRUCTION EXECUTION TIMES (ADD/MULTIPLY/DIVIDE)	WORD LENGTH	ESTIMATED SPEED ON MIX TABLE 3 KIPS	EST. MAX* PROCESSING SPEED
1. Hughes 4400	1.4/6.0/10.8	32	410-417	5-600
2. Litton 3070	2/8/8	32	587	4-500
3. CDC Alpha-1	2/9.7/17	32	447	350-450
4. RCA-215	3.3/7/16.5	32	305	2-300
5. Singer SKC-2000	4.1/8.5/8.5	32	168 (2.5 us memory)	200

*Single CPU

The instruction mix represents some improvement over a pure cycle time or add time comparison since it considers more instructions of the computer. However, there are several difficulties:

a) There is no simple means of determining weights or percentage of instructions in each class, which is representative of the application.

b) Generally mixes ignore any I/O or interrupts.

c) The full power of the instruction repertoire of a computer may be ignored by not being included in the mix.

d) There can be difficulties in using manufacturer specified instruction executiom times, particularly when instruction time is variable.

e) Some of the same factors ignored in the cycle/ add time comparison are also applicable to mixes. (e.g., word length, size ignored, etc.).

18-2.3 Benchmarks

As a general definition, a benchmark is a representative section of code executed or timed to give a measure of machine performance. For purposes of this report, three categories of Benchmarks are identified and termed: Kernel problems, Existing Programs, and Synthetic Programs.

18-2.3.1 Kernel Problems

A kernel problem is an algorithm which is coded for the machine and is timed. The timing is generally based on manufacturer specified execution time. Kernel problems are generally simple algorithms which are representative or typical of the application and differ from the other benchmarks in that they do not include general I/O or executive interfaces.

Some examples of kernel problems which have been used are polynomial evaluations, matrix inversion routines, table lookup, evaluation of special formulas, etc. Auerbach Corp. has developed a series of kernel problems which have been used to illustrate computer performance. These include sort problems, matrix inversion and others. Table VI is included to illustrate the result of an Auerbach matrix inversion kernel problem as operated on a Honeywell 632 computer.

MATRIX INVERSION

Standard Problem Estimates

Basic parameters: . . . general, non-symmetric matrices, using floating-point to 6.2 decimal-digit precision.

Timing basis: . . . . . using estimating procedure outlined in Users' Guide 5:200.312.

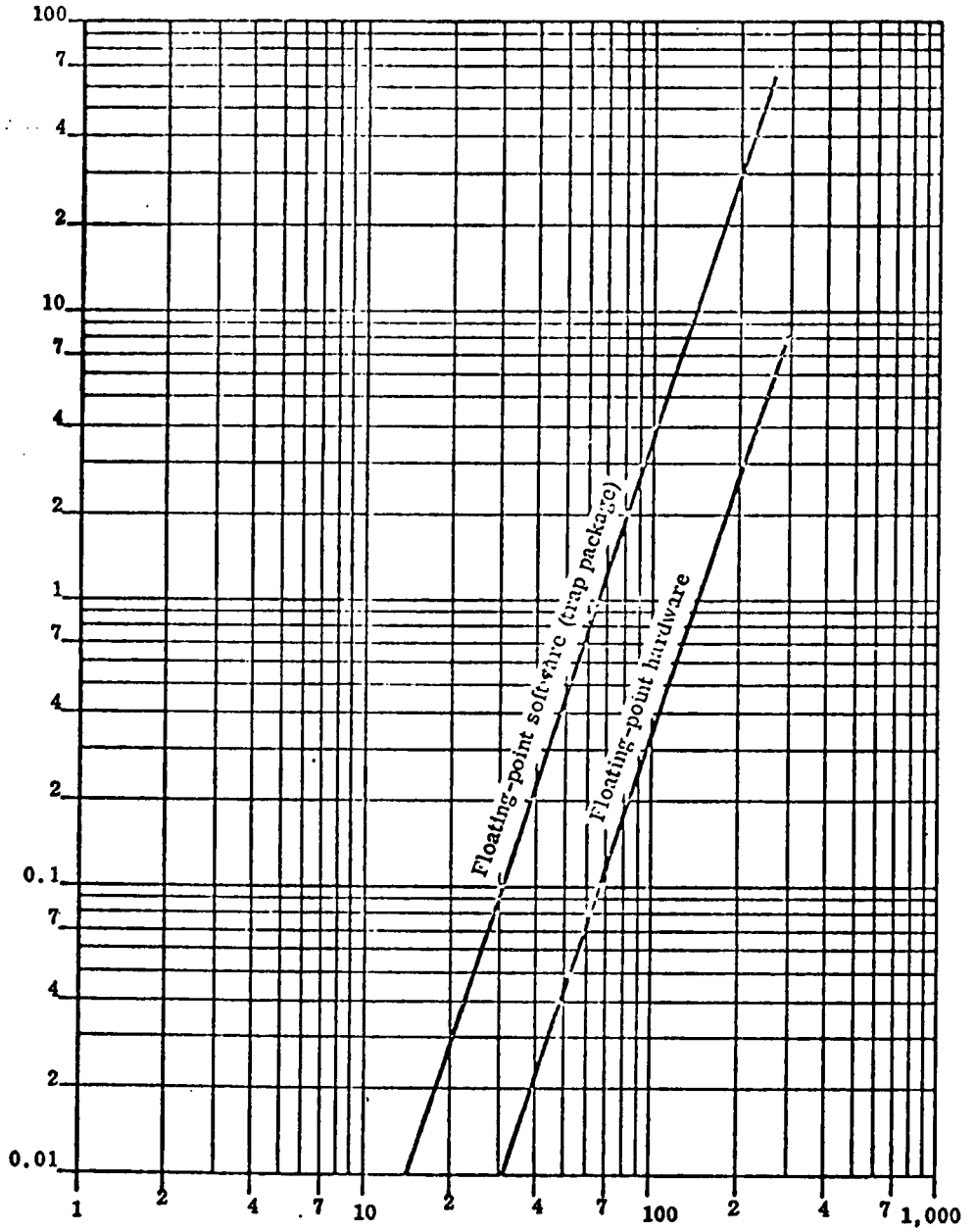


TABLE VI - AUERBACH KERNEL PROBLEM BENCHMARK

This type of benchmark has an advantage over mixes in that, although it is machine independent, it utilizes more instructions of the machine and can employ special features of the machine, usually ignored in a mix, to solve the problem. However, it does not generally include I/O or executive interfaces or multiprogramming considerations. Also, architectural factors in word size, etc., may still be ignored. Finally, it is not completely representative of the actual job stream. Although kernels can be combined to reflect the job stream, they are generally useful only as a relative measure of computer performance.

18-2.3.2 Existing Programs

This type of benchmark uses programs which exist (usually coded in a higher order language, such as COBOL, FORTRAN, etc.), and which can be compiled, executed and timed on the machine under evaluation. Since compilers for these languages exist for many computer systems, it is an approach often used in commercial computer acquisitions. The approach, however, contains both a measure of compiler and computer efficiency. This presents the difficulty that the performance measure for the power of the machine is potentially influenced by the performance of the compiler. (I.e., a machine with an efficient FORTRAN compiler can stand out). However, an approach to assess this difference is to code some of the routines in machine language and compare these to the compiled output of the same routine. Also, assessments of the multiprogramming aspects of the operating systems can be made by operating the benchmark programs first in series and then under the full multiprogramming features of the operating system for the computer.

18-2.3.3 Synthetic Benchmark Programs

A synthetic program is a type of benchmark in which an application program is coded for execution in the environment of the machine. It is coded with I/O and interfaces to the operating or executive system environment. It can be coded in either a higher level language or in machine language. In this sense it combines the features of both kernel problems and benchmarks.

The advantage of this type of benchmark is that the program can be prepared to reflect the actual application or to achieve measures of performance. Its major disadvantages are the costs associated with development of the benchmark and with its execution.

An additional disadvantage is that in aerospace computers the operating environment and executive system are not usually known beforehand and consequently assumptions must be made about these interfaces. If necessary, these may even be partially added to a part of the benchmark for execution.

18-2.4 Simulation

The most powerful technique for evaluation of computer system performance is through simulation. Simulation can be used at varying levels in evaluation of computer system performance. It is flexible and enables both microscopic and macroscopic evaluation of computer performance as discussed in previous chapters of this report.

The major drawback of simulation can be its cost. A relatively high development cost can be encountered depending on the scale of the simulation development and availability of existing tools.

Since Simulation has already been discussed in Chapters 6 and 17, it will not be discussed in detail in this chapter.

18-2.5 Performance Monitoring

This technique is used to collect information on the operation of a system for purposes of performance evaluation via hardware or software. It is generally used in conjunction with an existing system to determine bottlenecks in throughput or to evaluate change in hardware or software. Since this approach is discussed in Chapter 10 of this report, it will not be discussed in detail in this chapter. In addition, it is probably not an acceptable technique for aerospace computer selection since it requires the execution of the system.

18-3.0 Review of the Problems of Evaluation Techniques

Ideally the comparison of two candidate machines could be accomplished after the fact by proceeding to do the job on both, and then determining which one would perform better with respect to cost and execution time. Of course this is not practical. At the other extreme, people try to evaluate the instruction set of a machine by postulating some mix of instruction types and then evaluate the machine's execution time and memory

needs based on these instruction types. Memory tends to be the major cost in the hardware of a computer, and hence memory size often becomes the measure of system cost. Unfortunately, every machine has a different architecture, and a single postulated job mix becomes meaningless. What is really desired is to know how the machine performs when doing useful work and how well it does it relative to other machines.

18-3.1 Benchmark Programs and Problems

Often benchmark programs have been devised for comparative analysis, but they are seldom representative. They usually consist of a relatively simple set of routines that do some well-defined tasks such as matrix multiply, sort, etc., but these ignore the real characteristics of a job's execution and are inadequate. It is more important to know how the machine executes programs in the application environment. Subroutine calling and exiting, saving of special index registers, linking conventions, and addressing are of interest, but they are important only to the degree that they are utilized in the execution of actual programs.

The approach of using benchmarks is often followed in selecting a computing system by a general purpose commercial computer facility. This is aided by the widespread use of higher order languages. If Cobol or Fortran programs exist that are "representative of the daily workload", they are compiled on the other machine and relative comparisons made. The software (i.e., the compiler) as well as the hardware is tested in this fashion. It is only the success of the combinations of both that can produce good results and merits the ranking. It can be argued therefore that fair and reasonable overall conclusions may be obtained.

This approach, however, does not seem to be applicable in the case of aerospace data management computers. One of the primary reasons is a lack of compilers for aerospace computers. However, this does not rule out the possibility of a comparison of computers with respect to their performance in a compiler generated environment.

18-3.2 Cost of Determining Performance Evaluation

Another significant aspect of the performance measurement problem is the cost associated with determining computer performance. That is the time of programmers to devise and code kernel problems or benchmark programs and to collect and present the performance measurement results can be expensive. However, the costs must be evaluated in terms of the total acquisition costs. That is, if a single $40k computer for a single application is being purchased, then spending $100k in evaluating candidate

systems performance is not reasonable. If, on the other hand 50-100 computers each $150-300k are being purchased over a five-year period with $25 million of operational software, it definitely is appropriate to spend a reasonable amount of dollars in evaluating various candidate computer systems.

18-4.0 Higher Order Language Benchmarks for Aerospace Data Management Systems

Two higher order language benchmarks techniques are suggested as approaches to evaluating performance of several candidate computers for a particular aerospace Data Management System. By utilizing a higher order language more representative code of the application can be generated at lower cost and in a machine independent fashion. In other words, a more comprehensive benchmark can be obtained.

Both approaches utilize a hand translation of statements into the various candidate machines to obtain run time and memory useage for each machine. In addition, both approaches obtain benchmark statistics by execution of the benchmark in a general puspose machine for which there is a compiler. Both static distribution of statements (i.e., the number of times a statement exists in the program) and dynamic distribution of statements (i.e., the number of times statements are executed), are collected on the general purpose machine. The difference in these approaches is in the use of these statistics as described below.

18-4.1 Hand Compiled HOL Benchmarks

One approach to obtaining computer benchmarks involves creation of a pseudo-compiler for each machine, doing a hand compilation on representative programs, and then examining the efficiency in terms of memory and timing of the resultant code. This method also eliminates one source of discrepancy, the bagaries of the individual compiler writers and their chosen techniques. Since the same people and the same techniques would be used on all of the compilers, it can be argued that the results would be a fair measure of each machine's capabilities.

This approach could be as follows:

a) Data Management application software (guidance, navigation, checkout, etc.) that seems representative, will be coded in a HOL. Besides these real examples, other coding will be generated that is weighted by the statistics such as have been gathered by

Wichmann [5] and Knuth [6] in their surveys of existing source code.

b) The code will be compiled and executed on a commercial machine for which the HOL exists in order to authenticate this code.

c) A run time environment will be postulated for each candidate computer system with enough detail to define the working environment. For instance, if the machine has a "general register" set, then some registers will be accumulators, some will be base registers, and at least one will be a pointer.

d) A mechanical translation policy will be developed to transform the "intermediate code" of the HOL into equivalent assembly language statements for each computer. This need be done only for the various HOL constructs that are used in the benchmark programs and not for all the possible coded statements.

e) A manual translation of the actual intermediate language will be performed.

f) Statistics will be obtained from the translated code. Size data can be gathered by direct examination of the resultant code. Speed information can be inferred by counting instructions as they would be executed and by using the manufacturer's supplied data regarding machine instruction times.

g) Summary tables and conclusions will be drawn concerning the experiments. In addition to the data produced, both the good and the weak points of the individual computers uncovered during the translation process, including subjective evaluations concerning the suitability of the various computers, should be analyzed.

18-4.1.1 Sample FORTRAN Benchmark

The following is a simple example of a 3-line FORTRAN benchmark. For purposes of illustration, the statement of the example has been kept small since more complicated or realistic examples should include sequences of functions, subroutine calls, I/O, etc.

The only unusual aspect to the sample is the frequency of usage of formal parameters or dummy arguments as they are called in FORTRAN. However, they were intentionally chosen to emphasize the importance of efficient subroutine linkages and parameter handling. FORTRAN was chosen as the higher order language only because it was then possible to hand compile the statements on several computers. In fact, the model of the run time environment that was assumed was more dynamic than is customery for FORTRAN. The assumption was made that a base register or index register was required in order to access even local variables as well as formal parameter pointers.

The example is a FORTRAN subroutine named BLA:

```
SUBROUTINE BLA(A,B,C)...
DIMENSION A(6),X(6).....
1=I+1...................
A(I)=A(I+3)-X(I+4)......
B=C-2.0*A(5)+3.0........
RETURN
END
```

The benchmark was compiled or hand coded for nine computers. It was compiled on both the IBM 360 and PDP-10 computers using the Fortran G Compiler and hand coded for several Flight computers:

IBM 4Pi AP
UNIVAC 1832
SINGER SKC-2000
Autonetics D-216
Control Data CDC ALPHA

Two other non-flight computers were also hand coded: the PDP-11 which is a modern minicomputer with an interesting architecture and the Burroughs 6700 which is a stack oriented machine designed to execute a higher order language.

Since the benchmark is very small, no comprehensive statements on performance can easily be made. However, the following comments should be made:

1) Since the calculations were simple, no use was made of multiple accumulators and consequently those computers have not had their full computational power expressed in the result.

2) A dynamic addressive environment was assumed in the benchmark as expected in a real time aerospace environment. Consequently the availability of both base registers and either indexing or indirect addressing features were favored.

3) It is worth noting that the Burrough 6700 with a polish stream and stack oriented instructions was coded rapidly and without optimization and concern for index registers and accumulator assignments.

TABLE VII - RESULTS OF FORTRAN BENCHMARK

COMPUTER	# OF INSTRUCTIONS	# OF 16-BIT WORDS	# OF CONSTANTS	TOTAL MEMORY	EST. EXECUTION TIME
IBM 360	16	31	8	39	16.06
DEC PDP-10	14	28*	0	28*	80.28
IBM AP-1	14	21	0	21	34.36
CONTROL DATA ALPHA	18	33	4	37	51.66
UNIVAC 1832	13	23	4	27	54.75
AUTONETICS D216	20	22	4	26	53.75
KEARFOTT SKC2000	16	20	4	24	62.38
DEC PDP-11	15	27	0	27	36.80
BURROUGHS 6700	30	22.5	0	22.5	

*18 bits is the word quantum in this case

18-4.2 Statistical Approach to HOL Benchmark

Since it can be very expensive to fully translate benchmark programs into several candidate machines on a large enough scale to obtain a "representative" sample of machine code, an approach is suggested in applying statistics of statement static and dynamic frequency as derived from a general purpose computer. With the dynamic frequency of occurrence of HOL "operations" for the particular application, it is then possible to obtain a relative measure of execution time for each machine.

This approach to evaluation is made by extending a method presented by B.A. Wichmann[9]. Briefly, his method consists of defining a representative set of statements of the HOL, (in his case Algol) and making a set of time measurements, T_{ij}, for each representative HOL statement i (i=1 to n) on machine j (j=1 to m).

He then models these measurements as:

$$T_{ij} = F_i \; S_j \; R_{ij} \qquad \begin{matrix} 1 \leq i \leq n \\ 1 \leq j \leq m \end{matrix}$$

where F_i is a measure of statement complexity, S_j is a measure of machine performance, and R_{ij} is a factor related to the machine's relative performance for a particular statement.

The assumption is that the execution time of a statement is somehow directly proportional to the "complexity" of the statement and to the "performance" of the particular machine. The R_{ij} is then a measure of how much the particular T_{ij} measurement varies from the ideal.

After obtaining the T_{ij} measurements, the next step is to use these mn values and to determine the m + n values for the F_i and S_j. This is a valuable approach if the postulated measurements T_{ij} are the only ones obtainable. However, the results are less than satisfying since the relative frequency of dynamic occurrence of the statements of the actual application is not taken into account. An extension of this approach is proposed as a more satisfying view of the problem of determination of statement complexity and machine performance.

Suppose a larger sample of the specific DMS application software were coded in the HOL. If these programs were executed on a commercial machine, under instrumentation which can observe the relative frequency of dynamic occurrence of each statement type w_i, then a more meaningful measure of machine performance (inthis case, slowness: P_j) is given by

$$\sum_{i=1}^{n} w_i \; T_{ij} = P_j \qquad 1 \leq j \leq m$$

The P-values are analogous to Wichmann's S-values, but are renamed to avoid confusion. These P-values are computed from the measured statement execution times on the j machines as defined by the matrix T_{ij} adjusted by the statement execution frequency estimation for the Shuttle application software.

In an analogous manner the relative measure of the memory utilization can be obtained. Let M_{ij} be the amount of memory needed to represent the HOL statement i, and the machine j. The static distribution of HOL statements can be obtained for the benchmark by counting the HOL constructs in the code. Define σ_i as the static distribution. Then a relative measure of memory efficiency can be obtained by

$$\sum_{i=1}^{n} \sigma_i M_{ij} = A_j$$

The A_j values are relative measures of the memory sufficient for each machine.

Since the P_j have been determined, the statement complexities C_i in the Wichmann equation can be written as:

$$T_{ij} = C_i P_j Q_{ij} \qquad \begin{matrix} 1 \leq i \leq n \\ 1 \leq j \leq m \end{matrix} \qquad (18.1)$$

where the Q_{ij} and C_i are related to the R_{ij} and F_i of the Wichmann equation. This is mn equations in n(m+1) unknowns. To obtain a "best fit", we chose to minimize the variation of the Q_{ij} relative to the C_i, therefore define:

$$E = \sum_{ij} \sum (LQ_{ij})^2 = \sum_{ij} \sum (LC_i + LP_j - LT_{ij})^2$$

where the prefix L on a variable indicates the logarithm of that variable. This leads to

$$LC_i = \frac{1}{m} \sum_{j} (LT_{ij} - LP_j) \quad ,$$

and the Q_{ij} may then be computed from (18.1).

The interpretation to be placed upon the Q_{ij} values is that they reflect the inefficiency of machine j executing statement-type i, relative to how that machine executes other statement-types, independent of the statement-complexity and frequency of execution.

The values Q_{ij} then allow for an understanding of the structure of the machine with respect to the HOL. This would allow insight as to the ability of the machine to carry out particular functions not specifically considered in the weighting of the HOL statements.

Winchmann [5] used over forty different statements to compare each compiler. The statement classes used for comparison should be generated from two different considerations:

1) the different functions of the HOL: scalar operations, array operators, flow control within a program, modularization, input/output; and

2) those features which tend to differ in each machine architecture: integer operations versus floating point operations, the use of literals, both short and long, and of different precisions, etc.

Once these classes of representative operations have been obtained, only their implementation on the various machines need be considered. If a HOL compiler exists, then by actually executing the "translation" of the aerospace computer's implementation of the HOL statements, a time measurement can be made for each operation; similarly a memory space measurement can be made by examining the instructions. Since a compiler may not exist, a manual translation, as in the first method, can be performed.

The dynamic and static frequency of occurrence of these "HOL statements" for the benchmark are determined once and can be accomplished via execution on a machine which has a HOL compiler.

This method has the great advantage of being able to vary the postulated HOL statement mixes to determine how the relative merit of the machines changes.

18-4.3 Problems with the HOL Benchmark Approach

While these methods can obtain both an execution time measurement and a memory size measurement, it does not give the "complete picture".

1) The measurements used to obtain the execution time of the HOL statements on a given machine introduce several inaccuracies.

a) There is an assumption about the method by which the HOL translates into the machine language. This may not be accurate, particularly in the context of more complex statements than the representative HOL statement. Or conversely, perhaps the computer cannot generate as compact code as the assumed translation.

b) The execution time of the translated HOL statement is very difficult to obtain. If a HOL compiler was used, time must be "measured" on the object machine itself and usually the machine's timing mechanism has very large granularity, if the measurement is possible at all. Otherwise, some external clock would have to be used.

c) The last point indicates that one method to obtain more accurate measurements would be to embed the desired statement in a DO LOOP for a million executions. This has the bad side effect of creating a static environment for the given HOL statement. Even though it is being executed one million times, this is not necessarily equivalent to the presence of the statement one million times in a real dynamic environment where each occurrence would be from other, and different HOL statements. This criticism is equally valid of the manual translation approach.

2) The HOL statements cannot sufficiently take into consideration input/output. The interrelation of asynchronous computations in a multiplexed environment depends highly on the physical hardware characteristics, how they are interconnected, and upon the executive systems.

These "real time" problems which are characteristics of aerospace data management applications can be evaluated utilizing queueing theory approaches or other modeling and macrosimulation techniques [7] for evaluating throughput as discussed in Chapters 5 and 6.

REFERENCES

1. Lucas, H.C., "Performance Evaluation and Monitoring", (Computing Surveys, 3(3), Sept. 1971), pp. 79-91.

2. Knight, K.E., "Changes in Computer Performance", (Datamation, Sept. 1966), pp. 40-54.

3. Knight, K.E., "Evolving Computer Performance 1963-1967", (Datamation, Jan. 1968), pp.31-35.

4. Arbuckle, R.A., "Computer Analysis and Throughput Evaluation", (Computers and Automation, Jan. 1966).

5. Wichmann, B.A., A Comparison of ALGOL 60 Execution Speeds , (CCU Report #3, National Physical Laboratory, Tedington Middlesex Englang).

6. Knuth, D.E., An Empiracal Study of FORTRAN Programs , (Computer Science Dept., Report CS-186, Stanford Univ., Stanford, California, AD-715-513).

7. Stimler, S., and Brons, K.A., "A Methodology for Calculating and Optimizing Real Time System Performance", (CACM 11(7), July 1968), pp.509-516.

8. Bailey, W.O., "The Processor Figure of Merit", (Honeywell Computer Journal 5(4), 1971), pp. 201-204.

9. Smith, J.M., "A Review and Comparison of Certain Methods of Computer Performance Evaluation", (The Computer Bulletin, May 1968), pp. 13-18.

10. Ashley, D.W., A Methodology for Large Systems Performance Predictions , (IBM Report TR 00-1173, Sept. 1968).

11. Staudhanmer, J., Combs, C.A., and Wilkinsen, G., "Analysis of Computer Peripheral Interference Proceedings", (ACM National Meeting, 1967).

PART V

FACILITIES

CHAPTER 19

FACILITIES NEEDED FOR SYSTEM ANALYSIS

19-1.0 Introduction

We will now identify the facilities needed by a system analysis group to perform their job of designing, implementing, and testing the hardware and software of an aerospace data management system. Previously, in Chapter 17 we saw software system development divided into four phases. We will now expand these phases to a more general description of the development of system hardware and software. From an analysis of these development phases we will see the facilities that the designers need to aid them in their task.

19-2.0 System Design Phases

There are at least seven phases in the development of an aerospace data management system. These are:

1. Mission Requirements Phase: In this phase the overall goals and objectives of the mission are defined.

2. System Requirements Phase: In this phase the mission requirements are analyzed to determine the type of systems necessary to support the mission. An important part of this phase is reliability analysis to determine the type of redundancy necessary to enable successful system operation for the duration of the mission. In addition, software algorithms are also generated in this phase to help estimate the complexity of the needed systems.

3. System Analysis Phase: System problems are further studied in this phase with design aids such as macrosimulation, Markov analysis, Queueing theory, and running benchmark programs. The resulting data then enable design questions such as size of core memory, single processor vs. multiprocessor, hardware/software tradeoffs to be answered.

4. System Specification Phase: With all these design questions and tradeoffs answered, the system needed can be specified. That is, a detailed design can now be generated.

5. Implementation Phase: In this phase the system is built. Hardware is constructed, and software coded.

6. Test Phase: The system must now be tested for proper operation. Aids such as digital simulators, hybrid simulators and performance evaluation monitoring hardware are all useful in this very important phase of system design. Thorough testing of the system is necessary to insure the success of the mission and the safety of any crew-members.

7. Operation Phase: The system is now ready to be used in mission operations.

19-3.0 Needed Facilities

The design phases we discussed above show that at least three facilities are necessary for the system designer. These are the digital computer, the hybrid laboratory, and a microprogrammable computer. We will now discuss these facilities in more detail and give some interesting Apollo development examples.

19-3.1 The Digital Computer Facility

In this volume we have discussed many system design techniques that require the use of a digital computer for their application. For example, macrosimulation, Markov analysis, and reliability analysis all involve tedious computations that are best done by a digital computer. Thus, the system designer should have access to a computer either directly or by means of a remote terminal. The size of the needed computer depends heavily upon the complexity of the system being designed and the software tool being executed. For example, Apollo digital simulations were run on an IBM System/360, model 75, and required hours of execution time even on this powerful machine. However, less complex systems could of course be analyzed and simulated on smaller computers. Poseidon digital simulations required much less execution time on the 360, model 75, than Apollo and, hence, could have been conveniently run on a smaller machine, such as a 360, model 50. In any case, we recognize the fact that the digital computer is an essential system design facility.

19-3.2 The Hybrid Computer Facility

A facility that has been almost indispensible to system

designers in aerospace applications is the hybrid computer. The hybrid computer allows a mixed digital/analog (hybrid) simulation of a space mission so that the design of the overall system can be tested. An all digital simulation is sometimes not as efficient as a combined analog/digital approach because certain problems are more readily solved with analog hardware. For example, in simulating the performance of a space vehicle, such as the Apollo Command Module, the rotational dynamics equations of the vehicle require high speed solutions, but extremely high precision is unnecessary. This portion of the simulation is best handled by analog circuitry. On the other hand, the vehicle's trajectory calculations require high precision but are run at a lower rate. Thus, digital simulation is appropriate here. Communication between the two portions of the simulation can be performed with control lines, interrupt lines, A to D and D to A converter channels. Such a linkage between the analog and the digital simulations allows a hybrid simulation of the space vehicle's performance.

19-3.2.1 Types of Hybrid Computers

There are many types of hybrid computers; perhaps as many as there are applications. However, if the hybrid computer is classified by the way in which it combines the digital and analog hardware, we arrive at the following four categories [1, 2]:

a) an analog computer with added digital logic,

b) a digital computer with added analog components,

c) a digital computer and an analog computer linked with A to D and D to A channels, and

d) a hybrid computer in which neither the digital nor the analog portions are intended for independent use.

The hybrid simulation facilities for the Apollo mission are a variation of category (c). We will describe this facility as an example of the type of facility that the system designer can use to properly test his system's performance.

19-3.2.2 Apollo Hybrid Simulations

The Apollo hybrid simulation facility at the M.I.T. Draper Laboratory consisted of an analog computer representation of the spacecraft linked with certain flight hardware including the Apollo Guidance Computer (AGC) [3, 4]. Its purpose was to verify the AGC programs in a real-time simulation using flight hardware. Figure 19.1 shows a typical configuration, the Apollo 204 Command Module Entry Simulator.

Although flight software was primarily verified on an all digital simulator facility, the hybrid facility offered certain advantages. They are:

a) real-time operation,

b) manual access to the AGC during simulation, and

c) on-line monitoring of the simulation.

The major disadvantage is the difficulty in tracing an AGC program's execution.

The hybrid facility was essential in developing programs for the man-machine interface. That is, programs that accept keyboard input from the astronaut and output displays of requested data. The hybrid simulator has also been used as a training device for the Apollo GNC system. In fact, many astronauts received training on this facility.

19-3.2.3 An Additional Feature of the Hybrid Facility

The hybrid laboratory, in which an actual flight computer is included in a hybrid simulation, provides an excellent environment to monitor the computer system's performance. Since the system is running in real-time, additional hardware can be interfaced with the flight computer to measure the efficiency of its performance under approximate operating conditions. The collected data can then be used to optimize the design of the flight computer's hardware/software systems, as discussed in Chapter 10.

19-3.3 The Use of a Microprogrammable Computer

Another useful design aid is a microprogrammable computer. Such a machine can be microprogrammed to reflect various computer architectures, such as a HOL organization. The advantage of this fact is that several candidate flight computers can be evaluated merely by changing the microprogram of this computer. Running benchmark programs is a very useful aid in this evaluation.

When an architecture is finally chosen, this flight computer can be emulated on the microprogrammable computer. A digital simulator is then unnecessary because the microprogram will not only reflect the given instruction set, but it can contain diagnostic features for the debugging of flight software. Traces, traps, a coroner function, etc., can conveniently be provided in this microprogram. Thus, the testing of flight software can proceed without the extra burden of programming a digital simulator.

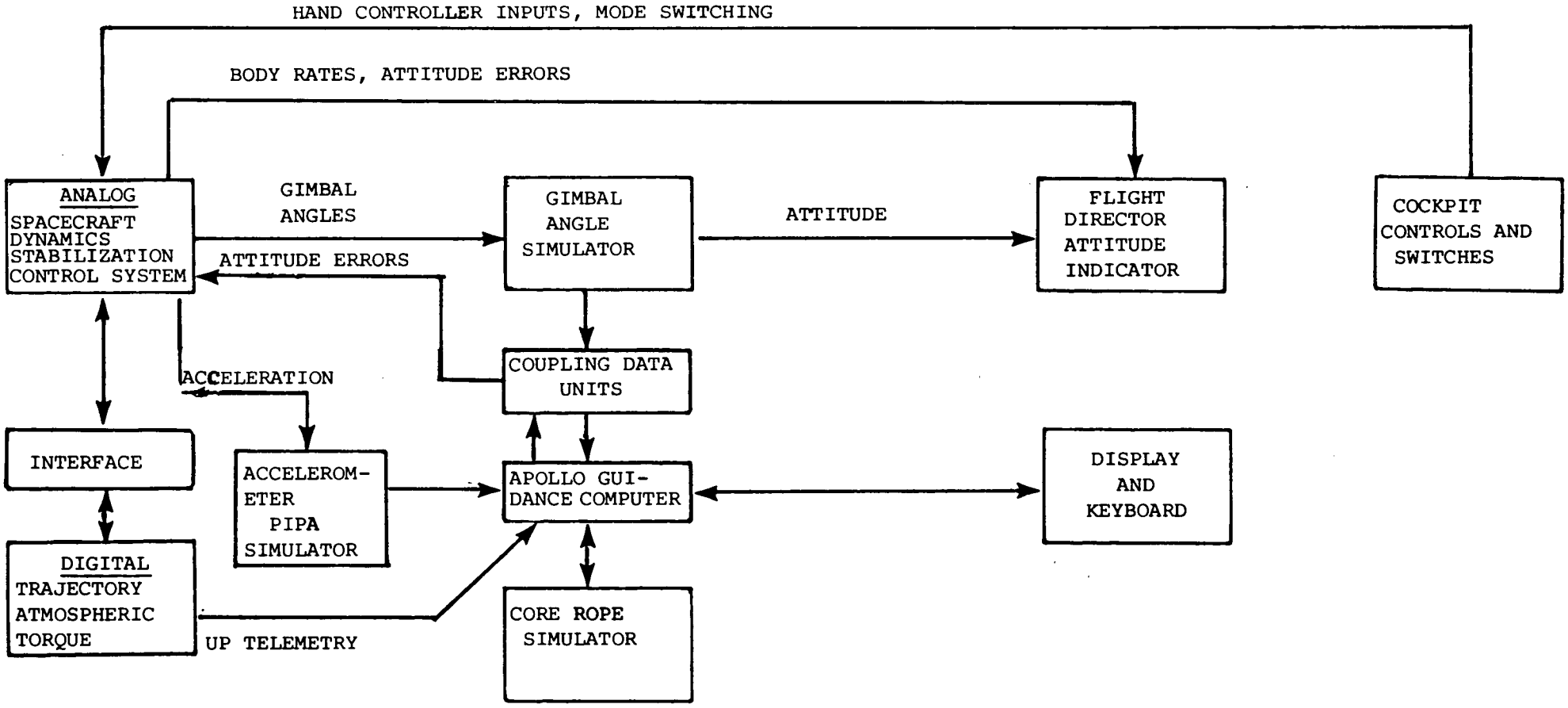


FIGURE 19.1: APOLLO 204 COMMAND MODULE ENTRY SIMULATOR CONFIGURATION

REFERENCES

1. Korn, G.A., and Korn, T.M., Electronic Analog and Hybrid Computers, (McGraw-Hill, New York, 1964).

2. Hagan, T.G., "Hybrid Computation", (Datamation, Oct. 1965), pp. 24-28.

3. Felleman, P.G., Hybrid Simulation of the Apollo Guidance Navigation and Control System, (M.I.T. Draper Lab, E-2066, Dec. 1966).

4. O'Conner, J.T., Hybrid Simulation of Apollo Missions for the Verification of Flight Software Crew Procedures, (M.I.T. Draper Lab., E-2531, Aug. 1970).

PART VI

APPENDICES

APPENDIX A

MATHEMATICAL BIBLIOGRAPHY

In developing a systems analysis group, one of the most important assets that the systems engineers and programmers should possess is a strong mathematical background. Many system design problems are very mathematical in nature, and those working on these problems should feel at ease with the mathematics. Presentations of purely mathematical topics to enhance the skills of a system analysis group are outside the scope of this contract. However, with a good bibliography an engineer or programmer can concentrate on those areas in which he feels he should increase his knowledge.

This appendix lists several fields of mathematics pertinent to systems engineering and programming, and within each field several references are given. It is by no means an exhaustive list, but is meant to direct an engineer or programmer to some of the standard references in the field.

A. General References in Applied Mathematics

These texts are encyclopedic in nature and contain sections on most of the more specific areas we will list below.

1. Courant, R., and Hilbert, D., Methods of Mathematical Physics, 2 Volumes, (John Wiley and Sons, New York, 1953).

2. Morse, P.M., and Feshbach, H., Methods of Theoretical Physics, 2 Volumes, (McGraw-Hill, New York, 1953).

3. Jeffreys, H.S., and Jeffreys, B.S., Methods of Mathematical Physics, 2nd Edition, (Cambridge University Press, 1950).

4. Pipes, L., and Harvill, L., Applied Mathematics for Engineers and Physicists, 3rd Edition, (McGraw-Hill, New York, 1970).

B. References in More Specific Mathematical Fields

1. Linear Algebra

   a. Hoffman, K., and Kunze, R., Linear Algebra, 2nd Edition, (Prentice Hall, New Jersey, 1971).

   b. Faddeeva, V.N., Computational Methods of Linear Algebra, (Dover, New York, 1959).

   c. Gel'fand, I.M., Lectures on Linear Algebra, (Interscience, New York, 1961).

   d. Jacobson, N., Lectures in Abstract Algebra, Volume II, Linear Algebra, (Van Nostrand, Princeton, New Jersey, 1960).

   e. Noble, B., Applied Linear Algebra, (Prentice-Hall, New Jersey, 1969).

2. Advanced Calculus

   a. Hildebrand, F., Advanced Calculus for Applications, (Prentice Hall, New Jersey, 1962).

   b. Franklin, P., Methods of Advanced Calculus, (McGraw-Hill, New York, 1944).

   c. Apostol, T., Mathematical Analysis, (Addison-Wesley, Reading, Mass., 1958).

   d. Buck, R.C., Advanced Calculus, 2nd Edition, (McGraw-Hill, New York, 1968).

   e. Rudin, W., Principles of Mathematical Analysis, 2nd Edition, (McGraw-Hill, New York, 1968).

3. Complex Analysis

a. Dettman, J.W., Applied Complex Variables, (MacMillan Co., New York, 1965).

b. Ahlfors, L., Complex Analysis, (McGraw-Hill, New York, 1953).

c. Carrier, G., et al., Functions of a Complex Variable, (McGraw-Hill, New York, 1966).

d. Churchill, R.V., Complex Variables and Applications, 2nd Edition, (McGraw-Hill, New York, 1960).

4. Linear Programming

a. Garvin, W.W., Introduction to Linear Programming, (McGraw-Hill, New York, 1960).

b. Hadley, G., Linear Programming, (Addison-Wesley, Reading, Mass., 1962).

c. IBM Corporation, An Introduction to Linear Programming, (New York, 1964).

5. Probability and Statistics

a. Feller, W., An Introduction to Probability and Its Applications, 2 Volumes, (John Wiley and Sons, New York, 1960).

b. Fraser, D., Statistics-An Introduction, (John Wiley and Sons, New York, 1958).

c. Barlow, R., and Proschan, F., Mathematical Theory of Reliability, (John Wiley and Sons, New York, 1965).

d. Parzen, E., Modern Probability Theory and Its Applications, (John Wiley and Sons, New York, 1960).

e. Wadsworth, G., and Bryan, J., Introduction to Probability and Random Variables, (McGraw-Hill, New York, 1960).

6. Differential Equations

a. Birkhoff, G., and Rota, G., Ordinary Differential Equations, 2nd Edition, (Ginn and Co., Boston, 1969).

b. Martin, W., and Reissner, E., Elementary Differential Equations, 2nd Edition, (Addison-Wesley, Reading, Mass., 1961).

c. Kaplan, W., Ordinary Differential Equations, (Addison-Wesley, Reading, Mass., 1958)

d. Garabedian, P.R., Partial Differential Equations, (John Wiley and Sons, New York, 1964).

e. Sneddon, I.N., Elements of Partial Differential Equations, (McGraw-Hill, New York, 1957).

f. Carrier, G., et al., Ordinary Differential Equations, (Blaisdell, Waltham, Mass., 1968).

7. Approximation Methods

a. Arden, B., and Astill, K., Numerical Algorithms: Origins and Applications, (Addison-Wesley, Reading, Mass., 1970).

b. Hildebrand, F., Introduction to Numerical Analysis, (McGraw-Hill, New York, 1956).

c. Cole, J.D., Perturbation Methods in Applied Mathematics, (Blaisdell, Waltham, Mass., 1968).

d. Bellman, R., Perturbation Techniques in Mathematics, Physics, and Engineering, (Holt, Rinehart and Winston, New York, 1964).

8. Fourier Analysis and Transform Methods

a. Sneddon, I.N., Fourier Transforms, (McGraw-Hill, New York, 1951).

b. Franklin, P., An Introduction to Fourier Methods and the LaPlace Transformation, (Dover, New York, 1949).

9. Algebra

a. Peterson, W.W., Error-Correcting Codes, (MIT Press, Cambridge, Mass., 1961).

b. Lin, S., An Introduction to Error-Correcting Codes, (Prentice-Hall, New Jersey, 1970).

c. Hall, M., The Theory of Groups, (MacMillan, New York, 1959).

d. Jacobson, N., Lectures in Abstract Algebra, 3 Volumes, (Van Nostrand, Princeton, New Jersey, 1960).

e. Herstein, I.N., Topics in Algebra, (Blaisdell, Waltham, Mass., 1964).

10. Queueing Theory

a. Morse, P.M., Queues, Inventories and Maintenance, (John Wiley and Sons, New York, 1958).

b. Saaty, T., Elements of Queueing Theory, (McGraw-Hill, New York, 1961).

c. Lee, A.M., Applied Queueing Theory, (MacMillan, London, 1966).

d. IBM Corp., Analysis of Some Queueing Models in Real-Time Systems, 2nd Edition, (New York, 1969).

11. Information Theory

a. Shannon, C., and Weaver, W., The Mathematical Theory of Communication, (University of Illinois Press, 1964).

b. Khinchin, A.I., Mathematical Foundation of Information Theory, (Dover, New York, 1957).

c. Lee, Y.W., Statistical Theory of Communication, (John Wiley and Sons, New York, 1960).

d. Raisbeck, G., Information Theory, (MIT Press, Cambridge, Mass., 1964).

e. Fano, R.M., Transmission of Information, (MIT Press, Cambridge, Mass., 1961).

f. Wiener, N., Cybernetics, 2nd Edition, (MIT Press, Cambridge, Mass., 1961).

12. Computability Theory

a. Minsky, M., Computation: Finite and Infinite Machines, (Prentice-Hall, New Jersey, 1967).

b. Rogers, H., Recursive Functions and Effective Computability, (McGraw-Hill, New York, 1967).

c. Brafford, P., and Hershberg, D., (eds.) Computer Programming and Formal Systems, (North Holland, Amsterdam, 1963).

13. Automata Theory and Mathematical Linguistics

a. Hartmanis, J., and Stearns, R., Algebraic Structure Theory of Sequential Machines, (Prentice Hall, New Jersey, 1966)

b. Ginsburg, S., Mathematical Theory of Context Free Languages, (McGraw-Hill, New York, 1966).

c. Hopcroft, J., and Ullman, J., Formal Languages and their Relation to Automata, (Addison-Wesley, Reading, Mass., 1969).

d. Chomsky, N., "Formal Properties of Grammers", in Handbook of Mathematical Psychology, Volume II., Luce et al. (eds.), (John Wiley and Sons, New York, 1963), pp. 323-418.

e. Cocke, J., and Schwartz, J., Programming Languages and their Compilers, 2nd Edition, (Courant Institute for the Mathematical Sciences, New York University, 1970).

f. Hennie, F., Finite State Models for Logical Machines, (John Wiley and Sons, New York, 1968).

g. Ginzburg, A., Algebraic Theory of Automata, (Academic Press, New York, 1968).

h. Minsky, M., loc. cit.

APPENDIX B

DIGITAL COMPUTER BIBLIOGRAPHY

A digital system designer should be very familiar with both hardware and software aspects of computer design. The following bibliography is a list of some standard works in computer systems design that should be familiar to all working in the field. Also included is a list of most of the pertinent journals.

Although not listed in this bibliography, previous journal articles provide an excellent source of information in this field.

A. Compiler Design

1. Gries, D., Compiler Construction for Digital Computers, (John Wiley and Sons, New York, 1971).

2. McKeeman, W.M., et al., A Compiler Generator, (Prentice-Hall, New Jersey, 1970).

3. Ginsburg, S., The Mathematical Theory of Context Free Languages, (McGraw-Hill, New York, 1966).

4. Donovan, J., System Programming, (Prentice-Hall, New Jersey, 1972).

5. Cheatham, T., "Notes for Course AM295: Theory and Construction of Compilers",(Harvard University Applied Mathematics Dept., 1968).

6. Schwartz, J., and Cocke, J., Programming Languages and Their Compilers, 2nd Edition, (Courant Institute for the Mathematical Sciences, New York University, 1970).

7. "Proceeding of a Symposium on Compiler Optimization", (ACM Sigplan Notices, 5(7), July, 1970).

8. Randall, B., and Russell, L., Algol 60 Implementation, (Academic Press, New York, 1964).

9. Hopgood, F., Compiling Techniques, (American Elsevier, Inc., New York, 1969).

B. Programming Languages

1. Sammet, J., Programming Languages: History and Fundamentals, (Prentice-Hall, New Jersey, 1969).

2. Wegner, P., Programming Languages, Information Structures and Machine Organization, (McGraw-Hill, New York, 1968).

3. Iverson, K., A Programming Language, (John Wiley & Sons, New York, 1962).

4. "Proceedings of a Symposium on Data Structures in Programming Languages", (ACM SIGPLAN Notices, 6(2), Feb., 1971).

5. "Proceedings of a Symposium on Languages for Systems Implementation", (ACM SIGPLAN Notices, 6(9), Oct., 1971).

6. Neuhold, E.J., "The Formal Description of Programming Languages", (IBM Systems Journal, 10(2), 1971), pp. 86-112.

7. Genuys, F., (ed.), Programming Languages, (Academic Press, New York, 1968).

8. Rosen, S., Programming Languages and Systems, (McGraw-Hill, New York, 1967).

9. Intermetrics, Inc., The Programming Language HAL-A Specification, (Cambridge, Mass., 1971 - Prepared under Contract NAS 9-10542, MSC Document #MSC-01846).

10. Higman, B., A Comparative Study of Programming Languages, (American Elsevier, Inc., New York, 1967).

C. Operating Systems

1. Cuttle, G., and Rolinson, P.B., (eds.), Executive Programs and Operating Systems, (American Elsevier, Inc., New York, 1970).

2. Colin, A.J.T., Introduction to Operating Systems, (American Elsevier, Inc., New York, 1971).

3. Proceedings of a Symposium on Comparative Operating Systems, (Auerbach, Princeton, New Jersey, 1969).

4. Organick, E., Guide to Multics for Subsystem Writers, (M.I.T. Press, Cambridge, Mass., 1972).

5. Denning, P., Resource Allocation in Multiprocessor Computer Systems, (Ph. D. Thesis, Electrical Engineering Dept., M.I.T., Cambridge, Mass., 1968).

6. IBM Corp., "IBM System/360 Concepts and Facilities", (New York, 1969, IBM #C28-6535-5).

7. IBM Corp., "IBM System/360 Supervisor and Data Management Services", (New York, 1968, IBM #C28-6646-2).

8. Sayers, A.P., (ed.), Operating Systems Survey, (Comptre Corp., Auerbach Publishers, Princeton, New Jersey, 1971).

9. Roos, D., ICES System Design, 2nd Edition, (M.I.T. Press, Cambridge, Mass., 1967).

10. Iliffe, J.K., Basic Machine Principles, (American Elsevier, Inc., New York, 1968).

11. ACM, "Proceedings of the National Symposia on Operating Systems Principles", (New York, 1967, 1969, 1971).

12. Saltzer, J., Traffic Control in a Multiplexed Computer System, Ph.D. Thesis, Electrical Engineering Dept., M.I.T., Cambridge, Mass., 1966).

D. Computational Algorithms

1. Knuth, D., The Art Of Computer Programming, (7 volumes, 2 published so far), (Addison-Wesley, Reading, Mass., 1967).

2. Flores, I., Computer Sorting, (Prentice-Hall, New Jersey, 1969).

3. Flores, I., Computer Software, (Prentice-Hall, New Jersey, 1965).

4. Arden, B., and Astill, K., Numerical Algorithms: Origins and Applications, (Addison-Wesley, Reading, Mass., 1970).

5. Barron, D.W., Recursive Techniques in Programming, (American Elsevier, Inc., New York, 1967).

6. Foster, J.M., List Processing, (American Elsevier, Inc., New York, 1967).

E. Computer Architecture and Hardware Design

1. Flores, I., Computer Organization, (Prentice-Hall, New Jersey, 1969).

2. Flores, I., The Logic of Computer Arithmetic, (Prentice-Hall, New Jersey, 1963).

3. Lorin, H., Parallelism in Hardware and Software: Real and Apparent Concurrency, (Prentice-Hall, New Jersey, 1972).

4. Bell, C.G., and Newell, A., Computer Structures: Readings and Examples, (McGraw-Hill, New York, 1971).

5. Husson, S.S., Microprogramming: Principles and Practices, (Prentice-Hall, New Jersey, 1971).

6. Hobbs, L.C., et al., (eds.), Parallel Processor Systems, Technologies, and Applications, (Spartan, New York, 1970).

7. Miller, J.S., et al., Multiprocessor Computer System Study, (Intermetrics, Inc., Cambridge, Mass., 1970, under Contract NAS 9-9763).

8. Buchholz, W., (ed.), Planning a Computer System, (McGraw-Hill, New York, 1962).

9. McCloskey, E., Introduction to the Theory of Switching Circuits, (McGraw-Hill, New York, 1965).

F. Real Time Systems

1. Martin, J., Design of Real-Time Computer Systems, (Prentice-Hall, New Jersey, 1967).

2. Martin, J., Programming Real-Time Computing Systems, (Prentice-Hall, New Jersey, 1968).

3. Martin, J., Systems Analysis for Data Transmission, (Prentice-Hall, New Jersey, 1972).

4. Wilkes, M.V., Time-Sharing Computer Systems, (American Elsevier, Inc., New York, 1968).

5. Organick, E., loc. cit.

G. Aerospace Computer Systems

1. Saponaro, J., Pepe, J., et al., Advanced Software Techniques for Data Management Systems, 3 Volumes, (Intermetrics, Inc., Cambridge, Mass., 1972, prepared under Contract NAS 9-11778).

2. Kosmala, A., et al., Engineering Study for a Mass Memory System for Advanced Spacecrafts, (Intermetrics, Inc., Cambridge, Mass., 1970, prepared under Contract NAS 9-9763).

3. Kosmala, A., et al., Standard Interface Definition for Avionics Data Bus Systems, (Intermetrics, Inc., Cambridge, Mass., 1971, prepared under Contract NAS 9-11477).

4. Intermetrics, Inc., Development of an MSC Language and and Compiler, (Cambridge, Mass., 1971, prepared under Contract NAS 9-10542).

5. Kosmala, A., et al., Central Processor Operational Analysis, (Intermetrics, Inc., Cambridge, Mass., 1971).

6. Kosmala, A., et al., Central Processor Memory Organization and Internal Bus Design, (Intermetrics, Inc., Cambridge, Mass., 1972).

H. Journals

Journal of the Association for Computing Machinery

Communications of the Association for Computing Machinery

Computing Reviews

Computing Surveys (back issues especially important)

IBM Systems Journal

IEEE Transactions on Electronic Computers

Computer Design

Modern Data

IBM Journal of Research and Development

The Computer Journal

Proceedings of Spring and Fall Joint Computer Conferences

Proceeding of ACM National Conferences

Computer Decisions

Datamation

The Computer Bulletin