NASA Contractor Report 3011

Design Study of Software-Implemented Fault-Tolerance (SIFT) Computer

J. H. Wensley, J. Goldberg, M. W. Green,
W. H. Kautz, K. N. Levitt, M. E. Mills,
R. E. Shostak, P. M. Whiting-O'Keefe,
and H. M. Zeidler

CONTRACT NAS1-13792

JUNE 1982

NASA

NASA Contractor Report 3011

Design Study of Software-Implemented Fault-Tolerance (SIFT) Computer

J. H. Wensley, J. Goldberg, M. W. Green,
W. H. Kautz, K. N. Levitt, M. E. Mills,
R. E. Shostak, P. M. Whiting-O'Keefe,
and H. M. Zeidler
SRI International
Menlo Park, California

Prepared for
Langley Research Center
under Contract NAS1-13792

National Aeronautics
and Space Administration

Scientific and Technical
Information Office

1982

CONTENTS

APPENDIX

ILLUSTRATIONS

TABLES

I INTRODUCTION

This report covers the research carried out by SRI on contract NASI-13792 (SRI project 4026) during the period 5 February 1975 to 5 February 1976. The primary goal of the research is to design a flyable SIFT computer that can demonstrate the feasibility of an integrated function, fault-tolerant computer in commercial aviation.

Prior research by SRI on contract NASI-10920 (SRI project 1406) [Refs. 1,2]* over the period October 1972 to October 1973 considered the design of fault-tolerant computer architectures and, in particular:

- The computational and reliability requirements of an advanced transonic commercial transport aircraft using fly-by-wire techniques with a unified digital computing system.
- The impact of modern digital circuit technology on the design of such a computer.
- Candidate architectures for a computer to satisfy the requirements.

One of the architectural concepts conceived in that study was given the name "SIFT" (Software-Implemented Fault-Tolerance). It showed great promise of satisfying the extreme reliability requirements of this application class. The detailed design of a computer based on the SIFT concept is the primary objective of the study reported here.

The goals of the effort were:

(1) To develop the SIFT design concept to a point at which its potential reliability may be evaluated with reasonable accuracy.

(2) To investigate alternate strategies for physical implementation, using available or specially designed components.

*Numbered references are listed at the end of the chapter.

(3) To prove the correctness of the hardware and software designs.

(4) To model the system and evaluate its effectiveness from a fault-tolerance point of view .

To achieve these goals, the research was directed at the critical aspects of the design, leaving less critical aspects to a later phase in the research program.

Some of the research results reported here have been previously discussed in the monthly technical progress reports and in a series of seven technical memos that have been issued during the course of this study. In addition, a Technical Plan for the Future Development of SIFT was issued in November 1975.

REFERENCES

1. J. H. Wensley, K. N. Levitt, M. W. Green, J. Goldberg, and P. G. Neumann, "Design of a Fault-Tolerant Airborne Digital Computer," Vol. I, Architecture, Final Report. NASA CR-132252, 1973.

2. R. S. Ratner, E. B. Shapiro, H. M. Zeidler, S. E. Wahlstrom, C. B. Clark, and J. Goldberg, "Design of a Fault Tolerant Airborne Digital Computer," Vol. II, Computational Requirements and Technology, Final Report. NASA CR-132253, 1973.

II SIGNIFICANT RESULTS AND OUTSTANDING PROBLEMS

In this section we summarize the significant research results achieved in this study, and we identify significant problems that remain.

A. Significant Research Results

The principal objective of the study was to carry out a refinement of the SIFT concept, thereby reducing uncertainties in the design. The intent was to prove the feasibility of a design based on the SIFT concept with an eventual goal of a flyable prototype (or "brassboard"). A significant result of our current study is that in this process of refining the design, no radical changes have had to be made. Indeed the fundamental SIFT concepts that distinguish it from other fault-tolerant computer architectures remain, namely:

- All fault-tolerance procedures (error detection, error correction, diagnosis, and reconfiguration) carried out by software.
- No essential special fault-tolerance--different replication possible for different tasks, or at different times for the same task.
- Very high reliability achieved without the need for high intrinsic reliability of subunits of the system.
- Reconfiguration on the basis of complete processor/memory modules or complete busses.
- An ability to use fairly standard units such as processors and memories, with an attendant gain in reliability by taking advantage of the stability of production processes with standard high-volume production.

The development of these concepts leads to a design with the following characteristics:

- Replicated units do not operate in lock-step mode but are only loosely synchronized. The communication between CPUs is asynchronous, thereby removing the need for an ultra-reliable system clock.

- Agreement between replicated units is verified only at the completion of program segments (tasks).
- Faulty units are not necessarily removed but can be either ignored or assigned to tasks having no overall effect.
- Transient faults do not necessarily cause permanent removal of the faulty units. Furthermore, the looseness of synchronization among sets of tasks makes it possible to enhance immunity from transients by providing that redundant versions of a computation may be done at different moments in time.
- The degree of fault-tolerance can be different for different tasks being performed and can be different at different times for the same task.
- No special hardware is used to carry out fault detection or correction.
- Communication between CPUs is minimized so that low bandwidth busses can be used, thereby facilitating physical separation of modules in environments where physical damage is a hazard.
- The design concept is independent of the way in which the units are built; i.e., no specialization of CPU or memory design is required for fault tolerance, thereby allowing the choice to be based on other properties, e.g., speed, availability.
- The total computing power of the system can be varied by using units of different speed or by changing the number of units.

During the current study all critical units of both hardware and software have been studied. The following are the key results obtained:

- There is no requirement for a central working memory, but there is justification for a back-up, nonvolatile memory, e.g., magnetic bubble memory (VI-C).*
- Viable structures for the input/output subsystems have been developed (VI-B).
- Trade-off studies of the bus system design have been carried out. Consideration has been given to cost, component count, delay, bandwidth, reliability, and structural simplicity of different bus structures, with a conclusion that a two-level structure is preferred (VI-A).

*Parenthesized notations indicate that chapter or section of this report in which the particular result is discussed in more detail.

- Adequate protection against external power transients effecting the power supplies can be provided, and fault-tolerance of the power system can be economically achieved (VI-E).
- Optical transmission offers a cost-effective way of protecting against induced transients in data paths.
- Satisfactory methods have been devised for allocating tasks between processors and for devising and representing suitable schedules (V).
- Reliability analyses show that a system employing five processors and four busses yields satisfactory reliability, with greater replication yielding even better reliability (VII).
- Formal proofs of the reliability properties of the system can be carried out in a rigorous manner, thus providing assurance of the correctness of the design, and also the correctness of the reliability model (VII).
- The design of the software, including the fault-tolerance features, can be specified in a formal abstract manner, thus enabling the proofs referred to above and assisting in transporting the design across different hardware implementations (VIII).

In carrying out the design study it has been necessary to develop a methodology for design and analysis. While this methodology has been aimed directly at the objectives of the current study, they have great relevance in the wider context of fault-tolerant computer design and beyond that to the design and analysis of computer systems in general. The principal features of this methodology are:

- Techniques for formal specification of complex computer systems.
- Techniques for formal proof of correctness of design.
- Techniques for the analysis of reliability of fault-tolerant computer systems.
- Techniques for allocating tasks among processors and for designing and representing schedules within processors of a multiprogrammed multiprocessor computing system.

These techniques represent a powerful rigorous methodology of design and analysis that can have a significant impact on future design efforts.

B. Outstanding Problems

While our study has considered all the critical design and analysis issues, there remain some outstanding problems both in the development of SIFT and in the design of fault-tolerant computers in general. The major outstanding needs that we see are for:

- Continued refinement of the SIFT design to include all design aspects and, in particular, to develop cost/performance/ reliability trade-offs to enable optimized versions of SIFT to be produced.
- More definitive data on the intrinsic reliability of different electronic technologies, particularly the newest ones, e.g., CMOS Large-Scale Integrated (LSI) circuits.
- Improved methods for the analysis of coverage, particularly of diagnosis techniques .
- More definitive data on the nature and incidence of massive transient disturbances, e.g., as caused by lightning strikes.
- A systematic study of the input/output units within an aircraft (sensors, actuators, etc.) and of their performance and reliability characteristics.

Most of these problems are considered in the technical plan for future development of SIFT.

III TECHNICAL PLAN FOR FUTURE DEVELOPMENT OF SIFT

A. Introduction

The material presented in this chapter was previously published in November 1975 as an informal document. It is included here so that this report can be a self-contained document.

The plan as presented here is that originally submitted. Subsequent discussions between NASA staff and SRI have resulted in a recommendation that certain tasks should be delayed from Step 2 to Step 3. These are the tasks "Test Procedures" and "Aircraft Test Interface" as shown in Figure III-1.

The plan is presented in detail for the period up to November 1976, by which time the design is expected to have been completed in sufficient detail to enable procurement of equipment. The specification of system software (local and global executives) will have been fully specified to enable program writing to commence. The plan is presented in more general terms for the period beyond November 1976. We discuss in this document the importance of strong interaction with other segments of industry such as the airlines, airframe manufacturers, avionics manufacturers, and semiconductor manufacturers, and also with other related research and development centers, e.g., NASA-Ames STP:AMD project and NASA Houston Space Shuttle Development.

B. The Relevance of Analytic Techniques, Simulation, Emulation, Experimental Models, Prototypes, and Flight Model in the Development Process

1. Introduction

The primary goal of the SRI effort is a flyable SIFT computer that can demonstrate the feasibility of an integrated-function, fault-tolerant computer in commercial aviation. Because of the complexity of such a computer and the importance of the demonstration, it is desirable to achieve a high level of confidence in the design before a flight model is built.

For a computer system, such confidence may be achieved through various means of validation such as human design review, formal analysis and proof, simulation, emulation, and the testing of physical prototypes. Furthermore, in any very complex system it is common practice to proceed with the validation in several steps, each step dealing with different levels of abstration and approximation. Validation exercises can be very expensive and time consuming, so it is desirable to choose a strategy for validation that will yield a high level of confidence quickly and with low cost.

The design approach SRI is using for SIFT is unusual in that both the hardware and the executive software will have precise, formal specifications. In this section we will present a plan for design validation that takes advantage of this design approach. We believe the plan is both effective and economical compared to reasonable alternatives.

2. Outline of the Plan

We propose the following steps:

(1) Abstract specification and proof of software and hardware.

(2) Design of programs and logic; investigation of nonlogical* design issues.

(3) Validation of the design including construction of a prototype computer.

(4) Testing of the prototype and construction and certification of an experimental model flight computer.

(5) Flight tests.

*By "nonlogical" we mean factors such as packaging, device performance, fault modes, etc., that are not included explicitly in programs or logic designs.

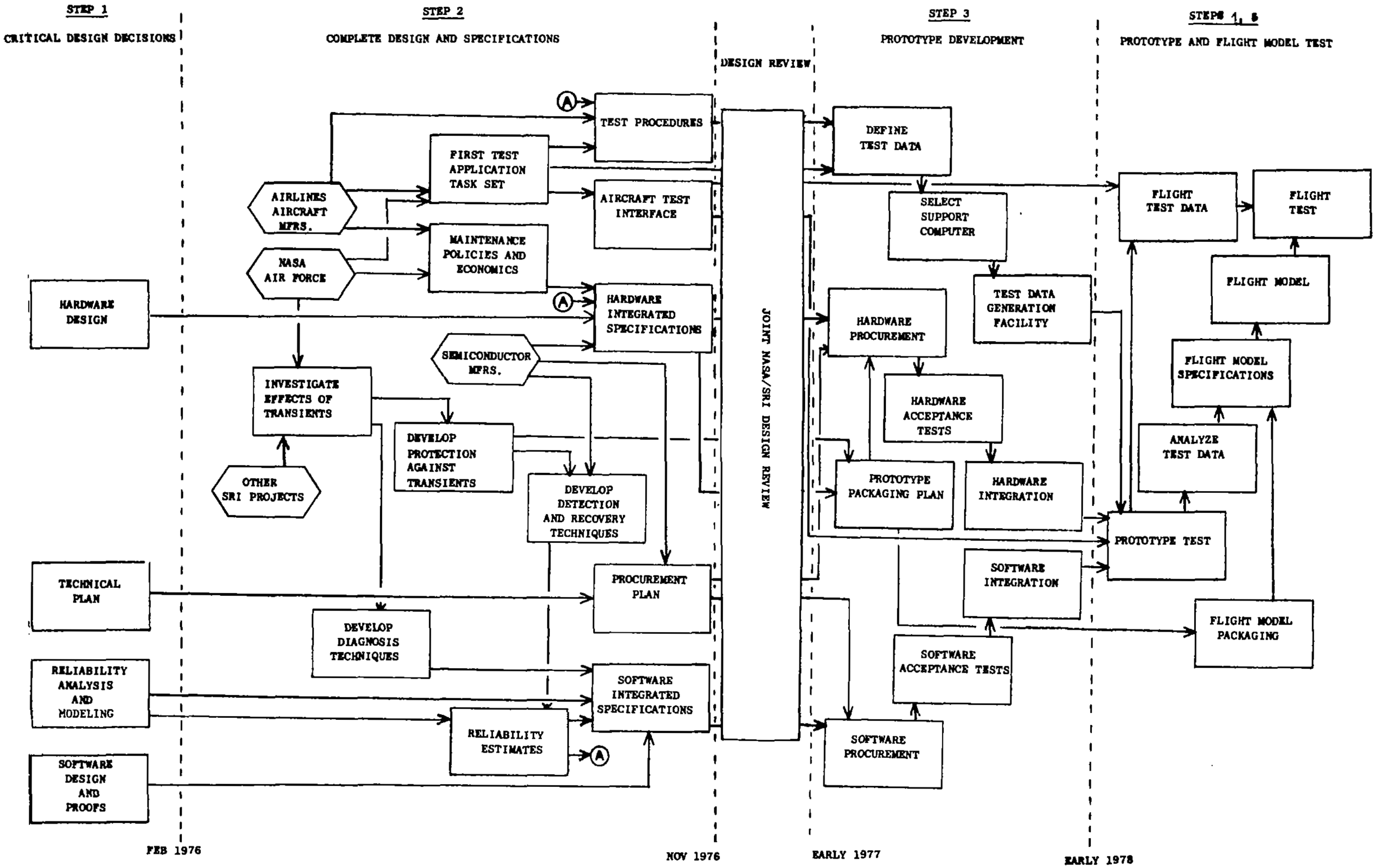

STEP 1
CRITICAL DESIGN DECISIONS
STEP 2
COMPLETE DESIGN AND SPECIFICATIONS
DESIGN REVIEW
STEP 3
PROTOTYPE DEVELOPMENT
STEPS 4, 5
PROTOTYPE AND FLIGHT MODEL TEST
A
TEST PROCEDURES
FIRST TEST APPLICATION TASK SET
AIRLINES AIRCRAFT MFRS.
AIRCRAFT TEST INTERFACE
MAINTENANCE POLICIES AND ECONOMICS
NASA AIR FORCE
HARDWARE DESIGN
A
HARDWARE INTEGRATED SPECIFICATIONS
SEMICONDUCTOR MFRS.
INVESTIGATE EFFECTS OF TRANSIENTS
DEVELOP PROTECTION AGAINST TRANSIENTS
OTHER SRI PROJECTS
DEVELOP DETECTION AND RECOVERY TECHNIQUES
TECHNICAL PLAN
PROCUREMENT PLAN
DEVELOP DIAGNOSIS TECHNIQUES
RELIABILITY ANALYSIS AND MODELING
SOFTWARE INTEGRATED SPECIFICATIONS
RELIABILITY ESTIMATES
A
SOFTWARE DESIGN AND PROOFS
JOINT NASA/SRI DESIGN REVIEW
DEFINE TEST DATA
SELECT SUPPORT COMPUTER
TEST DATA GENERATION FACILITY
HARDWARE PROCUREMENT
HARDWARE ACCEPTANCE TESTS
PROTOTYPE PACKAGING PLAN
HARDWARE INTEGRATION
SOFTWARE INTEGRATION
SOFTWARE ACCEPTANCE TESTS
SOFTWARE PROCUREMENT
FLIGHT TEST DATA
FLIGHT TEST
FLIGHT MODEL
FLIGHT MODEL SPECIFICATIONS
ANALYZE TEST DATA
PROTOTYPE TEST
FLIGHT MODEL PACKAGING
FEB 1976
NOV 1976
EARLY 1977
EARLY 1978

FIGURE III-1 SIFT DEVELOPMENT PLAN

Between Steps 2 and 3 we propose that a design review take place. This should be conducted by both SRI and NASA personnel or their representatives.

Step 1 will include the executive software and the major system modules, taken to a level of detail that comprises well-understood software and hardware functions. Some fault-tolerance functions will be parameterized to allow some user freedom in the choice of fault-tolerance policies. Auxiliary software such as diagnostic routines and system exercisers will not be included in this stage.

Step 2 will result in the complete specification of the computer system to a level of detail that will be sufficient for equipment and software procurement. The hardware specifications will include statements of functional capability, performance parameters, reliability constraints, interface specifications, and packaging constraints. The software specifications will also contain functional requirements and performance parameters and, in addition, will be accompanied by sample implementation schemes. Certain nonlogical design issues will be investigated. These include choice of device and interconnection technologies, packaging and shielding, power supply design, and specification of peripheral equipment such as displays and storage units. Information concerning fault modes will be acquired and applied to the logical design and executive programs.

At this stage a design review should take place. The purpose of this review is to make a detailed examination of the design, which had not been possible before a complete design existed. While we are confident that there will be little need for change in the basic system concepts, we do see the possibility of changes in some of the parameters of the system. This review should also examine the assumptions upon which the reliability analyses were based.

The prototype computer to be procured in Step 3 will be realized mainly in the same technology that is expected to be used in a flight-experimental model. The software will be extended to include sample application programs, in-flight diagnostic programs, and basic checkout programs. Instrumentation will be provided both in software and hardware, and an external computer will be programmed to drive the SIFT computer so as to simulate the aircraft environment. Hardware developments will include power supplies, clocks and interconnections, and connections with peripheral units such as bubble memories and aircraft circuits. The computer will be analyzed in order to guide (1) the final design of interconnections and packaging and (2) the design of test procedures. As discussed at the end of the next section, a limited amount of evolution will be allowed in the prototype. Examples of modifications that might be planned are (1) change in interconnection technology, e.g., the introduction of optical couplers and special power-supply circuits, and (2) replacement of some changeable memories by read-only memories, e.g., for microprograms or programs.

The flight computer of Step 4 will be ruggedized and shielded and will be provided with maintenance aids such as handbooks and diagnostic tools. A full set of application programs will be prepared. The computer will be certified for experimental aircraft installation. Our objective will be that the flight model will differ from the prototype to only a limited degree, and as such can be regarded as an evolution from it. This view is justified by the fact that the SIFT concept allows for the use of off-the-shelf units for the processors and memories. We can foresee changes between the prototype and the flight model in the bus system, the input/output system, and the scale of the system as a whole and, in addition, certain technology changes mentioned in the discussion of the evolution of the prototype itself.

3. Justification of the plan

The proposed plan departs from conventional practice in two major respects. First, formal system specification (Stage 1) is a much larger effort in the plan than in conventional practice, which typically specifies a system discursively. Our specification method is actually a very high-level form of programming that not only is precise (to some level of abstraction), but also provides a capsule intuitive view of the system.

The second departure is that the plan does not include a major component for testing, either by computer simulation or breadboarding. We believe this omission is justified because we believe that the specification and proof methodology we are employing will leave very few design questions unanswered, with the exception of certain nonlogical elements such as power supply and packaging (we plan to allow as much testing for these aspects as good engineering practice requires). Extensive testing would thus constitute a wasteful diversion of time and money. Moreover, we do not see the need for significant incorporation of innovative hardware technologies which would need to be tested in a breadboard version of the system.

Let us consider the usual arguments in favor of testing, and the reasons why we reject them.

The usual role of tests is to help designers see just what behavior is produced by the thing they have created. This purpose is obviated by our specification methodology.

Another argument often tendered in favor of testing is that the specifications may themselves be deficient, and that in the course of preparing tests, individuals will think of input conditions and sequences that may have been overlooked by the designers. The issue is how to validate or confirm a designer's understanding of the system problem. We believe that testing may be a weak tool for achieving this purpose, and that given the present cost and power of testing, other more human-oriented methods would be more effective. Such methods include:

(1) Good means for expressing, recording and displaying the design and its documentation.

(2) Careful design review methods, such as redundant design teams and "walk-through" (discussion of a design with an outsider).

It is for this reason that we propose a design review at the end of Step 2, i.e., before procurement of prototype equipment and software.

Yet another argument for testing is that it can expose assumptions made about primitve system functions. For example, the arithmetic operations of a particular processor may not support the computational methods assumed by the application programmer--or even the claims of the programming manual. This is indeed a significant issue. Computer simulation will, in general, not be useful to solve this problem. We are optimistic however that the problem can be deferred to the prototype stage (Stage 3) without creating the need for major redesign.

The final argument for testing or simulation is that formal validation methods cannot, in their present state of development, inform the designer about execution speeds. Such information may be necessary in order to set performance specifications for components, such as bus or memory bandwidth and processor cycle-time. We believe that such information can be obtained as needed by special analyses, including the use of computerized models. Such models would be much simpler and easier to create and use than general system simulations, emulations, or breadboards. Fortunately, the SIFT distributed-computer design places very mild performance requirements on system components for the chosen application domain, so we believe that extensive performance analyses will not be needed.

Some further remarks about the prototype are necessary. As described, the prototype will have essentially the same logic and use the same device types as the flight model. The two will differ mainly in size, completeness of program set, interconnection techniques, packaging, physical hardening, and the like. It might be argued that the design issues involved in these various qualities might be resolved without the necessity of building an operating prototype. It is perhaps too early in the course of the design to be certain about this issue; however, the justification for a separate prototype does not rest on its support of the design studies implied by the differences listed.

Despite our confidence in the prospect for validation of SIFT functional design, there are many nonlogical issues that may contain hidden implications on some aspects of the system logic. These issues include transient fault effects, accessibility and controllability for diagnosis, and postspecification shortcomings in device performance.

It would be desirable to uncover these "surprises" as early as possible. There is therefore a case for introducing physical parameters into the design of the prototype as early as possible.

To summarize, our view is that the issues that are typically resolved by simulation and breadboard models can be delayed to the stage of prototype testing because of the inherent flexibility of the SIFT concept.

C. Recommendations

We see the development of the SIFT concept as a series of five steps that can be briefly characterized as follows:

(1) Critical aspects of the design (current contract)

(2) Complete design of a SIFT system (extension to current contract)

(3) Building of a prototype of SIFT

(4) Prototype testing and procurement of a flyable model of SIFT

(5) Flight test and evaluation.

The remainder of this section details the above steps and defines the work to be accomplished in them. The involvement of different organizations (SRI, NASA, airframe manufacturers, airlines, semiconductor manufacturers, etc.) in the work is also discussed.

The technical plan is shown graphically in Figure III-1, with major groupings of expected results of the present contract under Step 1 (e.g., Hardware Design). The plan shows the work steps up to the latter part of 1976 in detail (with each item being defined in Section C2), and less detail for the steps beyond the end of 1976.

As can be seen from the chart, it is our view and recommendation that a flyable model of the SIFT computer can and should be implemented approximately in 1979. All of the preceding tasks have been designed with that goal in mind. The exact phasing of the many tasks can be a matter of negotiation, but the relative phasing is considered to be as indicated by the arrows in the chart.

We foresee that the tasks to be accomplished in Step 2 will require a level of effort slightly higher than that of the existing contract. This will be augmented with cooperation from many industry segments, such as airlines and semiconductor manufacturers. We show the point at which this cooperation begins in Step 2, but continuing cooperation is implied in the later steps, though this later cooperation is not specifically indicated on the chart. During the later phases, this cooperation will have to become one of active participation as equipment is procured and the prototype and the flyable model are physically tested. We see that the total level of effort in Steps 3, 4, and 5 will be significantly higher than at present and that it will be spread over many organizations. The exact details of the tasks to be performed in these later steps are the subject of the procurement plan that will be prepared under Step 2, at which time we expect to be able to detail the later steps so that a complete determination can be made of the required funding for the remainder of the program.

We consider that the major effort of Step 2 should be carried out by SRI, with active discussion by industry and other research segments. This combined participation will enable the effort to proceed with continuity of personnel, with the advantage of maintaining the existing capability and enthusiasm, while at the same time widening the community of those who are involved in the total effort.

The participation of other industry segments in Steps 3 to 5 is expected to increase very significantly. We see that the major role that SRI should play in these later steps is one of technical leadership and coordination, as well as being the research arm of the total effort in-resolving issues that are at present unforeseen but become known as work proceeds. In addition, we see that the scope of the total effort may be changed with changing circumstances, causing a need to carry out research that is beyond the scope of that which is currently envisioned.

1. Step 1 - Current Contract

The current contract calls for SRI to design and analyze all critical aspects of SIFT. The following items are included in this effort:

(1) General system requirements

(2) The definition and conceptual design of SIFT

  (a) Bus and processor memory bus interface

  (b) Processor

  (c) Memory

  (d) Input/output interface

  (e) Software

    (i) Executive program

    (ii) Sample application program

(3) Analysis and assessment of SIFT

  (a) Thoroughness of fault analysis

  (b) Proof of fault-tolerance procedures

  (c) Modeling

(4) Technical plan for further SIFT development (this document)

(5) Reporting and documentation.

SRI has currently completed about 70% of this research, with every indication that the objectives will be met. Very few new and significant problems have been uncovered. The only one of great significance is the matter of massive transients in the event of severe environmental distrubances, e.g., a lightning strike on the aircraft. The technical plan for Step 2 as presented below describes the actions that are proposed for dealing with this problem (see d and h under Step 2, following.

2. Step 2 - Complete design of SIFT

This subsection defines the tasks that constitute Step 2 in the technical plan. Each major task or group of tasks is separately defined in the sections a through j.

a. General - The overall goal is to specify fully a representative SIFT system to a level of detail satisfactory for procurement of all hardware and software components. The particular configuration of

SIFT that will be chosen for this purpose will be appropriate for carrying out a reasonable set of aircraft application tasks chosen after consultation with both airlines and airframe manufacturers. Throughout this study, close contact will be maintained with both semiconductor and avionics manufacturers so as to provide an efficient involvement of them in the building of a prototype to be carried out in Step 3 of this program.

b. Application Tasks - First implementation of the SIFT system should include only a subset of the total on-board data processing system that is projected for jet transports of the 1980-1985 period. This will make much more economical the task of realistic evaluation of the fault-tolerant techniques and procedures that are to be incorporated.

One objective will be the selection of those application tasks that can most readily and effectively be incorporated into the prototype system. The subset of tasks must be adequate and sufficiently varied to provide meaningful test results for analysis. In addition, the system must be planned to meet the constraints imposed by operational, economic, and procurement factors. The application task selection must therefore be based in part upon the recommendations, facilities, and equipment provided by various segments of the airline and data processing communities. Among those to be consulted in the task selection are the following:

- Airlines--As currently visualized, cooperation of one or more airlines will be sought to provide a commercial airline environment to enable generation of a true operational interpretation of system test results. Close liaison should be established with airline personnel at an early date. Operational and economic aspects of the operations of the airline(s) will most probably impose limitations on the functions that can in a practical sense be incorporated into a prototype test sytem.
- Research and development agencies--Throughout the past few years, a considerable amount of effort has been directed toward the design and test of digital systems for certain aircraft control and related functions. Agencies and companies such as NASA, the U.S. Air Force, and Boeing have been at the forefront of such programs. Liaison will be established with such organizations in order to exploit the results of those efforts.

- ARINC--It will be desirable to specify the prototype system to be consistent with existing and projected standards insofar as feasible, provided that the basic goals of the project are not jeopardized. Accordingly, direct contact will be made with agencies such as ARINC (Airline Radio, Inc.) to achieve the desired measure of system standardization and compatibility.

c. Hardware Design - A complete specification of all hardware components of a SIFT system will be prepared to a level of detail sufficient for procurement of a prototype from commercial vendors. The specifications must include all aspects including:

- Functional capability
- Size, speed, and performance parameters
- Reliability constraints
- Interface specifications
- Packaging constraints

To provide assurance that the specified system can be implemented effectively and economically within the projected time frame, contact will be established with vendors of computers, data communication equipment, interfacing hardware, and semiconductor products. In some cases, standard product lines can possibly be modified slightly to accomodate the requirements of the prototype system. For example, data communication elements will be modified to be compatible with the specified bus structure of the prototype system. Liaison will be established with appropriate departments of such organizations to facilitate specification of any such modifications that may prove to be necessary.

d. Software Design - The software of the SIFT will be fully specified. For the system software (Global and Local executives), the formal specifications prepared under the current contract will be augmented to the level of detail that can be used for procurement. This will involve preparing sample implementation schemes and providing firm estimates of those variables in the system that are currently parameterized in the formal specifications.

The set of application tasks (item 2b) will be specified in a similar manner to the system software.

The specifications will include full details of acceptance tests to be used for evaluation. The use of programming languages at different levels will be specified. Any special measures to be taken to assist in validation of the software will be specified.

Techniques should be investigated for providing a scheme for achieving fault-tolerant application software. Such studies should include consideration of the concept of "Recovery Blocks" as developed at the University of Newcastle, England, or derivatives of it specially adapted to the SIFT concept.

e. Aircraft Interface - The SIFT system will require digital data transmission from a multiplicity of instruments, sensors, radio-frequency units, and other peripheral input units, and transmission to a multiplicity of actuators and display units. One of the tasks will therefore be to specify the total data communication system. Some of the salient factors to be considered are as follows:

- The basic system--Various techniques for accessing and transmitting the data are possible. For example, a fully buffered system could be devised, or a simple polling system can be implemented at less cost in hardware but at greater cost in time requirements.

  A third alternative would be to provide an interrupt system whereby the individual data source notifies the computer via an interrupt that new data are available.

  Requirements, advantages, and tradeoffs of each of the candidate data communication techniques will be considered and recommendations made for each input and output variable.

- Protocol--For each data communication technique to be used, specific procedures will be established for proper control of the individual communication function.

- Synchronization--For some of the prototype system functions, relatively noncritical "macrosynchronization" among the various types of units and among replicated units of a given type will suffice for proper system operation. In general, synchronization of this type can be accomplished by proper polling sequences and the like. However, there may be some functions for which time synchronization will be more critical, and for which special synchronization must be provided. This possibility will be given special attention.

- Data validity--For a high reliability system, accuracy of data transmission will be extremely critical. The basic data communication system will therefore necessarily include data verification as a major consideration. Use of various techniques such as parity, hash totals, and check digits will be considered in the system specification.

- Data transmission noise--In computer installations such as those in aircraft where various system elements are widely separated, and where (electronic) data pulse fronts are relatively sharp, grounding problems sometimes lead to generation of "noise" pulses on the data lines that can cause errors in reception of the data. In addition, errors can be caused by such phenomena as lightning strikes on the aircraft and the consequent induced signals on the data lines. Potential problems of this type will be discussed with airline, research, and vendor personnel, so that system specifications can be prepared with a high assurance that problems of this type can be circumvented. It is possible that optical data transmission via glass or plastic fiber lines may prove to be satisfactory as a technique for interunit transmission for some or all of the data in the on-board system.

f. Maintenance Aspects - Ideally, any system--electronic or mechanical--would throughout its full life be free of any faults that would require either preventive (scheduled) or fault-forced (immediate) maintenance. The architecture of the SIFT system precludes the need for the latter type. It is necessary to consider scheduled maintenance procedures that do not include extensive disconnection of system components, probing of circuit boards with oscilloscopes and voltmeter probes, etc. Airline experience has consistently indicated that additional system faults are often caused by such well-intentioned but fault-prone procedures themselves. Rather, SRI considers that the scheduled maintenance should be a preflight "test and verification" (T/V) procedure, whereby the various system modules are exercised automatically, e.g., via prestored programs, and either approved as valid (verified), or flagged as faulty.* In the latter case, at least some minimal method of automatic diagnosis should be incorporated to designate, via printout or display, the faulty module so that it can be replaced with a minimum of effort and time.

The economic effect of different maintenance policies on airline operations should be considered. Analyses should be carried out to determine the conditions under which different maintenance policies are advisable; for example, one policy may be appropriate for a short-haul use while a different policy may be appropriate for long-haul use.

g. Reliability Analyses - As the design of the SIFT system becomes defined in progressively more detail, it is necessary that the reliability analyses carried out on the current contract be updated. As new data becomes available on fault statistics, it will be necessary to examine their effect on system reliability and in particular to determine if changes are therefore required in the design. The reliability analyses carried out under the existing contract will have to be extended to include consideration of the various input/output units, including the sensors and actuators of the aircraft. The analyses will also be extended to take into account all the different fault-tolerance procedures that are possible within the general framework of the SIFT concept.

h. Transient Behavior - As with fault-tolerant computer designs, a cause for concern is the possibility of a massive transient†

*Current plans are to incorporate, in a background mode, continuous testing of this type for all processor modules during real-time flight operations. It is further suggested that the status of each processor be indicated on the flight deck in some simple manner such as (1) green light--processor operation valid; (2) amber light--monetary (transient) fault detected; or (3) red light--processor outputs blocked from the system because of continuing faults.

†Such transients are typically caused either by lightning strikes or by other disturbances of the electrical system of the aircraft.

that causes multiple faults and perhaps multiple errors in the system. The approach to be used in dealing with this possibility consists of three parts.

First, it is necessary to collect data relevant to the problem of transients. Airlines and airframe manufacturers have significant data relating to this matter. Tests have been made on aerospace computers at NASA Houston and other sites. Other experiments at SRI and elsewhere have examined the effect of large electromagnetic fields on electronic equipment. It is hoped that these data plus consultation with experts in this field will enable an estimate to be made of the effects of such phenomena on an aircraft computer. Some tests may need to be carried out to answer specific questions on this issue.

Second, schemes must be developed to reduce the probability of such massive transients in the system. Such schemes must include aspects of shielding, improved grounding systems and, as just mentioned, the use of optical data links for the larger path-lengths external to the computer system itself.

Third, within the SIFT system, techniques must be developed for recovery from such transients. This may involve software techniques such as an automatic restart capability, and/or hardware techniques such as the provision of a highly protected, nonvolatile back-up memory to store critical state variables to assist recovery after a transient. Recovery speed requirements for these variables may require a special memory in addition to the general system back-up memory.

i. Diagnosis - The viability of the error detection and recovery strategies in SIFT relies on the freedom from faults of most of the SIFT units that are performing computations. Assuming a majority-vote strategy, there are double failures that cannot be tolerated. At the beginning of a flight, it is essential that all units, or possibly all units except one, are fault-free. We do not advocate the use of special test equipment to accomplish preflight checkout of the computer. Instead, the checkout is to be carried out by executing special diagnosing programs that flex the various system units, e.g., processors, memories, busses, I/O processors. During the present year's research, we also identified the need to carry out periodic, inflight, diagnosis of the hardware units that are not flexed during normal computation. This periodic diagnosis reduces the probability of multiple faults remaining undetected.

There has been extensive work on logic circuit diagnosis, from both a theoretical and a practical viewpoint. Much of the practical work has been carried out by the semiconductor manufacturers toward testing LSI chips as they emerge from production. This work is not adequate for our purposes since it relies on special test equipment (signal generators, probes, oscilloscopes) and since it does not guarantee complete coverage. The theoretical work has been concerned with developing diagnosing sequences that if applied to a circuit will determine if it is faulty. This work is attractive from our viewpoint since it relies on the circuit interfaces only. (The Computer Science Group of SRI has done extensive work in this area under commercial and NASA-ERC sponsorship). However, the theoretical work is not adequate since it typically assumes that the only fault mechanism is a gate being stuck at zero or stuck at one. It is known that LSI circuits exhibit a failure behavior which is significantly more complex.

Our approach will first involve technical discussions with semiconductor manufactures to determine the actual failure behavior. Preliminary discussions have indicated that the following failure behavior can be expected:

- All types of single gate failures, including input-output shorts, open-outputs, etc.
- Shorts between contiguous gates on a chip. This fault assumption precludes the development of test sequences that are based entirely on the logic diagram.
- Failures that occur only under maximal gate loading conditions. This fault seems to be manifested as input gate failures for some of the gates driven by the failed gate.

After identifying the failure behavior we will study the development of the sequences that will reveal the occurrence of the expected failures. During Step 2, we intend to develop appropriate techniques; the writing of actual diagnosing programs must await the procurement of hardware in Step 3.

j. Procurement Plan - Detailed plans will be drawn up for the tasks to be accomplished in Steps 3 to 5. These plans will incorporate for each task the following items:

- The specification of work to be accomplished
- The estimated time to accomplish the work
- The estimated cost of carrying out the work
- The qualifications required of organizations that could carry out the work
- Conditions of delivery and acceptance criteria
- A tentative list of candidate organizations to carry out the work.

The procurement plan will also define the interaction between the separate tasks. In particular, it will identify the critical path(s) in the development and the manner in which the procurement plan is intended to protect the plan as a whole from being jeopardized by failure to carry out any particular task.

The plan will consider methods that are possible for contracting this work, for example, the use of subcontracting or the issuing of independent contracts. The overall management of the development will be considered and recommendations made as to the way in which the separate efforts will be coordinated.

3. Design Review

Between Step 2 and Step 3 we anticipate a design review. This will be carried out by NASA personnel or their representatives in consultation with the SRI design team. The purpose of this review is to reexamine the design from the point of view of completeness and correctness and to check its appropriateness for the application set for which it is intended. At this point, it will also be possible to review the various estimates to timescale and funding that will have been prepared in the procurement plan in Step 2. Heavy involvement with airlines, airframe manufacturers, and avionics manufacturers will be desirable in this review.

4. Step 3

There are three major objectives of Step 3: hardware procurement and integration, software procurement and integration, and the development of a test facility. It is recommended that SRI continue to play a central role in this development step but that the major development of hardware and software be carried by organizations specializing in those fields. This way of organizing the development of a prototype has been used extensively by SRI with great success. In one case, we have carried out the role of system integrators for a mobile digital packet radio network, with radio and computing equipment being supplied by vendors in those fields. In another case, SRI is the system integrator in the development of a blind landing system for FAA. The design of SIFT greatly facilitates this kind of operation in that there is a high degree of functional independence among the various units of either hardware or software. It is thus relatively easy to specify individual procurements, with the final integration to be carried out after delivery. Our design methodology for preparing formal specifications and for defining the functional hierarchy of the system also makes independent procurement of parts of the system a practical strategy.

The integration of the various parts will involve the building of limited amounts of special hardware (e.g., the bus system), and also the writing of limited amounts of programs (e.g., the programs used for prototype tests).

a. Hardware Procurement - It is planned that as part of Step 2 we will have already determined those organizations that are qualified to act as suppliers of the processors and memories, which represent the major hardware components of the system. It will be necessary in Step 3 to prepare formal requests for bid from these organizations for each of the hardware units. Following evaluations of these bids, purchase orders or development contracts will be drawn up for procurement of equipment.

We anticipate that these actions will have been taken in the first few months of Step 3, thus enabling the eventual procurement to be completed after 9 months into Step 3. The remaining 3 months of Step 3 will be devoted to the integration of the hardware. This will be greatly facilitated by the prior development of the special hardware that is necessary for integrating the whole system. In procuring the major units of hardware, it is anticipated that, to a large extent, standard off-the-shelf units can be used with very minor modifications. We anticipate that many suppliers may be involved; for example, it may be desirable to procure main processors from an avionics computer manufacturer and input/output processors from an LSI microprocessor manufacturer.

As the prototype evolves, there will need to be continuing effort, primarily concerned with the details of the circuit technology that is used but also involving questions of packaging and interconnection. We expect that the first version of the prototype will use conventional technologies in these units but will evolve to become very similar to the eventual flight model that is planned in Step 4.

b. Software Procurement - The major software procurememt can be broken down into two parts, system software and application software. The system software will have been fully specified in Steps 1 and 2 and can be let out for bid using these specifications. The applications software will be specialized to the particular aircraft functions that are determined in Step 2 and will be greatly influenced by the type of aircraft that is to be the eventual test vehicle. Considerable gain may be had by procuring the application software from the same organization that is selected to supply the major hardware components, particularly if the latter is an avionics manufacturer.

We see that the organizing of the software procurement can be achieved in the first 3 months of Step 3, particularly when we take into account the preliminary actions that will have been taken in developing the procurement plan of Step 2, for example, the prior selection of one or more candidate organizations to accomplish the necessary work. It is expected that the actual procurement of the software will be possible in a period of 6 months, with the final 3 months of Step 3 devoted to the integration of the several components software.

c. Development of a Test Facility - In testing the prototype (Step 4 below) it will be necessary to provide an adequate test environment. This involved two major components, the connection of the prototype to simulated input and output units and the generation of appropriate test data. We foresee the setting up of a test generation facility based upon a general-purpose computer suitably programmed to generate the appropriate test signals.

d. Flight Model Packaging - Also to be included in Step 3 is the development of packaging techniques for the flight model. In this task we would expect that the experience of avionics equipment manufacturers would be directly applicable, and in such case we anticipate that this task would be a relatively small effort. The major novelty to be incorporated is the provision for protection against the effects of electromagnetic disturbances. We also see the possibility of some problems in incorporating optical coupling between units while maintaining the integrity of any required shielding.

5. Steps 4 and 5

The major activities of Steps 4 and 5 are the testing of the prototype and the building and testing of the flight model. The tasks to be accomplished in these steps are shown in the accompanying chart. As stated previously, we see that the flight model should be an evolution from the prototype rather than a completely new design.

We anticipate that the technology of the flight model will be very closely related to the prototype. One scheme would be to use a set of processors and memories from a minicomputer manufacturer, which in the prototype would be constructed using conventional circuit board techniques, and to use a ruggedized version of the same hardware in the flight model. This scheme is very attractive in that much of the supportive design work would not be changed in going from the prototype to the flight model. It would include the design of special equipment (busses, interfaces, etc.), the software system (executive and application programs), and the test procedures and facilities have been designed for the prototype.

The approach suggested above might preclude the use of advanced technology components such as LSI circuitry, but this is considered a small risk in view of two factors:

- It is unlikely that the flight model would be built using radically new technology because of the untried nature of such a technology and the lack of data on its reliability.
- Any LSI components that are suitable for the flight model will probably be preceded on the market by the same type of equipment implemented in a less advanced technology.

In testing the flight model, a suitable research aircraft environment will be required. We understand that NASA Langley is equipped with such a facility and anticipate that it can be used for testing. For this reason we see a strong involvement of NASA personnel in these steps.

IV THE SIFT CONCEPT

A. Introduction

In recent years, a number of fault-tolerant architectures [Refs. 1-4] have been devised and in some cases analyzed and implemented. Most of these architectures depend heavily on special hardware structures to achieve their fault-tolerance. While hardware mechanisms are fast and economical, they are severely limited in the kinds of faults they can treat. Also, such mechanisms cannot be easily modified to reflect changes in performance and reliability requirements.

The SIFT (Software-Implemented Fault-Tolerance) computer [Ref. 5] is founded on a new approach to fault-tolerant computing that puts strong emphasis on the use of software for achieving reliability, with corresponding de-emphasis on special hardware. The software that is critical to the reliability of the system is designed in accordance with a hierarchical design methodology [Refs. 6] that permits the stating and proving of formal properties relating to the system's correctness. A Markov process model is used to analyze SIFT's reliability as a function of various error-detection and reconfiguration strategies. The reliability model is incorporated into SIFT's formal description, permitting the demonstration that the model indeed reflects the behavior of the system.

The remainder of this chapter is concerned with the goals of the SIFT system and a narrative description of its operation.

We believe that the SIFT concept is useful in many application areas where high reliability is at a premium. Although a system might have extensive redundancy, if the software or hardware mechanisms that manage the redundancy are incorrect, the system will still be unreliable. Later chapters show how formal verification methods can be used to ensure that the present system is correct. We have attempted to develop a precise statement, in terms of a Markov-like model, of the behavior of SIFT in the hierarchical decomposition of the SIFT software to facilitate its verification. We believe this is the first attempt to specify formally a fault-tolerant system.

We think that it will be possible to verify formally the SIFT software, because it is relatively simple and because it is highly structured. Although SIFT exhibits some of the features of a modern operating system, e.g., task dispatching and (limited) memory management, it is much simpler than other systems being considered for verification [Refs. 6-7].

B. SIFT Performance and Reliability Goals

SIFT is a general-purpose computer intended for use as the central computer in advanced commercial aircraft. The computational requirements [Ref. 8] for the aircraft environment can be summarized as follows:

- The control features can be broken down to about 20 tasks, e.g., engine control, stability augmentation, and collision avoidance, that must be serviced. The computer is designed so that it could service the fastest tasks every 1 msec.
- The reliability requirement is dependent on the task. The tasks that are flight critical must exhibit a failure rate not exceeding 10^{-9}/flight-hour. This high reliability cannot be achieved with current hardware technology without redundancy.
- The programs and associated data that implement the tasks are of moderate size.
- A task might require input data from one or more other tasks (typically only a few words). No other type of communication exists between tasks.
- Input from aircraft sensors can be accomplished by reading multiple copies of sensors, and in some cases the output can be delivered to multiple actuators.

C. SIFT System Design

The SIFT computer (Figure IV-1) conists of a number of hardware modules, each composed of a memory and a processing unit. The individual processing units within the modules are connected to the corresponding memory units with wide-bandwidth busses. The intermodule bus organization (B_1,B_2,B_3) is designed to allow a processor to read from any memory but

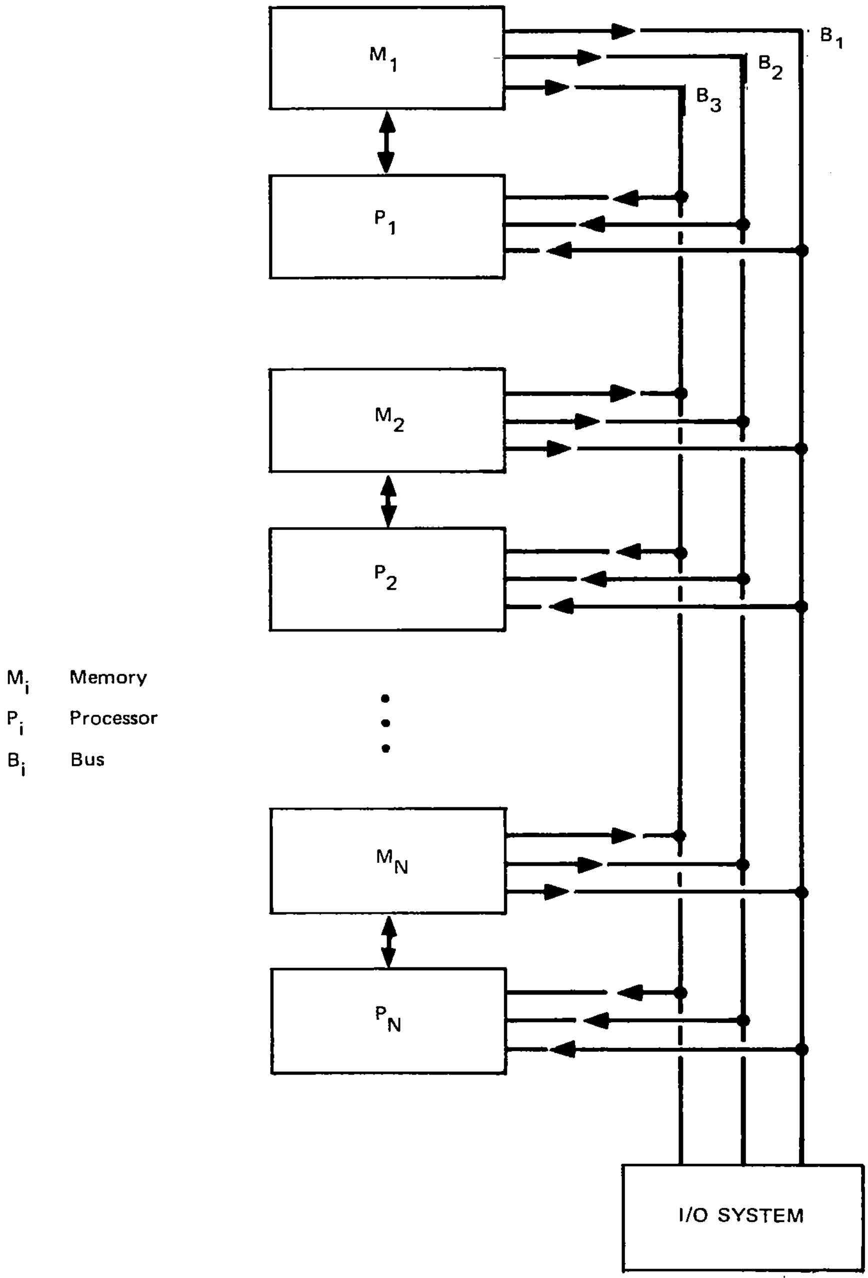


FIGURE IV-1 SYSTEM CONFIGURATION

not to write into other memory units. The intermodule bus is expected to have a much lower bandwidth than an intramodule bus because of the relatively low rate of information flow between tasks.

The input/output system assumed to be connected to the busses B_1, B_2, and B_3, as shown in Figure IV-1, consists of all the noncomputing units, for example, transducers, actuators, and sensors. The part of the total input-output that is carried out by program, such as formatting or code conversion, is handled in the same manner as for any other task; that is, it is replicated in several processors.

All large tasks are broken into a number of subtasks in such a way than no subtask requires more computing power than can be supplied by one processor. The tasks are given the designations, A, B, C,...; the processors are numbered 1, 2, 3... . Each processor is capable of being multiprogrammed over a number of tasks, as illustrated in Figure IV-2.

The control of the computing system is carried out by a number of functions that can be segmented into two classes:

(1) Local Executive: functions that apply to each processor (e.g., dispatching,* voting, reporting errors, loading new task programs).

(2) Global Executive: functions that are global to the system (e.g., allocation and scheduling of work load, reconfiguring).

A complete set of the software functions of the Local Executive is present in each processor; those of the Global Executive are carried out in a sufficient number of processors to provide the degree of fault tolerance required. The functions are realized by programs that have the same task structure as all other programs.

The normal operating mode for a processor carrying out a task is as follows: Data required for the task are assumed to have been computed by several processors (possibly including the same ones carrying out the task). The input data are read from the several processors where copies

*The bus logic envisioned does not use voting. The number of busses is variable. The number 3 is chosen for convenience of discussion.

TASKS	PROCESSORS 1	2	3	4	5	6	• • • n
A	X	X		X			
B			X		X	X	
C		X					
D	X		X	X			
E		X		X	X		
F			X		X	X	
G	X	X	X				
H	X			X		X	
I	X		X		X		
J	X	X	X	X	X	X	
• • • N	• • •						

FIGURE IV-2 EXAMPLE OF TASK/PROCESSOR ALLOCATION

exist. A validation is now carried out, typically by a vote among the several values of each datum. If any of the copies of the input data are found not to agree, this fact is noted for later processing by the executive. During the reading of the different versions of a data item, different busses are used in order to protect against errors in bus operations. The computation of the task is now carried out; the results are left in the memory of the module, and note is made (in the module) of the fact that the task is computed.

If discrepancies are detected between the several versions of a data object, diagnosis programs in the global executive determine which unit is at fault. Reconfiguration is achieved by having the several versions of the global executive indicate to each local executive which tasks should be performed and which other processors should replicate the calculations for each task. All the local executives examine each of the global executive versions and independently vote on these directions. That is, each local executive decides which of the reconfiguration directions it will accept, using a majority rule. A faulty processor might not heed the directions of the global executive, but, based on the instructions of the global executive, operative processors will ignore the faulty processor. Thus the worst impact of a faulty processor is that it will exert a slight load on the bus system.

D. The Design Methodology

The SIFT design has been specified in accordance with a formal design methodology that originated with D. Parnas [Refs. 9,10] and has been extensively developed at SRI [Ref. 6]. The chief reasons for using such a medium were (1) to impose a discipline on the design process assuring a clearly-structured, easily modified design; (2) to simplify verification of the correctness of that design; and (3) to facilitate the analysis of certain reliability properties. Previous use of the methodology has been concerned with only the first two of these aims. The SIFT effort is the first instance of its use in connection with fault-tolerant design.

The methodology can be viewed as a formalization of Dijkstra's step-wise refinement concept [Ref. 11]. The central idea is to decompose the design into a hierarchy of modules. The highest modules in the hierarchy provide an abstract, global description of the system's capabilities. Modules at lower levels of the hierarchy serve as building blocks for implementing the highest-level module. Modules at still lower levels are building blocks for implementing those at intermediate levels, and so on. The modules lying near the top of the hierarchy thus tend to be highly abstract, while those at or near the bottom tend to be more concrete. In the SIFT design, for example, descriptions of real machine hardware appear at the bottom level, and a set-theoretic model of the workings of the system appears near the top.

Each module in the hierarchy is specified in terms of a set of abstract data structures (called V-functions) plus a set of operations (called O-functions) that change the values of these structures. At any given moment, the state of the module is determined by the aggregate of the values of its V-functions. O-function calls thus cause transitions from one state to another. The V-functions and P-functions of each module are specified using a formal language. The specifications describe what happens when each of the functions of a module is called. Specifications for O-functions consist of assertions, i.e., logical formulas that relate the state (values of V-functions) of the module before an O-function call, to the state resulting from the call. Module specifications have other aspects [Ref. 12] that are discussed in greater detail in Chapter VIII.

E. Design Features of SIFT

This section is concerned with the more important design decisions that we have formulated for SIFT.

1. Task Dispatching

In the aircraft application, most computations are iterative. Thus, tasks are executed on a regular basis with a frequency that is dependent on the application. Noniterative tasks can also be handled within this scheme with no apparent difficulty. Dispatching is accomplished via a fixed schedule that is stored in each processor. Three periods, or frames, of a possible schedule are depicted in Figure IV-3. The maximum task iteration rate, or frame rate, is determined by the frequency of the "clock-ticks," which can be derived from an ultrareliable system-wide clock, or via a clock associated with the processor in question. For the latter option, the clocks in the respective processors are loosely synchronized. The synchronization requirement is that no processor is to commence iteration n of a task before iteration n-1 has been completed on all processors executing that task. Thus, the slowest processor should not slip behind the fastest processor by more than one frame.

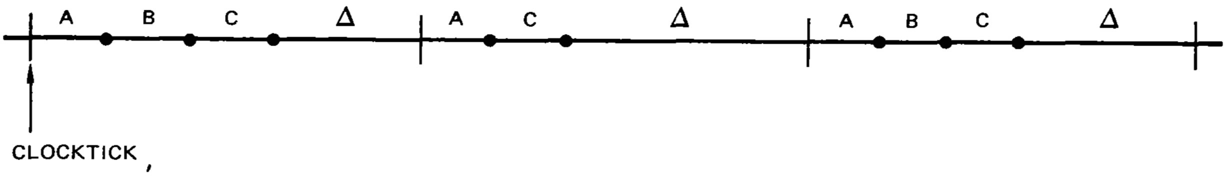


FIGURE IV-3 SNAPSHOT OF A SAMPLE SCHEDULE

In the example, tasks A and C are dispatched every frame, and task B every two frames. Note that task C is not dispatched at the same relative time in each frame. For the application being considered, this is an allowable perturbation. Each of these three tasks is intended to execute to completion during each frame in which it is dispatched, thus obviating the need for many mechanisms usually associated with multiprogramming. A task that for some reason does not complete the iteration by the end of its allotted time is halted in favor of the next task. In Figure IV-3, Δ designates an interval in which the processor is in a noncomputing state, awaiting the next clock-tick.

2. Task Communication

Each task is processed according to the following scheme:

```
READ DATA FROM EACH TASK SUPPLYING INPUTS
COMPUTE
```

WRITE DATA TO A BUFFER FOR EACH TASK THAT REQUIRES IT AS AN INPUT

We are assuming that any data a task requires for the execution of an iteration is obtained from output data computed by the previous iteration of the same and other tasks.

Since SIFT does not allow any processor to write directly into the memory of another processor, the WRITE DATA operation is accomplished by using a buffer that resides in the writing task's processor. If B is to write data for A, B deposits the data in a buffer that can be subsequently read by A, which may be executing in the same or in other processors.

The READ DATA operation, say by task A from task B, is implemented as follows: The data deposited by each version of B, in its own buffer, is read, and the majority value of the several versions of the data is computed by each version of A. In order to tolerate bus failures, each version of B is read via a different bus. The disagreements reported by the aggregate of processors are used to locate faulty processors and busses. Some of the data required by a task are obtained from external sources, which can themselves be viewed as tasks replicated for reliability enhancement. However, the various instances of a given input datum are not likely to be identical because of slight differences among real physical data sources. Such slight disagreements can be prevented from causing a vote disagreement by providing a mechanism whereby a task performing a read can specify the precision expected among the various instances.

When task A votes on the data computed by several instances of task B, these data must all be associated with the same iteration of B. Consider the task timing illustrated in Figure IV-4. Since all instances of the tasks executing in different processors are not assumed (nor intended) to be mutually synchronized, and if only one buffer were provided per processor per writing task, then iteration n of task A in P2 would read data from iteration n-1 from A in P2, but from iteration n from A in P1. This problem is resolved by providing two buffers in each

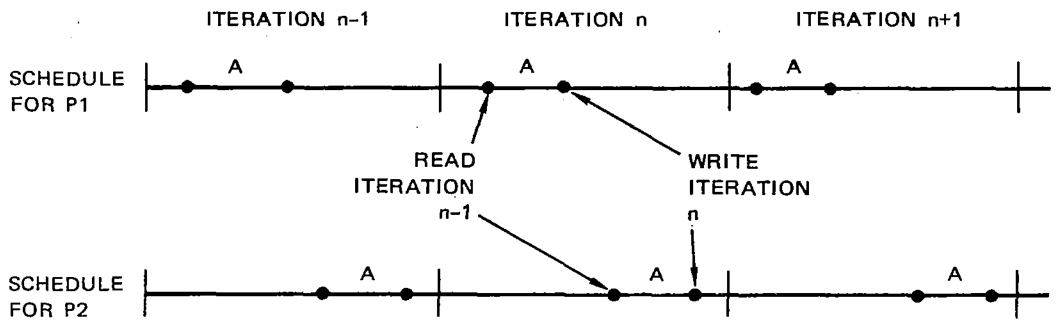


FIGURE IV-4 TASK SCHEDULES DEMONSTRATING THE NEED FOR TWO COMMUNITY BUFFERS

processor for each writing task, one which is written into on odd-numbered iterations and the other on even-numbered iterations.

3. Detection and Location of Processor and Bus Failures

In this section, we discuss the method whereby the global executive can determine which processor or bus is faulty, based on the error reports of each of the processors. For simplicity, we assume triplication of processors and busses, so that single faults can be tolerated and located. The generalization to general redundancy is not difficult.

On behalf of a task, a processor will read data from other processors that, in the absence of faults, should be identical. If one of the data instances, as read by a processor, is in disagreement, then the processor will record the identity of the disagreeing processor and the identity of the bus used. The global executive will examine the processor-bus discrepancies reported by each of the processors and attempt to identify the processors(s) and/or bus(ses) that are faulty.

The following four fault types cover all possible single processor and bus fault occurrences that could lead to erroneous results:

(1) Processor computation and/or voting (PCV)--A processor produces erroneous values in computing results for tasks and/or in performing a vote and deciding which input(s) to the vote is in disagreement with the majority.

(2) Bus transmission (BT)--A bus changes the value of a word as it is transmitted between processors.

(3) Processor-bus in initiating reading (PBIR)--A processor is incapable of initiating a read operation via a particular bus.

(4) Processor-bus in depositing data (PBDD)--A processor is incapable of depositing data onto a particular bus.

To illustrate the fault location algorithm, suppose that on each iteration, a task reads data from a previous iteration of itself. The task executes on three processors and uses three distinct busses for the read operation. For odd iterations, the bus assignment is as in Figure IV-5a and for even iterations as in Figure IV-5b. The interpretation of the matrices is as follows: the P1 row of Figure IV-5a indicates that when P1 reads from P1 it uses bus B1,* when P1 reads from P2 it uses Bus 2, and when P1 reads from P3 it uses B3. It is apparent

READING PROCESSORS \ READ-FROM PROCESSORS	P1	P2	P3
P1	B1	B2	B3
P2	B3	B1	B2
P3	B2	B3	B1

(a) ODD-ITERATIONS

READING PROCESSORS \ READ-FROM PROCESSORS	P1	P2	P3
P1	B3	B1	B2
P2	B2	B3	B1
P3	B1	B2	B3

(b) EVEN-ITERATIONS

FIGURE IV-5 BUS ASSIGNMENTS TO ENABLE SINGLE FAULT LOCATION

*It is, of course, feasible for a processor in reading from itself to use the internal processor-bus connection which is of a higher bandwidth than the inter-processor bus system. However, in this discussion we assume that the bus system is used for all read data operations. This avoids the need for a separate fault location algorithm when a task reads data from a task in its processor.

that for either assignment, the occurrence of any one of the four fault types leads to an error which is masked by the voting scheme. It remains to show that a fault can be pinpointed to a processor, a bus, or a processor-bus connection (type 3 or 4).

The fault location algorithm is illustrated in Figure IV-6, for a fault of each of the four types. In the case of a type 1 failure involving P1 (P1 computes erroneous results for the task and/or produces erroneous, and arbitrary, error reports), from the reports of P2 and P3 it is apparent that P1 is faulty and that its error reports should be ignored. The only possibility for ambiguity is between a type 1 failure (P1 voting incorrectly) and a type 3 failure (P1 unable to initiate a read via B1). Both failure types could produce the same error report (although it is unlikely that for a type 1 failure this would be the case), in which case the global executive would first suspect a type 3; the subsequent use of P1 could actually reveal the presence of a type 1 failure.

A faulty unit should be identified shortly after the detection of the failure by the voting processors. The global executive will then instruct all processors to ignore the faulty processor or not to use the faulty bus. This process is well known as adaptive voting, [Refs. 13, 14] and enables SIFT to tolerate some multiple faults that may occur before reconfiguration can be completed.

After the identities of the faulty units are known to the global executive, it initiates a reconfiguration process, so as to utilize effectively the remaining operative resources. After the reconfiguration is complete, the allocation of tasks to processors and busses could be entirely different from that prior to the reconfiguration. As a result of the reconfiguration, processors might be given new schedules, and the processor and bus allocation tables in each processor must be updated.

The formulation of new bus assignments after detection and location of a bus failure is easily carried out by the global executive. It is feasible for the global executive to compute in real time new task allocations and schedules in response to each processor failure.

ERROR REPORTS

FAILURE	P1-odd	P1-even	P2-odd	P2-even	P3-odd	P3-even	REMARKS
PROCESSOR-COMPUTE AND VOTE (P1)	?	?	(P1,B3)	(P1,B2)	(P1,B2)	(P1,B1)	MAJORITY AGREEMENT THAT P1 IS FAULTY, INDEPENDENT OF P1's REPORT
BUS TRANSMISSION (B1)	(P1,B1)	(P2,B1)	(P2,B1)	(P3,B1)	(P3,B1)	(P1,B1)	UNANIMOUS AGREEMENT THAT B1 IS FAULTY
PROCESSOR-BUS READ INITIATION (P1,B1)	(P1,B1)	(P2,B1)	-	-	-	-	ONLY P1 REPORTS ERRONEOUS RESULTS, ALWAYS INVOLVING B1
PROCESSOR-BUS, DEPOSITING DATA (P1,B1)	(P1,B1)	-	-	-	-	(P1,B1)	TWO DIFFERENT PROCESSORS REPORT ERRONEOUS RESULTS INVOLVING P1 AND B1

FIGURE IV-6 ILLUSTRATIONS OF FAULT-LOCATION ALGORITHM

An alternative approach is to precompute the allocations and schedules for all possible processor fault occurrences. We are now using this latter approach, since the storage requirements are small.

A global executive instance could reside in each processor, and thus assume the entire responsibility for reconfiguring that processor, based on the error reports of all processors. Besides deriving new schedules and bus assignments for its processor, each global executive updates tables and loads new tasks into the processor. However, in order to minimize the executive computational load on the processors, we have decomposed the executive tasks into two parts: (1) a global executive task, residing in at least three processors, which computes new allocations and schedules for each processor, and (2) a local executive residing in each processor, which determines what its new configuration should be by voting on the three global executive instances.

F. The Logical Structure of SIFT

The preceding discussion has summarized the primary operation of SIFT. In this section, we consider a hierarchical decomposition of the system.

The logical structure of SIFT consists of a hierarchical layering of modules, which we designate as system modules, and some programs, namely the application tasks and the global and local executives, that utilize the facilities of the external interface of system modules. For simplicity, we will say that the tasks call the functions of the interface. Each system module may be considered as an abstract machine that maintains a state (represented by V-functions) and provides operations (O-functions) to modify the state. The application tasks and executive may then be considered as programs that run on the abstract machines. The data required by the tasks are distributed among the system modules.

Each task, including the global executive, executes in some subset of the processors. The fault status and fault schedules modules, which are accessed only by the global executive, appear only in processors executing the global executive.

It is assumed that the real-machine module describes the ordinary machine instructions, e.g., add, store. In reality, the tasks and all modules above the hardware will call these instructions and thus should be depicted as connecting to the real machine. We will not concern ourselves with these connections here since they are not essential to the fault-tolerance, reliability, and scheduling properties of SIFT.

G. Discussion

The SIFT concept embodies a number of ideas whose usefulness extends beyond the particular application for which the system is designed. Because conventional, off-the-shelf processing units comprise the bulk of the hardware, the system can be easily and inexpensively adapted to a broad range of needs. Moreover, because the degree of reliability achieved by the system depends on the number of processors used and on scheduling strategies rather than on built-in aspects of the design, it can be varied according to performance and cost requirements of the application.

The use of a formal design medium for purposes of specification, validation, and reliability modeling can be expected to play an important role in future designs of fault-tolerant computers. While a system might make extensive use of redundancy, the system will not be reliable unless the software or hardware mechanisms that manage the redundancy are correct. Similarly, the formulation and use of elaborate reliability models is of little value if it cannot be demonstrated that these models actually reflect the behavior of the system. We believe that SIFT constitutes a major step in the direction of fault-tolerant systems whose correctness and reliability can be verified.

REFERENCES

1. A. Avizienis, G. C. Gilley, F. P. Mathur, D. A. Rennels, J. A. Rohr, and D. K. Rubin, "The STAR (self testing and repairing) computer: An Investigation of the Theory and Practice of Fault-Tolerant Computer Design," IEEE Trans., Vol. C-20, No. 11, pp. 1312-1321 (November 1971).

2. A. L. Hopkins, Jr., "A Fault-tolerant Information Processing Concept for Space Vehicles," IEEE Trans., Vol. C-20, No. 11, pp. 1394-1403 (November 1971).

3. F. P. Maison, "The MECRA: A Self Reconfigurable Computer for Highly Reliable Process," IEEE Trans., Vol. C-20, No. 11, pp. 1382-1388 (November 1971).

4. "Design of a Modular Digital Computer System," NASA CR-123655, 1972.

5. J. H. Wensley, "SIFT-Software Implemented Fault Tolerance," Proceedings of the Fall Joint Computer Conference, Vol. 41, pp. 243-253 (AFIPS Press, Montvale, New Jersey, 1972).

6. L. Robinson, K. N. Levitt, P. G. Neumann, and A. K. Saxena, "A Formal Methodology for the Design of Operating System Software," in Current Trends in Programming Methodology, Vol. 1, R. T. Yeh (ed.) (Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1976).

7. P. G. Neumann, L. Robinson, K. N. Levitt, R. S. Boyer, and A. Saxena, "A Provably Secure Operating System," final report under Contract DAAB03-73-C-1454, Stanford Research Institute, Menlo Park, California (June 1975).

8. R. S. Ratner, et al., "Design of a Fault-Tolerant Airborne Digital Computer," Volume II--Computational Requirements and Technology, Final Report. NASA CR-132253, 1973.

9. D. L. Parnas, "Information distribution aspects of design methodology," in Proc. IFIP Congress 1971 (North-Holland Publishing Company, Amsterdam, 1972).

10. D. L. Parnas, "A technique for module specification with examples," Comm. ACM., 15, 5, pp. 330-336 (May 1972).

11. E. W. Dijkstra, "Notes on structured programming," in Structured Programming, C.A.R. Hoare (ed.) (Academic Press, New York, New York, 1972).

12. L. Robinson, "Specification Techniques," Proceedings of the Thirteenth Design Automation Conference (June 1976).

13. W. H. Pierce, "Adaptive Vote-Takers Improve the Use of Redundancy," in Redundancy Techniques for Computing Systems, pp. 229-250 (Spartan Books, Washington, D.C., 1962).

14. J. Goldberg, K. N. Levitt, and R. A. Short, "Techniques for the Realization of Ultra-Reliable Spaceborne Computers," Final Report. NASA CR-80019, 1966.

V TASK STRUCTURE, ALLOCATION AND SCHEDULING

A. Introduction

This section of the report describes effective procedures for the following major elements in the design of a fault-tolerant computer system:

- Analysis of tasks to derive applicable flight phase descriptions.
- Determination of the numbers and sizes of the redundant processor and memory units required for the reliable execution of the specified tasks.
- Allocation of tasks to specific processor and memory units to achieve some balance of processing and memory loads.
- Specification of task scheduling and reconfiguration procedures and symbologies in a concise, efficient from suitable for implementation.

The analysis considers representative tasks for those aircraft functions that have been previously described as candidates for control by SIFT. While this set is more comprehensive than the set of tasks that will be initially implemented in the prototype, it is important to develop a design methodology for SIFT that will be applicable up to the desired conceptual level of technological sophistication. This approach assures an upward compatible design that will meet the requirements of a smaller scale prototype while not constraining or invalidating future levels of system expansion.

This section also considers schedule implementation factors such as schedule storage, the impact of system degradation, the practical implications of task criticality class, the change of schedules with flight phase change, and approaches to deriving new task schedules dynamically during flight.

B. Flight Phase Analysis

The set of flight-related application tasks previously described in the SRI report entitled "Design of a Fault Tolerant Airborne Digital Computer" [Ref. 1] was examined to establish configurations of active tasks sufficient to support the major flight phases potentially encountered in normal flight. The intent of this study was to determine the variations in the task profiles of the various phases. It became necessary to distinguish between tasks that were actively being executed, those that were serving a vital backup role (referred to as passive allocation), and those tasks that performed secondary roles by actively confirming and augmenting the results of the primary tasks. In some instances, certain tasks were not required at all for certain phases. The derived flight-phases (Table V-1) constitute the major operational modes for which SIFT must provide task allocation and scheduling among the multiple processors and memories, as discussed in the following portions of this section.

An anomaly state presenting a "Navigational Failure" is also specified in Table V-1 as an illustrative example. Although it is represented as a phase, it is clearly a state that may need to be accommodated during several of the described phases. The primary change is the shift of the navigational support system from the VOR/DME and Multiple-DME to the Omega or satellite equipment as primary support, with reliance on Air Data for secondary support. Thus, the task schedules could be modified by simple replacement of the preferred primary and backup navigation systems by the appropriate secondary set of primary and backup systems.

C. Review of Task Characteristics for Flight Phase Assignment and Processor-Memory Unit Allocation

The set of flight tasks that were to be considered for initial SIFT implementation are listed in Tables 3 and 4 of the SRI report entitled "Design of a Fault Tolerant Airborne Digital Computer" [Ref. 1]. Further qualification and modification of the characteristics of these tasks have been made to facilitate schedule development. Values of some properties

Table V-1

SYSTEM CONFIGURATION ANALYSIS

Task		Flight Phase						
Code	Application	Takeoff	Climb Descent	Cruise	Initial Approach	Landing	Missed Approach	Navigational Failure
A1	Attitude Control	—	P	P	P	—	P	P
A2	Flutter Control	—	P	P	—	—	—	—
A3	Load Control	—	S	S	S	P	S	S
A4-6	Autoland	—	—	—	—	P	—	—
A7	Autopilot	—	P	P	P	B	P	P
A8	Attitude Indicator	P	P	P	P	P	P	P
B2	Inertial	P	P	P	P	P	P	P
B3	VOR/DME & Multiple DME	—	P	P	P	P	P	—
B4	OMEGA or Satellite	—	B	B	B	B	B	P
B5	Air Data (Navigation)	—	—	—	—	—	—	B
B7	Flight Data	—	P	P	P	P	P	B
B8	Airspeed, Altitude	P	P	P	P	P	P	P
B9	Graphic Display	—	P	P	P	P	P	—
B10	Test Display	S	S	S	P	P	P	S
C1	Collision Avoidance	P	P	P	P	P	P	P
C2	Data Comm., A/C	P	P	P	P	P	P	P
C3	Data Comm., Air/Ground (DABS)	S	S	S	P	P	P	S
D1	AIDS	S	S	S	S	S	S	S
D2	Instrument Monitor	S	S	S	S	S	S	S
D3	System Monitor	S	S	S	S	S	S	S
D4	Life Support	—	P	P	S	—	—	P
D5	Engine Control	P	P	P	P	P	P	P

Symbols: P: Prime; S: Secondary; B: Backup; —: N/A needed to be assigned, and ranges of values required unique assignments before schedule development could proceed. In addition, the local and global executived have now been specified to a degree permitting values with a sufficient level of confidence to be assigned to their description. Likewise, the flight-phase-change tasks and the reconfiguration tasks can be specified more accurately. A summary of the modifications and additions that have been made to the tables are given in Table V-2.* The resulting revised set of task modules and their properties is shown in Table V-3. These data can now be used to allocate task sets to processor-memory units and to develop a suitable task scheduling algorithm.

D. Task Allocation and Schedule Generation

Procedures are outlined here for the allocation of the various tasks to specific (redundant) processor and memory units, and for the organization of the scheduling of those tasks.

The problems of allocating and scheduling these tasks are considered in some detail. Assumptions regarding the characteristics of the tasks include:

- All tasks pertinent to a flight phase are resident in main memory during that phase.
- Reconfiguration can occur due to flight phase change, processor-memory failure or pilot intervention.
- Software task replication in separate processor-memory units is used to achieve fault-tolerance.
- Replicated tasks will be only loosely synchronized.
- The tasks operate on a real-time basis and have stringent execution periodicity requirements.

*The alphanumeric task designators are those used in Table V-1.

Table V-2

ADJUSTMENTS AND MODIFICATIONS
TO THE INITIAL TASK SPECIFICATIONS

- Tasks A4, A5, and A6, the Autoland Tasks, were coalesced into one task.
- Task B1, the Supervisor Task, was merged with the Local Executive, LE.
- Task B4, DME/OMEGA, was combined with Task B3, VOR/DME, since they would not run concurrently. Also, B4 was assigned the same MIPS as B3.
- Task B5, Air Data, was assigned an iteration rate per second of 5 and a MIPS of 0.001.
- The criticality class of Task C2, Data Comm. A/C, was set to 1 assuming that the backup is under pilot direction.
- Variable task criticality class assignments were fixed by assuming the highest value.
- Task D3, System Monitor, was assigned an iteration rate per second of 5. Multiple or variable iteration rates were chosen to have the highest value.
- Task LE was added to the table for the Local Executive and is totally replicated. It has to run as frequently as the most frequent module, that is, with an iteration rate of 670 per second. The number of instructions per iteration was determined to be 50.
- The Global Executive, GE, was also added and was assigned an iteration rate of five per second and 200 instructions per iteration.
- Infrequent requirements -

	Instructions per Iteration*
Reconfiguration - Global	100
Reconfiguration - Local	120
Flight phase change - Global	100
Flight phase change - Local	120
Program load	20 X number of words

*Criticality class = 1.

Table V-3

TASK MODULE PROPERTIES FOR SCHEDULING ASSIGNMENTS

Task	Iteration Rate/Sec.	Period in Sec. (X10^3)†	Instructions Iteration	MIPS	Criticality Class	Memory
A1	20	50	1150	0.023	1	2075
A2	250	4	276	0.069	1	92
* A3	240	4 (3)	58	0.014	3 *	60
* A4,5,6	160	6.25 (6)	344	0.055	1 *	1025
A7	5	200	200	0.001	4	250
* A8	30	33.3 (30)	2567	0.077	1 *	1310
B2	25	40	1360	0.034	2	2250
* B3	5	200 (180	800	0.004	4 *	300
B4	5	200	800	0.004	4	505
B5	5	200	200	0.001	4	135
B6	0.2	5000	5000	0.001	4	315
B7	5	200	5600	0.028	4	550
* B8	16	62.5 (60)	562	0.009	4 *	430
* B9	8	125 (120)	4000	0.032	4 *	6250
B10	10	100	1900	0.019	4	9340
* C1	670	1.5	31	0.021	4 *	1200
* C2	5	200	1200	0.006	1 *	610
* C3	4	250	250	0.001	4 *	562
* D1	4	250	500	0.002	5 *	1300
* D2	5	200	2800	0.014	4 *	1900
* D3	5	200	200	0.001	1 *	1000
* D4	0.5	2000	2000	0.001	1 *	1000
* D5	33	30	3606	0.119	1 *	1500
* LE	670	1.5	50	0.034	1 * T	320
* GE	5	200 (180)	200	0.001	1 *	1100
* REC-GE	IRREGULAR		100		1 *	
* REC-LE			120		1 * T	
* FPC-GE			100		1 *	
* FPC-LE			120		1 * T	
* PROGRAM LOAD			20‡		1 * T	

Abbreviations:
FPC - Flight phase change
LE, GE - Local and Global Executives
REC - Reconfiguration
T - Totally replicated

[*] Most demanding phase, Autoland

[†] The values in parentheses are period assignments somewhat shorter than the desired requirements, but representing convenient multiples of the smallest period (1.5 msec.), and of subsequent higher multiples, for schedule derivation. Note that in no instance was the period changed by more than 30%.

[‡] Times the number of words

Task allocation in the SIFT context embodies the selection and assignment of tasks to one or more processor-memory units. The objectives of allocation are:

- To achieve balanced task loading in terms of both processing and memory requirements.
- To allow enough spare capacity to permit reconfiguration with one less memory-processor unit.
- To accomodate supplementary "passive" allocation of critical tasks that demand processing within a time frame that does not allow for reconfiguration.
- To permit either full task allocation or single flight-phase task allocation with reconfiguration to achieve flight-phase change.

Schedule generation has a similar set of objectives:

- A technique with an unequivocal set of task sequence assignment rules.
- A schedule specification that can be efficiently stored, applied, and changed.
- A derivation methodology that is flexible and can be shown to achieve the required task execution periodicity.
- A representation that is easily interpreted and implemented.

Prior to detailed allocation algorithm development, consideration should be given to the way that task schedules will be implemented. Of particular interest here are such factors as:

- The resource penalty for loading all normal tasks belonging to any flight phase schedules so that no reconfiguration will be necessary.
- When a failure or set of failures occurs, the method by which new schedules are derived and implemented.
- The extent to which tasks can be replaced by pilot control so that schedules may be revised by simply eliminating failed tasks.
- The form in which schedules are to be stored and/or the way they will be generated on-line.

The resolution of these design factors is dependent on the task replication scheme, passive allocation techniques, failure state procedures, and the total number of processor-memory units. If a generalized approach is taken to addressing these factors, the factors might be evaluated from several alternative approaches to determine the impact of these assumptions on either the required processor-memory resources, the simplicity of schedule derivation and implementation, or the reconfiguration method. For instance, if we triply replicate all tasks and, for all flight phases simultaneously, allocate them over five processor-memory units, would resources of any of these units be exceeded, and what is the size memory units that is required? Likewise, what is the resulting accumulated distribution of processor and memory resources? These and other similar questions are examined in this section.

The problems associated with reconfiguration resulting from system failures are considered first and may determine the amount of spare capacity that must be designed into the system to assure redistribution of tasks on a failed processor-memory unit onto the other operational units. A less critical task that is adequatly replicated initially may not need to be reassigned. Very critical tasks either must be reassigned to another processor-memory unit, must have adequate backup via another task that is operational, or must have been passively allocated on some other processor. Tasks that cannot tolerate missed iterations and that belong to a high-criticality class should be passively allocated to guarantee immediate takeover of the critical task processing from a failed unit.

One approach is to take all the tasks in any of the phases and distribute them across processors. However, this distribution would lead to rather heavily loaded processors, and one or more additional processor units would be required to support it. Such a distribution would, however, facilitate reconfiguration and flight phase change.

If there is only one flight phase per allocation or if, at least, not all tasks are loaded into the system all of the time, then the occurrence of a failure requires either accessing stored schedules or having a method of automatically generating them. Such methods are considered in detail in the next section. Another approach is to depend upon pilot intervention for certain noncritical support functions and to more than triply replicate the critical tasks. While these approaches are all viable, the current documents need only demonstrate the feasibility of at least one such allocation and scheduling scheme.

In order to estimate the minimum adequate number of processors to forestall degradation because of a single processor or memory unit loss, a simple approach might be taken. First, all tasks above minimum criticality must be at least triply replicated to assure an adequate level of confidence. Three replications are sufficient, but four give an additional margin that allows one processor-memory to fail while the module continues to execute reliably without configuration. Assuming, however, that reconfiguration is acceptable and necessary, a simple approach to estimate the number of processors sufficient for the task is:

Let PT be the processing time to carry out one iteration of the task,
T be the iteration period of the task,
MR be the memory requirement of the task,

then, the total processor requirement is

$$P = \sum_{\text{all tasks}} \left(\frac{PT}{T}\right) \quad ,$$

the total memory requirement is

$$M = \sum_{\text{all tasks}} MR \quad .$$

To illustrate application of possible allocation techniques, two approaches are taken. Instead of focusing initially on the flight phases, it was decided to attempt to schedule all tasks across all processors. These processors are assumed to have 0.5 MIPS (millions of instructions per second) capacity and have a 20-kiloword memory.

First, all tasks were assigned triple replication. Then the number of processors was calculated:

$$\text{Number of Processors} = \sum_{j=1}^{N} \text{MIPS} \times 3 \quad \times \frac{1.2}{0.5} \ + 1 = 4.8 \rightarrow 5 \quad .$$

In this example, N = the number of tasks, MIPS_j is the millions of instructions/second required for task $_j$, 3 is for triple replication, the 0.5 factor is the machine MIPS, and the 1.2 factor provides a safety margin. When the resulting number is rounded up to achieve an integer number of processors, the appropriate number is found to be five. Thus, when these tasks are allocated over five processors, assigning tasks of the highest criticality class first, and allocating solely on the basis of processor resource utilization (MIPS) and not on memory, the allocation found in Table V-4 is obtained. Of particular note is the degree of MIPS balancing achieved. The totals are consistent to within approximately ±0.2%.*

The following steps define a more refined method for distributing replicated tasks over a number of processor-memory units while assuring that some balance of loads is achieved.

- Define the flight phase and tasks to be resident in the memory units.
- Tabulate for each task: (1) the task MIPS and the fractional processor utilization for the assumed processor type, and (2) the task memory requirement in thousands of words, and the fraction of total memory required (for the size of memory unit to be used).
- Determine the required replication per task based on the criticality class and whether passive allocation will be required.
- Accumulate the sum of the MIPS required for all tasks of the worst-case flight mode to determine either: (1) the total number of a prespecified processor type that will be required; or (2) the processor speed required to provide one complete flight-mode processing capability per processor (including a reasonable safety margin). From an overall reliability viewpoint, it is desirable to plan for at least five processors (as discussed in Chapter VII).

*However, the balance may not always be this close.

Table V-4

ALLOCATION OF ALL TASKS TRIPLY REPLICATED ACROSS FIVE PROCESSORS

Allocation Sequence:

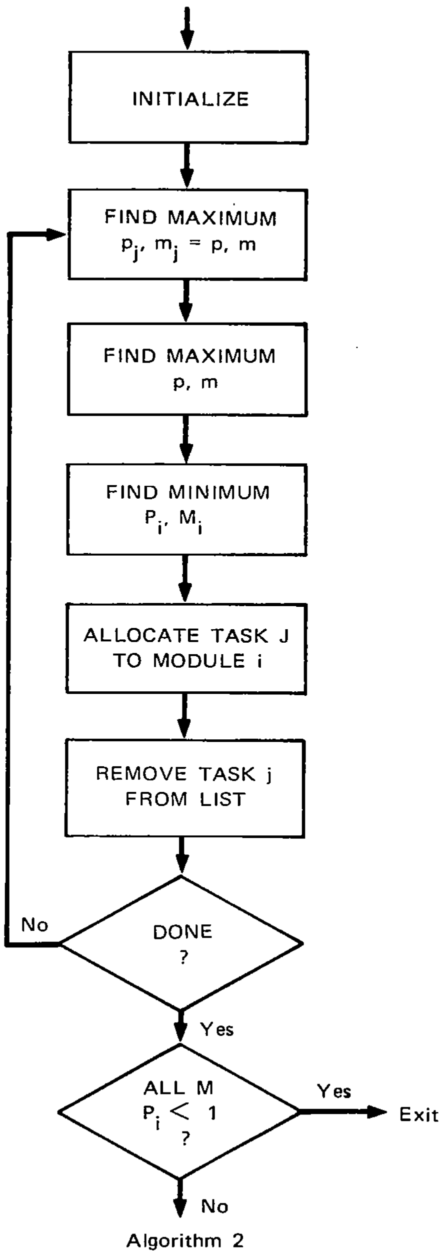
- Take tasks with criticality class 1-2 and distribute them by maximum MIPS; then
- Take tasks with criticality class 3-5 and distribute them by maximum MIPS.

CC	MIPS	1	2	3	4	5
1	0.119	D5	D5	D5		
1	0.077			A8	A8	A8
1	0.069		A2		A2	A2
1	0.055	A4			A4	A4
2	0.034	B2	B2	B2		
1	0.023	A1			A1	A1
1	0.006		C2		C2	C2
1	0.002		D3,4		D3,4	D3,4
4	0.032	B9	B9	B9		
4	0.028			B7	B7	B7
4	0.021		C1		C1	C1
4	0.019	B10			B10	B10
3	0.014	A3	A3	A3		
4	0.014	D2	D2		D2	
4	0.009	B8		B8		B8
4	0.004		B3	B3		B3
4	0.004		B4		B4	B4
5	0.002			D1	D1	D1
4	0.001	A7	A7	A7		
4	0.001	B6			B6	B6
4	0.001		C3	C3		C3
4	0.001	B5	B5		B5	
Total		0.322	0.322	0.321	0.322	0.321

- Accumulate the sum of the memory requirements for all of the allocated tasks of the worst-case flight mode (including a reasonable safety margin) for each of the processors. In addition, memory must be provided for the passive allocation of those critical tasks for which, during reconfiguration, no missed iterations can be allowed.
- Begin allocation by selecting the unallocated task requiring the highest fraction of either processor MIPS or memory storage, whichever is greater. In the event of a tie, select the one with the highest combined processor-memory total. For the given flight phase, passive allocation should be based on the memory requirement only. That is, the required memory storage is considered for passive allocation, along with the active modules. However, the execution of time for these tasks is treated as zero for processor resource utlization.
- Using the determined allocation criteria (either memory or processor), assign each selected task in turn to the memory or processor, as appropriate, with the most available unassigned capacity. Assign tasks to the indicated replication level. The only constraint is that a given task may be assigned at most once to a given processor-memory unit.
- Continue assigning tasks of lower load requirement until all tasks have been assigned. Tasks that are totally replicated, such as the Local Executive, can be assigned at any point during the procedure. For the examples given here, these tasks are assigned last.
- Check accumulated capacities on all units to verify that none have been exceeded. Of any have been exceeded or if either resource is badly misbalanced, a reallocation should be made to achieve better balance. However, for a given task, reallocation can take place only to units that have not already been allocated that task.

A flowchart representing these basic steps is given in Figure V-1.

Next, applying this algorithm to the flight phase requiring the most resources, we encounter more interesting conditions. An examination of the flight phases in Table V-1 leads to the Landing Phase as the most demanding of the phases. The critical data for allocation are shown in Table V-5. The allocation based on these data is shown in Table V-6. No preference was made for criticality class. Triple replication was assumed. Also, the five processors used in the previous example were used here. This is to allow for passive allocation as well as to allow modules from other phases to be present to facilitate rapid phase change.



p_j = Processing Requirement for Task j
m_j = Memory Requirement for Task j
P_i = Processing Load Already Allocated to Module i
M_i = Memory Load Already Allocated to Module i

FIGURE V-1 ALLOCATION ALGORITHM

Table V-5

TABLE OF AUTOMATED FLIGHT PHASE TASKS AND THE CHARACTERISTICS USED TO DISTRIBUTE THEM OVER PROCESSOR-MEMORIES

Task	MIPS	Fraction of 0.5 MIPS Processor	Memory (K)	Fraction of 20K Memory
A3	0.012	0.024	0.06	0.003
A4	0.055	0.110	1.02	0.051
A8	0.077	0.154	1.31	0.065
B3	0.004	0.008	0.30	0.015
B8	0.009	0.018	0.43	0.021
B9	0.032	0.064	6.25	0.312
C1	0.021	0.042	1.20	0.060
C2	0.006	0.012	0.61	0.030
C3	0.001	0.002	0.56	0.028
D1	0.002	0.004	1.30	0.065
D2	0.014	0.028	1.90	0.095
D3	0.001	0.002	1.00	0.050
D4	0.001	0.002	1.00	0.050
D5	0.119	0.238	1.50	0.075
GE	0.001	0.002	1.10	0.055
LE(T)*	0.034	0.068	0.32	0.016

*T = Replicated totally

Table V-6

ALLOCATION EXAMPLES--DISTRIBUTED ASSIGNMENT OF
AUTOLAND PHASE TASKS OVER FIVE PROCESSOR-MEMORY UNITS

		Accumulated Task MIPS per Processor					Accumulated Task Memory (K) per Memory Unit				
Task	M/P*	1	2	3	4	5	1	2	3	4	5
B9	M	0.032	0.032	0.032			6.25	6.25	6.25		
D5	P			0.151	0.119	0.119			7.75	1.50	1.50
A8	P	0.109	0.109		0.196		7.56	7.56		2.81	
A4	P	0.164	0.164			0.174	8.58	8.58			2.52
D2	M			0.165	0.210	0.188			9.65	4.71	4.42
D1	M		0.166		0.212	0.190		9.88		6.01	5.72
C1	M	0.185			0.233	0.211	9.78			7.21	6.92
GE	M			0.166	0.234	0.212			10.75	8.31	8.02
D3	M	0.186			0.235	0.213	10.78			9.31	9.02
D4	M		0.167		0.236	0.214		10.88		10.31	10.02
C2	M			0.172	0.242	0.220			11.36	10.92	10.63
C3	M	0.187	0.168			0.221	11.34	11.44			11.19
A3	P	0.199	0.180	0.184			11.40	11.50	11.42		
B8	M	0.208			0.251	0.230	11.83			11.35	11.62
B3	M		0.184	0.188	0.255			11.80	11.72	11.65	
LE	M	0.209	0.185	0.189	0.256	0.231	12.15	12.12	12.04	11.97	11.94

*M - Allocation based on fraction of memory requiring largest capacity
P - Allocation based on fraction of processor requiring largest capacity

Notes: 1. Autoland requires the greatest processing.
2. Memory assumed -- 20 kilowords
3. Processor assumed -- 0.5 MIPS.

In this case, the accumulated MIPS and memory are reported in each column. The results indicate that the distribution is not well equalized for the processor (about ±16% deviation) while the memory distribution is closely balanced (to about ±1%).

Thus, viable allocation schemes have been demonstrated that are easily implemented and satisfy all stated allocation objectives.

E. Schedule Derivation

Now that some flexible allocation techniques have been described, the problem of schedule derivation can be addressed. Properties that are basic to scheduling of tasks in SIFT include:

- Replicated tasks executing on different processor units need not be in lock-step synchronization, but are only loosely synchronized.
- Tasks may be preempted, but such a procedure can make program proving more difficult.
- Fixed sets of tasks represent flight phases and are executed with a given periodicity.
- The only mandatory description of a given schedule execution is one of flight change or reconfiguration.

In view of these conditions, several assumptions can be made that somewhat simplify the approach to scheduling.

- Tasks are assumed to be assigned to schedules as single units that cannot be preempted. The only exceptions occur when tasks execute for a period of time that equals or exceeds the shortest task period (equal to the period of the Local Executive). These tasks do require preemption and will be considered more closely.
- Tasks are executed according to a fixed task sequence in which each task is allocated one or several time blocks. Except for the clock routine and global executive flight failure or phase change processing, no task is interrupt-driven or initiated outside of the fixed sequence.
- The schedule is based on the maximum execution time of each of the member task modules (with provision for a reasonable safety margin).
- The schedule maintains a totally reproducible ordering of tasks that assures all tasks of the requisite periodicity.

Furthermore, to accomplish the objectives laid out at the beginning of this section, a simple, compact method of representing the schedule must be available.

F. Schedule Representation and Notation

A convenient notation is needed that allows representation of a task schedule in a concise, easily derived, and readily interpreted form. One such approach is the adoption of a notation resembling that of regular expressions. The notation and interpretation of this formalism has been modified and expanded to accommodate the SIFT scheduling requirements.

The following notational conventions have thus far been adopted:

Symbol — Interpretation

$(\ )^n$ or $(\ )^*$

The parentheses enclose a sequence of tasks or sets thereof,[†] separated by commas, that are to be executed in sequence. The "n" superscript indicates that the event sequence defined within the expression is to be repeated n times and then terminated. The asterisk superscript means that the expression is to be repeated from left to right until externally terminated. Note that the asterisk may not be used more than once in a given expression and if used, must qualify the outermost parentheses. Each task is allocated a time slot equal to the maximum time required for normal execution (extended to provide a safety margin).

$[\ \ ]^i$

The square brackets enclose tasks, or expressions containing sets of tasks, separated by commas. Only one of these elements is to be selected for scheduler processing each time this expression is encountered. Items are selected sequentially from left to right in turn as the expression is encountered, such that for n-items, one task for each of these n-items will have been executed after n-iterations through this expression. Again, cycling wraps around to the first

[†] Clearly the Task Dispatcher in the Local Executive must run following each task execution to refer to the schedule in effect and to determine the next task to be executed. Likewise, the clock routine must be run. We assume that these small, fixed-length blocks of instruction can be treated as though they were part of each task rather than being explicitly included as separate tasks. This approach sacrifices nothing technically but greatly simplifies the representation.

element. The superscript indicates that this expression is to be repeated i-times at that point in the schedule.

Δ integer — The Δ indicates an idle or unallocated processor time block where the integer is the available time in microseconds. This may be used to execute irregular tasks or tasks that are so infrequent or require so much processor time that they must be preempted a number of times to allow more-frequent tasks to run.

$[\quad]_j$ — Subscripts indicate the periodicity in milliseconds of a given item in a given scheduling expression. These are used for convenience and need not be present.

Examples of the application of such a notation are shown in Figure V-2. Also shown in this figure are alternative link-connected diagrams and the long forms of these expressions.

Basically, the convenience of the modified regular-expression formalism is that it greatly facilitates the derivation of a schedule, as well as its storage and execution, while the graphical representation aids in the comprehension of a completed schedule.

G. Sample Schedule Derivation

Using the techniques discussed in the previous section, a sample schedule is now developed using the tasks in the Landing Flight Phase. This phase received attention since it demands more system resources than do other operational phases. The data required for scheduling are the task periodicity and the task module execution time. The step-by-step development of the sample task schedule is shown in Figure V-3, and the corresponding alternative representation is given in Figure V-4.

The procedure used for deriving such a schedule is:

Let N = the number of tasks,
For the j^{th} task, $(j = 1, \ldots N)$, let
p_j = the periodicity (in milliseconds)
e_j = the execution time (in microseconds)

The flowchart given in Figure V-5 describes the procedure for deriving the schedule expression.

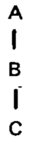
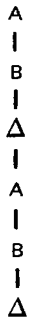
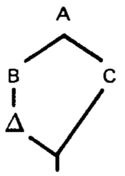
SCHEDULE EXPRESSION	ALTERNATE REPRESENTATION	INTERPRETATION
$(A,B,C)^*$		A,B,C,A,B,C, A . . .
$(A,B,\Delta)^2$		A,B, Δ,A,B, Δ
$(A, [(B, \Delta),C])^*$		A,B, Δ,A,C,A,B, Δ, . . .

FIGURE V-2 SCHEDULE REPRESENTATION EXAMPLES

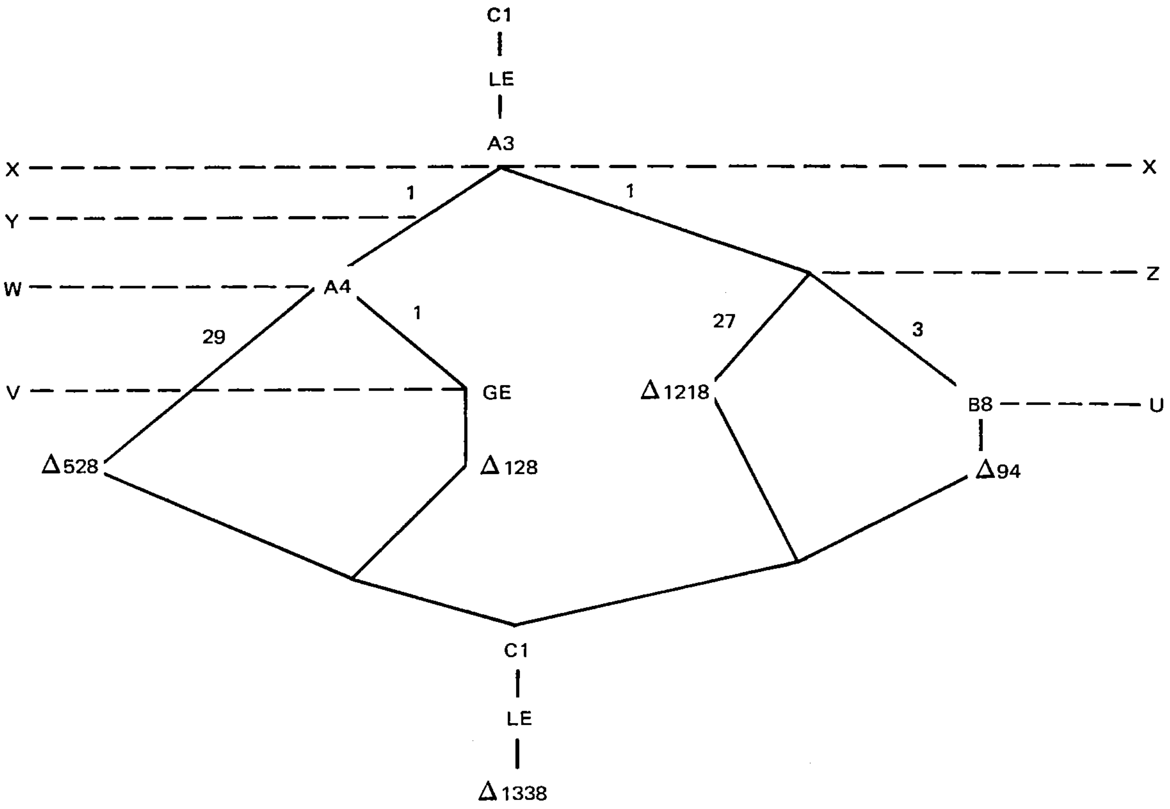
In representing the schedules, we can use a convenient shorthand notation in which we define clusters of tasks according to the scheme illustrated in Figure V-4. The total storage required to represent a schedule can be greatly reduced, as can be seen by comparing the cluster representation of Figure V-4 with the longhand expression illustrated in Figure V-3.

TASKS[†]	ASSIGNED PERIODICITY (MILLISEC)	EXECUTION TIME[‡] (μSEC)	EXPRESSION
C1	1.5	62	$(C1,\Delta 1438)^{*}_{1.5}$
LE	1.5	100	$(C1,LE,\Delta 1338)^{*}_{1.5}$
A3	3.0	116	$(C1,LE,A3,\Delta 1218,C1,LE,\Delta 1338)^{*}_{3}$
A4	6.0	688	$(C1,LE,A3,[(A4,\Delta 528),\Delta 1218]_{6},C1,LE,\Delta 1338)^{*}_{3}$
B8	60.0	1124	$(C1,LE,A3,[[(A4,\Delta 528),\Delta 1218]^{9},[(A4,\Delta 528),$ $(B8,\Delta 94)]_{60}],C1,LE,\Delta 1338)^{*}_{3}$
GE	180.0	400	$(C1,LE,A3,[[[(A4,\Delta 528),\Delta 1218]^{9},[(A4,\Delta 528),$ $(B8,\Delta 94)]]^{2},[[(A4,\Delta 528),\Delta 1218]^{9},[(A4,GE,\Delta 128),$ $(B8,\Delta 94)]_{180}]_{60}]_{6},C1,LE,\Delta 1338)^{*}_{3}$ WHICH CAN BE SIMPLIFIED TO: $(C1,LE,A3,[(A4,[\Delta 528^{29},(GE,\Delta 128)])_{180},[\Delta 1218^{27},$ $(B8,\Delta 94)^{3}]_{180}]_{6},C1,LE,\Delta 1338)^{*}_{3}$
A8	30.0	5134	(THIS SET OF TASKS WILL REQUIRE PREEMPTION, AND THE TASKS ARE LOGGED IN A SEPARATE EXPRESSION IN ORDER OF INCREASING PERIOD SIZE.)
B9	120.0	8000	
B3	180.0	1600	$(A8,B9,B3,\Delta)$

[†] These tasks are allocated in the order of decreasing iteration rates, except for those tasks for which the execution time equaled or exceeded the period for the most frequently executed tasks. These tasks require preemption and are assigned to an expression separate from that for the main schedule.

[‡] Assumed processing rate of 0.5 MIPS

FIGURE V-3 SAMPLE SCHEDULE DERIVATION FOR THE LANDING FLIGHT PHASE



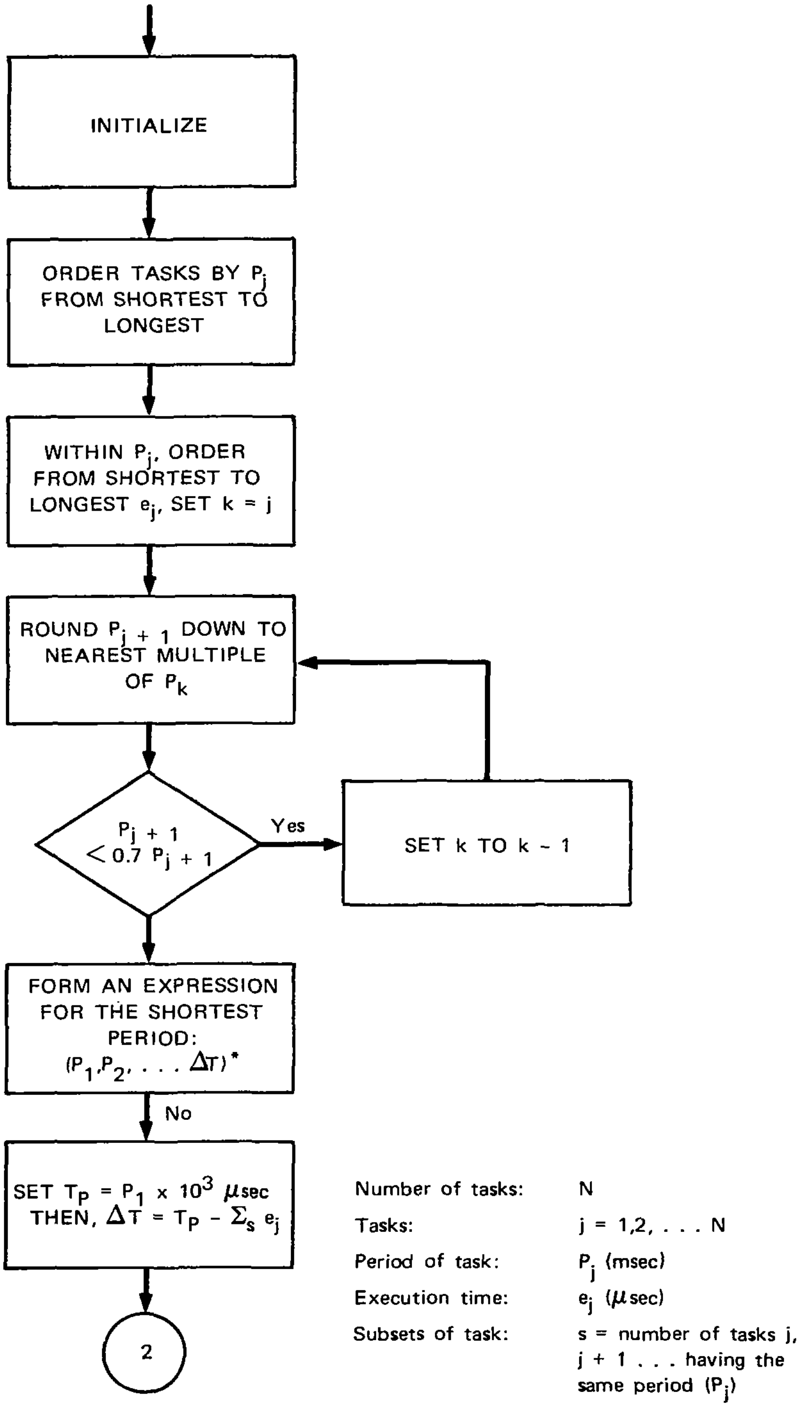
NOTE: Clusters may be defined in increasingly greater detail:

$X = [Y,Z]$
$Y = (A4,W)$
$W = [\Delta 528^{29},V]$

$V = (GE, \Delta 128)$
$Z = [\Delta 1218^{27},U]$
$U = (B8, \Delta 94)^{3}$

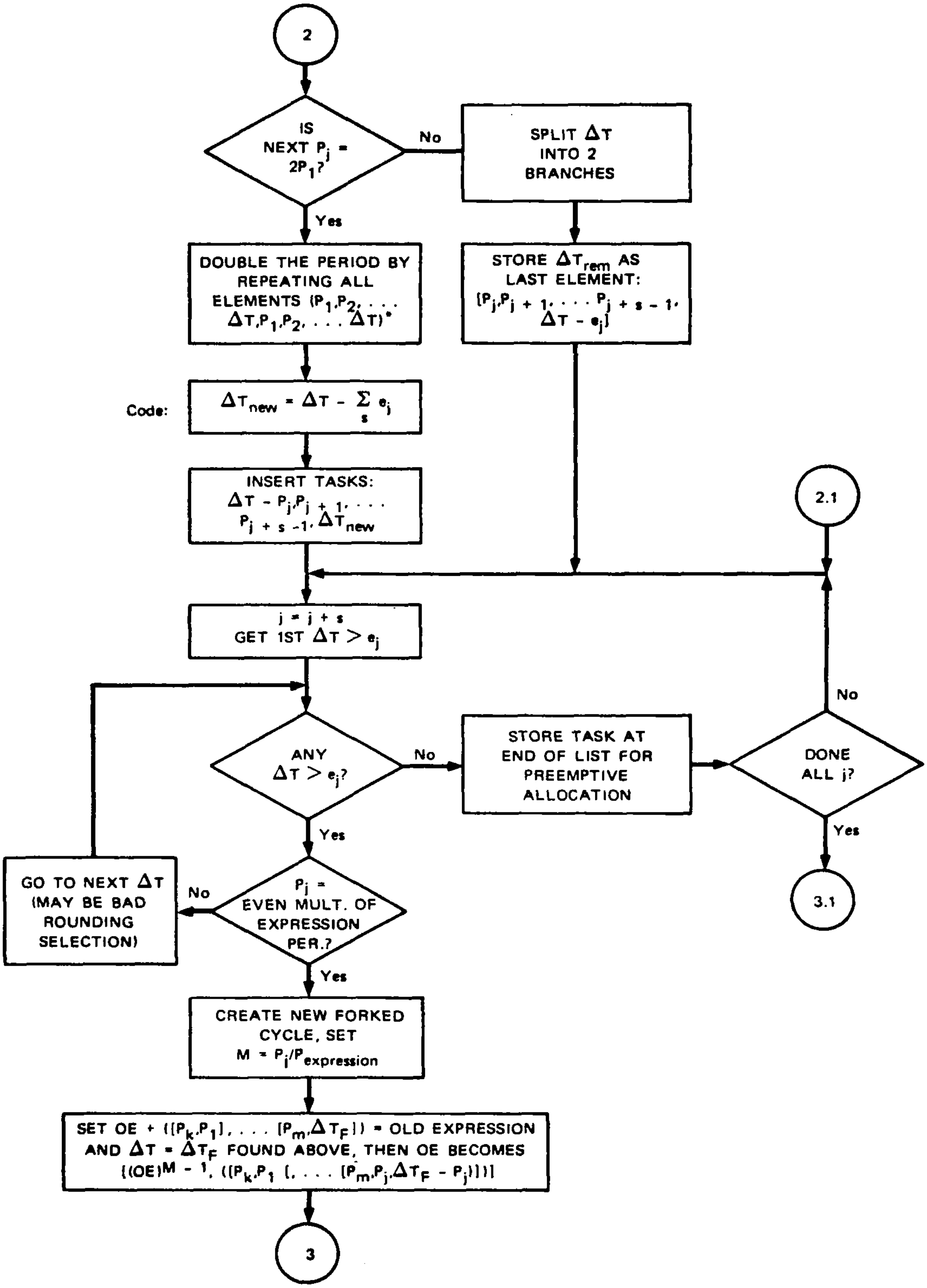
FIGURE V-4 ALTERNATE SCHEDULE REPRESENTATION

There are two procedures that need further discussion. The first involves the reason behind choosing successive multiplicity factors rather than factors based on the "lowest common denominator." It is indeed true that the latter approach would lead to an acceptable schedule. However, the intention has been to derive a schedule description that was concise and simple. With this in mind, examining the initial stages of schedule development, one writes a simple expression with a period as big as the shortest of the tasks. One then assigns portions of the remaining time block to successive tasks. In the cited example


INITIALIZE
ORDER TASKS BY P_j FROM SHORTEST TO LONGEST
WITHIN P_j, ORDER FROM SHORTEST TO LONGEST e_j, SET $k = j$
ROUND P_{j+1} DOWN TO NEAREST MULTIPLE OF P_k
$P_j + 1 < 0.7\ P_j + 1$
Yes
SET k TO k – 1
FORM AN EXPRESSION FOR THE SHORTEST PERIOD: $(P_1, P_2, \ldots \Delta T)^*$
No
SET $T_P = P_1 \times 10^3\ \mu$sec THEN, $\Delta T = T_P - \Sigma_s\ e_j$
2

Number of tasks:	N
Tasks:	$j = 1, 2, \ldots N$
Period of task:	P_j (msec)
Execution time:	e_j (μsec)
Subsets of task:	s = number of tasks j, j + 1 . . . having the same period (P_j)

FIGURE V-5 SCHEDULE DERIVATION FLOWCHART



2
IS NEXT P_j = $2P_1$?
No
SPLIT ΔT INTO 2 BRANCHES
Yes
DOUBLE THE PERIOD BY REPEATING ALL ELEMENTS (P_1,P_2, . . . ΔT,P_1,P_2, . . . ΔT)*
STORE ΔT_{rem} AS LAST ELEMENT: [P_j,P_j + 1, . . . P_j + s - 1, ΔT - e_j]
Code:
$\Delta T_{new} = \Delta T - \sum_s e_j$
INSERT TASKS: ΔT - P_j,P_j + 1, . . . P_j + s -1, ΔT_{new}
2.1
j = j + s
GET 1ST $\Delta T > e_j$
No
ANY $\Delta T > e_j$?
No
STORE TASK AT END OF LIST FOR PREEMPTIVE ALLOCATION
DONE ALL j?
Yes
Yes
3.1
GO TO NEXT ΔT (MAY BE BAD ROUNDING SELECTION)
No
P_j = EVEN MULT. OF EXPRESSION PER.?
Yes
CREATE NEW FORKED CYCLE, SET M = $P_j/P_{expression}$
SET OE + ([P_k,P_1], . . . [P_m,ΔT_F]) = OLD EXPRESSION AND ΔT = ΔT_F FOUND ABOVE, THEN OE BECOMES [(OE)$^{M-1}$, (([P_k,P_1 [, . . . [P_m,P_j,ΔT_F - P_j])))]
3

FIGURE V-5 SCHEDULE DERIVATION FLOWCHART (Continued)

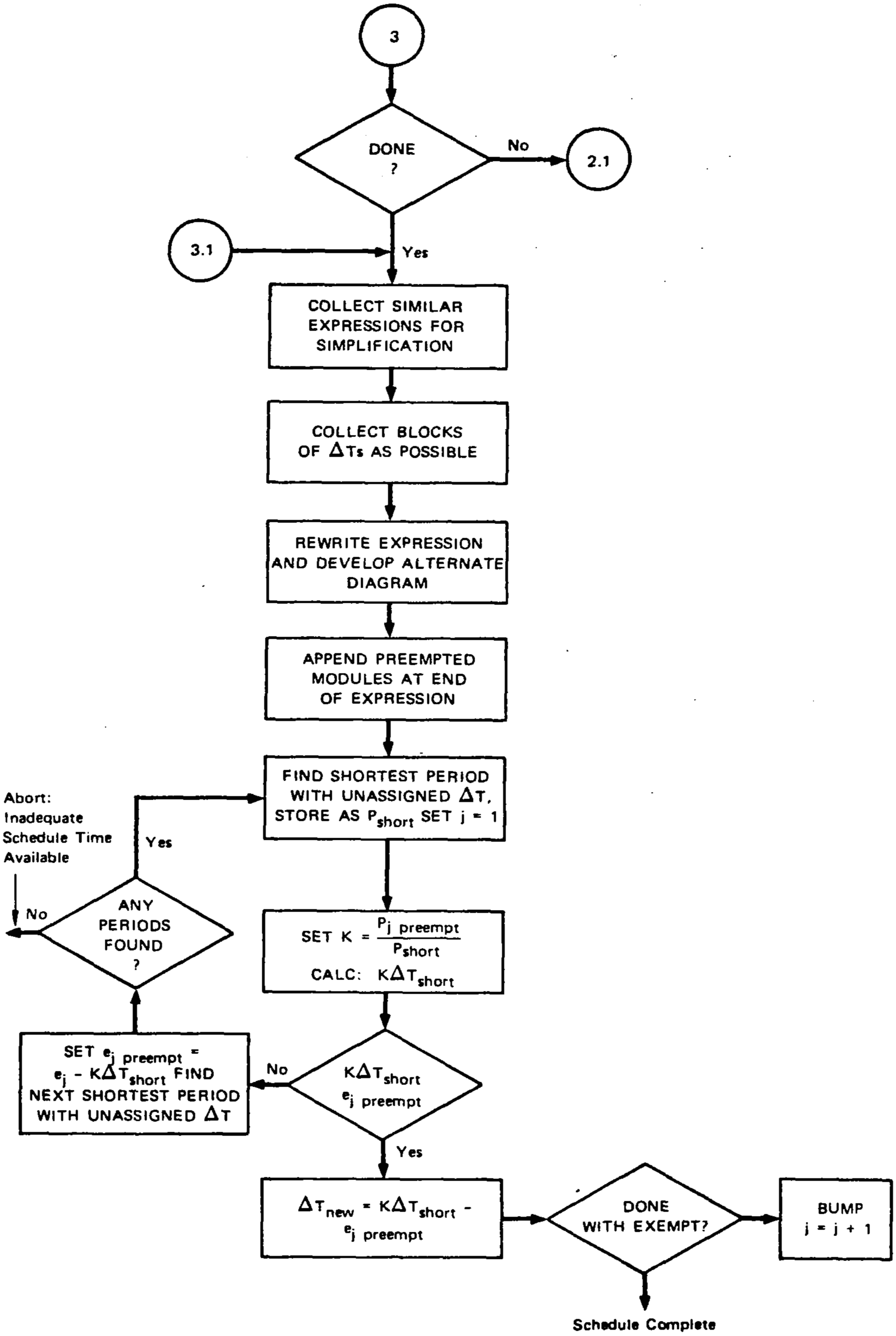

3
DONE ?
No
2.1
3.1
Yes
COLLECT SIMILAR EXPRESSIONS FOR SIMPLIFICATION
COLLECT BLOCKS OF ΔTs AS POSSIBLE
REWRITE EXPRESSION AND DEVELOP ALTERNATE DIAGRAM
APPEND PREEMPTED MODULES AT END OF EXPRESSION
FIND SHORTEST PERIOD WITH UNASSIGNED ΔT, STORE AS P_{short} SET j = 1
Abort: Inadequate Schedule Time Available
Yes
No
ANY PERIODS FOUND ?
SET K = $\frac{P_{j\ preempt}}{P_{short}}$
CALC: $K\Delta T_{short}$
SET $e_{j\ preempt}$ = $e_j - K\Delta T_{short}$ FIND NEXT SHORTEST PERIOD WITH UNASSIGNED ΔT
No
$K\Delta T_{short}$
$e_{j\ preempt}$
Yes
$\Delta T_{new} = K\Delta T_{short} - e_{j\ preempt}$
DONE WITH EXEMPT?
BUMP j = j + 1
Schedule Complete

FIGURE V-5 SCHEDULE DERIVATION FLOWCHART (Concluded)

the total expression had a periodicity of twice the smallest period. We then partition the available time blocks (the Δs) typically into two subexpressions, only one of which is executed each time the expression is processed. These subexpressions will typically have their own Δ time blocks as elements. Note that the period of these elements is a multiple of the period of its parent expression. Thus as each task is added to the expression, a decision must be made as to which Δ time block should be partitioned to accommodate the task. If it is possible to take advantage of the expression having the largest period already assigned, by using the Δ time block remaining within that expression, then the embedded expression hierarchy is greatly simplified by not proliferating many new disjoint subexpressions with new periodicities. In addition, this choice of the Δ time block leads to more efficient use of available Δ time blocks, leaving larger time blocks unfragmented and whole.

The second procedure deals with the use of remaining time blocks to satisfy the needs of task requiring preemption. It is assumed that each of these tasks is run to completion before another is started. Here large contiguous time blocks that are maintained intact and belong to the shorter periods make the preemption task analysis much more straightforward. This allows use of very regular time intervals to verify that the period needs can be satisfied for those preempted tasks in the worst case of all tasks requiring processing simultaneously. For example, in the sample schedule, if all three preemptive type tasks came due for execution at the same time, there are Δ time blocks of 1338 μsec available every 3 millisec. This means that by 30 millisec (the period of the most frequent task), there are 13,380 μsec available. Since all three tasks require a total execution time of 14,734 μsec, if we used no other Δ time blocks in the expression, A8 and B9 would have been executed, and B3 could be started when A8 came due for execution again. If we then examine the longest consecutive sequence of tasks that could occur before a Δ time block came available, it could be Cl, LE, A3, B8, Cl, LE, which consumes a time block of 1564 μsec. Then in the worst case, if this sequence occurred at the time the three preempted tasks needed execution, a lead time in scheduling their execution ahead of their required periodicity to assure no scheduling problem in this instance can be calculated. It is desired to execute A8 prior to the onset of this "no break" situation. Then it should be scheduled at:

$$\text{Max Time}_{\text{No}\ \Delta} + \text{Exec Time}_{\text{A8}} = 6.7 \text{ millisec} = 23\% \text{ of period}$$

Hence, if the period of this task is decreased by 23% for scheduling purposes, then the period all three tasks should be satisfied in this worst-case situation. A similar analysis can be carried out on the other preempted tasks. Additional consideration of criticality class and missed iterations may weaken this requirement.

H. Conclusion

In summary, then, methods for determining the required number of processor memory units has been described and allocation of tasks can be readily performed. Furthermore, a schedule derivation method has been presented that could be performed on-line. However, all normal flight schedules would be best stored in a regular expression and invoked as required. This is because the derivation algorithm would require more execution time and data access than would the retrieval of stored schedules. While critical tasks will be passively allocated, there will be instances where it may be necessary to derive a new schedule. A method has been described that could be readily implemented. If tasks will revert to pilot control, then they need only be deactivated during reconfiguration. This enhances the criticality of the display screens to the system.

As to schedule storage, the regular expression formalism can clearly be mapped into a compact storable stack of tasks with appropriate delimiters and flags. This would lead to a way to store the schedules both efficient and useful.

A simpler, more visual schedule representation has also been derived that allows for ready comprehension of the task execution as a function of time and the periodicity of isolated task sequences.

Lastly, the schedule derivation is such that preemption is required for only a subset of the tasks, namely those with exceptionally long execution times that encroach upon the time periods of the more frequent tasks, and the period of these preempted tasks is verified. This concludes a derivation of a suitable schedule representation for SIFT.

REFERENCE

1. R. S. Ratner, E. B. Shapiro, H. M. Zeidler, S. E. Wahlstrom, C. B. Clark, and J. Goldberg, "Design of a Fault-Tolerant Airborne Digital Computer," Vol. II--Computational Requirements and Technology, Final Report. NASA CR-132253, 1973.

VI HARDWARE DESIGN

A. Bus Interconnection Network

1. Introduction

The purpose of the bus interconnection network in the SIFT computer is to provide communication between each processor (main or I/O) and all memory units, except possibly the single memory unit already connected directly to that processor by a high-bandwidth link. This communication could be established with a separate connection between all processor-memory pairs. However, since only a few of the total number of possible communication paths would ever be in use at the same time, a multilevel interconnection network, similar to those employed in telephone systems, should be considered in the hope of achieving a net saving in equipment. A multilevel realization may turn out to have some desirable fault-tolerance features as well. A two-level arrangement having four to six intermediate busses was proposed in the original SIFT design concept.

In this section, some alternative designs for the interconnection network are explored. Comparisons are made between a single-level network of direct connections (no busses), a two-level network (single set of busses), and a three-level network (a cascade having two separate sets of busses). Bit-serial, byte-serial, and all-parallel data transfer modes are evaluated. The principal cost measures used for these comparisons of the several cases are:

g or G = number of equivalent NAND gates, a measure of hardware complexity.

t or T = number of terminals.

d or D = number of clock cycles of delay for a full memory access.

Lower-case letters apply to a single module or unit, and upper-case letters to the grand totals for the entire network. Other important but less quantitative criteria are:

- The degree to which the final network can be conveniently modularized into MSI or LSI semiconductor chips, either custom designed or commercially available.
- The complexity of calculations needed to generate the routing codes for the two- and three-level networks.
- Algorithms required for checking and diagnosis of the interconnection network to achieve the desired degree of fault tolerance.

Typically about half of the processors will be I/O microprocessors, rather than main processors, and their memory units will be correspondingly smaller. They may not need the full number of address and data-word bits, and communication paths will probably not be required between each microprocessor and the other microprocessor memories. However, the savings in time and equipment resulting from these simplifications are not expected to be great and have therefore been neglected at the present stage of the design.

The main results of this analysis are expressed in Figures VI-7 and VI-8, which are described in detail in the following section.

2. Design Alternatives

Figure VI-1 shows the interconnection network within its immediate context in the SIFT computer. Its overall function is to provide bilateral communication paths between a set of p processors and a set of p memory units. Requests normally originate with a processor, which injects onto the forward connection the number of the memory unit (M) with which it wishes to communicate, bus routing information (B) as appropriate, and the address (A) within the memory. The return connection carries a data word (W) from memory, or else a single acknowledgment digit.

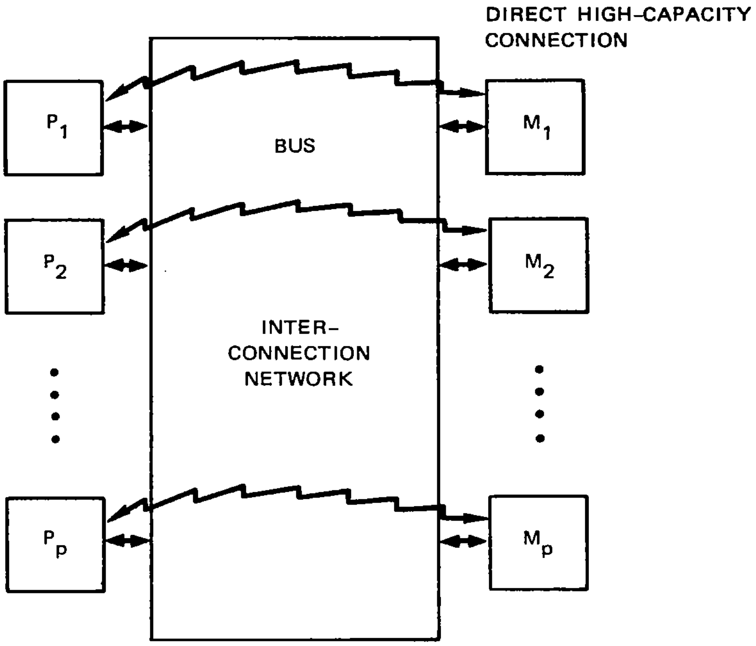


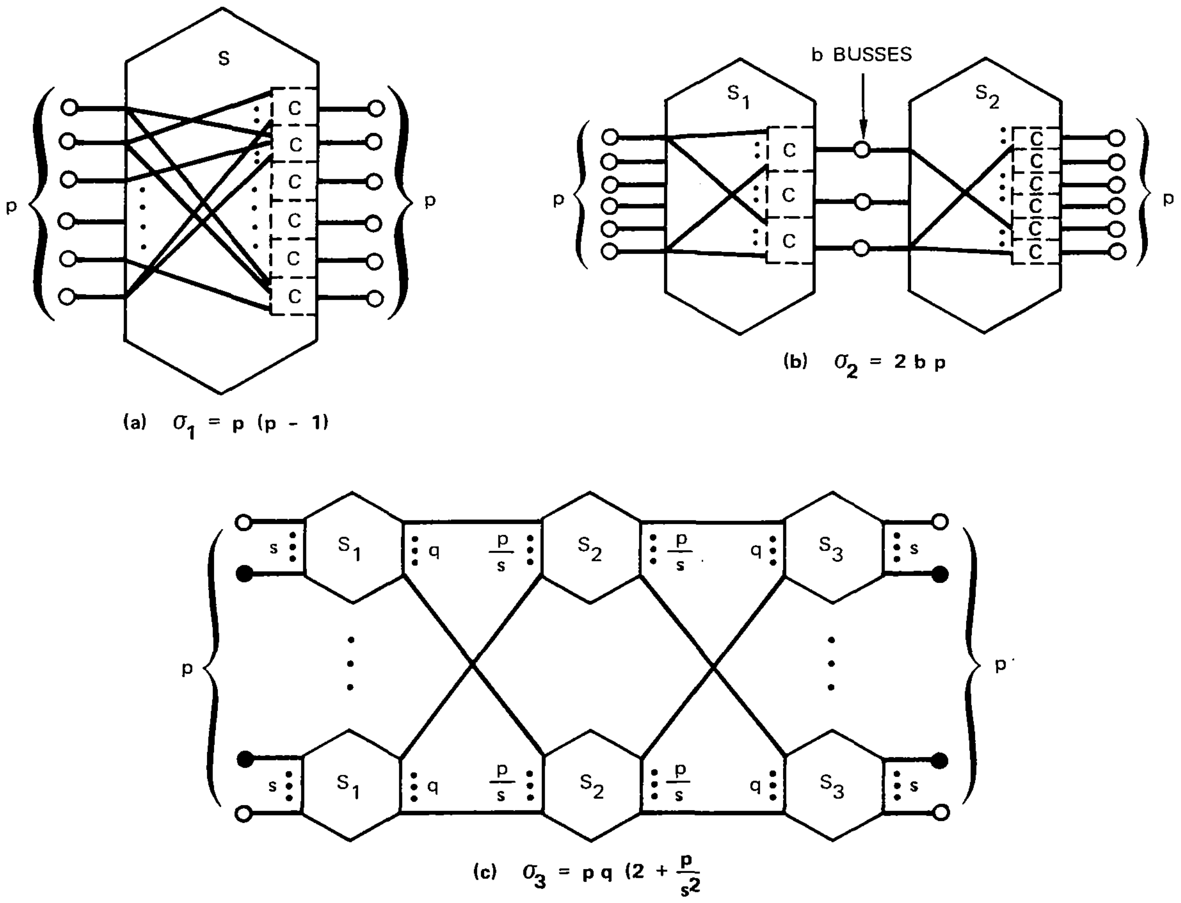
FIGURE VI-1 INTERCONNECTION NETWORK

The following list gives the principal independent parameters and their expected ranges:

Parameter	Minimum	Typical	Maximum
p = number of processors = number of memories	9	12	18
b = number of simultaneous paths needed (= number of busses for 2-level case)	3	4	6
n_w = number of bits in data word	16	24	32
n_a = number of bits in memory address	16	20	24

The number of bits needed for memory selection and for bus selection can be derived from p and b, respectively, assuming a convenient coding.

Three possible interconnection schemes are shown in Figure VI-2; switching unit S in this figure is assumed to be capable of making one-to-one connections between its left-hand terminals and its right-hand terminals in all (or almost all) ways--in the fashion of a crossbar, for example. The quantity σ_ℓ designates the total number of simple switches that would be required if each path through every switching unit were provided by a separate switch. (The subscript ℓ designates the number of levels in the network.) In Figure VI-2(a), for example, we have $\sigma_1 = p(p - 1)$. (Recall that a network connection from processor k to memory k is not needed.) The two-level arrangement in Figure VI-2(b) reflects the assumption that no more than b connections are ever needed at the same time. The three-level network of Figure VI-2(c), to be described in detail later, also has the flexibility to provide as few as



(a) $\sigma_1 = p\ (p - 1)$

(b) $\sigma_2 = 2\ b\ p$

(c) $\sigma_3 = p\,q\ (2 + \frac{p}{s^2}$

FIGURE VI-2 POSSIBLE INTERCONNECTION SCHEMES

$b < p$ paths, at a saving in circuit complexity. Other schemes having more levels are also possible, but these three alternatives will be seen to be the most competitive for present purposes.

The set-up of the paths in the interconnection network must be executed in a sequence of steps. First, a processor initiates a request in the form (B, M, A), where B consists of 0, 1, or 2 bus numbers, depending on the number of levels; M is the memory number; and A is the address within that memory (job number and local address). This request is sent to the first receptor--either a memory unit [Figure VI-2(a)] or a bus [Figures VI-2(b) and VI-2(c)]. Each output controller C of each receptor continuously scans all request lines incident upon it whenever it is not busy holding a connection. When the scanning is successful, the receptor enters the busy state and closes the connections for the forward transfer of the address and next bus (if any) and for the reverse transfer of data. At the next level, the same action is repeated by the succeeding receptor. At the final level (1, 2, or 3), the address A is handled by the memory unit itself.

Actual transfer of data (and later release of established paths) occurs somewhat differently, depending upon the mode of bit communication through the network. For parallel transfer, the memory address A arrives at the memory unit coincident with the data request. The return of the data word W automatically signals completion of the operation. The request is then removed by the processor, all gates along the path are opened, and each controller is released from the busy state and resumes scanning. In the case of serial transfer, receipt of a request at the memory unit triggers the return of an acknowledgment digit to the source processor. This processor then spews forth its stream of address digits, on completion of which the memory unit returns its stream of data-word digits. Release of the path then follows as in the parallel case. Transfer to and from I/O processors takes place in an identical but possibly abbreviated manner, since communication may be needed in only one rather than both directions.

3. Parallel Transfer

It should be clear from this description that each controller in each receptor (memory unit or bus) requires two parts, a scanner and a switch. The manner in which these two parts function together is shown in block diagram form in Figure VI-3 and as a logical circuit in Figure VI-4, for the single-level case and parallel mode of transfer.

In the functioning of each scanner (Figure VI-4), a (p - 1)-stage unary counter cycles continuously as long as no request is received on one of its memory select lines M. The first such request that is encountered stops the counter, and the counter state m and busy signal are passed on to the switch.

Each switch is a simple two-way multiplexor for connecting one of the p - 1 processors to the corresponding output lines. Thus it has $(p - 1) \cdot (n_a + n_w)$ left-hand terminals, which connect to the processors, and $n_a + n_w$ right-hand terminals, which connect to the memory proper. The gate realization is straightforward. The complex of lines at the left side of Figure VI-4 corresponds to the nearly complete crossing of connections within S in Figure VI-2(a). Note that the total of p(p - 1) M-lines ties directly to the scanners, p - 1 of them to each scanner. The other lines are paralleled to or from the controllers.

a. One- and Two-Level Networks

The cost measures for a single controller may now be written down directly for the single-level case. For the scanner we have

$$g_{sc} = 12(p - 1) + (p - 1) + 2 + 6 = 13p - 5 \text{ equivalent gates}$$

$$t_{sc} = 2p \text{ terminals (excluding clocks and power)}$$

$$d_{scmin} = 1, \; d_{scmax} = p - 1 \text{ clocks;}$$

and for the switch, letting $n = n_a + n_w$,

$$g_{sw} = np \text{ gates}$$

$$t_{sw} = (n + 1)p \text{ terminals}$$

$$d_{sw} = 0 \quad .$$

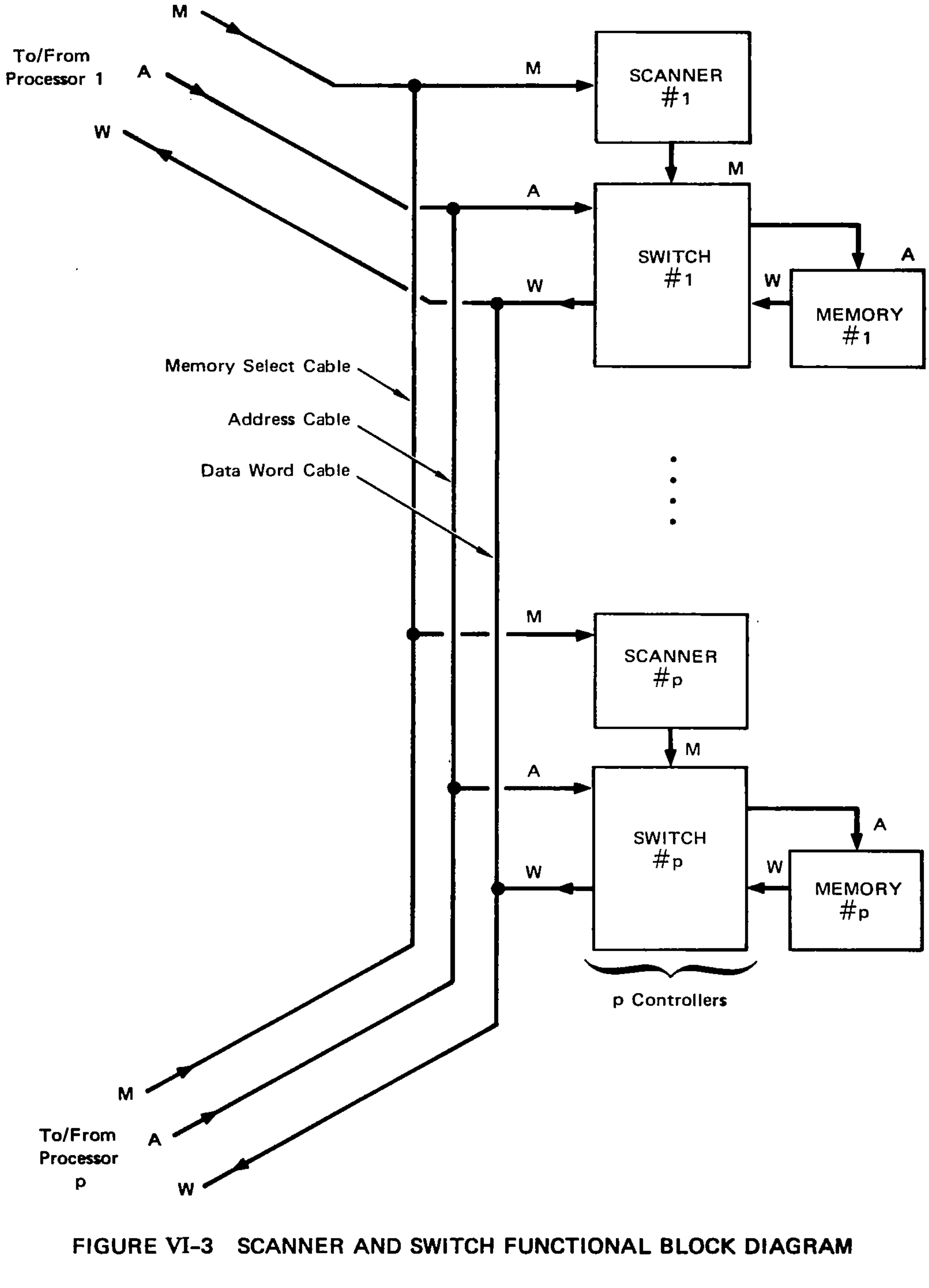


FIGURE VI-3 SCANNER AND SWITCH FUNCTIONAL BLOCK DIAGRAM

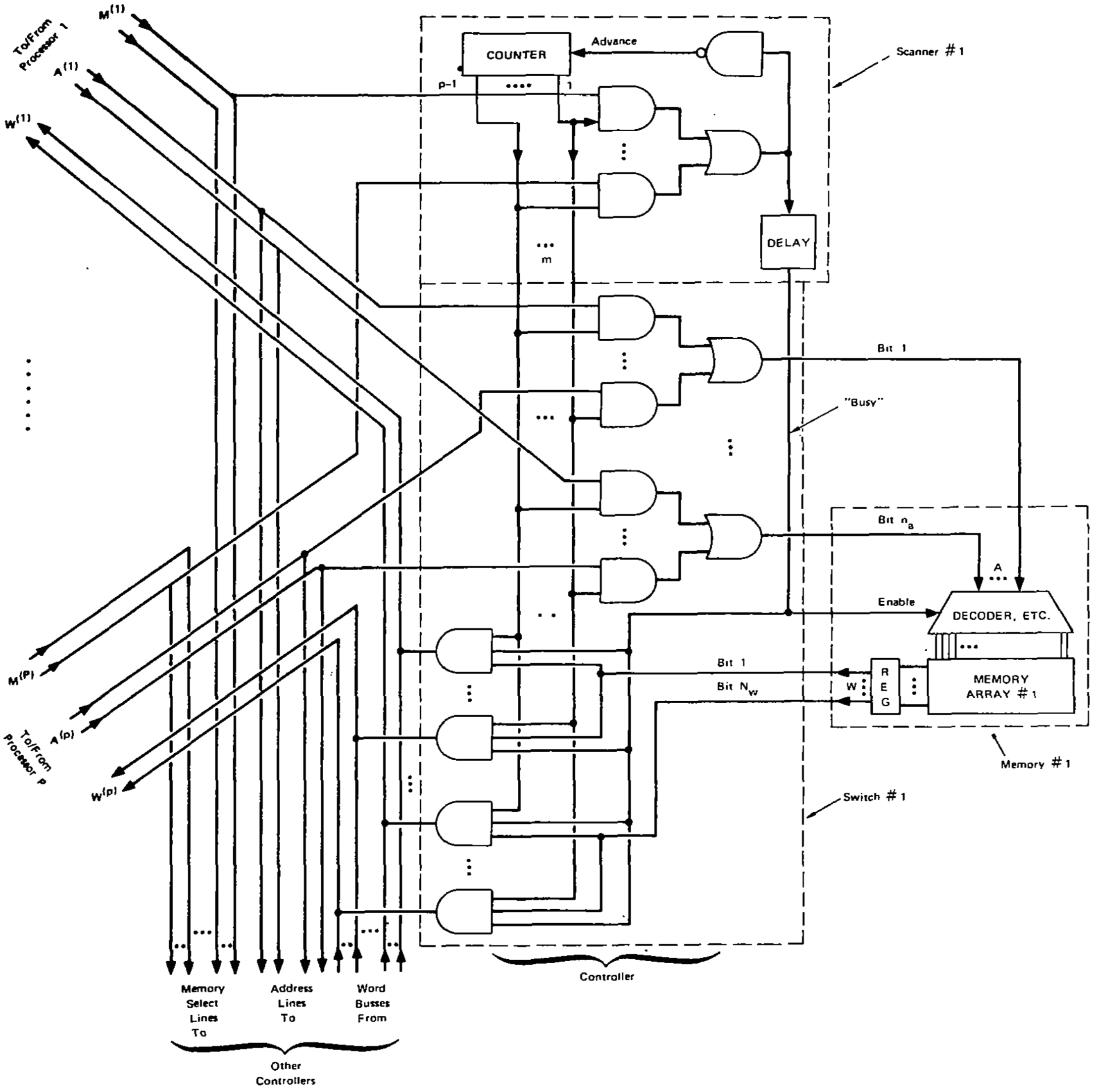


FIGURE VI-4 SCANNER AND SWITCH LOGIC CIRCUITRY

We have assumed here a cost of 12 gates per stage for the counter, and 6 equivalent gates for the single-digit delay. (These costs are all approximate, but the final results do not depend critically upon them.) Wired-OR output gating is assumed for the data-word lines returning to the processors.

In the *two-level* case, shown in Figure VI-5, the bus and memory receptors are the same as in the one-level case except for the different numbers of inputs and outputs. In particular, the number of scanned positions increases from p - 1 to p in the first level and reduces from p - 1 to be in the second level. Thus,

Level 1

$$\left.\begin{aligned} g_{sc} &= 13p + 8 \\ t_{sc} &= 2p + 2 \\ d_{sc} &= 1 \text{ to } p \end{aligned}\right\} \qquad \text{(b of these)}$$

Level 2

$$\left.\begin{aligned} g_{sc} &= 13b + 8 \\ t_{sc} &= 2b + 2 \\ d_{sc} &= 1 \text{ to } b \end{aligned}\right\} \qquad \text{(p og these)}$$

The number of bit lines to be switched increases from n to n + p in the first level, in order to include the routing digits B, but remains at the value n in the second level. Thus

Level 1

$$\left.\begin{aligned} g_{sw} &= (n + p)(p + 1) \\ t_{sw} &= (n + p + 1)(p + 1) \\ d_{sw} &= 0 \end{aligned}\right\} \qquad \text{(b of these)}.$$

Level 2

$$\left.\begin{aligned} g_{sw} &= n(b + 1) \\ t_{sw} &= (n + 1)(b + 1) \\ d_{sw} &= 0 \end{aligned}\right\} \qquad \text{(p of these)}$$

Note that the circuit arrangements presented in Figures VI-3 and VI-5 constitute a simplification over that described previously in the Project 1406 Final Report. Specifically, memory-unit selection is done here with a unary rather than a binary code. This choice presumes that each processor requests each individual memory unit with a separate line. The previous DATA REQUEST line can then be combined with this line. If the memory units were selected with a more compact binary code having, say, n_p digits, where $n_p < p$, then the number of bit lines to be switched in the first level would be reduced slightly in the two-level case $[g_{sw} = (n + n_p)p]$, but a more costly scanner must be used $[g_{sc} \approx 13p + 15 + n_p(p + 11)]$. We conclude that the unary code leads to a more economical design.

The grand totals may now be calculated. Summing over all p memory-unit controllers, the single-level case yields

$$G_1 = p(13p - 5) + (n)p^2 = p^2(n + 13) - 5p$$

$$T_1 = 2p^2 + (n + 1)p^2 = p^2(n + 3)$$

$$D_{1min} = 3,\ D_{1max} = p + 1;$$

while the two-level case yields

$$\begin{aligned} G_2 &= b(13p + 8) + (n + p)b(p + 1) + p(13b + 8) + n(b + 1)p \\ &= (n + 8)(2bp + p + b) + bp(p + 11) \end{aligned}$$

$$\begin{aligned} T_2 &= b(2p + 2) + b(p + 1)(n + p + 1) + p(2b + 2) + \\ &\quad p(n + 1)(b + 1) \\ &= (n + 3)(2bp + p + b) + bp(p + 1) \end{aligned}$$

$$D_{2min} = 4,\ D_{2max} = p + b + 2 \quad .$$

Note that in both cases two clocks have been added to the total transfer time for acceptance of the memory address and return of the data word. G and T designate the total number of gates and terminals, respectively, for all controllers, assuming one scanner module and one switch module in each controller.

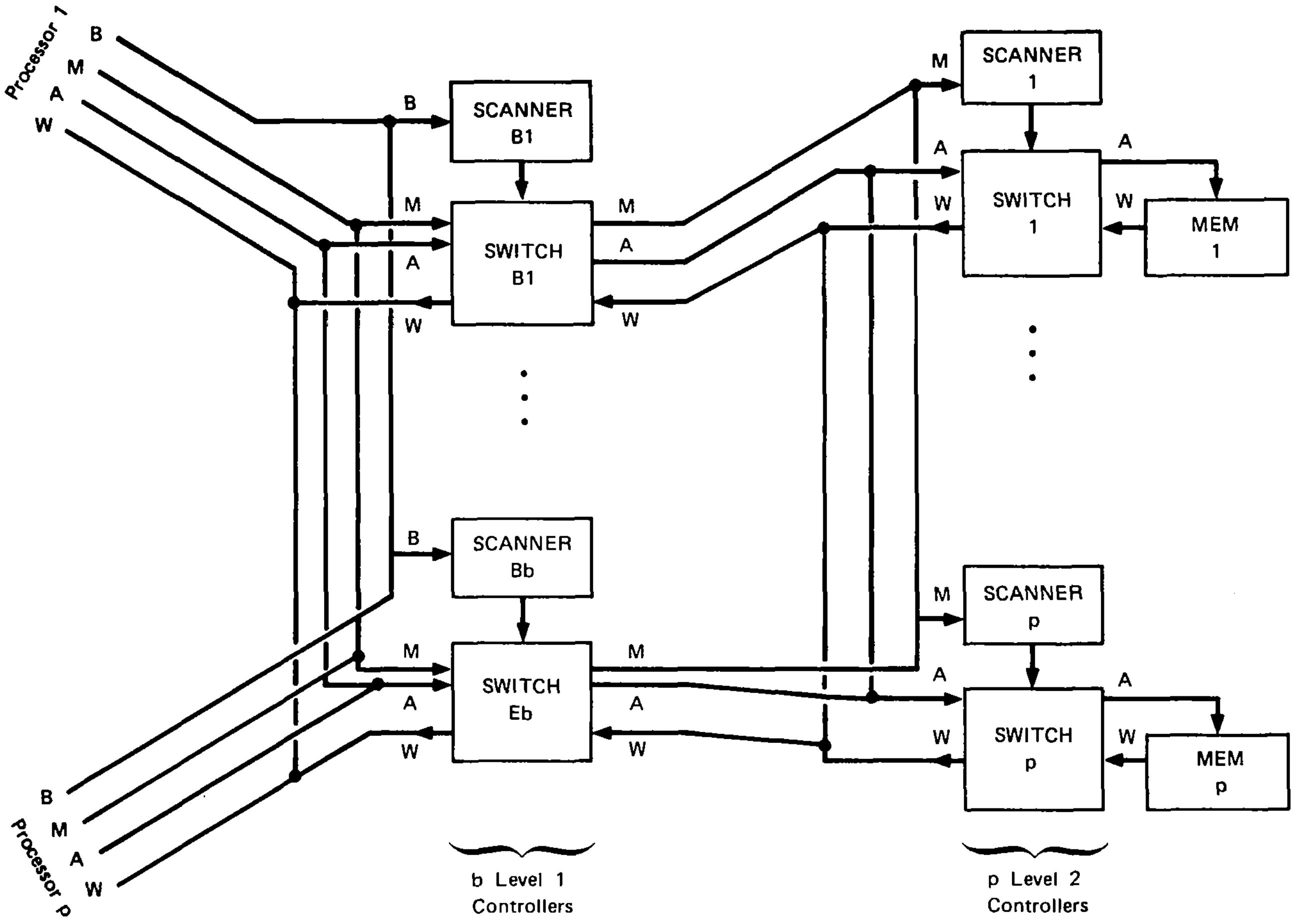
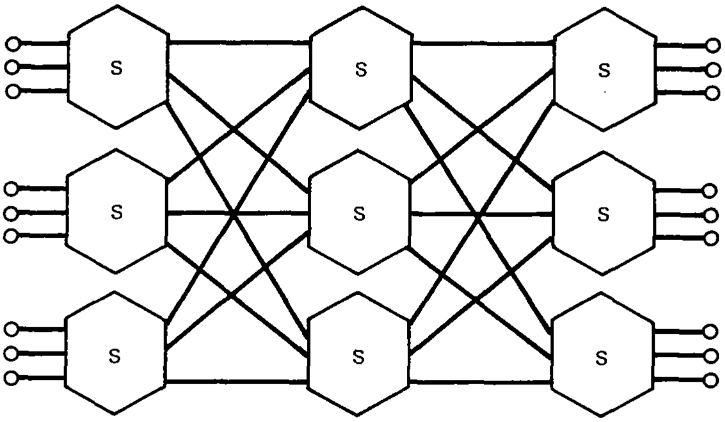


FIGURE VI-5 TWO-LEVEL NETWORK BLOCK DIAGRAM

b. The Three-Level Network*

The general form of a three-level interconnection network was shown in Figure VI-2(c) [Ref. 1]. In the first level, the set of p network inputs is handled s at a time, by qp/s controllers in p/s groups of q each. Each has s inputs. The second level has qp/s controllers, now in q groups of p/s each. Each has p/s inputs. The third level, antisymmetrical to the first, consists of p controllers in p/s groups of s each. Each has q inputs. An example for p = 9, s = q = 3, is given in Figure VI-6.



p = 9, s = q = 3

FIGURE VI-6 EXAMPLE OF A THREE-LEVEL NETWORK

The parameters q and s should be chosen to optimize the design; we use here the cost parameters G_3, T_3, and D_3. The values of q and s measure the richness of interconnectability, through the number b of simultaneous parallel paths provided, just as in the two-level network. First, s must be selected in the range $2 \leq s \leq p/b$ for the network of

*The calculations reported earlier in Technical Memo No. 5 contained an algebraic error, which affected the numerical results and the comparison of the three-level case with the others. This error has now been corrected and the conclusions modified accordingly.

Figure VI-2(a) to be meaningful. To achieve b simultaneous paths we need $q = s$ if $s \le b$ and $q \ge b$ if $s > b$.

Telephone switching theory provides several directly applicable definitions and results.

A rearrangeable network is one that provides all possible one-to-one input-output connections, just as assumed for the switching unit S itself; i.e., it is a permutation network. In general, however, if some of these connections have already been established along certain paths, it may be necessary to reroute them in order to set up additional connections. A nonblocking network also has full permutation capability, but in this case such rerouting is never necessary, regardless of the order and the particular routing with which prior connections are set up. These two classes of switching networks are useful theoretical models for telephone switching, but are never used in practice because of their high cost, and because only a small fraction of all telephones are ever in use at the same time.

The parameters of the three-level network are constrained as follows:

- For a nonblocking network, $q \ge 2s - 1$.
- For a rearrangeable network, $q \ge s$.
- For most telephone networks, $q << s$.

For the bus interconnection network, it has been assumed sufficient to have as few as $b < p$ simultaneous connection paths. Consequently, a rearrangeable or nonblocking capability is not needed; however, such capability may be an asset if it can be achieved at a small additional cost--a definite possibility, since the utilization ratio b/p is larger here than in telephone practice.

Cost parameters for the scanners and switches in the individual levels may now be readily calculated, just as was done in the two-level case. For parallel data transfer:*

*Actually, a scanner having only two positions is somewhat less complex than the above expressions indicate. The value $g_{sc} = 21$ has been used in this case instead of the value given by these formulas ($g_{sc} + 34$).

Level 1

$$\left.\begin{aligned}
g_{sc} &= 13s + 8 \\
t_{sc} &= 2s + 2 \\
d_{sc} &= 1 \text{ to } s \\
g_{sw} &+ (n + \tfrac{p}{s} + q)(s + 1) \\
t_{sw} &= (n + \tfrac{p}{s} + q + 1)(s + 1) \\
d_{sw} &= 0
\end{aligned}\right\} \quad (\tfrac{pq}{s} \text{ of these})$$

Level 2

$$\left.\begin{aligned}
g_{sc} &= 13\tfrac{p}{s} + 8 \\
t_{sc} &= 2\tfrac{p}{s} + 2 \\
d_{sc} &= 1 \text{ to } \tfrac{p}{s} \\
g_{sw} &= (n + \tfrac{p}{s})(\tfrac{p}{s} + 1) \\
t_{sw} &= (n + \tfrac{p}{s} + 1)(\tfrac{p}{s} + 1) \\
d_{sw} &+ 0
\end{aligned}\right\} \quad (\tfrac{pq}{s} \text{ of these})$$

Level 3

$$\left.\begin{aligned}
g_{sc} &= 13q + 8 \\
t_{sc} &= 2q + 2 \\
d_{sc} &= 1 \text{ to } q \\
g_{sw} &= n(q + 1) \\
t_{sw} &= (n + 1)(q + 1) \\
d_{sw} &= 0
\end{aligned}\right\} \quad (p \text{ of these})$$

Summation of these individual contributions leads to complex expressions for G_3 and T_3. Those for T_3 are identical in form and similar in value to those for G_3, differing only in some of the constants. Consequently, it will be sufficient to deal with G_3 only, in optimizing q and s in terms of p, b, and n.

First, note from the contributing terms that the dependences on q and s are strictly positive and inverse, respectively, so that G_3 will be least when q is minimized and s is maximized, subject only to the constraining inequalities cited above. Two cases must be distinguished. If $p \le b^2$, then we have $q = s \le p/b \le b$. Selection of the largest possible value of s gives $s = p/b$, yielding

$$G_3 = p^2[p + (2n + 27)b]/b^2 + p(p + 3n + 24) + pb(b + n + 15).$$

On the other hand, if $p > b^2$, then we have $q = b$ and $b < s \le p/b$. The maximum value of s is the same, yielding

$$G_3 = b^4 + (n + 16)b^3 + 2(n + p + 8)b^2 + 2(n + 13)pb + p(n + 8).$$

The corresponding values of D_3 are

$$D_{3min} = 5$$

$$D_{3max} = \begin{cases} \frac{2p}{b} + b + 2 \text{ when } p \le b^2 \\ \frac{p}{b} + 2b + 2 \text{ when } p > b^2 \end{cases}$$

4. Bit-Serial Transfer

For serial implementation of the controller, the scanner has the same costs, while the number of bit lines in the switch is reduced from n to just 3: a single address line, an acknowledgment line, and a data-word line. However, the time required for actual transfer is now increased by n. Thus,

$$g_{sw} = 3p$$
$$t_{sw} = 4p$$
$$d_{sw} = n$$

For all p memory units in the single-level case, then

$$G_1 = 3p^2 + p(13p - 5) = 16p^2 - 5p$$

$$T_1 = 2p^2 + 4p^2 = 6p^2$$

$$D_{1min} = n + 3, \; D_{1max} = n + p + 1$$

For the two-level case, assuming parallel transmission of memory selection lines through the bus controllers, but serial transfer of all addresses and data,

$$\begin{aligned} G_2 &= b(13p + 8) + b(p + 1)(3 + p) + p(13b + 8) + p(b + 1)3 \\ &= b(p^2 + 33p + 11) + 11p \end{aligned}$$

$$\begin{aligned} T_2 &= b(2p + 2) + b(p + 1)(4 + p) + p(2b + 2) + p4(b + 1) \\ &= b(p^2 + 13p + 6) + 6p \end{aligned}$$

$$D_{2min} = n + 4, \; D_{2max} = n + p + b = 2 \quad .$$

For the three-level case, taking $s = p/b$ and either $q = s$ or $q = b$, as before:

$$\begin{aligned} G_3 &= p^2(p + 33b)/b^2 + p(p + 33) + bp(b + 18) \text{ for } p \le b^2, \\ &= b^4 + 19b^3 = 2(p + 11)b^2 + 32pb + 11p \text{ for } p > b^2 \quad . \end{aligned}$$

$$\begin{aligned} D_{3min} = n + 5, \; D_{3max} &= \frac{2p}{b} + b + n + 2 \text{ for } p \le b^2, \\ &= \frac{p}{b} + 2b + n + 2 \text{ for } p > b^2 \quad . \end{aligned}$$

In these calculations, it is assumed that the circuitry required for serialization and parallelization of addresses and data is integrated into the processors and memories, respectively, at no appreciable change in hardware cost within these units. If this assumption is not justified for the technology chosen for processor and memory implementation, then the effective value of G_ℓ will increase over that calculated here.

5. Byte-Serial Transfer

Similar calculations apply if the transfer is effected using $r = \lceil n_a/\beta \rceil + \lceil n_w/\beta \rceil$ successive β-bit bytes. Here, $n \rightarrow 2_\beta + 1$ in G and T, and the delay D increases by r over the value for the parallel mode. The results for the single-level network become:

$$G_1 = (2\beta + 14)p^2 = 5p$$
$$T_1 = (2\beta + 4)p^2$$
$$D_{1min} = r + 3,\ D_{1max} = r + p + 1 \quad .$$

For the two-level network:

$$G_2 = bp^2 + (4\beta + 29)bp + (b + p)(2\beta + 9)$$
$$T_2 = bp^2 + (4\beta + 9)bp + (b + p)(2\beta + 4)$$
$$D_{2min} = r + 4,\ D_{2max} = r + p + b + 2 \quad .$$

Finally, for the three-level network:

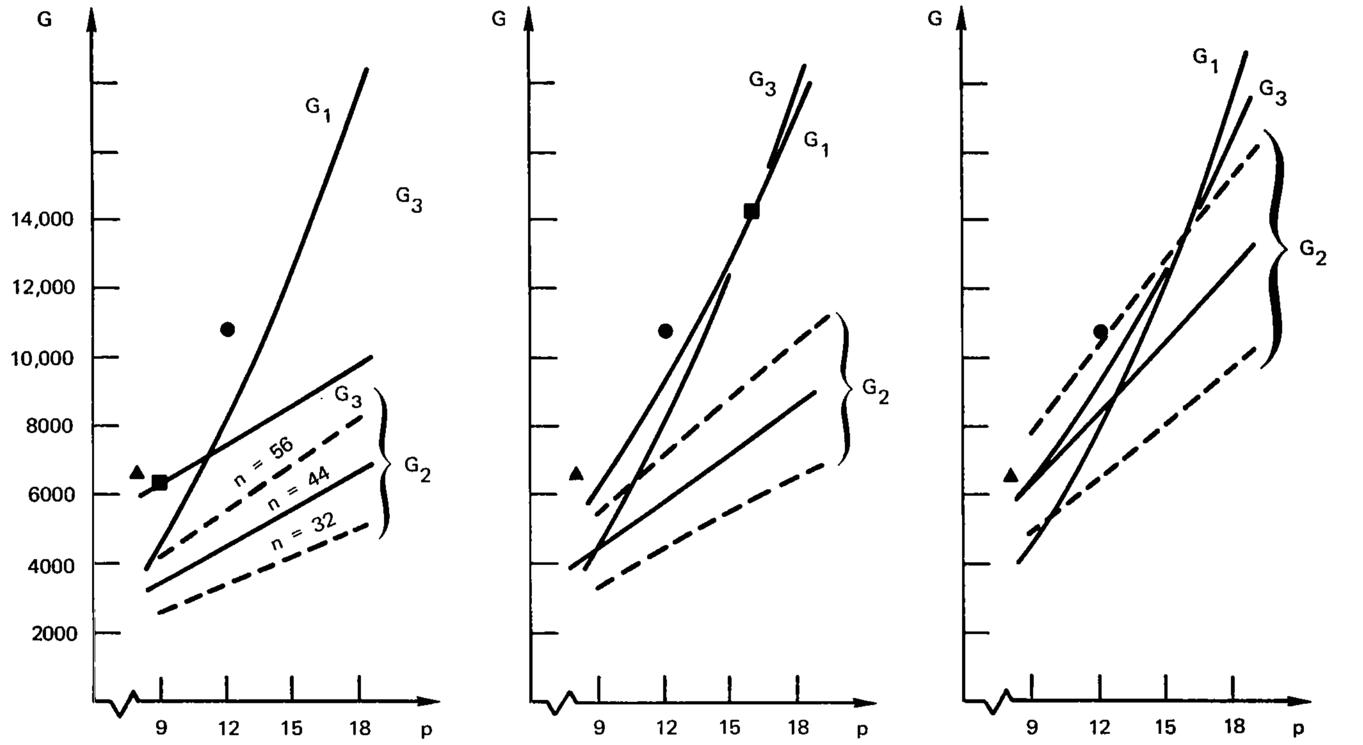
$$\begin{aligned} G_3 &= p^2[p + (4\beta + 29)b]/b^2 + p(p + 6\beta + 27) + \\ &\quad pb(b + 2\beta + 16) \text{ for } p \le b^2, \\ &= b^4 + (2\beta + 17)b^3 + 2(2\beta + p + 9)b^2 + 2(2\beta + 14)pb + \\ &\quad p(2\beta + 9) \text{ for } p > b^2 \quad . \end{aligned}$$

$$\begin{aligned} D_{3min} &= 5 + r, \\ D_{3max} &= \frac{2p}{b} + b + r + 2 \text{ for } p \le b^2, \\ &= \frac{p}{b} = 2b + r + 2 \text{ for } p > b^2 \quad . \end{aligned}$$

6. Comparative Analysis of Cost Measures

Figures VI-7(a), (b), and (c) display collectively the magnitude of G as a function of p, b, n, and ℓ over the ranges of interest of these parameters, for the parallel mode of data transfer. These curves are shown for the typical value of $n = n_a + n_w = 44$ bits (solid curves), with values for the minimum and maximum ($n = 32$ and $n = 56$, respectively) designated by dotted curves for the two-level case. (The others are very similar.)

All of the formulas for T are so similar in form and relative value to the corresponding ones for G that there is no need to display their values separately. It may be safely concluded that the ranges of optimality of design parameters based on the total number T of scanner and switch terminals are very nearly the same as those based on the total number G of gates.



(a) b = 3

(b) b = 4

(c) b = 6

FIGURE VI-7 GATE COSTS FOR PARALLEL TRANSFER

Reference to these figures leads to the following conclusions for the parallel case.

For b = 4 and p large the two-level network is preferred by a wide margin, but as b is increased or p decreased, both the one- and three-level networks become more competitive in terms of gate cost. In particular, the two-level network is superior to the three-level over the entire range of parameter values of interest, and to the one-level network as well over all of this range except when $p \lesssim 2b$; however, this extreme combination of values is very unlikely to occur in the design of the SIFT computer. The dependence of the minimum gate cost on all three parameters p, b, and n is linear and nearly proportional; in fact the approximate formula

$$G_{min} \approx 2\left(p + \frac{1}{2}\right)\left(b + \frac{1}{2}\right)(n + 8)$$

holds within 3% over the entire range of interest. This formula allows one to estimate quickly the effect of a change in one of the design parameters on the circuit complexity.

Gate costs for the serial mode of transfer are shown in Figure VI-A-8 for the case b = 4. Again, the superiority of the two-level implementation is apparent.

For the typical case: p = 12, b = 4, n = 44, and $\beta = 4$, Table VI-1 lists maximum delay times $D_{\ell max}$ for 1, 2, and 3 levels and for all three transfer modes. Dependence on p, n, and b is linear (where there is any dependence at all), so the sensitivity of D_{max} to changes in these parameters can be readily estimated.

It is clear that, within each mode, the one- and three-level networks are preferred from the standpoint of minimizing the maximum delay time. However, the differences are only about 25% to 30% for the parallel mode, are truly negligible for the bit-serial mode, and are in between for the byte-serial mode.

It may be concluded that the operating speed of the interconnection network is not critically dependent upon the number of levels in the network, but (not surprisingly) depends rather critically upon whether

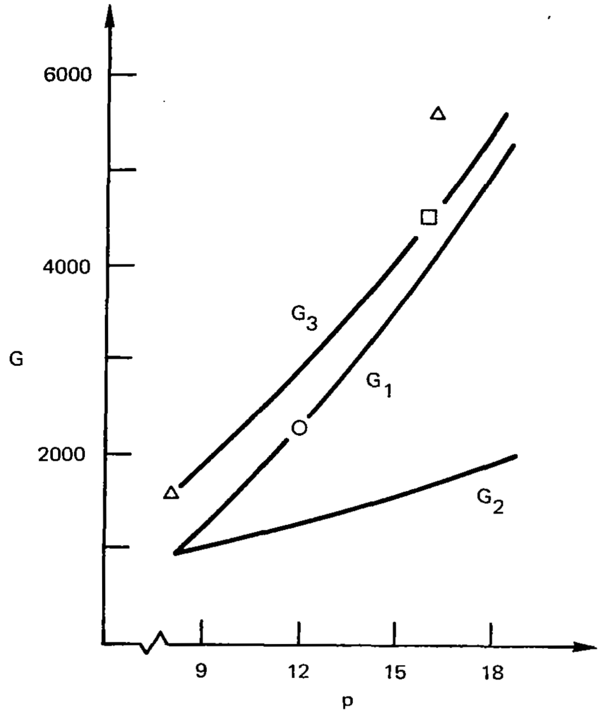


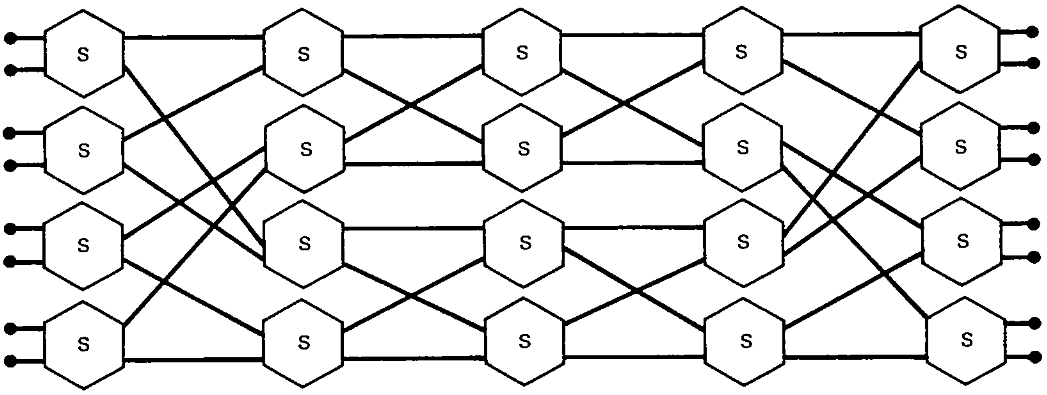
FIGURE VI-8 GATE COSTS FOR BIT-SERIAL TRANSFER b = 4

the parallel, byte-serial, or bit-serial mode of transfer is employed. In the parallel case there is a secondary dependence on the number p of processors and number b of busses, as expressed in the equations

$$D_{1max} = p + 1, \quad D_{2max} = p + b + 2, \quad D_{3max} \approx 3b + 2 \quad .$$

7. Networks with More Than Three Levels

For sufficiently large p, the costs G_ℓ and T_ℓ can be reduced by employing an interconnection network having more than three levels. This can be done by holding q and s at small values (2, 3, or 4) and applying recursively to each controller in the center level the one-to-three-level transformation implied by Figures VI-2(a) and VI-2(c). For example, a five-level network so generated from Figure VI-2(c) for p = 8, s = q = 2 is illustrated in Figure VI-9. The relevant concern here is whether such an alternative is preferred over the two-level case for the range of values of p likely to be encountered in the SIFT interconnection network.



p = 8, u = 3, p = 5

FIGURE VI-9 EXAMPLE OF A FIVE-LEVEL NETWORK

The limiting situation q = s = 2 may be examined first. In this case each of the S-units has two inputs and two outputs. The network is the well-known Benes-Waksman arran [Ref. 2] shown in Figure VI-9 for $p = 8$. This network is known to have complete permutation capability ($b = p$), and for $p = 2^u$ has $2u - 1$ levels and $p(u - 1/2)$ S-units. (A less symmetrical version has a slightly smaller number $p(u - 1) + 1$ of S-units, but would probably require a more complex routing algorithm.) Thus,

$$\ell = 2u - 1,$$
$$\sigma_\ell = 2p(2u - 1) \quad .$$

Each S-unit requires a pair of route-selection lines from the source processor. At the ith-level, then,

$$g_{swi} = 3[n + 2(\ell - i)] \quad .$$

Summing over all levels to get the grand total

$$G_\ell = \sum_{i=1}^{\ell} p(g_{sw} + g_{sc}),$$

we obtain, taking $g_{sc} = 21$ as before,

$$g_\ell = 6.2^{u-1}(2u - 1)(n + 2u + 5), \text{ where } u = \log_2(p)$$

For $p = 16$ we have $u = 4$ and $\ell = 7$, giving $G_7 = 19{,}152$ for $n = 44$ and $G_7 = 5736$ for the serial mode. These values are much greater than the minimum values plotted in Figures VI-7 and VI-8. The corresponding costs for $p = 8$ (Figure VI-9) are $G_5 = 6600$ and 1680. These values are indicated by small triangles in Figures VI-7 and VI-8. We may conclude that the exclusive use of 2 × 2 S-units, using as many levels as needed, gives rearrangeability but is more costly than all other solutions in the total number of gates required. The delay D_{max} would also increase in these five- and seven-level realizations, of course.

Exclusive use of 3 × 3 S-units does not lead to a fully utilized network having $\ell > 3$ levels until $p = 3^3 = 27$. For $p = 12$, however, a hybrid five-level network for $s = q = 3$ can be formed by realizing each of the three 4 × 4 S-units in the central level in Figure VI-2(c) in the form of a three-level, six-element subnetwork of 2 × 2 S-units. The final network, shown in Figure VI-10, has a total gate cost $G = 10{,}860$ for $n = 44$ and $G_5 = 2496$ for the serial mode. Another five-level version using 2 × 2 and 3 × 3 S-units in alternate levels has exactly the same cost. These values are indicated by small circles in Figures VI-7 and VI-8. Again, these costs are quite high and indicate the undesirability of increasing the number of levels above three.

8. Modularization of the Bus Interconnection Network

The most complex scanner encountered in the previous discussions had $g_{sc} = 13p + 8 \geq 242$ gates and $t_{sc} = 2p + 2 \leq 39$ terminals. Consequently, there would be no difficulty in implementing the scanner as a separate semiconductor module, should this be desired. The switch, on the other hand, has a typical complexity given by

$$g_{sw} = (n + \gamma)(\delta + 1)$$

$$t_{sw} = (n + \gamma + 1)(\delta + 1)$$

where δ varies from 0 to p and δ from b or s to p. Except for the serial mode ($n = 3$), then, the switch will need to be partitioned into two or more parts to fit on any but the largest LSI modules. This may be conveniently done in the present instance by taking advantage of the

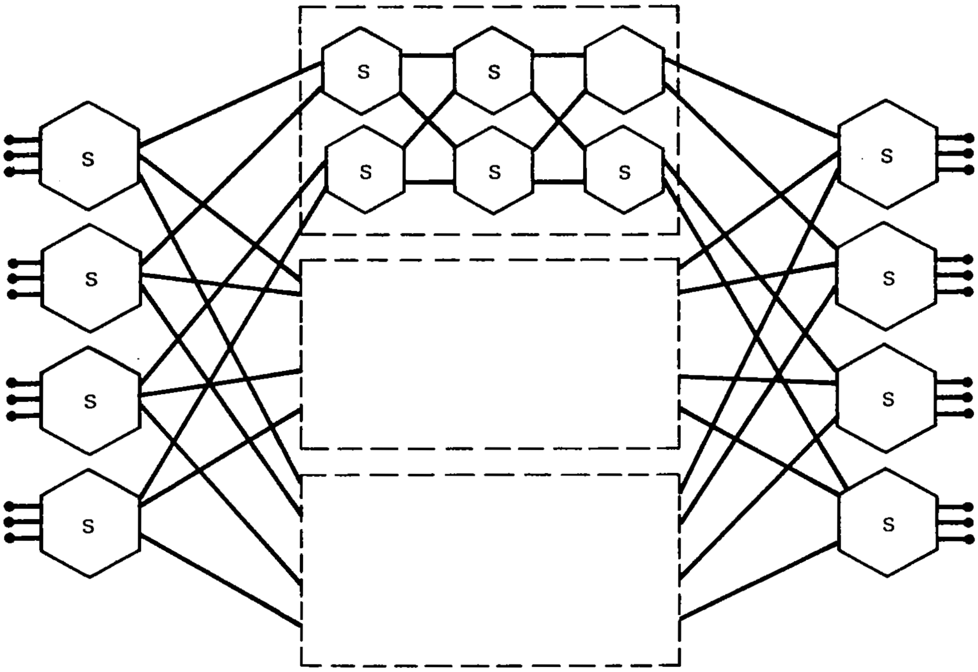


FIGURE VI-10 FIVE-LEVEL NETWORK USING 2 X 2 AND 3 X 3 S-UNITS IN ALTERNATE LEVELS

iterative form of the circuitry for the n parallel bits. Only the δ gate control lines (from the counter in the scanner) need be repeated in each module, and all switch modules at the same level could be essentially the same.

9. Comparison of Delay Times

The maximum delay times through the interconnection network for different values of ℓ are shown in Table VI-1.

A particularly attractive design alternative for realization would be one in which all controllers have the same number of inputs and outputs, so that a common module might then be utilized in all three levels. This would result in a minimum of wasted gates and terminals, merely those corresponding to some of the routing selection digits in levels past the first. With reference to Figure VI-2(c), this condition

Table VI-1

SUMMARY OF MAXIMUM DELAY TIMES $D_{\ell \max}$

$(p = 12, b = 4, n = 44, \beta = 4)$

ℓ	Parallel	Byte-Serial	Bit-Serial
1	13	24	57
2	18	29	62
3	12	23	56

requires that $s = q = p/s$, so that $p = q^2 = b^2$, for the three-level network. Hence,

$$G_3 = 3b^2[b^2 + b(n + 14) + (n + 8)]$$

for the parallel case, and with n replaced by 3 for the serial case. Each of the $3b^2$ modules now requires $g_3 = g_{sc} + g_{sw} = 2b^2 + b(n + 15) + (n + 8)$ gates and $t_3 = t_{sc} + t_{sw} - p = b^2 + b(n + 5) + (n + 3)$ terminals. Results for the only two cases of interest, corresponding to b = 3 and b = 4, are tabulated in Table VI-2 for parallel transfer (n = 44) and for serial transfer. The network for b = 3 is shown in Figure VI-6.

It is immediately apparent that the modules are pin-limited rather than gate-limited for any modern semiconductor technology. A serial controller might fit nicely on one semiconductor chip, but the parallel version would still need to be bit-partitioned to make this possible. Nevertheless, this possibility is an attractive one from the standpoint of implementation, despite the nearly double total gate cost (small squares in Figures VI-7 and VI-8).

If this high gate cost can be afforded, the single-level network must be reconsidered, since all controllers are also identical in this case:

$$G_1 = p^2(n + 13) = 5p$$

$$g_1 = p(n + 13) - 5$$

$$t_1 = p(n + 3) \quad .$$

Table VI-2

SUMMARY OF COSTS FOR NETWORK REALIZATION
USING ALL-IDENTICAL MODULES

			Three-Level Network						One-Level Network					
			Parallel, n = 44			Bit-Serial			Parallel, n = 44			Bit-Serial		
b	p	No. of Modules	G_3	g_3	t_3	G_3	g_3	t_3	G_1	g_1	t_1	G_1	g_1	t_1
3	9	27	6,345	247	203	1,917	83	39	4,572	508	423	1,251	139	54
4	16	48	14,400	320	259	4,560	115	54	14.512	907	752	4,016	251	96

The total number of gates required is now somewhat less, but the number of terminals (and gates) per module is substantially greater, so more modules would be required.

All of these alternatives are compared numerically in Table VI-2.

10. Fault Tolerance Aspects

We follow the principle that recovery should be possible from all single faults and from all double and most other multiple faults that occur sufficiently far apart in time to allow a delay for the fault analysis programs to identify the faulty unit and reconfigure the machine to avoid its use.

This degree of fault tolerance can be achieved by defining and isolating units in such a way as to limit the extent of a "single" fault. Note that the extent of a fault is determined not only by how the domain of improper operation propagates electrically, but also by the precision with which the diagnostic routines are able to pinpoint the location of the fault. For example, if these routines are so weak that a fault in A can be narrowed down only to the combination (P_1,A), then the extent of the fault must be considered to be (P_1,A) and not just (A). With reference to Figure VI-11, this ability to limit means that any one of the units P_1, P_2, P_3, A_1, A_2, or A_3 may fail as a result of a single fault, but

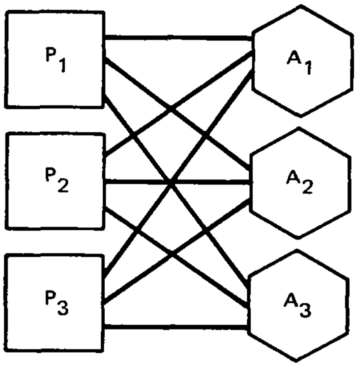


FIGURE VI-11 ARRANGEMENT OF UNITS TO ACHIEVE FAULT TOLERANCE

that a fault in unit A_1, for example, must never cause A_2 of A_3 to become inoperative. Similarly, a fault in P_1 must never incapacitate P_2 or P_3. It is possible, though presumably less likely, for a fault to develop on an actual connection path--from P_1 to A_1, for instance--that prevents communication between these two units. Such a fault might occur without blocking proper operation of P_1 in conjunction with A_2 and A_3, or of A_1 in conjunction with P_1 and P_2. On the other hand, such a fault might also disable either P_1 or A_1 or both. What we do insist, however, is that if A_1 must be declared faulty, then A_2 and A_3 will not be rendered inoperative by the same fault, and if P_1 is declared faulty, P_2 and P_3 are not affected.

In terms of the connection graph for the units of the entire computer, this condition is equivalent to requiring that a single fault may disable (a) any one branch, (b) any one node, (c) any one node with an incident branch, or (d) any two connected nodes and their incident branch.

Proper definition of what constitutes a separate unit with respect to single faults requires only that replicated parallel subnetworks be regarded as separate units, just as one would expect. Isolation of these separate units must then proceed according to the principle that parallel replicated units should never be connected together in any way.

At the circuit level this condition of isolation requires that merging lines must be electrically isolated at all points of fan-in and fan-out of these replicated units. This might be implemented as simply as employing resistors or diodes for input clamping and for interunit connections at all points of fan-in and fan-out, in order to prevent propagation of faults between units. However, optical channels or special coupling circuits might be preferred, depending upon the nature of the actual signals and the magnitudes of the fault probabilities.

To prevent the most likely types of double faults (those confined to single units) from blocking fault-free units from use, an additional condition on the three-level interconnection network should be imposed: that all switch units should have a fan-in and fan-out of at least three. Thus,

$$s \geq 3, \quad q \geq 3, \quad \frac{p}{s} \geq 3 \quad .$$

This requirement is satisfied by all of the realizations discussed previously.

Diagnostic routines should have no difficulty in pinpointing the location of a fault to any switch, since this unit is purely combinational and is interposed in data paths between a processor and a memory. Thus all stuck-at and short-circuit faults will appear as errors in address or data transmission and can be readily detected and located. Some of the switched lines carry routing and memory selection information, but errors in this information will also show up as data errors whenever the wrong memory is selected. To provide more positive protection against memory selection errors, it may be desirable to assign replicated files to different address blocks within the various memory units.

The scanner, while a simpler unit, operates sequentially and must therefore be checked externally for both "do-nothing" faults and faults that cause two otherwise nonsimultaneous actions to occur at the same time. The former type of fault will presumably be detected by whatever overtime monitor is used for processor operations. The latter type could cause small delays if a processor tries to communicate with a memory unit through more than a single path through the interconnection network, or it could cause data errors if a processor becomes connected to more than a single memory unit. Both of these types of faults are readily detected. However, the second may require some special attention in preparation of the fault location routines so that the fault can be properly localized.

11. Routing

When the number of levels in the interconnection network is two or more, the allocation program in the system executive must contain a routine for assigning busses or a route for each processor-to-memory access. This routing information is forwarded to the processor as part of the execution program; in effect it becomes a portion of the memory-select and address information. However, the assignment of access routes cannot be made independent of one another. The scheduling of all of the various tasks must be considered, in order to avoid conflicts and delays that would otherwise result from two processors demanding overlapping routes. Thus, route assignment must take into account both the scheduling of tasks and the detailed interconnection possibilities within the interconnection network.

For the two-level network, all bus controllers connect symmetrically to all processors and also to all memories. Consequently, potential routing conflicts can be circumvented by simply avoiding the assignment of the same bus to two concurrent accesses. If the scheduling constraints are not too severe, such an assignment might be handled by simply rotating the assignment of busses to processors. This allocation rule would be used until faults occur. As particular bus controllers, processor-to-bus connections, and bus-to-memory connections are recognized as potentially faulty and are taken from use, the assignment algorithm would become more constrained. It could still operate on a "next available" basis, or by whatever algorithm is used for handling defective processors and memories.

If the number of levels is three or more, the choices between possible routes between processors and memories are no longer equally preferable. As indicated previously, these interdependencies could be completely avoided by employing a nonblocking interconnection network. This form of network was seen above to be nearly twice as costly as a rearrangeable network, however, and its use is probably not justified in the present application. In view of the small number of connections likely to be set up simultaneously, relative to the total number of processors, routing conflicts would appear to be the exception rather than the rule, even if a simple "next available path" assignment algorithm were used. Telephone theory provides sophisticated algorithms aimed at minimizing the blocking probability for average low-level use of a switching system [Ref. 1]. In the present case, however, the number of processors is probably too small to make such elaborate schemes either necessary or beneficial.

It seems sufficient, therefore, for the allocation subroutine to maintain and update a simple table, which contains for each processor (row of the table) and memory unit (column) a list of the paths that have been provided for in the design--normally 3 or 4--and the status of each such path: (a) fault-free and available at the moment, (b) fault-free but busy, (c) potentially faulty, or (d) faulty. Except for the multiple choice of routes, this table is the same in kind as must be maintained to store the status of all units in the SIFT computer.

12. Conclusion

The foregoing analysis indicates the superiority of a two-level interconnection network over alternatives employinp only one level or three or more levels for virtually all ranges of parameters likely to be encountered in the SIFT computer and for all three transfer modes, parallel, byte-serial, and bit-serial. In terms of the various cost measures, the two-level network is less complex in total gate and terminal counts for all parameter ranges of interest, in most cases by a wide margin. It has the same (or a little greater) maximum access delay.

The bus interconnection network is readily decomposed into circuit modules, although some sacrifice in gate cost can be expected if all of these modules are to be made alike.

These conclusions are not yet based upon a reliability analysis, one that will take into account the various fault probabilities in each type of unit and especially failures in interunit connections. The final design choice for the interconnectio network will depend to some extent upon this analysis, as well as upon the scheduling algorithms and diagnostic strategy adopted and upon system tradeoffs involving speed requirements, relative hardware costs, and the parameters assumed specified: p, b, and n.

The principal unanswered question at this stage of the design concerns the matter of how the controller should be realized in hardware. No two controllers occupying similar replicated positions in the network should share the same chip. However, most of the controllers are terminal-limited rather than gate-limited, and this poses a problem for good economy of realization.

B. Input/Output Subsystem

1. Introduction and Summary

The input/output subsystem of the SIFT computer is used to connect to the aircraft environment. It clearly must satisfy the same kind of reliability requirements as the remainder of the system. In addition, it is highly constrained by the character of the devices within the aircraft; for example, replicated air pressure sensors will not produce identically the same readings, so the voting on this data will have to allow for this fact.

The basic scheme for the input/output subsystem is:

- Critical sensors are replicated, and the programs that require the data read all the versions and carry out a voting procedure as with any data that are read.
- Critical actuators must be replicated, each of them containing sufficient local logic to be able to read the several versions of the output data that they require and carry out local voting, possibly by mechanisms similar to those currently employed with multiple actuators on aircraft, e.g., forced sum voting.
- Noncritical sensors and actuators are not replicated but are connected to the system in the same way as critical ones in order to preserve the same fault-isolation rules on the input/output subsystem as are used between processing modules.
- The central computing elements of the SIFT system are isolated from noncritical sensors and actuators.

The manner by which the above objectives are achieved is described below, starting with the design of a system for critical sensor and actuator input/output, followed by a discussion of appropriate structures for noncritical units. The section concludes with considerations of the aircraft bus structure and the question of problems of the positioning of the logic of sensors and actuators at the central computer or at the units themselves.

2. Critical Input/Output Units

Figure VI-12 shows how critical input and output units would be connected to the central SIFT computer system. We assume that data to and from the SIFT system flows on a multiple bus system, which is connected to the main bus system of SIFT via logic that is realized by a specially programmed microprocessor (marked as P in Figure VI-12).

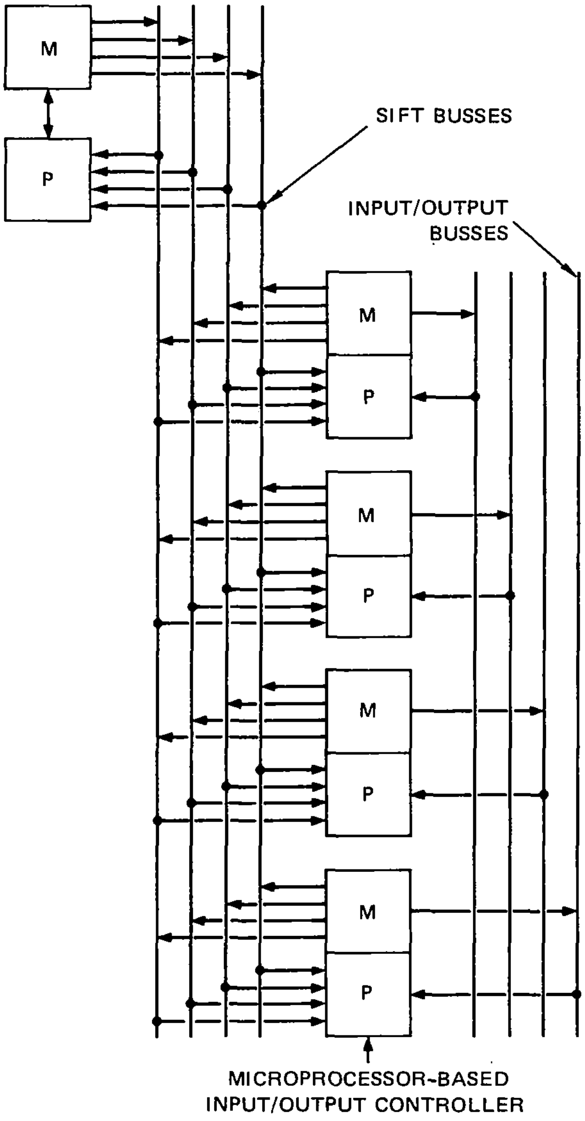


FIGURE VI-12 INPUT/OUTPUT FOR CRITICAL SENSORS AND ACTUATORS

Each microprocessor operates in the same manner as the main processors of SIFT, except that the tasks that are to be performed are much smaller and the executive that resides in them is a reduced version of the LE/GE combination in the central processors. The reductions that are made are:

- No global executive is present in the microprocessors, as the functions normally performed by it are either not necessary or are carried out by the GE of the central processors.
- The LE contains only the voter, scheduling, and dispatching functions, together with sufficient of the global/local interface to enable it to determine its schedules by reading the central GE tables.

In all other respects the I/O processors operate according to the same general rules as the central processors. This includes voting on multiple input to achieve error detection and correction, reconfiguration by change of scheduling tables, and the restriction that a processor may only read data from other processors and may not write into the memory of other processors.

Figure VI-12 shows the logical structure but does not indicate the physical placement of the I/O processors. It is anticipated that some or all of the I/O processors could be placed close to the sensors and actuators that must be controlled. Such considerations depend on the economics and reliability predictions for the various bus system technologies.

3. Noncritical Input Units

In a SIFT system that is carrying out both critical and noncritical tasks, it is necessary to maintain a separation between the tasks because the noncritical tasks may not receive as much validation and verification as the critical tasks and thus may corrupt them. A method of protection, whereby a program cannot affect the memory outside that allocated to it, prevents programs for the noncritical tasks from interfering with the critical ones (see Subsection VI-C-4). The same degree of protection against the possibility of errors in the hardware is achieved by the use of a microprocessor-based unit that connects to the main bus system and is used to read from the sensor and deposit the results of the read operation in its own memory. These results can then be read by the main processors of SIFT. This scheme effectively isolates potentially unreliable equipment from the other units of the system.

4. Noncritical Actuator Units

Noncritical actuators can be dealt with in the same manner as noncritical sensors, by interposing a microprocessor-based unit between the busses of SIFT and the units themselves. To some extent this may not be necessary if the units themselves are so connected into the bus system that they can only read from the SIFT module memories. This constraint on the operation can be achieved in the bus control mechanism, as is the case for the connection of SIFT modules themselves. The advantage of the use of a microprocessor is that attention can be given to the design of its interconnection and the same logic can be used many times, whereas if the units are themselves connected, then it is necessary to ensure that the logic and physical design preclude the propagation of errors or damage. This validation would have to be carried out for each individual type of unit that is connected to the system.

C. SIFT Memory System Design

1. Introduction and Summary

In this discussion of SIFT memory system design, we consider the storage of programs and data for high-rate control and display functions, which are served by the distributed memories of the basic SIFT scheme, and also the low-rate, high-volume storage that may be required in a practical, general-purpose aircraft computer.

At the present state of development of the SIFT architecture, the following questions about memory are the most pertinent:

- How many levels of memory are needed to accommodate the range of storage capacities and speeds in the SIFT prototype?

- What technologies are appropriate to the various storage functions?
- What special logical functions may be needed within memory modules to support SIFT processing and communication modes?
- What fault-tolerance capabilities are appropriate to the various levels of the memory hierarchy?
- What are the basic performance requirements for SIFT memories?

These questions are discussed in order.

The following conclusions are derived in the discussion:

- Two levels of memory are needed in an Air Transport SIFT: a set of high-speed random access memories (RAM) for program execution and a high-capacity block-access byte-serial store. A large central RAM is not needed.
- The following fault-tolerance schemes appear attractive:
  - Single-error-correction, double-error-detection codes exist for RAM data channels.
  - Software reconfiguration of contiguous blocks of words in RAM is provided.
  - Either arithmetic sum checks or longitudinal parity checks are made for a byte-serial store.
  - Redundant address information (to some degree of precision) may be usefully appended to each RAM word.
  - The appropriate form of redundancy for the block access memory needs further study. Dual redundancy appears satisfactory.
  - A section of read-only memory may be employed usefully in each processor memory.
  - A reliable form of nonvolatile writable RAM would be beneficial, but is not presently available. Awareness of future developments is desirable.
  - The feasibility of marginal checking for contemporary semiconductor memories should be investigated. It is potentially of great value in SIFT.
  - Memory design should allow for the possible use of tagged architecture. Tagging of words appears to have several beneficial uses, e.g., in protecting data from erasure because of erroneous address calculation.

- The problem of unflexed fault tolerance circuits is significant, but solutions are apparent, e.g., permanently string data patterns in memories that can test error-detection logic.

2. Memory Hierarchy

The most pressing concern about the SIFT prototype memory hierarchy is the need for a central fast memory. Our studies[3] have determined that the high-rate aircraft control programs may be served adequately by the memory modules associated with the distributed SIFT processors. It also appears necessary to have capability for storing high-volume data with relatively low requirements for access speed. Data of this type include:

- Copies of all programs, to be used in an extreme situation when normal reconfiguration fails.
- Infrequently used programs, e.g., for diagnosis.
- Sequences of recent input-output activity, to be used for system recovery or off-line system analysis.

The volume and rate required for this class of data are now known precisely, but it appears that (1) transfer of a block of data, with latency on the order of a few milliseconds, is satisfactory (see Section V), and (2) capacities on the order of 10^7 bits and data rates of 10^6 bits per second are reasonable to demand. Such characteristics are provided by current technologies in the form of block-organized shift registers.

These considerations indicate that the SIFT prototype should provide for serial-mode, block-transfer storage as well as multiple, fast, random-access storage units for the distributed processing of high-rate programs. Given the disparity in speeds and sizes of the two memory types, it is natural to consider the use of an intermediate level of storage, such as a large-capacity random-access memory.

The following uses for such a memory are apparent:

(a) Direct program execution, one benefit of which would be an economy in storage, since failures in a processor would not require abandonment of memory, as in the present distributed processor design. A second benefit would be a reduction in the overhead required for the memory mapping that supports reconfiguration of storage under faults.

(b) Storage of rarely used programs, a benefit of which would be a reduction in the required capacity of the distributed processors for programs which, when needed, require rapid access.

(c) Storage of flight-critical programs for rapid loading of distributed memories during fault recovery.

(d) Storage of future very large programs that exceed the individual capacity of present distributed memories. A benefit would be to simplify reconfiguration procedures, since the distributed memories would be reserved for small programs.

Of these points, only (a) requires special architectural provisions, since the direct execution of programs from a central memory requires a major increase in bus data rates and/or a redesign of the bus concept. The major benefits appear to be economic, on the grounds that in a central memory, failure of a processor does not cause a reduction in memory; hence a lower amount of memory redundancy is needed. The economic impact would appear to be small, because processor failure is expected to contribute little to system failure rate compared to memory itself.

Point (b), storage of rarely used programs, can be satisfied by the block-organized storage level, provided that the store is capable of loading a 1K program block in 12 ms [Ref. 3].

Point (c), storage for rapid loading of critical programs, appears not to have significant benefit. All the critical programs surveyed are small and can be transferred between a pair of distributed memories with sufficient speed to meet recovery requirements. If, in the future, critical programs are added that are so large as to prohibit rapid transfer, satisfactory reconfiguration could be achieved by employing a higher order of replication than is currently expected for application programs.

Point (d), storage of abnormally large programs, does not constitute a compelling reason for distinguishing a separate level of memory. It might require that some of the distributed memories be of larger than average size. This would tend to constrain the flexibility of reconfiguration, but it does not seem to be a serious problem.

We conclude that an intermediate level of storage, in the form of a large random access memory, is not justified.

It should be noted that the arguments given are based on the particular computational needs of the air transport problem environment and on currently feasible memory organizations. The appropriate memory hierarchy for a SIFT computer used in different problem areas, e.g., time-shared computing or communications processing, would have to be reexamined. New memory developments might also have an impact. For example, an extremely low-cost serial store might be a useful attachment to each dedicated processor-memory.

3. Memory Technologies

The primary impact of memory technology on SIFT architecture is felt in the following issues:

- The choice of magnetic core or semiconductor storage for the processor memories.
- The feasibility of a block-access 10^7-bit secondary store.
- The need for fixed or nonvolatile storage.

In the July 1974 study [Ref. 3] it was argued that semiconductor memories are preferable to magnetic-core memories on the ground that (1) they tend to use much fewer circuit connections and manual assembly operations, (2) the drive circuits operate at lower, hence less stressful power levels, and (3) low-level sense signals are restricted to the interior of the devices, hence are more immune to noise. We believe that these factors still apply, and our following discussions assume the use of semiconductor memories for the processor memories. We also observe that magnetic-core memory technology continues to evolve appreciably. Therefore the comparative value of the two memories should be reviewed periodically.

Recent developments have established the feasibility of two novel technologies for block-access stores: charge-coupled devices (CCD) and magnetic bubble storage (MBS). Both are amenable to byte-serial shift-register type structures and are well suited for block-structuring, with block lengths of 10^4-10^5 bits. Data rates appear to favor CCD by a factor of two to three. Contemporary CCD units have a maximum rate of about 5 Mb/s, while MBS units have a maximum rate of about 2 Mb/s. CCD appears to be capable of higher speed through design refinements, but significant increases in MBS speeds may require a breakthrough in technology.

An important function of a block-store memory is to retain data between flight periods. At such times, aircraft power may be off. It is therefore important to consider the feasibility of means for preserving data with power-off in the two schemes. MBS appears to have a significant advantage in nonvolatility of data, in respect to both (1) power loss and (2) interference due to strong environmental signals such as lightning. This advantage results from the use of static magnetic biasing fields closely adjacent to the storage surface. The problem of power loss in CCD (and other semiconductor memories) may be mitigated by the use of small batteries, and the problem of interference may be solved by careful shielding. The use of "holding" batteries is common in current semiconductor memories, but it is not yet fully accepted as a solution. Shielding as a protection against lightning strikes also has proved generally satisfactory for low-power digital circuits. Power loss remains an inadequately studies problem.

Based on nonvolatility, MBS would appear to be the medium of choice at this time. The issue, however, may be decided on economic grounds. While the primary issue for the air transport application is not cost but reliability, the two are closely connected, since high-volume production has a strong effect on intrinsic device reliability.

With regard to economic factors, CCD has a strong current advantage over MBS, derived in part from the strength of the existing semiconductor industrial technology base. CCDs are themselves facing strong competition from ordinary random-access memory (RAM) technologies. Apparently the somewhat higher intrinsic cost of RAMs (due to higher fabrication and testing complexity) may be outweighed by their presently much higher production volumes and by the system advantages of their lower latency times.

In summary, a reliable, low-cost, block-oriented mass store appears feasible at this time. If the magnetic bubble technology proves not to be economically viable, then special effort should be made to assure the capability of semiconductor memories to withstand transient interference and to have data maintained during power-off periods of several days.

The problem of data-volatility also pertains in the distributed RAMs used for program execution. The reason for concern is that a massive power failure or noise impulse may erase critical programs or data and require a time-consuming reloading of programs or recomputation of critical state data.

The most critical data, in order, are:

- The local executive program
- Any copy of the global executive program
- Flight-critical state information for high-iteration-rate control programs
- Flight-critical state information for low-rate control programs
- Non-flight-critical programs.

The following several candidate functional types of random-access memory are considered. It is generally feasible to mix memory types within a single memory unit.

The most nonvolatile memory is a read-only memory (ROM). Fortunately, a ROM is entirely feasible for the local executive because the same program appears in each memory, and it should not be subject to change. The use of a ROM has the important benefit that it would permit system initialization without the use of external memory units.

The use of ROM for the global executive is more controversial, because not every memory needs a copy of the global executive and because the program may be more subject to change. The nonvolatility of a ROM may be attractive enough for the purpose of rapid recovery after massive transient errors to justify some extra copies of the global executive. It should not be necessary to have a copy in each memory, merely enough to cover the expected needs for spare copies. Furthermore, it may be possible to partition the global executive program into ROM and RAM portions to achieve some economy in the redundant copies; every memory would then have a copy of the ROM portion, but only as many RAM-portion copies would be carried as are expected to be needed.

ROM technology is not appropriate for Items (c), (d) and (e). For critical-state information, it would be attractive to be able to use one of the various forms of writable nonvolatile semiconductor memory, a function usually referred to as programmable read-only memory (PROM). The so-called MNOS technology, for example, has the desirable characteristic that information is retained without applied power. It has several disadvantages, such as deterioration with extensive use and slow writing speed, which seem to rule it out for the present application. Several claims have been made recently in various trade journals about improved PROM technologies for RAMs and CCDs. Such developments deserve continued attention.

In summary, the use of ROM for a portion of the SIFT working memories should be assumed. The use of PROM technology would be beneficial if some reliable form appears that is compatible in timing with the primary RAM devices.

4. Special Logical Functions

In this section we discuss the possible need for special logical functions in memory units other than for fault tolerance. We consider both the distributed RAM memories and the block-structured mass memory.

In SIFT, each processor handles a mix of tasks, some of which are of high criticality while others may have no strong reliability requirements. It is expected that the high-criticality programs will be subjected to a variety of verification procedures (including formal proof) to ensure their correctness. The tasks of low criticality may not be verified to the same high degree, and thus it is necessary to ensure that these latter tasks cannot adversely effect the correct operation of the high-criticality task through the presence of programming errors. A powerful mechanism to guarantee the separation of tasks is the use of "bounds" checking on memory references. This mechanism allows a task program to write only into the area of memory that is allocated to it.

Various known address-control mechanisms should be incorporated in the processor. It may also be cost-effective to include some redundancy within the memory. For example, a short data field may be attached to each data word that can carry some identification to aid in program protection. Such identification could be either a unique program label, or perhaps, a label indicating the level of certification reached by a given program.

Such appended data fields are known as tags. The extensive use of tags has been advocated by several authors (Feustal), e.g., for security enhancement and for indicating the "type" of a datum (integer, alphanumeric, etc.). It is employed in at least one line of commercial computers. Decisions about tagging properly belong to processor design. For the SIFT RAMs, the concept simply implies additional word length of from three to ten bits. Logical operations on the tags would be accomplished within the processors.

The second class of memory function is the block-oriented bulk memory. This memory is not intended for direct program execution, so it need not be associated with a regular processor. Nevertheless, in order to receive data for recording, it must actively request it. Some form of processor is therefore needed to provide this function, as well as such functions as block address interpretation and output. Only a fraction of the logical capability of a standard SIFT processor would be needed, but using one is probably the most cost-effective approach.

5. Fault Tolerance

The basic SIFT concept assumes the use of modules of standard design for both memories and processors. The primary mechanism for fault tolerance is reconfiguration over modules, but the use of some fault-tolerance mechanisms within modules is not excluded, and may, in fact, be cost effective. This section discusses the use of fault tolerance within SIFT memories. The major emphasis is on the primary distributed memories.

The five relevant issues in memory fault tolerance are:

- Diagnosis
- Error detection
- Error correction
- Reconfiguration
- The "unflexed fault tolerance circuit" problem.

These issues will be discussed in order.

a. Diagnosis

The need has been established for in-flight diagnosis in addition to preflight diagnosis for the Air Transport SIFT. The architectural issues are (1) should diagnostic data have a special memory port, or should it flow via the normal data part, and (2) are any special logical capabilities needed within the memory devices or subsystems to aid in diagnosis?

With regard to the first of these, it is clear that additional ports would introduce sources of error whose control might be very expensive in terms of added system fault-tolerance mechanisms and increased complexity of reliability analysis. All efforts should be made to employ the data paths and processor control functions used for actual computation.

The second issue is difficult to address in the absence of particular memory designs. In general it is desirable to avoid special mechanisms, both to reduce the number of fault sources, and to avoid special, low-volume production runs that might be required for special features, but two functions may justify some added equipment: (1) marginal testing, and (2) partitioning of memory to enhance fault location.

We deem marginal testing to be potentially a very valuable facility, especially because it may help to uncover incipient faults, and thus give time for reconfiguration before computing errors occur. The actual benefit for contemporary memory systems needs to be ascertained prior to the statement of engineering specifications. One item of concern is that the control of such marginal states needs to be protected. One one hand, the use of program control introduces new hardware and software sources of error (in order to limit the damage due to such errors, marginal checking should be controlled independently in each processor). On the other hand, it would be an unreasonable burden on the flight crew to have the control completely manual. It may be acceptable to use program control together with a crew-visible indicator to indicate the application of a marginal state. This would tend to protect against a stuck-on marginal state.*

The use of internal logic to enhance fault location could be beneficial if it were desired to use internal reconfiguration for fault tolerance (e.g., by modifying memory address-mapping). Its benefit would be to accelerate diagnosis by isolating sections of memory. RAMs have very uniform structures, which are amenable to systematic testing. It is therefore not clear that the amount of the acceleration of diagnosis would justify the cost and fault-hazard of the added logic. Some further investigation of this point would be justified. In any event, since memory diagnosis is not feasible for recovery of high-criticality faults, its acceleration is a relatively minor issue.

*The trade-offs discussed here also appear in the more general issue of run-time system diagnostics, since any diagnostic mode can introduce some sources of vulnerability to mission-directed computation.

b. Error Detection

Error detection in SIFT memories may have several benefits. For example,

- As an adjunct to error correction, it can give warning of an incipient memory-unit failure.
- It can strengthen the decision of a programmed voting check; that is, if one version of an input datum disagrees with the one (or more) other versions, a memory-error indicator can confirm the identification of the faulty data source. If there is only one other version (i.e., if the redundancy is "dual"), then the faulty source may be identified without further diagnosis. This would tend to justify increased use of dual-mode redundancy.

Many effective techniques are known for error detection in memories. Several important examples are: (1) generalized parity-check codes for words of data, (2) special codes for numerical data, (3) arithmetic sumchecks for blocks of data, and (4) address tags (for verifying address-selection logic in RAMs).

Parity-check codes are very cost-effective for RAMs, and are widely used. The use of up to ten percent additional number of bits is probably justified.

The use of separate codes for numerical data is not justified. They are inefficient for memories, and ineffective for LSI-realized processors, especially in a voting-and-whole processor-reconfiguration scheme such as SIFT.

The use of arithmetic sum-checks for data blocks appears attractive for a serial mass-memory, since it would tend to catch shift-control faults (that would cause loss of duplication of characters). The technique has no additional contribution over word-parity checks for RAMs.

The use of redundant address tags on words is attractive, since certain faults in some RAM word-selection logic-circuits tend to cause selection of single incorrect words (other faults may cause selection or partial selection of several words). Single-word selection errors would not be detected by parity checks. The cost of detection would depend on the precision of the redundant address information. This could range from one bit to the full address. The value of this technique should be estimated in the context of particular memory system designs. For example, the word selection logic may be such that most failures give either non-selection or multiple-word selection.

c. Error Correction

The benefit of error correction for the data of SIFT memories is that it may be a very cost-effective way to prolong the useful life of by-far the largest portion of SIFT hardware. It appears to be unsurpassed in cost-effectiveness protection against one or perhaps two faults per memory unit. Furthermore, it is usually inexpensive to obtain error detection of one unit more than the amount of error correction. Thus, given the programmed voting check of SIFT, the occurrence of two errors in a single-error-correction, double-error-detection memory unit would still allow indication of which version of a dual-redundant computation is correct.

While the use of error detection alone is potentially very effective in SIFT, as discussed previously, the added cost of single-error-correction with double-error-detection appears to be highly justified.

The degree of redundancy required for a block-access mass memory will depend on technological factors that are presently unknown. Considering the relatively low criticality of the data, dual redundancy may be satisfactory. Since the memory will probably have a module realization, it may be that dual redundancy might be effectively applied over storage blocks within a memory, provided that the driving circuitry can be protected.

d. Reconfiguration

The major value of memory reconfiguration is to deal with multiple faults, since error-correction schemes using coding are almost always more cost-effective for one or two faults. Reconfiguration may be applied at the level of a block of words, a bit-plane or a memory device.

The least expensive form is block reconfiguration. In the current SIFT design, software memory mapping tables are employed to interpret addresses for all interprocessor communication. These tables greatly facilitate program relocation among processor-memory units. It is a trivial step to assign values to the mapping tables so as to bypass any contiguous block of words in a memory that has been determined to contain faults. The major cost would seem to be the size of the program (and the cost of its verification) needed to analyze the fault pattern and to define the boundaries of the forbidden region.

Block reconfiguration is effective for faults in memory cells and for some faults in word selection circuits, but it is not effective for faults that affect an entire bit circuit. Such faults are well covered by error-correcting codes, for one or two bit circuits per memory. If a larger number of bit-circuit faults must be tolerated, some form of switching of logical bit-planes may be effective. Such switching is very expensive and fault-prone. Considering the numerous other fault tolerant mechanisms available in SIFT, we deem it not to be a cost effective measure.

A device-level reconfiguration scheme has been described [Ref. 3] that is very cost-effective for large numbers of faults. In order to employ this scheme some modifications to memory chip design are needed, together with a rather powerful diagnosis and reconfiguration program. The initial cost of the scheme appears to be prohibitive for the present application. Its greatest value is for very long unattended life.

e. The "Unflexed Fault Tolerance Circuit" Problem

The general "unflexed problem" is the problem of determining that a functional element is operable prior to its use in a real-time computation. This problem is especially serious for fault-tolerance mechanisms. In a straightforward design, some failures of such a mechanism will be observed only when the fault condition it is designed to treat occurs. An apparent dilemma exists in that if a fault condition is artificially simulated so as to test the mechanism, some action (such as program reconfiguration) may be initiated that could harmfully degrade performance; therefore, to avoid such degradation, the consequent activity must be inhibited. Such inhibition, of course, may also be a source of trouble, and a complete test must assure that the inhibition itself can be terminated.

This problem applies to both hardware and software mechanisms. In the case of memories, the controlled flexing of error detection and correction circuits is clearly desirable. One attractive way to achieve this would be to record permanently a set of erroneously encoded words in an ROM section of each memory. Such words could be read by a diagnostic program which would be designed to interpret correctly the outputs of the error-detection or correction circuits. The problem of correctly inhibiting undesired reconfiguration would be passed upward to the executive program.

This scheme avoids the need for special circuitry to defeat the normal data encoding circuits. Such defeating would be needed in order to simulate a defective memory.

6. Performance Specifications

In this section we summarize the performance required for SIFT memories. At this stage of SIFT development, the various requirements have different degrees of certainty. The discussion references the sources of the requirements and indicates the issues that must be examined in order to achieve more precise values.

a. Distributed-Processor Memories

The following requirements pertain to the random-access memories used locally to each SIFT processor:

- Word length

  Data field: 24 bits [Ref. 3]

  Tag field: 4 to 8 bits (length to be determined by analysis of higher-level mechanisms for correction of faults in memory addressing)

  Error-detection/correction field: 9 to 12 bits (assume 30 bits for data and tag) (length to be determined by the trade-off in cost and reliability between memory and coding-decoding logic)

- Maximum capacity: [64K-128K] words [Ref. 3]

  Actual values will vary with the application, and may be much less than the maximum. The expected value required to cover the entire set of air transport applications is [24K]. The maximum amount assumed will determine the address size required for processor design. The value of maximum-capacity stated is consistent with trends in modern minicomputer technology and architecture. It is very conservative with respect to the full set of computations surveyed in Reference 3. It is conceivable that some new requirements may develop that greatly increase required system memory-capacity, e.g., elaborate graphic displays. The SIFT architecture can be expanded (by at least a factor of three, and perhaps higher) by the addition of new processor-memory pairs, so as to meet a greatly increased requirement.

- Access modes: (1) whole word, read/write, random-access (access to subfields at the memory interface is an unnecessary feature and would greatly complicate error-detection/correction); (2) whole word, read-only, random-access (data are preset at manufacture). Assembly should allow easy change of ROM portion by maintenance personnel. Total amount of the ROM portion is expected to be less than 20% of the distributed-processor memory.

- Special features

  (1) Single error correction with double error detection, using encoding and decoding logic at the (word) data interface.

  (2) Possible use of programmed marginal checking of memory circuits.

- Interfaces

  (1) High bandwidth interface to the local processor.

  (2) The interface to the bus system must incorporate a means to protect the memory from continuous accesses by a faulty bus. Section VI-B describes such an interface which is identical to the means used by a bus to protect itself against repeated accesses by a faulty processor.

b. Block-Access Memory

Requirements on the block-access memory are less certain at this time than those on the random-access memories, because the retrieval functions are less critical. Also, some relevant architectural issues remain to be settled, such as tolerance to transient interference.

The key characteristics are data rate, delay in accessing the beginning of a data block, and capacity. The capacity of the memory system will be determined by numerous critical and noncritical data functions. A range of 10^6 to 10^8 bits appears likely. The data rate and access delay will depend upon the critical functions. These appear to be of two classes, i.e., temporary storage of input and output data, and storage of duplicate copies of critical programs. A maximum recovery time of 12 milliseconds is assumed. Separate block memories may be needed for the two classes. The following estimates apply:

- Data rate: for temporary I/O: 10^6 bytes/sec
  for program access: 0.5×10^6 bytes/sec.
  (based on transfer of a block of 4K word, 30 bit/word, in 50 ms).
- Access delay for a random block: less than 1 ms.
- Special features:

  (1) fault tolerance probably in the form of dual redundancy, with independent control of storage and retrieval. Possible use of longitudinal error-detecting codes.

D. Processors

The essential features that are required in a SIFT processor for use in an advanced commercial transport are described in this section. As the fault-tolerance of the complete system is achieved by software aided by the overall system structure, there is little need for any special fault-tolerance hardware in the processors themselves. Rather, the requirements on the processor are confined to a small number of critical features that enable the software to achieve the fault-tolerance.

The critical issues in the specification of the processors are:

- Interface to the bus system
- Interface to the memory
- Features to assist diagnosis in the processors
- Indirect and indexed addressing
- Interrupt system
- Internal clock
- Memory access bounds checking.

We address these points in the order listed.

The interface to the bus system is the most important issue in the processor specification. It is this interface that enables the SIFT system to achieve fault isolation and damage isolation between individual processor memory modules. The fault isolation is achieved by the fact that the processors cannot write data onto other units, and the damage isolation can be achieved by the use of appropriate circuit design techniques such as the use of high impedance drive circuits. The interface must be capable of transmitting a data-read request to the appropriate bus and of transferring the accessed data back to the requesting processor. A data-read request contains the following elements:

- Bus designator (max 3 bits)
- Process designator (max 4 bits)
- Task designator (max 6 bits)
- Offset within task (max 14 bits).

The above estimates for the number of bits assumes that the following maxima are used in the SIFT specification: 8 busses, 16 processors, 64 tasks and 16K words within a task. It also assumes that a simple binary code is used and that codes for error detection and correction are not employed. The above control data must be communicated to the bus system. The 27 maximum bits of this data can be achieved with the filling of a register of two words length.* This implies the existence of this special register attached to the conventional output unit of the processor. The use of such a special register can enable conventional processors to be used in this application with only the requirement of being able to load a pair of external registers, as will be possible with all commonly available processors. Following action by the bus system on the control data a word will be transferred back to the processor. This word of data will be placed in an external register that can be read by the processor in a conventional manner.

The interface to the memory units would be the conventional interface as typically supplied with a processor and no special requirements arise in the case of the SIFT computer.

The use of special features to assist in the diagnosis of processors and memories would lead to an opportunity for more powerful diagnosis techniques. In the SIFT system we intend to base the diagnosis of equipment on the behavioral characteristics of that equipment and thus we see little need for special built-in test equipment (BITE) or its use in built-in testing (BIT).

The reading of data in one computing module by another demands that indirect addressing be available, because the location of a word in one memory unit is unknown to the reading unit. This implies that a table of base addresses for data segments be kept in each memory and that the access to a word in those memories would use indirect accessing to the required word. The same comments apply also to the use of indexed access

*Assuming a word length of 16 bits as discussed below.

to data. These remarks are also restrictions upon the specification of memory units but are included here because the indirection and indexing into memory is often carried out in the associated processor.

The provision of an interrupt system in the processor is required for some of the fault tolerance techniques that are part of the SIFT design. Included in this are the interrupt for time-out when a task has overrun the time allotted to it and the interrupt on the occurrence of a clock tick. The latter occurs whenever a new task starts. No external interrupts are envisioned for the SIFT system. An internal clock is required to provide for the above-mentioned clock ticks at the start of each task frame.

An important feature of the processor is the need to provide for checking of the address in each task calculation to ensure that data of other tasks is not corrupted. This can be accomplished by the use of array bounds checking as in conventional data processors.

Within the limits of the above, there are few requirements on the processor beyond the speed requirement that the processor be capable of operating at an instruction execution rate of approximately 0.5 MIPS. This is a relatively modest requirement compared to modern minicomputers, but is currently beyond that achievable by present-day LSI microcomputers, although it is expected that by the mid-1980s this speed will be achievable by such units.

E. Power Supply System

We are concerned in this section with reliability aspects of the power supply system for SIFT. Two issues are important, protection against damage propagation and maintaining adequate power supply to SIFT in the event of individual power source failures.

The primary power sources used on modern civilian commercial aircraft vary with each aircraft type. For an example, the DC-10 and the 747 both use a three-phase 400 cycle alternator driven by each jet engine. The DC-10 has three jet engines and three alternators while the 747 has four. In both aircraft the power generated by the alternators is rectified, regulated, and fed to a common 28-volt bus, which in turn supplies current to the 28-volt battery bank.

The alternators are used in a feedback system in which the regulators monitor the input voltage (and current) and feedback a proportionate amount to the alternator field thus controlling the alternator output voltage. Figure VI-13 depicts such a single alternator system. Figure VI-14 shows the power sources on a DC-10 aircraft with three primary and two auxiliary alternators. One of these auxiliary units is driven by a turbine for use when the aircraft is on the ground and the main engines are at idle or when engines are being used to taxi the aircraft, as the varying engine speeds used during taxi operation would cause unacceptable voltage fluctuations. The second auxiliary unit on this aircraft is driven by the flow from an external air scoop when the aircraft is in flight. It is strictly for emergency purposes in event of failure of the main alternators.

The 747 aircraft system is similar but has four main engine driven alternators and two auxiliary gas turbine driven alternators. Each of these aircraft primary power systems is a good example of parallel redundancy. Under emergency operation, any one alternator feeding the battery bank could supply adequate charging current for a long enough period for safe completion of the flight.

Primary power systems of this nature provide a parallel redundant system in which the main alternators are all on line and contributing to the common load. Up to the limit of the storage capacity of the battery bank, the excess power is being stored. The regulators are typically adjusted so that each alternator is contributing an approximately equal amount to the common load. When the battery pack is fully charged, the regulators furnish less field and the alternator is allowed to idle.

Failure of an alternator can be caused either by failure of the driving engine or by a failure of the alternator itself. In either case the removal of that alternator from the system is affected by the series combination of (1) the rectifier diode and (2) by the regulator before manual (switching) intervention is affected. The remaining alternators

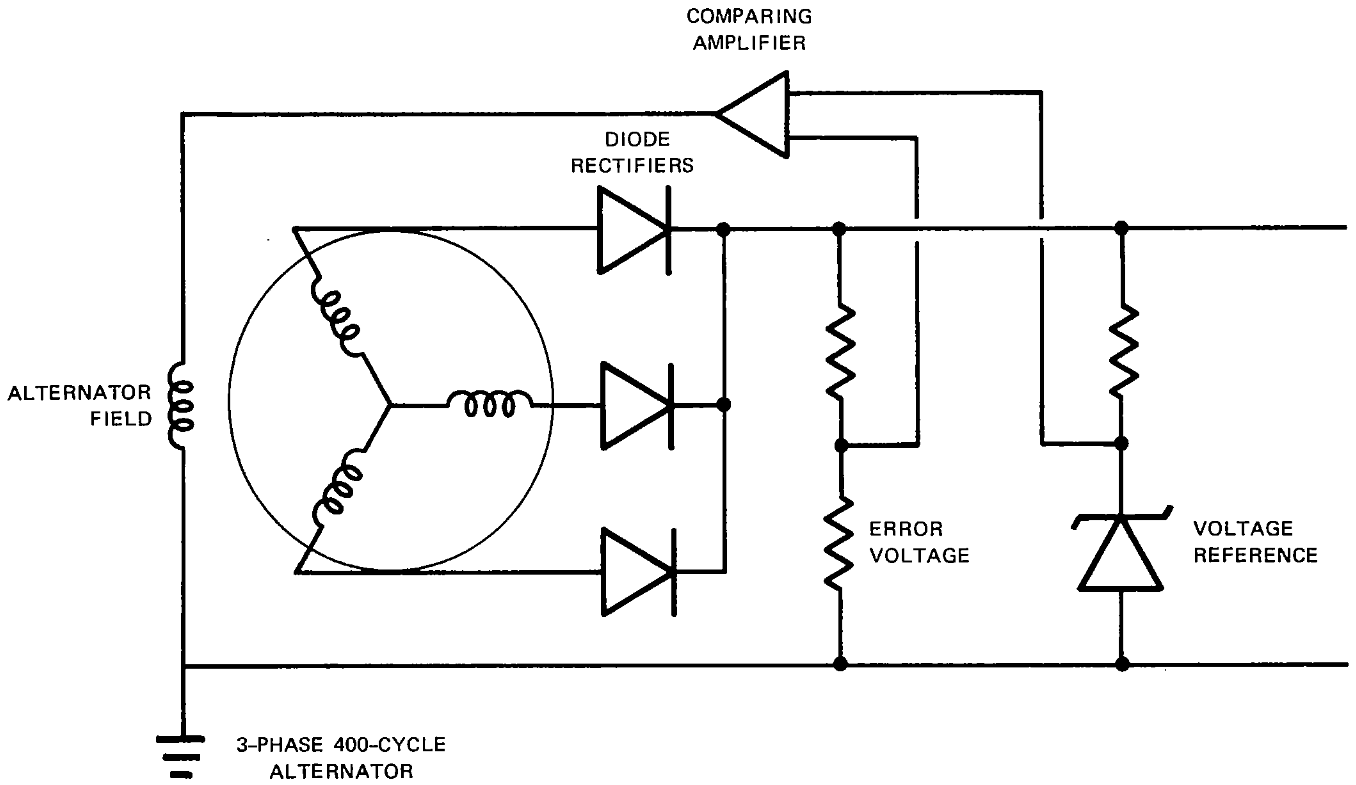


FIGURE VI-13 TYPICAL AIRCRAFT ALTERNATOR/REGULATOR SYSTEM

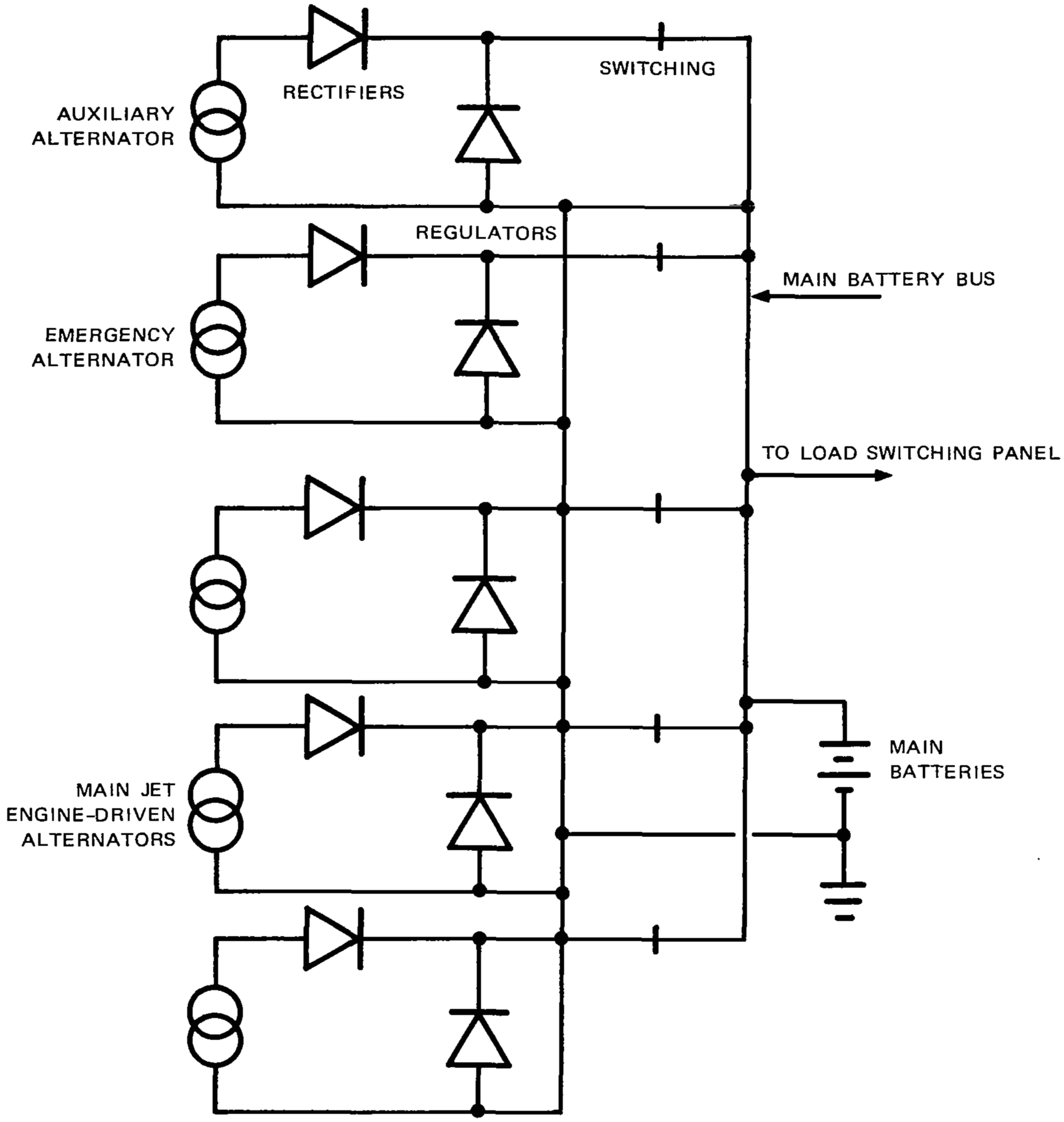


FIGURE VI-14 SCHEMATIC OF DC-10 POWER SYSTEM

automatically pick up the extra load which was being carried by the failed unit in addition to their own previous load.

Failure of any single diode by open circuit condition will not necessarily cause a system failure. The ripple will increase and the rectifier output voltage will drop, resulting in increased field excitation requirement to the alternator. The unit failures are normally detected and corrected by the flight engineer. A shorted diode would be more serious and could cause a failure in the unit either by burn out of the phase winding or damage to the regulator circuit. (A simple fuse could be inserted in each phase leg to ensure an open circuit instead of a short and thus prevent component damage.)

One suggested protection device is an overvoltage protection device, Figure VI-15. This device would be located at the load input. The circuit has the ability of fast action in the event that a regulator failure resulted in an overvoltage from the associated alternator. In that case the device would turn on the SCR thus shooting out the fuse and removing the offending system. The remaining alternators would be unaffected except that they would be required to pick up the additional load.

The availability of the five generating systems plus the main battery bank provide a parallel redundant system which has proved capable of failure-free operation for the time periods normally required in commercial flights. The attention of a flight engineer to monitor the system and make the necessary switching decisions is mandatory in present failure situations.

Consider a fault tolerant computer power system in which the flight engineer does not play such a part. The use of multiple processors would allow us to take advantage of the redundancy of both the power sources and the processors. Figure VI-16 shows such a system applied to the five power sources available on a DC-10 aircraft, for the case of a SIFT system with air processors. Figure VI-17 is a simplification in which the Xs indicate that a connection is made, i.e., that the processor is obtaining power from the associated source.

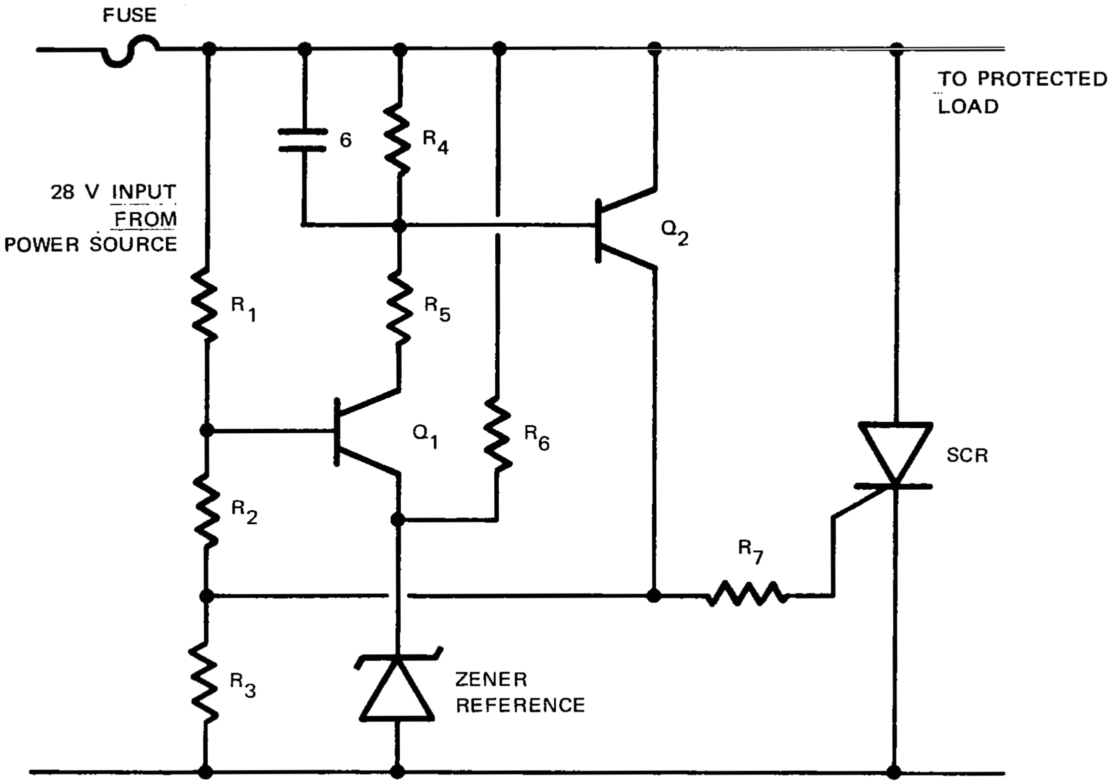


FIGURE VI-15 OVERVOLTAGE PROTECTION CIRCUIT

Assume a failure of main alternator No. 1 in this system. Processors A, C, D, and F would each lose one of their three power sources. Two sources would remain to sustain operation. The case is similar for loss of alternators 2 or 3. In each instance, two power sources remain. In multiple failure cases, four processors are obtaining power from the essential battery bus.

We conclude that a reliable power supply for the SIFT system can be designed by using two techniques:

- Incorporate protective devices in the local power system controllers so as to provide protection against damage propagation.
- Use an interconnection scheme between power sources and processor so that failure of up to two power sources does not effect the SIFT system and failure of three power sources causes failure of at most one processor/memory module.

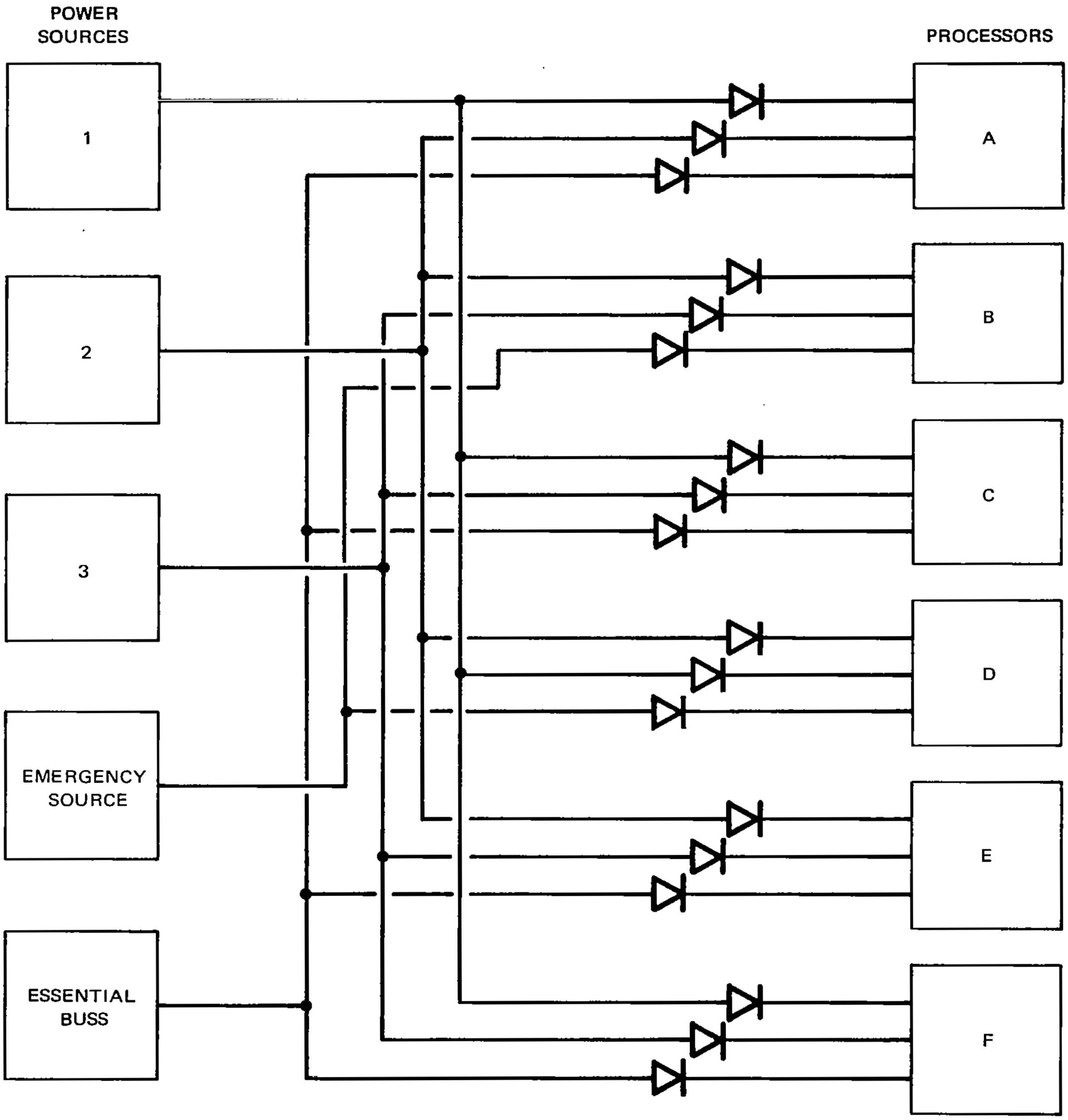


FIGURE VI-16 CONNECTION BETWEEN POWER SOURCES AND PROCESSORS

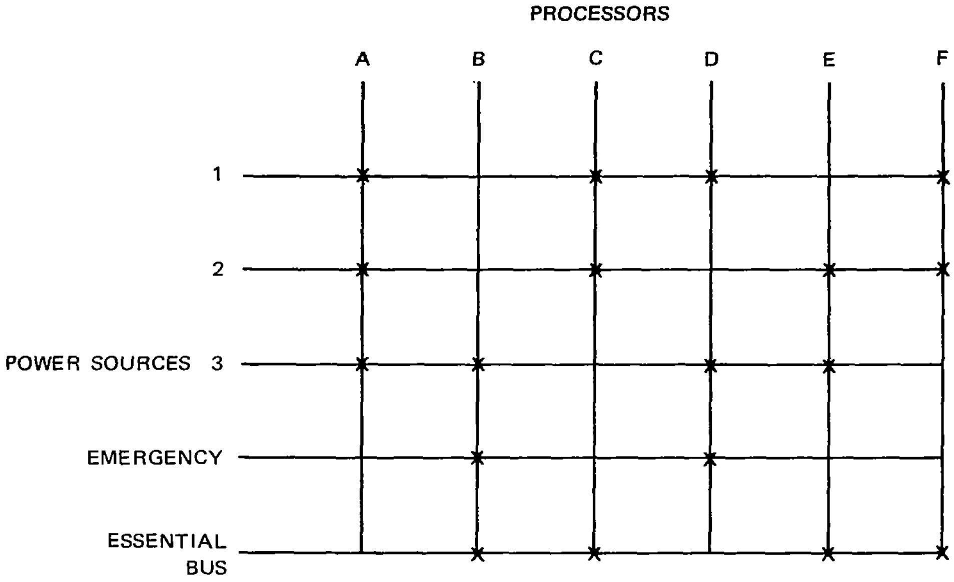


FIGURE VI-17 POWER SOURCES AND PROCESSORS CONNECTION PATTERN

REFERENCES

1. V. E. Benes, "Mathematical Theory of Connecting Networks and Telephone Traffic" (Academic Press, New York, 1965).

2. A. Waksman, "A Permutation Network," Journal Assoc. Comp. Mach., Volume 15, No. 1, pp. 159-163 (1968).

3. J. Goldberg, K. N. Levitt, and J. H. Wensley, "An Organization for a Highly Survivable Memory," IEEE Transactions on Computers, pp. 693-705 (July 1974).

VII RELIABILITY ANALYSES

A. Summary

Several models of SIFT system are examined to determine failure probabilities and failure rates under circumstances where random permanent faults or transient errors interfere with normal operation. The system is modeled by a time-homogeneous Markov process, and analytical techniques are developed for convenient solution of the associated state graphs. Principal parameters of the analysis are the numbers of working processor/memory and bus units remaining at a paeticular time, their respective failure rates and the length of time involved in reconfiguring the system on detection of error. Each model is implemented as a FORTRAN program from which tabulated values of failure probabilities may be calculated for any desired values of relevant model parameters. The most significant conclusions of the study are

- Assuming typical failure rates for electronic components and an acceptable total system failure rate of 10^{-9}/hr (as recommended by the FAA), a SIFT system composed of five or more processors and four or more busses should adequately meet the reliability requirements.
- System survival is limited by the "weaker" collection of the two types of units (processors or busses). That is, if there are too few processors available at a given time, it does not improve system reliability to have a large number of available busses, and conversely.
- For either permanent or randum transient faults, system performance is not significantly degraded by reconfiguration times an order of magnitude greater than the expected value of a few milliseconds.
- Uncorrelated transient errors significantly decrease system reliability only if their rate of occurrence is comparable with that of permanent failure.

B. Motivation

According to an old anecdote, one electrical instrument manufacturer during the mid-1930s used to test his products for reliability by kicking them down a flight of stairs. Apparently the procedure was effective, since his equipment enjoyed the reputation of extreme ruggedness. The story illustrates an early and not very scientific application of destructive testing to reveal weaknesses in design concepts. Destructive testing is still widely used and is particularly effective in the areas of mechanical and structural engineering where failure rates may be accelerated in well understood ways by applying excessive loads, stresses, heat, etc. Another way of obtaining information about failure rates is by life testing many similar units under normal operating conditions. This policy is the one more usually employed when it is unclear how to accelerate component failure in a predictable manner.

For the estimation of the reliability of SIFT, neither of the above methods of testing can be usefully applied to the total system for two reasons. First, it is not clear how to "abuse" the system in such a way that a predictably increased rate of failure would occur. Second, the design-goal failure rate for the total system is so low ($\sim 10^{-9}$/hour) that actual life testing would be prohibitively slow and expensive.

As an alternative approach, one can decompose the SIFT system into parts each of which has known reliability properties or is susceptible to separate reliability analysis.

Then the interaction between these parts can be accurately described, one can make confident predictions about the behavior of the system as a whole. Except for a choice of analytic techniques, this partitioning scheme is the essence of a reliability model, and our conclusions will depend on the accuracy of our knowledge about the behavior of its component parts. What we hope to learn from the reliability model will be outlined in the following section.

C. The Reliability Model

We deal with a physical system (a collection of hardware and programs) and a complicated set of possible "events." For convenience one can distinguish between

- Normal events; e.g., initiation and termination of scheduled programs, changes in flight phase, pilot intervention, etc.
- Abnormal events; e.g., hardware failure or transient errors.

The partitioning of the system could be carried out to any level of detail, but for purposes of the following analysis we distinguish processors (with their associated memories) and individual communication busses as component parts subject to separate reliability analysis. For these system-hardware units good reliability estimates exist, based on total count of active devices and much experience with similar electronic equipment. For example the failure-rate of a typical processor in the SIFT system is estimated with some confidence as about 10^{-4} per hour, while failure of a bus-unit is estimated to be 10^{-5} per hours on the basis of a component-count of approximately 10% that of a processor.

As part of the SIFT system, programs may also be partitioned into subsets for reliability analysis. The main interaction between programs and hardware from a reliability standpoint concerns the duration and criticality of the programs. The role of program criticality is discussed elsewhere in this report, however, it is clear that short, rapidly executed programs have less likelihood of being disturbed by transients or onset of hardware failure. In particular, reliability estimates turn out to be sensitive to the execution time of a program necessary to recovery or reconfiguration of the system.

We intend to model the functioning of the SIFT system by a finite set of distinct "states" with transitions between states occuring in response to particular events. A state of the system can represent any condition that seems important to consider in the reliability analysis. For example a specific state of the model may represent the combined conditions that one processor has failed and that a reconfiguration program is currently being executed on a different processor. Of the events that cause transitions between states of the model (particularly abnormal events), many will occur at random times. We want to analyze the model to determine the probabilities that the system will be found in a designated state at or before or after a particular time. In particular we are interested in the probability that the system will have reached the FAIL state before the mission time T has elapsed.

In addition to the estimate of system-failure probability the above model yields several other useful types of information.

- The probability of reaching conditions (states) that would require special actions to be taken. For example the abandonment of some noncritical tasks or some form of pilot intervention.
- Differential failure rates. That is, the failure rate of the system after it has been in operation in say n hours. This is important in establishing a policy for equipment maintenance and replacement.
- Mean time between failures (MTBF) for the system or parts of it. This is informative if uses are contemplated where no maintenance is feasible, e.g., an extended mission in space.

D. Analytical Techniques

The most generaı formulation of the reliability model that we use consists of a directed graph whose vertices correspond to states of the system, while edges correspond to possible transitions. In this case each transitıon has an associated rate that may be time dependent and also history dependent. That is, the transition rate of a particular edge might depend upon the manner in which the attached state-vertex was reached. To analyze such a system requires the solution of a system of nonlinear differential equations for which routine analytical methods do not exist. Consequently, with this type of model, recourse would have to be made to numerical integration programs, and to the computation of many particular cases to determine how solutions behave with respect to different parameters of the model.

Fortunately, we will not need such complete generality in dealing with the SIFT reliability models. One simplification is that we will always be able to choose the states of the model in such a way that the probability of occupying a state is independent of how the state was reached. For example, if we designate one state of the model to represent the condition that two processors have failed, it should be immaterial which one failed first (history independence). Another simplification occurs because most of the transitions of our models are in response to "abnormal" events of stochastic nature, like component failure, whose failure rates are constant (time independence). Note that this assumption is approximately true for semiconductor devices but would not be true for say the clutch in an automobile which wears out with continued use.

With both of the foregoing simplifications, our reliability model falls into the category of a finite-state, continuous-time, simple Markov process. For such Markov processes, there are elegant and powerful methods of obtaining complete closed-form solutions. When transition rates are time dependent (but not history dependent), as will be the case if a transition depends on say the fixed execution time of a program, then the model corresponds to a semi-Markov process.

Here a variety of solution techniques have been suggested in the literature. For our purpose it seems very desirable to retain the simplicity of the pure Markov model and to have a uniform analytic procedure that can be applied in all cases. For this reason, the method we favor is to approximate the behavior of time-dependent state-transitions with a collection of redundant states having constant transition-rates, and whose collective behavior simulates the desired time dependence. This artifice has been called "the method of stages" and a discussion of its use may be found in Reference 1. For those unfamiliar with Markov processes a short description with applications to the modeling problem is presented in the appendix to this report. The appendix also illustrates the method of stages as applied to the modeling of a fixed-duration event and a transient event.

We wish to find the most convenient and potentially useful way of obtaining the desired insight and the actual numerical results from a given SIFT model. The Markov-process description of the model yields a system of linear, first-order, constant-coefficient differential equations that completely determine all of the state-transition probabilities as a function of time. In the usual formulation, these relations are called the Chapman-Kolmogorov differential equations. Since we are more interested in the possibility of occupation of particular states of the model, we will consider a slightly different set of differential equations that relate occupancy probabilities rather than transition probabilities. The system of equations is

$$P'_q(t) = \sum_{i=1} a_i P_i(t) - \sum_{j=1} b_j P_q(t) \tag{1}$$

where Pq(t) is the probability of occupying state q at time t, and a_i, b_j are the (constant) transition-rates associated with edges entering state q from state i and leaving stage q to state j, respectively (for a derivation of these relations see the appendix).

As is well known, the solution of a system of equations like the above set can be carried out by a purely mechanical process. Given an initial vector of occupation probabilities, say $P_1 = 1$ and $P_q = 0$ for all other states, then each $P_q(t)$ can be expressed as the sum

$$P_q(t) = \sum_{1}^{n} A_i e^{\lambda_i t} \tag{2}$$

where n is the number of states in the model, A_i are numerical coefficients that depend only on the initial probabilities, $P_1(0) \ldots P_n(0)$, and λ_i are the eigenvalues of the matrix of coefficients of P_i associated with the system of equations given by (1). Trivial complications occur if all of the eigenvalues are not distinct, in which case terms of the form

$$\sum_{j=1}^{k-1} t^j A_j e^{\lambda k^t} ,$$

where k is the multiplicity of an eigenvalue, need to be included.

In principle one can rely on standard routines for finding eigen values and eigen vectors of a finite matrix to solve any particular reliability model. One important problem, however, is that expressions like that of equation (2) can lose almost all of their computational accuracy when very low failure-rates are involved. To illustrate this point consider the very over-simplified model of a SIFT system shown in Figure VII-1.

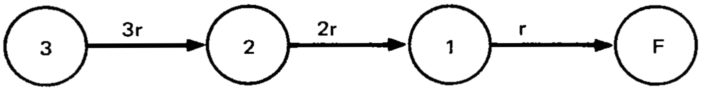


FIGURE VII-1 A SIMPLIFIED SIFT MODEL

Here we depict a system that begins in state 3 with three active processors whose individual failure-rates are r (per hour). This model shows a rate of "decay" 3r, into the situation where two processors survive, followed by a rate of failure 2r into a state where one processor is left, followed by a rate of failure r into a failed state F. Suppose we are interested in the probability of being in state 1, with one processor still surviving. Using standard numeric techniques as described above we could obtain the following expression for this probability,

$$P_1(t) = 3e^{-rt} - 6e^{-2rt} + 3e^{-3rt} \qquad (3)$$

where the above coefficients might not actually be exact due to round-off errors. Now if we attempt to evaluate this expression for small rt the answer may be almost meaningless unless high-precisions arithmetic is employed throughout all of the calculations. The reason is that each of the above exponential terms has a value nearly equal to unity. The value of $P_1(t)$ for small rt is actually close to $3(rt)^2$, but this fact might not be evident from an imprecise evaluation of (3), particularly for very small values of rt where much of the significant behavior of the SIFT system occurs.

A second problem involved in using automatic numeric solution techniques of the eigenvalue-eigenvector type is that the representation of a solution in a form involving computed numeric coefficients conceals the way that different parameters of the model interact. Thus, it is more difficult to determine how particular parameters are affecting the behavior of the model than it would be if explicit parameterized expressions for the state occupation probabilities were available.

For these (and other) reasons we prefer to use Laplace Transform techniques in dealing with the system of equations (1). Using this approach one can reduce the solution of the system to algebraic manipulation of polynominals in a transform variable s, that is related to the probabilities $P_i(t)$ by the transformation

$$P_i^*(s) = \int_0^\infty P_i(t)e^{-st}dt \tag{4}$$

When the transformation is applied to (1), one obtains a set of linear equations of the form

$$sP_q^*(s) = \sum_{i=1}^{n} a_i P_i^*(s) - \sum_{j=1}^{n} b_j P_q^*(s) \tag{5}$$

or

$$(s + \beta)P_q^*(s) = \sum_{i=1}^{n} a_i P_i^*(s) \tag{6}$$

where β stands for the sum of the rates b_i of edges emanating from state q. The solutions of this system are simple ratios of polynomials in s which can be inverted by standard methods to obtain the corresponding time-dependent solutions for each of the state-occupancy probabilities $P_i(t)$.

Using this method on the diagram of Figure VII-1 one obtains

$$P^*_1(s) = 6r^2/(s+3r)(s+2r)(s+r) \tag{7}$$

from which one can immediately deduce the exact solution already given in (3). However, the expression (7) can also be seen by inspection to have approximate value $6r^2/s^3$ for sufficiently large s. So, from a knowledge that t^n transforms to $n!/s^{n+1}$ one can also conclude by inspection that $P_1(t)$ behaves like $3(rt)^2$ as $t \to 0$. Several simple but useful relations of this type are mentioned in the appendix.

E. Models and Programs

In this section we will discuss four reliability models of the SIFT system and describe computational programs based on these models. A primary assumption is that the principal failure modes of a model correspond to transient or permanent faults occurring either in a processor/memory unit or in a communication bus unit. The modeling so far has not considered a further subdivision of the system components, since strategies for making use of "partially failed" devices have not been considered in any detail. We also assume that there is no difference in criticality to the system in the failure of any particular processor or bus. This assumption may not be strictly true since a processor that is executing an executive program may be more critical to survival than one employed in some inessential task. By choosing to ignore the latter possibility we obtain results that are on the pessimistic or "safe" side of the actual reliability situation.

The main parameters of the models considered here are the number of processor/memory units and number of buses available at the start of a mission, their respective permanent-failure rates, and mission time. Two other parameters of importance are the expected time to reconfigure the system after detection of a fault and a measure of the expected degree of success in diagnosing a transient fault and restoring the system to operation without loss of equipment. For each model we assume that a failed state has been reached if less than two processors or buses survive. This would correspond to a situation in which voting among remaining hardware units would no longer be meaningful.

The models listed below were selected with the intent of discovering the effects of differing failure modes separately and in combination.

Model I: Permanent faults, instantaneous reconfiguration.

Model II: Permanent faults, finite reconfiguration time.

Model III: Transient faults alone.

Model IV: Transient and permanent faults with finite reconfiguration time.

For each of the above models the corresponding Markov-process state-graph was analyzed using Laplace Transofmr methods to obtain closed form solutions (or approximate solutions) for the probability of reaching the failed state as a function of mission time. These analytical expressions are used by several small interactive FORTRAN programs to print tables of failure probabilities for any desired values of the model parameters. A sample output giving the failure probabilities for a SIFT system composed from 10 or fewer processors and 7 or fewer buses under Model I assumptions is shown in Table VII-1. Here the sample output is parameterized with failure rates of 10^{-4}/hr and 10^{-5}/hr for processor and bus failure rates respectively, while the mission time was taken as 10 hours. These figures are interactively supplied by the program user.

Consistency between different models may be checked by running their corresponding programs with parameter values that cause one case to degenerate into another. For example Model II, for reconfiguration time = 0, gives the same results as Model I, and Model IV with 100% probability of transient recovery gives the same reults as Model II.

For convenience, we have also modified the Model II and Model IV programs to provide differential probabilities of failure. That is, the probability of failure during the next one-hour period of a system that started with n processors and m buses, as a function of mission time. A table of failure rates for the particular reconfiguration time of one second as computed by Model IIA is shown in Table VII-2. This information is important if one must make policy decisions concerning when replacement or maintenance of defective units should occur. For example, Table VII-2 shows that with the assumed initial configuration several 10-hour missions could safely be undertaken without between-flight maintenance. According to the table, after 60 hours of operation, the expected hourly failure rate is still only 5 x 10^{-10}, which is within the nominal acceptance value of 10^{-9}.

Table VII-1

SAMPLE OUTPUT

```
MODEL I. *

PROCESSOR FAILURE RATE:   1E-4

BUS UNITS FAILURE RATE:   1E-5

MISSION TIME, IN HOURS:   10

FAILURE PROBABILITY TABLE:

PRO/BUS   2          3          4          5          6          7

 2      2.20E-03   2.00E-03   2.00E-03   2.00E-03   2.00E-03   2.00E-03

 3      2.03E-04   3.02E-06   3.00E-06   3.00E-06   3.00E-06   3.00E-06

 4      2.00E-04   3.40E-08   4.00E-09   3.99E-09   3.99E-09   3.99E-09

 5      2.00E-04   3.00E-08   8.99E-12   4.99E-12   4.99E-12   4.99E-12

 6      2.00E-04   3.00E-08   4.01E-12   6.48E-15   5.98E-15   5.98E-15

 7      2.00E-04   3.00E-08   4.00E-12   5.07E-16   7.03E-18   6.97E-18

 8      2.00E-04   3.00E-08   4.00E-12   5.00E-16   6.79E-20   7.97E-21

 9      2.00E-04   3.00E-08   4.00E-12   5.00E-16   6.00E-20   1.60E-23

10      2.00E-04   3.00E-08   4.00E-12   5.00E-16   6.00E-20   7.01E-24
```

* RECONFIGURATION TIME ASSUMED TO BE NEGLIGIBLE.

Table VII-2

FAILURE RATES FOR RECONFIGURATION TIME OF ONE SECOND AS COMPUTED BY MODEL IIA

MODEL II-A. *

PROCESSOR FAILURE RATE: 1E-4

BUS UNITS FAILURE RATE: 1E-5

NO. PROCESSORS: 5

NO. BUSSES: 4

RECONFIG. TIME, IN SEC: 1

FAILURE PROBABILITY TABLE:

TIME: HR.	FAIL. PROB.	HOURLY RATE.
1	5.59E-11	5.59E-11
2	1.12E-10	5.59E-11
4	2.24E-10	5.61E-11
6	3.37E-10	5.65E-11
8	4.51E-10	5.73E-11
10	5.67E-10	5.86E-11
12	6.87E-10	6.04E-11
16	9.42E-10	6.60E-11
20	1.23E-09	7.50E-11
40	3.75E-09	1.96E-10
60	1.06E-08	5.10E-10
80	2.65E-08	1.11E-09
100	5.81E-08	2.08E-09
200	7.99E-07	1.53E-08
400	1.17E-05	1.13E-07

• DATA EXTRACTED FROM MODEL II.

F. Computational Results and Interpretations

The reliability models referred to above yield quantitative data on the separate and combined effects of finite reconfiguration time and imperfect transient recovery on a SIFT system that also suffers spontaneous permanent faults. Model I depicts the most idealized (and most optimistic) situation in which only permanent faults are considered. Perfect reconfiguration strategies and transient recovery schemes cannot improve the reliability estimates of Model I unless "salvage" of working parts of the system at a level smaller than a processor/memory or bus-unit is feasible. We have not considered the latter possibility in this analysis.

A state-diagram for Model I is shown in Figure VII-2. Starting with n processors and m busses in the initial state, we assume constant failure rates of p and q for each of these devices respectively. It is assumed that a failed state will have been reached only if less than two processors or busses survive after a mission time of duration T.

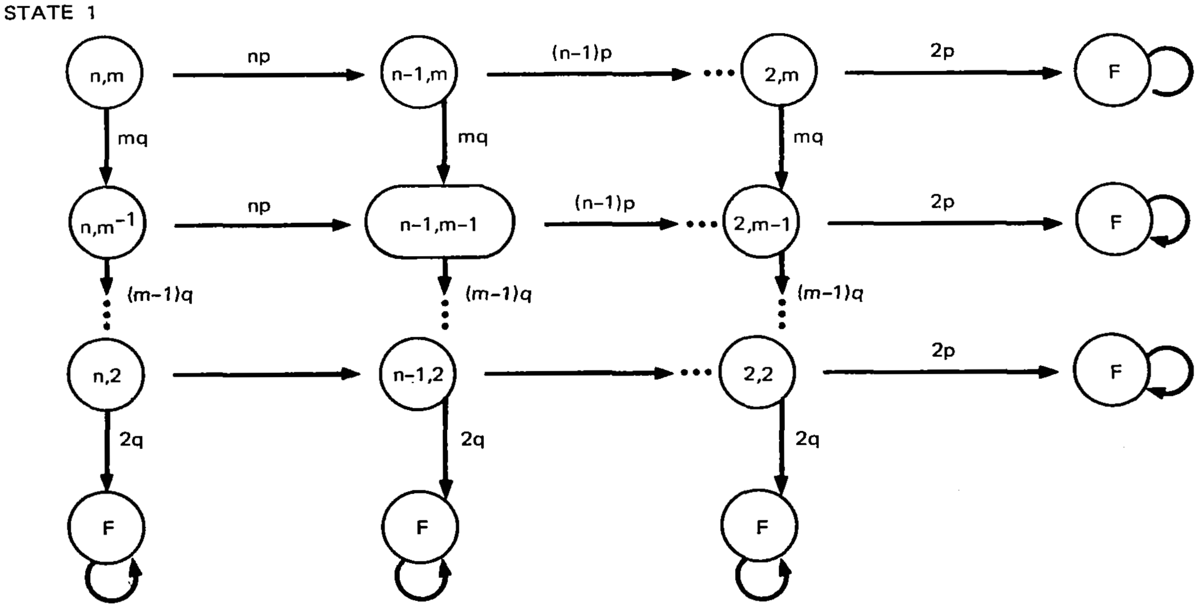


FIGURE VII-2 MODEL I STATE-DIAGRAM

In the rectangular array of states representing the surviving numbers of processors and busses respectively, note that the failure-rates associated with each exit arrow are proportional to the number of remaining units of the same type. The states designated F represent failure.

Observe that the model corresponds to one in which processor and bus failure are considered to be independent and uncorrelated events. Therefore an analysis of failure probability can be carried out without "solving" for each state occupancy probability. However, we retain the generality of the full Markov-process representation because we might wish in some later analysis to consider some special action to be taken in one of the particular states of Figure VII-2. In each of the subsequent models to be discussed, a similar state graph is assumed, except that the transitions between states are complicated by the insertion of intermediate states that represent the effects of reconfiguration delays or transient-fault recovery.

An analysis of Model I yields closed-form expressions for failure probabilities parameterized in terms of processor/bus failure rates and mission time as shown in Table VII-1, where $p = 10^{-4}$, $q = 10^{-5}$ and $T =$ 10 hours. This tabulation shows the intuitively expected result that survival probability for a system having few processors or few busses is not significantly improved by having a large number of the other type of unit available.

To illustrate, referring to Table VII-1, we see that with an initial configuration employing four processors the system failure probabilities are essentially identical if four or more busses are available initially. Similarly if only three busses are initially available then it does no good to have more than five processors. The system reliability can be said to be processor-limited or bus-limited. It therefore makes sense to think of "enough" busses for a given number of processors (and conversely).

The calculated values of Table VII-1 are typical. In this prototype case, a SIFT system composed of four processors and four buses is seen to have a 10-hour mission probability of failure of 4 x 10^{-9}. On a per-hour failure rate basis this would be 4 x 10^{-10}, which is within the FAA acceptable limit of 1 x 10^{-9}/hour. The situation is not improved by adding another bus, which would change the failure rate to 3.99 x 10^{-10}. However, the addition of another processor (making the count 5 and 4) leads to a failure rate of about 10^{-12}, which again is not significantly improved by adding yet more processors.

For this model, the significant features can be summerized by the graph of Figure VII-3, which shows system-failure probability plotted

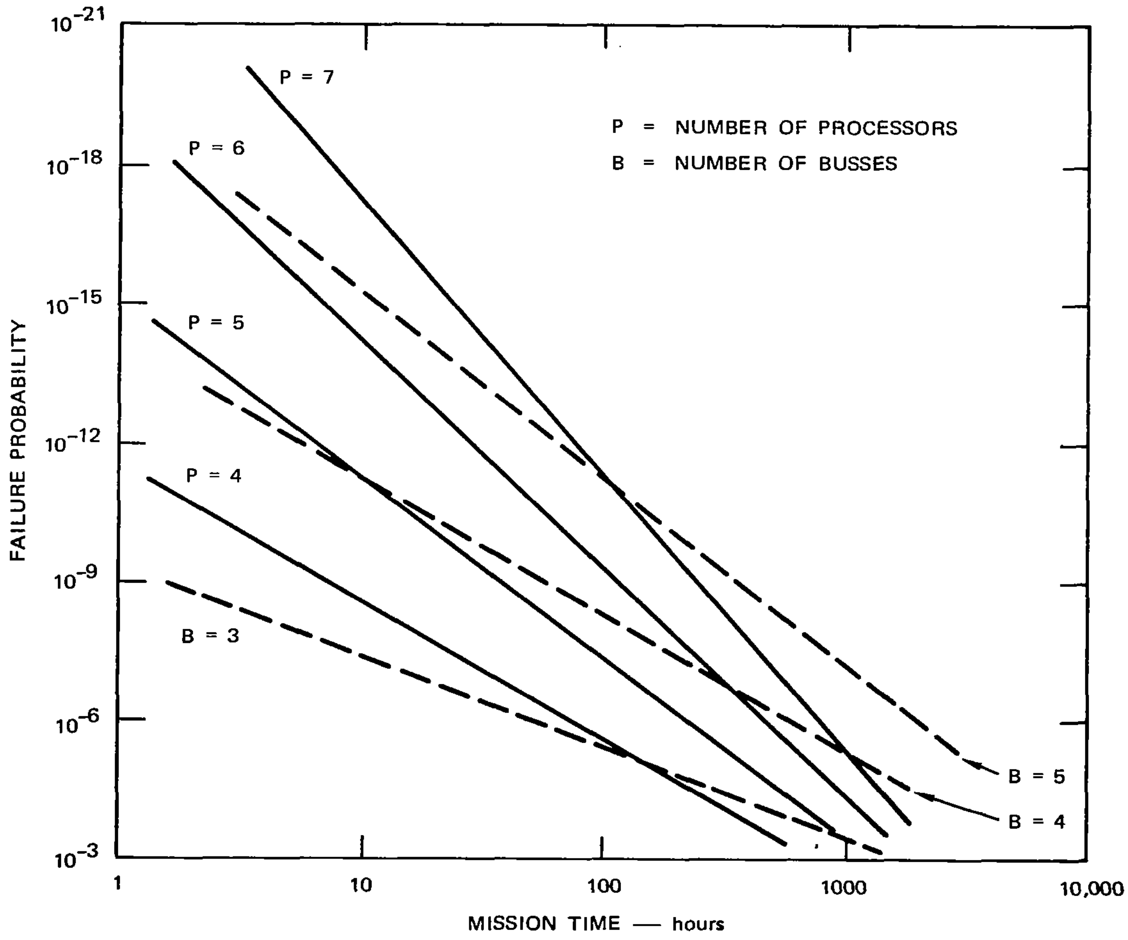


FIGURE VII-3 MODEL I BEHAVIOR

against mission time for various numbers of processors and buses. In particular, the point corresponding to a mission time of 100 hours and a failure probability of 10^{-11} (10^{-9}/hour) is seen to be barely achievable with 6 processors and 4 buses; the need is for nearly 7 and 5 respectively.

In Model II we consider the effect that a finite reconfiguration time has in degrading the performance of Model I. Here it is assumed that upon detecting the presence of an error in a processor or bus some diagnostic and/or reconfiguration programs of fixed durations must run before safe system operations can be resumed. If another processor or bus failure occurs during this short interval of time it could cause serious problems in recovery. For the purpose of Model II we consider the latter event to be fatal. Therefore, Model II presents a rather pessimistic interpretation of failure probabilities due to nearly simultaneous permanent faults. The state-diagram of Figure VII-2 is also appropriate to this case if each transition is reinterpreted to contain a delay-state of fixed duration and a direct entry to the failed state as shown in Figure VII-4

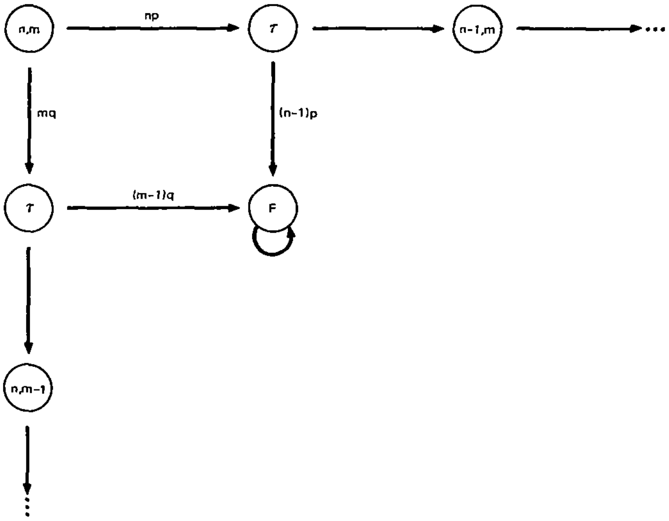


FIGURE VII-4 MODEL II STATE-DIAGRAM

The fixed delay states labeled τ in Figure VII-4 are handled in the manner described in the appendix to this report. Actually, we obtain exact closed-form expressions for the various state-occupancy probabilities just as was done for Model I.

A typical output from the Model II program is shown in Table VII-3. Here again, we have taken processor and bus failure rates of 10^{-4} and 10^{-5} as reasonable values with a mission time of 10 hours. The value of τ, the reconfiguration time, stated as 10 seconds is a gross overestimate of the probable time for such a procedure, which might take at the most perhaps 100 ms. We take the larger value of τ to illustrate the effects of the finite reconfiguration time on the results of Model I.

From an inspection of Table VII-3 it can be seen that in each row and column corresponding to a fixed number of processors (buses), in system failure probabilities first diminish and then increase with larger number buses (processors). The interpretation of this result can be understood by considering our previous assumption that a system failure would occur if any pair of functional units failed within time τ. Obviously, this possibility increases with the number of active processor/bus units involved. However, the distribution of tasks over many processors would allow voting over the results of several units instead of say three-- so that some double failures might in fact be tolerated. This means that the results of Model II may be considered to be pessimistic. Figure VII-5 shows how failure probability varies with extended mission times with reconfiguration time as a parameter. The curves represent an initial configuration of five processors and 4 buses.

The dashed line representing zero reconfiguration time corresponds to data obtained from Model I. The effects of a finite reconfiguration time are most apparent for short mission times where failure probability would have been very low according to the assumptions of Model I.

For very long mission times failure probability is limited by the numbers of available processors and buses and becomes independent of the value of τ. In the range of mission times near 10 hours, there is an increase of failure probability of about one order of magnitude for each order of magnitude increase in τ. It should be noted that for about a 10 hour mission and a reconfiguration time of 10 seconds, the failure probability is about 5×10^{-9} or 5×10^{-10} on a per-hour basis.

Table VII-3

TYPICAL OUTPUT FROM MODEL II PROGRAM

```
MODEL II. *

PROCESSOR FAILURE RATE:   1E-4

BUS UNITS FAILURE RATE:   1E-5

MISSION TIME, IN HOURS:   10

RECONFIG. TIME, IN SEC:   10

FAILURE PROBABILITY TABLE:

PRO/BUS   2           3           4           5           6           7

  2     2.20E-03    2.00E-03    2.00E-03    2.00E-03    2.00E-03    2.00E-03

  3     2.03E-04    3.03E-06    3.00E-06    3.00E-06    3.00E-06    3.00E-06

  4     2.00E-04    3.73E-08    7.36E-09    7.38E-09    7.40E-09    7.44E-09

  5     2.00E-04    3.56E-08    5.59E-09    5.61E-09    5.64E-09    5.67E-09

  6     2.00E-04    3.83E-08    8.36E-09    8.38E-09    8.41E-09    8.44E-09

  7     2.00E-04    4.17E-08    1.17E-08    1.17E-08    1.17E-08    1.18E-08

  8     2.00E-04    4.56E-08    1.56E-08    1.56E-08    1.56E-08    1.57E-08

  9     2.00E-04    5.00E-08    2.00E-08    2.00E-08    2.01E-08    2.01E-08

 10     2.00E-04    5.50E-08    2.50E-08    2.50E-08    2.51E-08    2.51E-08
```

* FINITE RECONFIGURATION TIME.

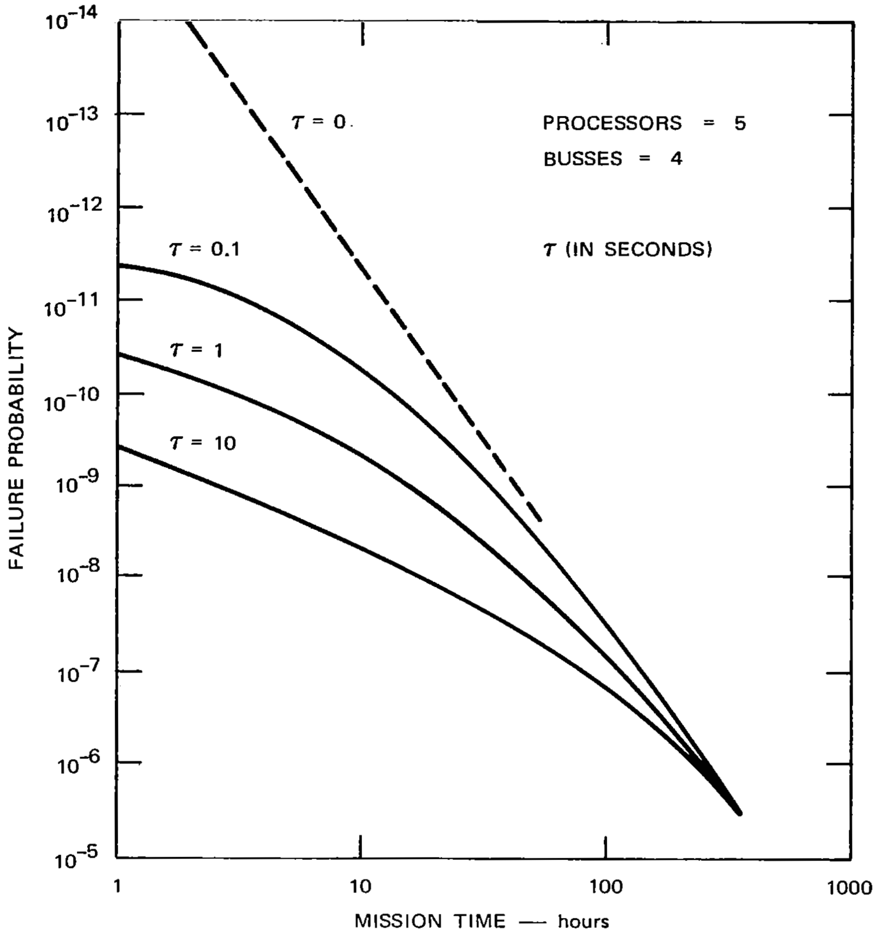


FIGURE VII-5 MODEL II BEHAVIOR

Therefore, 10 seconds would be an acceptable reconfiguration time for this combination of processors and buses. Actually we have estimated that reconfiguration should take no more than a few milliseconds, so that a failure probability about 100 times smaller than the above figure would be expected in this case.

Model III was designed to measure the effects of imperfect recovery from transient errors. As such, it does not provide a realistic model of SIFT since it assumes that no permanent faults occur. The usefulness of Model III was mainly to have an independent computational check on the results of Model IV which includes Model III as the special case where both processor and bus permanent failure rates have value zero.

Model IV is a superposition of Models I, II, and III. It attempts to account for the effects of permanent faults (randomly occurring), finite reconfiguration time and transient recovery. The state transitions associated with one node of the state diagram are shown in Figure VII-6.

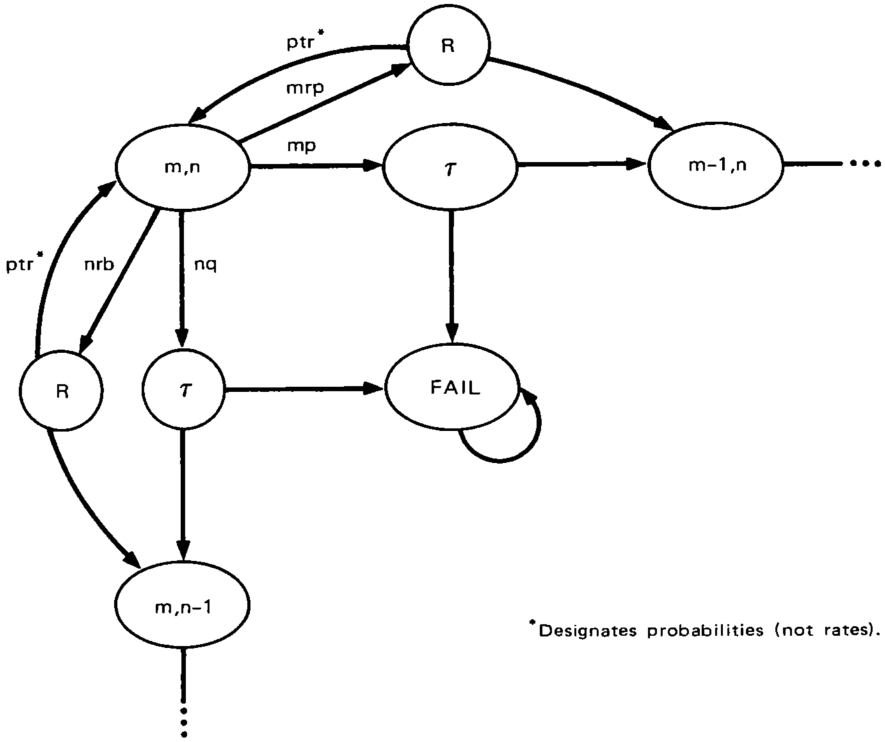


FIGURE VII-6 MODEL IV STATE-DIAGRAM

For the typical node of the state-transition graph of Model IV there are these possibilities:

a. A transient error may occur with average rate rp or rb, causing a transition to one of the two states labeled R in Figure VII-6. Following this occurrence we assume that detection and correction in a return to the original state. Otherwise, we conclude that a permanent fault has occurred and proceed to a state with one less processor or bus in the active system.

b. A permanent fault may occur with average rates p or q (as in Model II) causing a transition to one of the states having one less processor or bus if configuration is successful. If two permanent faults occur within time τ, the FAIL state is entered.

Since Model IV has seven parameters, including two permanent failure rates (for processors and buses respectively), two corresponding transient failure rates reconfiguration time, transient recovery probability, and mission time--it is hard to present a comprehensive picture of how all these parameters interact. The best way to explore this seven-dimensional space (nine dimensions of processor and bus counts are included) is to run the interactive Model IV FORTRAN program using parameter values near the region of interest. We have already focused attention on one set of parameter values that seem to be reasonable or typical for the proposed SIFT system, i.e., $p = 10^{-4}$ $q = 10^{-5}$, $T = 10$ hours, $\tau = 100$ms, $m = 5$ processors, $n = 4$ buses. To see how transient errors can affect failure probabilities we may assume the above values and compute Model IV for various values of transient recovery probability and transient error rates. Results corresponding to one choice of recovery probability (.9) and transient error rate (10^{-5}) are shown in Table VII-4.

A composite graph showing the effect of different assumptions about recovery rate and transient error rate is shown in Figure VII-7. It is observed that the effects of transients are rather closely approximated by simply adding to the processor and bus permanent failure rates a rate equal to ($1\text{-}ptr^*$) times the corresponding transient failure rate.

In Figure VII-7 the curve labeled A represents the situation in which transient recovery is successful 100% of the time i.e., $ptr^* = 1$. Curve B represents a case where transient error rates are assumed to be identical with permanent fault rates and the probability of recovery is zero. Thus, for any recovery probability between 0 and 1 the failure probability of the system lies between the curves A and B--and very close to A for high recovery probabilities. The curve D represents zero probability of recovery from transients occurring at 10 times the rate of

Table VII-4

TRANSIENT RECOVERY PROBABILITY AND TRANSIENT ERROR RATES

MODEL IV. •

PROCESSOR PERMANENT FAILURE RATE: 1E-4

PROCESSOR TRANSIENT FAILURE RATE: 1E-5

BUS UNITS PERMANENT FAILURE RATE: 1E-5

BUS UNITS TRANSIENT FAILURE RATE: 1E-5

MISSION TIME, HOURS: 10

RECONFIG. TIME, IN SEC: 0.1

RECOVERY PROBABILITY: 0.9

FAILURE PROBABILITY TABLE:

PRO/BUS	2	3	4	5	6	7
2	2.24E-03	2.02E-03	2.02E-03	2.02E-03	2.02E-03	2.02E-03
3	2.23E-04	3.09E-06	3.06E-06	3.06E-06	3.06E-06	3.06E-06
4	2.20E-04	4.04E-08	4.15E-09	4.15E-09	4.15E-09	4.15E-09
5	2.20E-04	3.64E-08	6.75E-11	6.25E-11	6.28E-11	6.32E-11
6	2.20E-04	3.64E-08	9.07E-11	8.56E-11	8.59E-11	8.63E-11
7	2.20E-04	3.64E-08	1.25E-10	1.20E-10	1.20E-10	1.20E-10
8	2.20E-04	3.65E-08	1.64E-10	1.59E-10	1.60E-10	1.60E-10
9	2.20E-04	3.65E-08	2.10E-10	2.04E-10	2.05E-10	2.05E-10
10	2.20E-04	3.65E-08	2.60E-10	2.55E-10	2.56E-10	2.56E-10

* RECONFIGURATION WITH TRANSIENT RECOVERY.
(UNCORRELATED BETWEEN DEVICES)

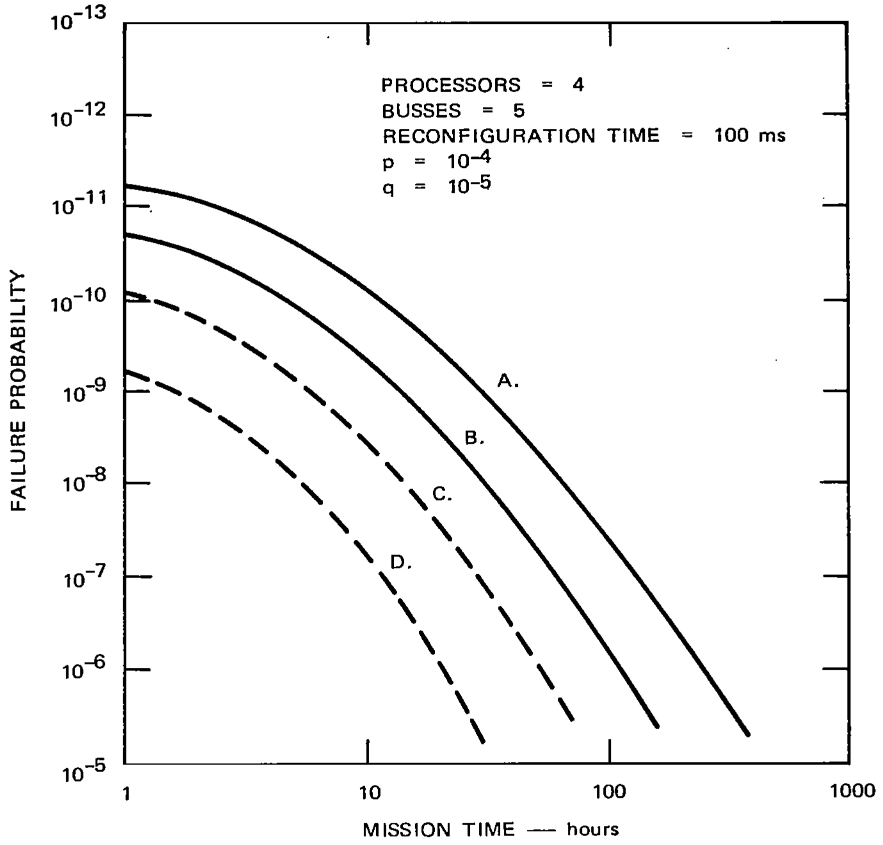


FIGURE VII-7 MODEL IV BEHAVIOR

permanent faults. Finally, curve C represents a 70 percent probability of recovering from transient errors also occurring at 10 times the permanent failure rate.

We do not presently have reliable data on the expected rates for transient faults, but the rates covered in Figure VII-7 are probably higher than those that would occur in practice. Observe that even with the high rates of 10^{-3} and 10^{-4} for processor and bus transient errors, the system failure rate for a 10 hour mission is about 3 x 10^{-8}. This is well within the nominal 10^{-9} per hour failure rate considered satisfactory by the FAA.

REFERENCE

1. D. R. Cox and H. D. Miller, The Theory of Stochastic Processes, Wiley (1965), pp. 187-189.

VIII THE HIERARCHICAL DESIGN METHODOLOGY

This section was previously issued as Technical Memo No. 6. It is a tutorial description of the software design methodology that is being employed in the design of the SIFT computer and other software systems. This particular approach is an instance and an extension of what has come to be called "structured programming." It has been developed in its present form mainly at SRI for the creation of large and complex programs, including operating systems.

This new methodology appears to have considerable generality, but for this introductory description only those aspects that are relevant to the SIFT computer design are covered in detail. Emphasis is placed on concepts that are either fundamental but unfamiliar, or are especially critical in the SIFT computer.

The new methodology claims several significant advantages over conventional software design techniques, namely

- The costs of program production are reduced.
- The final program is more amenable to formal proof of correctness than a program developed on an ad hoc basis.
- The reliability of the resultant program is improved.
- The program is specified in a way that enhances its understandability.
- The program is flexible to future design modifications.
- The program is created in such a way that it may be augmented to have additional special properties--e.g.,
    - Security provisions can be added to prevent unauthorized access, leakage or modification of information.
    - The operational reliability can be assessed quantitatively.

Most of these features are important design objectives of the SIFT computer.

Large and difficult problems of any type--basic research, engineering design, product development, and product organization--are invariably handled by some sort of <u>decomposition</u> of the large problem into several smaller ones. Of the many possible decompositions that could be used, one is normally selected for which three conditions are satisfied:

(1) Each smaller problem is <u>well defined</u>, so that there is no ambiguity about whether a proposed solution is really a solution.

(2) The decomposition is <u>complete</u>--that is, solutions to all of the smaller problems will be sufficient to solve the large one.

(3) The smaller problems are <u>easier to solve</u> than the large one.

The small problems can be further decomposed according to the same conditions. This decomposition may be repeated as often as necessary until only readily resolved questions, tests, measurements or experiments remain. If the three conditions are satisfied at each step, then one may be sure that the decomposition process will terminate, and that the original large problem has a solution when the final residual problems are solved.

These conditions for the iterability of a decomposition are well-known, but are stated here in this particular way because of their direct relevance to the creation of complex software systems--a relevance that has resisted formal treatment during the past two decades of development of computer programming. These three conditions, when properly restated in formal terms, are sufficient to insure that large and very intricate programs may be created by repeated decompositions, as finely as desired, so as to achieve the advantages listed above.

The concepts that are central to an understanding of this type of program decomposition will be introduced by extension from two familiar engineering design problems. These examples utilize decompositions so natural and familiar that the iterability conditions are not usually dealt with explicitly.

The first point to be made is that the results of a decomposition may be abstract rather than concrete.

Consider the design of a large unit of engineering hardware such as an automobile or a spacecraft. For an automobile a traditional approach is effective: the overall problem is decomposed into portions corresponding to the various physical parts of the vehicle--engine, steering, brakes, body, electrical system, and so on. Each of these portions can be similarly decomposed, and so on. Specifications are written for each portion to permit the various portions to be designed independently, with the assurance that they will all fit together in the final assembly.

For spacecraft, however, it has proven more effective to make the initial decomposition on the basis of function--e.g., attitude control, propulsion, scientific experiments, communications, etc., as suggested by Figure VIII-1. Each such function may but need not correspond to a particular unit of hardware. Finer decompositions are made in terms of function and/or hardware. A given unit of hardware may be used to provide several functions or subfunctions. The first two iterability conditions are satisfied at each step by (1) describing each function or unit by a specification that prescribes its behavior completely, without getting involved in how this behavior is implemented; and (2) showing how the overall design specifications will be met if only the various functions or units meet their own specifications.

In addition, if each step involves a simplification of function [Condition (3)], the entire iteration will converge into an effective design. For example, the communication function might be broken down into data acquisition, data reduction and data storage, followed by a transmitter and an antenna system. Data acquisition could be further decomposed into sub-functions such as scanning, multiplexing, A→D conversion, etc.; and so on. As noted, some subfunctions will be executable on common subunits of hardware, such as an onboard computer provided to implement not only data acquisition and reduction, but data storage, portions of attitude control and the scientific experiments as well.

This example should illustrate clearly that design decompositions, including full specifications, need not be made in terms of hardware, but that abstract nonphysical concepts such as "function" can be used as well.

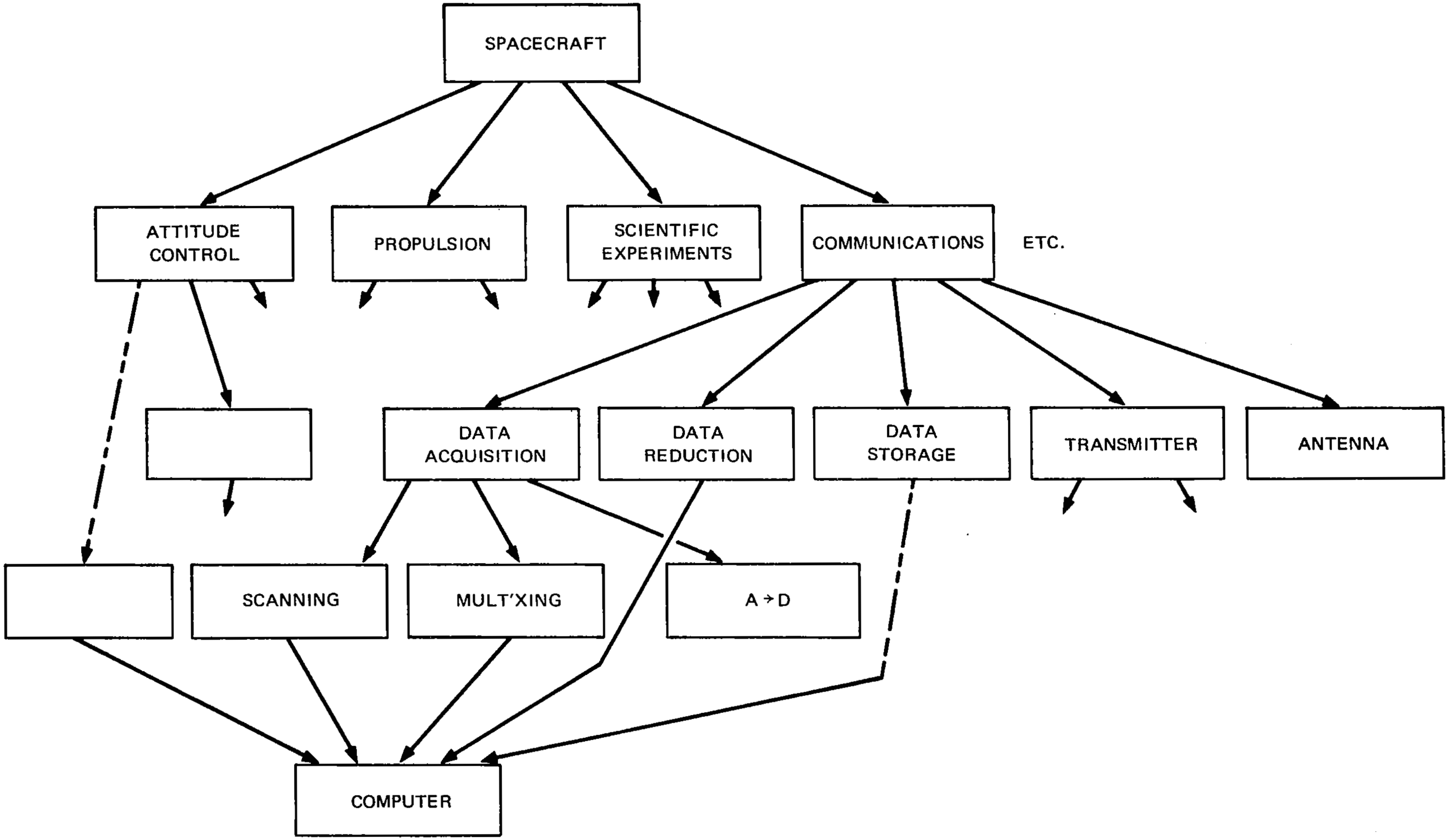


FIGURE VIII-1 DECOMPOSITION IN TERMS OF FUNCTION

A second illustration is provided by the set of subroutines that are commonly employed in the writing of a program. Assuming that no recursion is allowed, the "calling" operation can effectively organize these subroutines into a hierarchy with the main program (or programs) at the top (Figure VIII-2). This hierarchy represents an iterated decomposition of the original program (problem) into a succession of successively smaller programs. One may even suppose that the simplest subroutines are themselves decomposed into sequences and instructions taken from a common programming language at the lowest level. The syntax of conventional programming normally forces the three iterability conditions to be satisfied automatically.

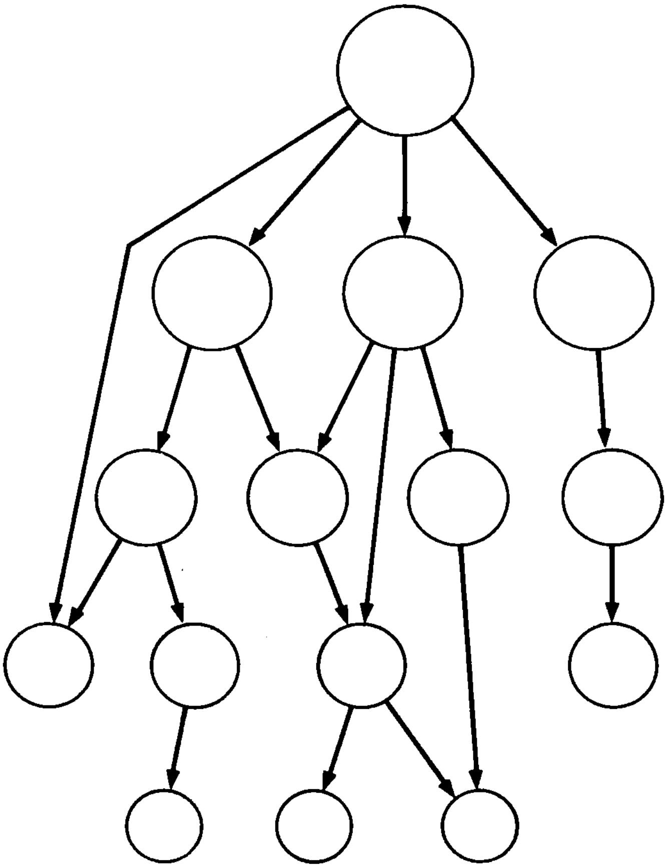


FIGURE VIII-2 ILLUSTRATION OF A FUNCTIONAL HIERARCHY

This second example illustrates how a programming hierarchy can be set up. It is desirable that this hierarchy possess certain features, the most important of which is the independence of the data handled by a subroutine from direct manipulation by those higher-level subroutines that may call the subroutine in question. It would be preferred if the data structures used by any one subroutine (or procedure) could be intimately associated with the operations of that subroutine, so that all accesses and changes to the data must pass through the subroutine itself. While these kinds of constraints may be incorporated into a conventional program in any specific instance, what is really needed is a methodology that automatically controls the allowed ranges of operations on stored data and programming states when creating programs in general.

This requirement may be satisfied inherently by placing the data structures to be manipulated by the program in the same decomposition hierarchy as is used for the subroutines themselves. If this is done properly, the specifications for a particular function in the hierarchy can be written to provide strict control over the corresponding elements of data. As a result, each subroutine will be self-contained and fully specifiable with respect to both the operations it performs and the corresponding data structures. Note here that, just as the fundamental instructions for the hardware correspond to the lowest levels in the hierarchy, so also are the fundamental elements of data located at this level. As one moves upward through the hierarchy, data structures composed from these data elements may become larger and more intricate, just as the functions performed upon them become more complex and powerful.

A type of program hierarchy will now be described that satisfies this additional data-structuring requirement in an effective and practical way.

The fundamental element in the new hierarchy will be designated a module, a term due to Parnas (who is also responsible for other ideas in structured programming). A typical program hierarchy has the same form as that illustrated in Figure VIII-2. Each module, depicted by a circle in the figure, is related to certain other modules below it by a dependency relation, which is indicated by an arrow and will be defined subsequently. The uppermost module(s) represents the user program(s), and the lowermost modules represent the minimal level of implementation--e.g., computer instructions or hardware.

In essence, a module consists of a collection of data structures and a collection of operations on these data structures. For example, a module called MATRIX might maintain n × n matrices of real numbers and functions for inversion, transposition, element change, access, etc. A module called STACK might be used to push and pop characters on the top of a "stack" of stored characters.

By virtue of its data structures, a module may be said to possess a storage state which will change from time to time as operations are performed in the module. To specify a module completely, it is first convenient to define its data structures by declarations of variables, parameters, etc., in a conventional way, plus a set of value-functions called V-functions. These V-functions collectively and completely describe the storage state of the module, though without presuming any particular configuration of the data elements in a physical or other form. For example, it is of no concern at this point whether the characters in a STACK module are stored in the form of a bidirectional shift register, an array with a pointer, or as a linked list; the only property of interest is that the characters entered by push be returned by pop in inverse order of entry. In this case, the set of V-functions describes the set of all past characters pushed in that have not yet been popped.

For the second part of a module specification, the operations performable in the module are described by a set of operation-functions or O-functions. These O-functions are expressed in terms of the effects they have on the set of V-functions of the module. That is, each O-function describes in V-function terms how a prior storage state is transformed into a new storage state. Again, no presumptions are made here as to how an O-function is to be realized or implemented in terms of simpler constructs.

To complete the specification of a module, initial values must be specified for all V-functions. Also, exception conditions may be indicated for both O- and V-functions--e.g., an input variable is out of range, allocated storage space is full, or a disallowed operation is requested. English-language explanatory comments may also be included if desired, but the specification must be complete without these, of course. Finally, the module is given a name.

The total specification now satisfies the first iterability condition, namely, that the module is completely defined as far as its behavior is concerned.

To satisfy the second iterability condition, each module is implemented by a set of lower-level modules in the hierarchy. The modules on which a given module depends for its implementation are said to be in a dependency relation to it, and are connected to it in Figure VIII-2 by an intermodule arrow pointing downward from it. Consider a typical module M_k and all those other modules on which it depends--its dependency set--as illustrated in Figure VIII-3. Let us suppose that an implementation of M_k has already been accomplished, so that M_k and all of the modules in its dependency set have been specified as described above. Thus, each has its own set of O- and V-functions, initial values, etc., appropriate to its purpose. To "implement" M_k it is now necessary to somehow relate its own specification to the specifications of the modules in its dependency set.

To this end, a correspondence between M_k and its dependency set is recognized. First, the states of M_k and those of its dependency set are put into correspondence. This is done by mapping the nonhidden V-functions in the dependency set onto the set of V-functions of M_k. For example, the states representing the contents of matrices in a MATRIX module would be expressed in terms of the states representing vectors in a lower VECTOR module on which it depends. A state mapping is actually a mapping of data structures. It can be expressed by a set of assertions or equations, or in simple cases merely as a tabular listing.

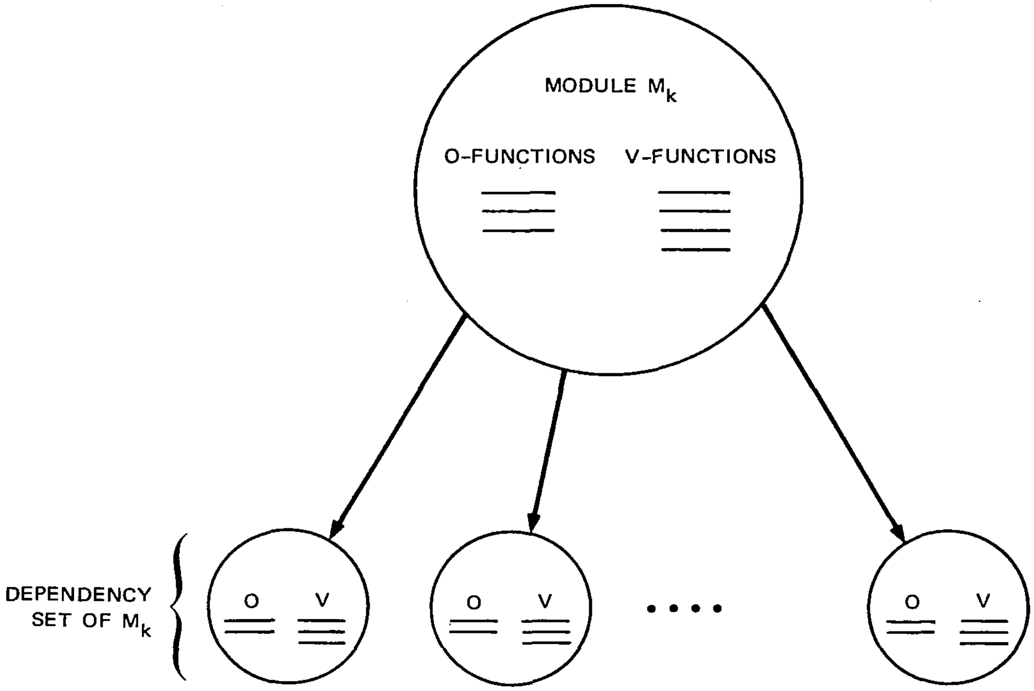


FIGURE VIII-3 DEPENDENCY SET

In general, certain portions of the total state in the dependency set may be transparent to M_k, in which case the mapping will be many-to-one rather than one-to-one. The corresponding V-functions are said to be <u>hidden</u> (HV), e.g., all stored characters except the most recently entered one in a STACK module would normally be transparent to a calling module above, and would be represented by such a function. HV-functions in the dependency set do not participate in the mapping onto M_k, but the HV-functions in M_k must all be accounted for by the set of V-functions in the dependency set.

Second, every O-function and V-function of module M_k must be implemented as a program (that is a sequence of calls) in terms of the O- and V-functions of the modules of its dependency set. Note that these programs for function implementation are <u>abstract</u> programs, in the sense that the modules on which they might be run are themselves <u>abstract machines</u>.

This particular definition of a program hierarchy of modules naturally assumes that there is no recursion in the dependency order. However, it is permitted for two or more modules to depend upon the same lower-level module. Consequently, the graph that describes the hierarchy is a directed graph without cycles.

The set of nonhidden V-functions of a module may also contain *derived* or DV-functions. These are redundant and are created for convenience only. DV-functions need not participate in the assertions that describe mapping correspondences, but must be defined directly or indirectly as programs, just as for the other accessible V-functions.

As a convenience one may also speak of an *OV-function* as an inseparable concatenation of an O-function and a V-function. It need not participate in the mapping, but like O-functions requires a statement of effects in its specification.

The second condition of iterability, the completeness of decomposition, is therefore satisfied provided each O- and V-function of every module can be realized as a program of O- and V-function taken from lower-level modules of the hierarchy, where the data structures of the corresponding modules are related by a complete V-function mapping as defined above.

The third iterability condition is satisfied in the course of design, provided each module is implemented with other modules that are less complicated than itself. This condition is not automatically satisfied by the design methodology. However, the methodology so structures the design that it is easier for the designer to maintain control over the complexity of the implementations at each level, compared to a conventional design.

These concepts and definitions will now be illustrated by means of the simple example illustrated in Figure VIII-4. An upper module STACK is to be implemented by means of a lower module ARRAY. The function of STACK is to carry out the usual push and pop operations on a conventional stack--a data structure consisting of a finite ordered list of elements on which elements may be inserted (pushed) or removed (popped) only at

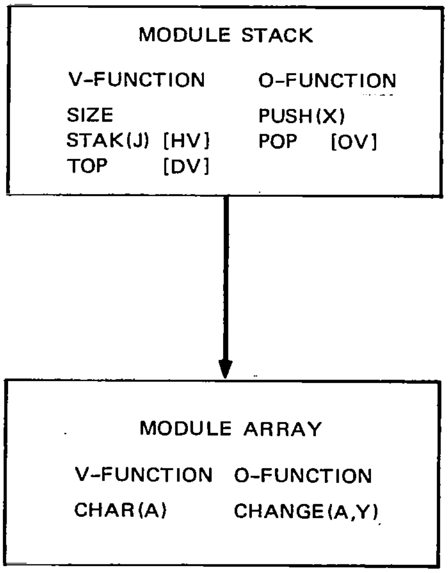


FIGURE VIII-4 EXAMPLE OF CONCEPTS AND DEFINITIONS

the top of the list. The data structure in ARRAY is an unordered set of elements.

A complete specification of the module ARRAY is listed in Table VIII-1. First, variables are declared as to their types--that is, whether they are integer, real, complex, boolean, etc., and parameters of the array are defined. Next, the single V-function CHAR(A) is specified. After stating its purpose, an exception condition is given to cover the case when the independent variable A is out of range. Then, the initial value of the V-function is indicated for all possible values of A. Next, the O-function CHANGE(A,Y) is specified, including its purpose, exceptions conditions (either A or Y out of range), and its effect upon the V-function of the module. This module could be directly realized in a random access memory, in which case the V- and O-functions would correspond to nondestructive read and simple write operations, respectively. Note that there is nothing about the specification of ARRAY that presupposes this particular hardware realization, however.

Table VIII-1

MODULE ARRAY

Declaration:	Integer A, Y	
Parameters:	AMAX	Size of Array
	CHMAX	Maximum value of stored element
V-Function:	CHAR (A)	
	Purpose:	To return the A-th element of the array
	Exceptions:	AOUT: $A < 1$ or $A > AMAX$
	Initially:	$\forall\ (0 \le i \le AMAX)(CHAR(i) = 0)$
O-Function:	CHANGE (A, Y)	
	Purpose:	To replace the A-th element of the array by Y
	Exceptions:	AOUT: $A < 1$ or $A > AMAX$ YOUT: $X < 0$ or $Y > CHMAX$
	Effects:	$CHAR(A) = Y$

The specification of the module STACK shown in Table VIII-2 follows similar lines. The V-function SIZE reflects the number of entries that are currently stored in the stack. A HV-function STAK(J) represents the entire contents of the stack; it is invisible and cannot be called by higher-level modules. A DV-function TOP is derived from STAK(J). The effects of the O-function PUSH(X) are to inject a new element X onto the top of the stack and to increase SIZE by 1. The OV-function POP accomplishes the reverse--the top element is both returned and deleted from the stack, and SIZE is reduced by 1. Again, the STACK module could be implemented in a variety of ways and none are assumed or precluded by the specification given here.

Implementation of STACK by means of ARRAY requires first that the V-functions of the two modules be placed in correspondence. To this end, the parameters are related first. The nonderived V-functions of STACK are expressed in terms of (nonhidden) V-functions of ARRAY. Note that the elements of STAK and CHAR are placed in one-to-one correspondence, except that one extra element of CHAR is reserved for SIZE, which is to be used as a pointer in this implementation.

Table VIII-2

SPECIFICATION OF MODULE STACK

Module Stack

```
Declarations:  Integer J, X

Parameters:    SMAX     Maximum size of stack
               CHARMAX  Maximum value of stored element

V-Function:    SIZE
               Purpose:  To return the number of elements currently in
                         the stack
               Exception:  None
               Initially:  SIZE = 0

HV-Function:   STAK(J)
               Purpose:  To represent entire contents of the stack
               Initially:  ∀i(0 ≤ i ≤ SMAX)(STAK(i) = Undefined)

DV-Function:   TOP
               Purpose:  To return top element in the stack
               Derived:  TOP = STAK(SIZE)
               Exceptions:  EMPTY:  SIZE = 0

O-Function:    PUSH(X)
               Purpose:  To augment stack with an additional Element X
               Exceptions:  FULL:  SIZE = MAXS
                            XOUT:  X < 0 or X > CHARMAX
               Effects:  SIZE = 'SIZE' + 1
                         STAK(SIZE) = X

OV-Function:   POP
               Purpose:  To return and remove top element of the stack
               Exception:  EMPTY:  'SIZE' = 0
               Effects:  POP = STACK('SIZE')
                         SIZE = 'SIZE' - 1
```

Mapping

```
Parameters:    CHMAX = CHARMAX
               CHMAX = SMAX
               AMAX = SMAX + 1

V-Functions:   ∀J(1 ≤ J ≤ AMAX - 1)(STAK(J) = CHAR(J))
               SIZE = CHAR(AMAX)
```

Initialization

```
CHANGE (AMAX,0)
```

Implementation

```
V-Function:    SIZE = CHAR(AMAX)

DV-Function:   TOP = CHAR(CHAR(AMAX))
               EXIT:  EMPTY:  CHAR(AMAX) = 0

O-Function:    ASSERTION:  FULL:  CHAR(AMAX) = AMAX - 1
                           XOUT:  X < 0 or X > CHMAX
               PUSH(X):  CHANGE (AMAX, CHAR(AMAX) + 1)
                         CHANGE (CHAR(AMAX), X)

OV-Function:   ASSERTION:  EMPTY:  CHAR(AMAX) = 0
               POP:  POP = CHAR(CHAR(AMAX))
                           CHANGE (AMAX, CHAR(AMAX) - L)
```

Next, ARRAY must be initialized to conform to the initial conditions of STACK. Only the extra element CHAR need be set to a defined value.

Finally, the implementation is described by expressing each function of STACK as a program in terms of the V-function CHAR and the O-function CHANGE of the module ARRAY. Exit conditions expressed in the same terms describe non-normal returns from the lower level to the upper level.

Several properties of this realization may be noted at this point as a way of summarizing some important general features of a hierarchical design:

(1) Each module specification is essentially independent of those of all other modules.

(2) The effects of O-functions and the definitions of DV-functions are expressed in terms of nonderived V-functions of the same module, and are implemented solely in terms of functions occurring in that module's dependency set.

(3) The implementation employed for a module is not visible to those upper modules that may depend upon that module for their own implementations.

(4) The state of a module is determined by the complete set of values of all of its nonderived V-functions, over all allowed values of their arguments.

(5) The mapping of V-functions between modules presumes explicit relationships between the parameters of corresponding modules.

(6) HV-functions have no exception conditions; O- and OV-functions have no initial conditions; and none of the V-functions have effects.

With this background, the steps of design according to the methodology may be outlined as follows. The starting point for the design consists of a specification of the uppermost module in the hierarchy--a concise description of what the overall program is to accomplish. If the hardware on which the final program is to be implemented is prescribed, a list of the lowest-level elements or functions out of which the system is to be composed will also be specified. Then:

(1) The uppermost module function is decomposed into a hierarchy of modules, the function of each of which is cursorily described in words.

(2) The function of each module is defined precisely by a specification, as explained previously.

(3) The V-function mappings are worked out between each module and those of its dependency set.

(4) All V-functions and O-functions are implemented as programs in the V- and O-functions of modules of their corresponding dependency sets.

Like most designs, these four steps are not independent of one another. They are not executed in a single sequence but are passed through repeatedly in trial-and-error fashion until all conditions are satisfied and all cost and quality measures (such as the number of programming steps, the running time, and amount of storage required) are suitably optimized.

Steps 2 and 3 above are largely formal, but steps 1 and 4 are more creative. The first step requires a perspective view of the entire hierarchical program. In decomposing the various module functions, the designer must use his past experience to anticipate how they should best be decomposed, both in terms of the operations performed and the data structures appropriate to these modules. The fourth step can often benefit from ingenuity at a more detailed, logical level.

Two idealistic approaches to hierarchical design are fashionable. In the bottom-up approach the modules are defined in succession from the bottom of the hierarchy to the top, in such a manner that no module is created until all modules in its dependency set have been created first. The user program at the top is created last. In the top-down approach one starts with the uppermost module and creates lower-level modules in succession as they are needed, until all of the lowest-level modules are defined. The bottom-up approach is more of a synthesis, in the sense that the simplest elements of the system are gradually assembled into more and more powerful units in order to realize the function at the top. In one way it is a more orderly progression. The top-down viewpoint is more analytical, in the sense that functions, data structures and operations are repeatedly broken down into simpler parts until only primitive versions remain.

Both of these approaches suffer from the same disadvantage. Namely, it is normally difficult to define modules at intermediate levels merely from a knowledge of module specifications at the uppermost and lowermost levels of the hierarchy. While the decomposition of the system into a functional hierarchy greatly simplifies the overall design problem, the step of selecting intermediate-level modules still requires a broad view of the manifold possibilities of analysis and synthesis. These are presently best acquired only through experience.

Consequently, the bottom-up and top-down approaches actually mark extremes at the ends of a continuum of possibilities. The practical approach lies in between. The total number of competing design alternatives are reduced by working at all levels of the hierarchy simultaneously, using the top and bottom levels only as starting points.

If a computer program has been designed hierarchically as defined above, formal program proving techniques can be applied appropriately to the module specifications, to the V-function mapping correspondences, and to the O- and V-function implementations, to prove correctness of the program. These techniques are now undergoing refinement in related SRI projects in which the hierarchical design methodology is being applied to other software systems.

This hierarchical approach also creates a design in which it is easier to control security--maintaining control over different users' rights to access and/or change various data elements--including a capability for proving that the desired security is indeed achieved. Finally, a program resulting from the new methodology is amenable to determination of its reliability, under an assumed set of failure probabilities for its lowest-level module functions, and to proving that a required reliability level has actually been achieved in the design.

IX HIERARCHICAL ORGANIZATION OF SIFT

In this section we discuss the design of the SIFT system as a hierarchical layering of abstract machines. First, we briefly discuss the hierarchical design methodology, with respect to our concept of the SIFT requirements and the handling of faults and errors. This methodology comprises five stages of design and implementation. The realization of SIFT is discussed relative to these five stages. The proposal for this project suggested that the operating system be described using flowcharts. This suggestion has not been followed because the five stages provide a description which is easier to understand and also is easier to verify and analyze.

A. The Hierarchical Methodology Relative to SIFT

In Section VII the hierarchical methodology was discussed as a general approach to designing and proving systems. Below, we briefly review the methodology and present some augmentations to handle hardware faults, their impact on all abstract machines and an approach toward developing a credible reliability assessment of SIFT. The methodology involves the following stages.

Stage 0--Express the problem to be solved in abstract and perhaps imprecise terms. For SIFT this stage entails expressing the intent of the SIFT system with regard to dispatching application tasks and the handling of errors.

Stage 1--Conceive of a set of abstract machines that seem appropriate for solving the problem. Each abstract machine has a state space and operations to change the state. As suggested by Parnas [Ref. 1], we use V-functions to represent the state and O-functions to correspond to operations. The machines are organized hierarchically, i.e., as nodes in a directed acyclic graph. An edge from node A to node B indicates that the machine at node B implements the machine at node A. The machines corresponding to leaf nodes in the graph are called primitive machines since the operation of the entire system is dependent on these machines. In the design process it is useful to view each abstract machine as maintaining a particular type of abstract object for use by an abstract machine directly above it in the hierarchy. Some of the functions of each abstract machine can be called by programs that run on the system. We designate these functions as comprising the system interface. For SIFT the programs that call the interface functions are simply the application tasks. It should be clear that the machine functions that are not part of the interface are not accessible to the application programs.

All of the O- and V-functions of a machine accessible to a higher level machine are called the visible functions of a machine. Some of the V-functions just serve to aid in defining the state and cannot be accessed; these functions are called hidden V-functions. The set of V-functions of a machine that are essential in defining the state space of a machine are called primitive V-functions. Some other V-functions, called derived V-functions, return a value that is dependent on the value of more primitive V-functions. The role of derived V-functions is to provide a mechanism for referring to a collection of states by a single function. These functions are described in more detail in Section IX-D.

In order to formalize the concept of a fault occurrence, we use the mechanism of hidden O-functions. That is, some of the abstract machines will contain the O-function cause-fault, which is not called by any program, but occurs asynchronously with other processing, with a probability dependent upon the hardware and transient fault mechanisms. The object that is "damaged" by the fault is dependent on the particular abstract machine that is subject to failure. In our analysis of SIFT the primitive objects that are subject to failure are processors and busses. This represents a coarse treatment of faults as compared with one in which faults are assumed to affect memory words or processor registers. Our course approach is realistic for an LSI implementation and moreover, does not produce overly pessimistic results. Hidden V-functions are included in certain machines to record the occurrence of faults. These functions are of necessity hidden since an observer of the machine's behavior will not detect the fault occurrence until the machine is appropriately exercised.

Stage 2--In this stage a formal specification is written for each of the abstract machines. The exact format of the specifications is discussed in the next section, but it suffices for this discussion to say that the most significant part of a specification is the effects section for each O-function which gives the new V-functions in terms of the values of V-functions immediately prior to a call on the O-function. These effects are written as assertions in a language to be described below. In the case of machines whose specification is intended to portray the results of faults, the effect of the generic O-function cause-fault is to change the values of hidden V-functions that record the fault occurrence. The fault can produce a perceivable error when an O-function is invoked whose specifications are dependent on the above hidden V-functions.

Stage 3--In this stage the states of each non-primitive abstract machine are represented in terms of the states of the lower level machines that comprise its implementation. This representation is a partial mapping from the state-spaces of the lower level machines onto the upper level state-space. It is partial since not every lower level state will have an image in the upper level, and it is onto since each upper level state must be the target of a mapping. Two distinct states S1, S2 in the upper level machine must have distinct images in the lower level machine, otherwise in the implementation the states will not be distinguishable. On the other hand, an upper level state S1 can be represented by more than state T1, T2,... in the lower level machine. The meaning of this multiple representation is that at any instant only one of the states Ti actually represents s1, but which state is selected depends on the implementation (stage 4) and possibly external inputs, e.g. faults. If the lower level consists of more than one abstract machine it is convenient to view the aggregate of such lower level machines as a single machine with a state space that is the cartesian product of the component state spaces.

It is also convenient to carry out the state mapping in two steps. In the first step all of the states in the lower level machine that have images in the upper level are defined. In the second step the upper level target states are selected for each state defined in step 1.

As discussed for stage 2 above, certain states of a module record the occurrence of a fault. In some cases an occurrence of a fault of one machine also results in a fault at a higher level machine. In such cases the fault states, similar to other states are mapped upward. However, in other cases the occurrence of a fault is masked by the upper level machine. Thus, a fault-free and its counterpart faulty state both map up to the same state in the upper level machine.

Since the states of an abstract machine are defined by values of primitive V-functions, the intermodule representations are written as expression relating the lower level primitive V-functions and the upper-level V-functions. The process of defining the lower level states that map upward involves writing an expression in terms of the primitive V-functions of the lower level machine.

Stage 4--In this stage, the nonprimitive abstract machines are implemented in terms of the machines directly below them in the hierarchy. Since the visible O- and V-functions of a machine are callable, it is these functions that are to be implemented in terms of the visible O- and V-functions of the lower level machines. At present, we write these implementation programs in a simple language which we call an <u>abstract programming language</u>. These programs can serve as a plan for the ultimate implementation programs written in assembly language or perhaps some higher level language. We intend to study the possibility of using some augmentation of an existing high level language as the abstract programming language. That is, the abstract programs could be compiled directly into assembly code.

As discussed above for stage 1, the occurrence of faults is handled by the invocation of the O-function cause-fault by a hidden asynchronous process. If a fault in a lower level machine is to be apparent in the machine directly above, then there must be a cause-fault O-function in the upper level machine. The upper level cause-fault O-function is the "implemented" by a combination of cause-fault O-functions of the lower level machine.

Figure IX-1 depicts the state changes and state representation mappings for upper level machine S which is implemented by machine T. In S the effect of the O-function when the machine is in state S1 is to cause a transfer to S2. The states T1 and T2 of T map up to S1 while T3 and T4 map up to S2. If T is in state T1 then the implementation of the O-function is a program which causes T to undergo numerous state transitions, but only the initial and final states (T1 and T4) map up to S. A fault in T could cause a transition from T1 to T2, but this fault is invisible to S since both T1 and its faulty counterpart both map up to S1.

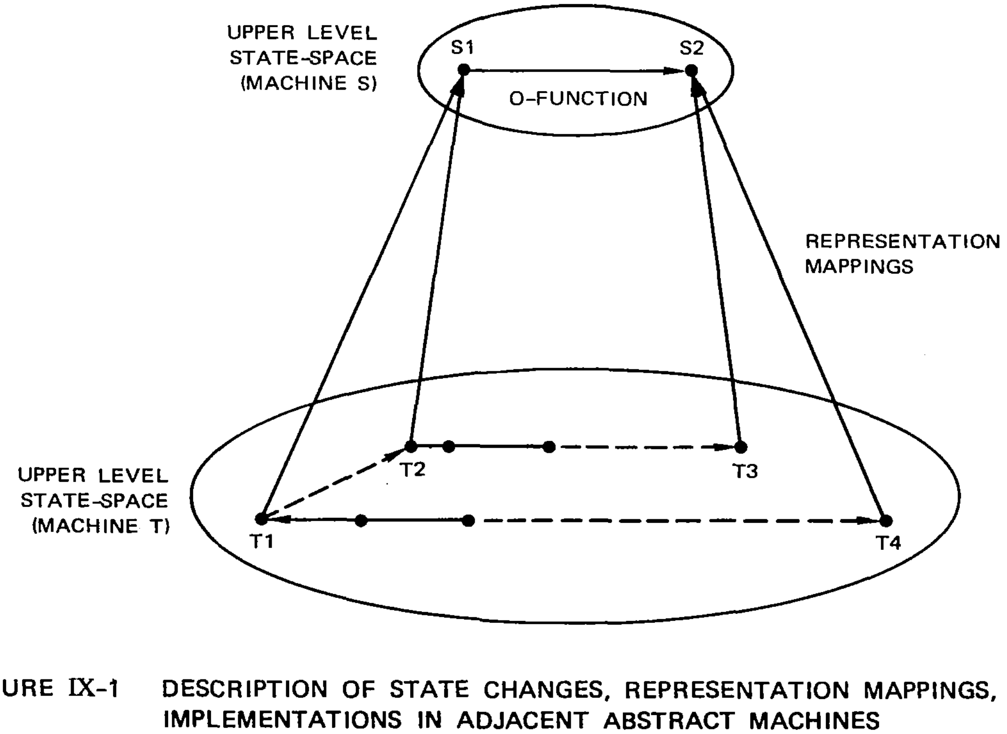


FIGURE IX-1 DESCRIPTION OF STATE CHANGES, REPRESENTATION MAPPINGS, AND IMPLEMENTATIONS IN ADJACENT ABSTRACT MACHINES

B. Stage 0 of the Methodology for SIFT

In this stage the intent of SIFT is described in imprecise terms in order to aid in developing a hierarchical organization. This description will also serve in formulating precise assertions about SIFT for purposes of verifying the design. The primary purpose of SIFT is to dispatch application tasks when their service is needed, even if the hardware units fail. As discussed in Section V, two types of application tasks are handled by the SIFT system: scheduled tasks which are guaranteed to be dispatched at a fixed rate, and priority tasks, each of which is dispatched if its deadline has expired and if its priority is highest of all such tasks whose deadline has expired.

The basic scheme for all application tasks, for each iteration is as follows:

```
READ DATA FROM EACH TASK SUPPLYING INPUTS
COMPUTE
WRITE DATA TO A BUFFER FOR EACH TASK THAT REQUIRES IT
AS AN INPUT
```

If the n-th iteration of task A requires data from the (n-1)st iteration of task A, then A is included in both the input task set and the output task set for A. A particular feature of the selected avionics task set is that a task running at iteration rate f does not read data from a task running at iteration rate f1, f1 > f. Thus a task A need only write data for a less frequently dispatched task B, as often as B is dispatched.

When B writes data for A, B deposits the data in a buffer that is shared with A. When A is dispatched its read data operation involves reading the contents of the buffer. For purposes of redundancy, tasks are executed on more than one processor. Thus the read operation for B from A yields a result which is the majority value of the data over all instances of execution of A. If no majority value exists then several policies can be invoked, one of which is to temporarily suspend B's execution. Several issues regarding the vote operation are significant to the design of the SIFT operating system.

- The application programmer for B should not have to know which processors are running A nor even how many such processors exist at any instant. The SIFT operating system should maintain such information and appropriately process the read input data command.
- When B's read from A involves a vote over more than one instance of A, it is essential that all such instances produce data for the same iteration (excluding those instances on faulty processors). As we observe below this is the only synchronization requirement on the execution of instances of tasks.
- When available, different busses are used for the read over instances of a task, in order to permit the vote mechanism to mask single bus failures, in addition to single processor failures.
- The primary error detection mechanism is via a disagreement on a vote. If the redundancy is sufficient relative to the number of faulty processors and busses, it is possible to uniquely identify the faulty units.

The units that are individually subject to failure are processors and busses. At a finer grain it will be necessary to consider failures than involve a processor's interaction with a bus. (We have conducted some analyses of this latter failure type and have incorporated mechanisms in the design to accommodate it, but the work is not yet complete.) When a failed bus is detected and identified, then the reconfiguration process will modify the system tables such that this bus is avoided in all future read input data operations.

When a failed processor is detected and identified, the reconfiguration process is to reallocate tasks to operative processors such that the operative processors are used in an optimal manner. One simple policy that can be pursued here is to allocate the tasks of the failed processor to a spare processor, if one exists, or else accept reduced redundancy for these tasks. The design does not impose the use of this simple reconfiguration policy, but instead allows for the storage of task allocations to processors as a function of faulty processors. This table could be computed prior to a flight or when a processor failure is detected.

For any policy of allocating tasks to processors, after a processor failure, the following steps must be carried out:

- The program code for a task assigned to a processor is loaded into that processor. In SIFT the loading process involves the reading of the program code from other instances of the program, by a special loading program in the processor.
- The tables of all processors executing reallocated tasks must be updated so that the read input data operations are directed to the appropriate processor. Since all data required by a task is obtained by read input data, once the program code is loaded and the tables are updated, the task is ready to be dispatched.

Some tasks will require service independent of the actions of the reconfiguration process. Thus it is essential that at least one instance of each critical task be dispatched as needed during the reconfiguration process. In order to achieve this continuity of service, the reconfiguration policy changes only one processor's task allocation at any time. In addition, that processor's reallocation is completed before attention is directed to another processor.

This section has presented an informal, but complete, description of the external interface of SIFT system, and SIFT's mechanisms and feasible policies in handling faults. The next section presents a hierarchical decomposition of SIFT.

C. Stage 1 As Applied to SIFT

The decomposition of the SIFT system as a hierarchy of abstract machines is shown in Figure IX-2. Each of the solid-line boxes corresponds to an abstract machine that maintains abstract data objects, i.e., contains O and V-functions. Each of the dashed-line boxes is an abstract program that does not contain a state; the programs just contain code to call the functions of lower level abstract machines, and perhaps, constants.

The state of the system is maintained entirely by the abstract machines. In order to simplify the description we show only a single instance of each machine. However, it is understood that each processor

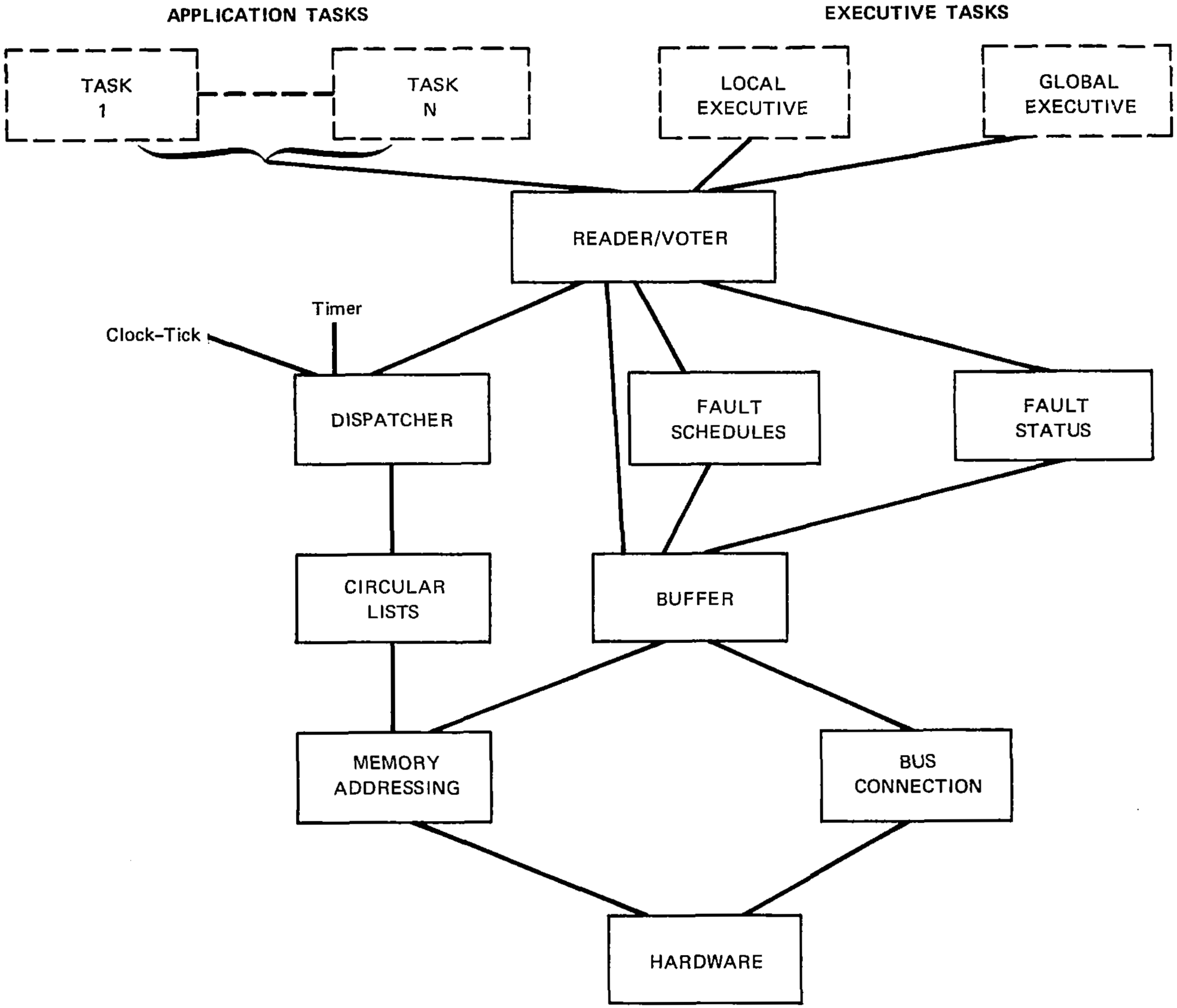


FIGURE IX-2 HIERARCHICAL STRUCTURE OF SIFT

in the system provides some of the functions of these abstract machines at its interface. In machine specifications (Section IX-D) we formally handle the situation of multiple instances by incorporating an argument identifying the processor on which the function was called. This formality is strictly for purposes of specification since a task instance can only call functions provided by the processor on which it is running.

A task, when it is dispatched, will acquire information by calling the interface V-functions of the abstract machines and can change the state of the abstract machines, say for purposes of transmitting information to another task, by calling the interface O-functions of the abstract machines.

Below we briefly discuss the abstract machines and two executive tasks; detailed discussions are given in Sections IX-D and IX-E.

Hardware: This machine provides the basic primitive processing instructions (arithmetic, logical, control) associated with each of the processors, in addition to the few instructions associated with the bus system. None of the bus system instructions are visible to the application tasks. The basic machine instructions are visible, with the exception that all main memory references by tasks are processed by the machines above the hardware. These extra levels of indirection ensure that an errant task will not be able to access memory outside of its workspace.

Memory Addressing: A task during its execution requires access to memory in order to read and write local data, and to read program instructions. The memory of concern here is the memory associated with the processor that is executing the task. The memory addressing abstract machine ensures that each task's accesses are within the preset bounds for the task. This level of indirection need not result in inefficient processing of memory instructions if hardware base and bound registers are used.

Bus Connection: This machine provides the mechanism for processor to connect with another processor by a particular bus. A task can establish a bus connection only indirectly, by performing a read input data operation.

Buffer: When task A is to deliver data to task B, the channel is a buffer in the processor on which A is executing and which can be read by B. Corresponding to each task for which A computes data, there exists two buffers. As we discuss in Section IX-D two such communication buffers are required since the execution of different instances of tasks is not tightly synchronized. The tasks do not access the buffers directly; the access is via the reader/voter abstract machine. The buffers are, of course, ultimately implemented in terms of areas of real memory that only the buffer machine abstract programs can access. Also, the buffer machine will effect the needed bus connections in order to carry out the interprocessor read operations.

Dispatcher: The dispatcher abstract machine holds, for each processor, a schedule giving the order in which tasks are to be dispatched. For scheduled tasks (see Section V) the schedule is a circular list of task names. During a frame interval successive tasks on the list are dispatched in turn. Each frame contains some spare time in which priority tasks are considered for dispatching. At the beginning of the next frame any priority task in execution is interrupted, and the next scheduled task on the circular list is dispatched. A processor can be given a new schedule after a processor failure or when a change in flight phase occurs.

Circular Lists: This machine provides functions for maintaining and accessing the circular lists that comprise the schedules for scheduled tasks. In this machine the circular lists are stored compactly as regular expressions.

Fault Schedules: As noted in Section IX-B, it is feasible to precompute schedules for each of the processors that are to be invoked under all possible processor fault conditions. These schedules are stored in the fault schedule machine for access as needed. It is likely that each processor could be preloaded with all of the schedules that it will need during a flight; the appropriate schedule, for a given state of the system, could be selected by the global executive.

Fault Status: This machine is used to store the status (operative or failed) of the processors and buses of the system. As we will note below it is used by the global executive in deciding the reconfiguration state after a fault.

Reader/Voter: This machine serves two purposes: (1) it is called by a task whenever it writes data into a buffer for another task or reads data from another task, and (2) it records the occurrence of faults. With regard to (1) the machine stores the identification of the processors executing tasks and the buses to be used in reading data. For a read operation, in which task A reads from a buffer location of task B, the reader/voter returns the majority value over all instances of task B. With regard to (2) the reader/voter records the occurrences of processor and bus faults. A fault becomes manifested as a detected error if it causes a disagreement in a vote. Multiple faults can cause a task to fail it if they cause one-half or more at the instances of a task to produce an erroneous result in an output buffer of the task. The reader/voter records such fatal error occurrences.

Each reader/voter instance, upon detecting an error as a disagreement among one or more of the voted inputs, attempts to identify the offensive units. It accomplishes this diagnosis by recording the processor-bus combinations that produced an input in the minority. The global executive, described below, analyzes the reports of all of the reader/voter instances, accounting for the possibility of erroneous reports from a failed processor.

Above we have briefly described the abstract machines of the SIFT system; more details are given in Section IX-D relative to the specifications. The global executive and local executive programs are briefly discussed below.

Global Executive: The global executive (GE) is simply a task that manages the system reconfiguration after the detection of a fault. It has access to the global status of the system and hence can determine the new system state. The GE should be dispatched often enough to guarantee a rapid transition to a recovery state. A preliminary analysis of the effects of reconfiguration time on the system reliability has determined that the GE should be dispatched as often as the highest-rate application task. Since the GE is a critical task, it must be executed redundantly--at least triplicated.

The operation of (each instance of) the GE is as follows. It reads the error reports of all reader/voter instances, and attempts to identify the failed processors and/or buses, if any. The GE will correctly identify the failed units provided for each vote only a minority of the inputs are in error. If the GE has identified a failed bus it computes a new bus assignment, and informs each local executive of this new assignment by communication through a shared buffer. If a processor has been deemed to have failed then the GE communicates with the local executives that must follow new schedules. In order to maintain the servicing of tasks during a reconfiguration, only one processor is permitted to be in the reconfiguration state at any instant. Hence, the GE selects a processor to follow a new schedule, and then awaits the completion of this processor's reconfiguration before selecting the next processor. The processing time for the GE is dependent on the existence of an error and on the number of reader/voter instances reporting an error. However, the maximum anticipated processing time should be small enough such that the GE can be considered as a scheduled task to be dispatched every frame.

The abstract implementation of the GE is discussed in Section IX-F. By virtue of the abstract functions provided in the SIFT interface, the program is relatively simple and should be amenable to proof.

Local Executive: Each processor contains a local executive (LE) which is a task which controls the reconfiguration of the processor as dictated by the GE. The LE is dispatched every frame and its operation is simply to read the buffer shared with the GE, in order to determine if the GE wishes to change the state of the LE's processor. This simple read operation consumes only a few machine instructions.

However, if the processor is to be significantly reconfigured then it is probably necessary to suspend the scheduling of tasks until the reconfiguration is complete. The LE can accomplish this suspension by calling the dispatcher machine to invoke a schedule which contains only the LE. Based upon the computation of the GE, the reconfiguration of a processor could involve: (1) informing the LE that the assignment of tasks to processors (excluding the LE's processor) is changed, or (2) informing the LE that the bus assignment for reads is changed, or (3) informing the LE that it is to change the schedule of its processor. The reconfigurations associated with (1) and (2) are not time consuming, merely requiring the updating of the reader/voter assignment tables.

The reconfiguration associated with (3) is significant, however, since it possibly involves a major change in the schedule of the processor. If new tasks are to be allocated to the processor then a loader program, which could be a separate task or just a subprogram of the LE, must be invoked to store the program code of the new tasks in the processor's memory. The loading is accomplished for each task by voting on each of the instances of the task's program code. The LE must assign memory locations in the processor for the new tasks and must update the tables of the buffer and memory addressing machines to reflect the new assignment of tasks. Once the loading and table updating operations are complete, the LE can invoke the new schedule and cause the processor to resume its servicing of tasks.

The LE is a more complicated program than the GE, but we feel that it should still be amenable to formal proof. Some of the complexity can be handled by decomposing the LE into three subprograms, corresponding to items (1) through (3) above.

D. Formal Specification of SIFT

1. Introduction

SIFT is specified as a hierarchy of Parnas modules. Each module is regarded as an abstract machine, having its own data structures (V-functions) and operations (O-functions).

At any given time, the state of each machine is just a description of the instantaneous values of its V-functions. The O-functions of a module are operations that cause the state to change.

The highest module in the hierarchy is an abstract, global description of what the system does. Modules at lower levels of the hierarchy can be viewed as building blocks for implementing the highest-level module. Modules at still lower levels are building blocks for implementing those at the intermediate levels, and so on.

The specifications that follow in subsection IX-D-1 describe each module independently. By themselves, they say nothing about how the lowever level modules are actually used to implement those at higher levels. This information is provided, rather, by mapping functions and abstract programs. Mapping functions implement the V-functions of a given module with the V-functions of lower-level modules; abstract programs implement the O-functions of a given module with programs written in terms of the O-functions and V-functions of lower-level modules. The mapping functions and abstract programs for SIFT will have to be specified before the system can be coded. It should be emphasized however, that the properties we wish to prove about the SIFT design depend only on the module specifications--not on the mapping functions or abstract implementation. (In the same way, the correctness of a FORTRAN program depends only on the program itself--not on the compiler.)

Parnas modules are specified according to a rigorous syntactic discipline much like a programming language. Each module specification is composed of several segments--one for declaration of type variables, one for defining V-functions and O-functions, and so forth.

The purpose of the various sections of a module specification might best be explained in relation to a specific example. Consider, then, the reader/voter module, which has five sections--one for DECLARATIONS, PARAMETERS, DEFINITION, EXCEPTIONS, and FUNCTIONS. This last section is actually the most important, since it declares the V-functions and O-functions of the module.

2. The FUNCTIONS section

The first function declared in the FUNCTIONs section of the reader-voter is the V-function task-set. The function header VFUN task-set(proc) = st gives the name of the V-function, its formal argument list (proc), and the result identifier st. The identifiers proc and st are declared in the DECLARATIONS section to be of type PROC (processor) and SET-OF TASK (set of tasks), respectively. The V-function task-set is thus a data structure which is indexed on PROCs and which stores sets of TASKs. The INITIALLY phrase in the declaration indicates that at the time the module is initialized, the value of task-set (on each argument proc) is the single-element set $\{\ell e\}$. The intended interpretation is just that at initialization of the system, each processor is loaded with only the local executive task.

The next declaration in the FUNCTIONs section introduces the V-function proc-bus-assignments, which is keyed on two arguments (a PROC and a TASK) and which stores sets of PAIRs. Unlike the declaration for task-set, this one has an EXCEPTIONs subsection. Exception conditions are boolean expressions used to restrict the domain of a V-function, much as array bounds in an ALGOL array declaration restrict the domain of the array. While V-functions may in general have an arbitrary number of exception conditions, proc-bus-assignments has only one: task-not-in-proc(proc task). Like all other exception conditions, task-not-in-proc is defined in the EXCEPTIONS section of the module specification (to be described later). As is evident from the definition, task-not-in-proc (proc task) is TRUE for a given value of proc and task if and only if task is not currently a member of task-set. Thus, just as it is erroneous to try to read an array outside the limits of its indices, it is erroneous to try to read proc-bus-assignments at ⟨proc,task⟩ if task $\notin$ task-set (proc) at that time. More generally, it is erroneous to try to read any V-function on arguments that violate any of its exception conditions at this time.

Declarations that contain a header, a comment, zero or more exceptions, and an initialization part account for the great majority of

V-functions in a typical module specification. In addition, there are a few special kinds of V-functions that are declared somewhat differently:

A DERIVED V-function is one whose value is determined completely by the values of other V-functions in the module. The V-function error-detected, for example, is DERIVED. Its value is determined completely by that of the V-function disagreement-set. More particularly, for a given value of its argument proc, error-detected has the value true if and only if the set disagreement-set(proc) is non-empty. Although derived V-functions are, strictly speaking, superfluous, they often add clarity to specifications. Note that derived V-functions have no INITIALLY specification since their initial values are determined by the initial values of the V-functions from which they derive.

A HIDDEN V-function is one that is not intended to be available to the user of the module. The V-function fault, for example, is HIDDEN reflecting the fact that the module does not provide direct access to information about what processors and/or busses are faulty. Note, however, that the values of HIDDEN V-functions do impact F-functions that are visible. For example, the DERIVED V-function read is in part derived from fault. Apart from the designation HIDDEN, HIDDEN V-functions are specified in exactly the same way as ordinary V-functions.

A third special kind of V-function, the OV-function, will be treated later.

In addition to declarations for the storage elements of the module, the FUNCTIONS section contains declarations for the operators, or O-functions, that change the values of those elements. As with V-function declarations, each O-function declaration begins with a header giving its name and formal argument list. Since O-functions do not store values but only change the values of V-functions, no "result" identifier is given. As with V-function declarations, an EXCEPTIONS subsection may be present, restricting the range of acceptable arguments. Once again, the definition of each exception condition appears in the EXCEPTIONS section of the module specification.

The substance of an O-function specification is contained in its EFFECTS subsection--the section that describes exactly what the O-function does. More precisely, the EFFECTS section contains a statement of the relationship between the state of the module (i.e., the values of its V-functions) before the O-function is called, and the state just after it is called. The O-function delete-task(proc task) in the reader-voter is a typical example. This O-function has two effects. It changes the value of the V-function task-set, deleting task from the set task-set(proc). It also changes the value of the V-function proc-bus-assignments, causing it to be undefined for the argument pair ⟨proc,task⟩. In the specification, quotes are used to distinguish the values of V-functions before the call from those after the call. Thus, 'task-set (proc)' refers to the state before the call while task-set(proc) refers to that after the call.

It is quite important to note that the statements in EFFECTS sections are assertions, i.e., mathematical statements of a relationship among states. They are not in any way procedural as would be, say, assignments in some programming language. For example, the assertion task-set(proc) = 'task-set(proc)' ∪ {task} could equally well be written

'task-set(proc)' ∪ {task} = task-set(proc)

since the = stand for equality, not for assignment.

Just as HIDDEN V-functions are used to mask certain state information, HIDDEN O-functions are used to mask certain changes in state. The O-function cause-fault is of this type. The cause-fault operation simulates a hardware failure that impacts certain reads. Since this operation is not really available to the SIFT system but is rather part of the internal affairs of the module, it is made hidden.

In addition to V-function and O-function declarations, the FUNCTIONS section contains declarations for a function which is a combination of the two--the OV-function. An OV-function may be viewed either as an O-function that returns a value, or as a V-function whose invocation produces a side effect. The OV-function vote-read is of this kind since it both returns a value of word and potentially changes the values of the V-functions fatal-error and disagreement-set.

3. The DECLARATIONS Section

Part of the specification of any Parnas module is a section declaring the types of the identifiers (such as formal arguments to V- and O-functions) used in other parts of the module specification. In most programming languages, declarations are used for two purposes: to direct allocation of storage, and to provide for type-checking. In the Parnas context, storage is associated only with V-functions, which are declared in the FUNCTIONS section. The DECLARATIONS section of a module specification concerns itself exclusively with the typing of formal arguments (including the "result" arguments of V-functions).

In the DECLARATIONS section of the reader-voter, there are declarations for integer and boolean identifiers much in the style of ALGOL. Unlike ALGOL, however, the language of module specifications provides an extensible type facility, that is, the designer of a module may introduce new, abstract types.

For this purpose, the DECLARATIONS section may include a TYPE subsection in which types (as opposed to objects of a given type) are declared. The TYPE subsection of the reader-voter contains declarations for three new primitive types (PROC, TASK, and MACHINEWORD) and a new compound type (PAIR). The word "DESIGNATOR" indicates that the new type is primitive, i.e., is not constructed from existing types. The name PROC suggests that objects of this type are intended to designate processors, as indeed they are. From the formal point of view, however, objects of type PROC have no intrinsic properties other than that they are distinct from all objects of any other type.

The new type PAIR, on the other hand, is defined in terms of more primitive types. The type designation STRUCTURE (PROC: proc, integer: bus) indicates that objects of type PAIR are composed of two parts, one of which is a PROC and the other of which is an integer. The identifiers proc and bus in the declaration of PAIR are <u>selectors</u>; given an object p of type PAIR, p.proc refers to the component of p which is of type PROC, and p.bus refers to that of type integer.

New types may also be developed using the SET-OF or BAG-OF constructs. The identifier sp, for example, is declared as a set of objects each of which is of type PROC. Similarly, votes is declared as a bag (that is, a kind of set in which a given member may have repeated instances) of MACHINEWORDS.

4. The PARAMETERS Section

The PARAMETERS section contains declarations for the constants of the design. Parameters can be viewed as V-functions whose values are fixed once and for all at the time the module is implemented. They are frequently used in exception conditions to designate the maximum or minimum values V- or O-functions arguments may take. The integer parameter max-tasks, for example, indicates the maximum number of tasks a processor can accomodate. Max-tasks appears in the definition of the exception condition too-many-tasks. Note that the syntax of parameter declarations mirrors that of declarations in the DECLARATIONS section.

5. The DEFINITIONS Section

Just as assembly language macros save programmers the labor of writing out repeated instances of a routine, definitional macros allow the specifier of a module to avoid repeated instances of an expression. Each definition begins with a MACRO header giving the name of the macro, a (possibly empty) formal argument list, and a result identifier (used for type checking).

Two definitional macros are used in the reader-voter module--majority-opinion(proc task offset) and dissenting-pairs(proc task offset). As it happens, each definition is used only once--in the EFFECTS section of the OV-function read-vote. Macros were used in this case not to save writing, but to make the EFFECTS section easier to read and understand.

6. The EXCEPTIONS Section

Exceptions were described earlier as boolean conditions used to restrict the intended argument domains of V-functions and O-functions. Because a single exception frequently applies to several V- and/or O-functions, all exceptions are defined as macros. The syntax used is exactly the same as that used for macros in the DEFINITIONS section.

Memory Addressing

The functions of the abstract machine instance in processor proc are called by abstract machine instances above memory addressing in the hierarchy and tasks executing on proc. These tasks will use memory addressing in order to execute instructions in their programs and to access temporary data locations. The machine includes some simple protection mechanisms in order to prevent a task from writing beyond the limits of its address space.

Initially the only task known to memory addressing, as indicated by the value of the V-function task-set, is the local executive (LE). The V-function mem-area-write defines the memory area allocated to a task for writing. Initially a fixed area is assigned to the LE; the remaining area is free as indicated by the value of the V-function area-free. The O-functions assign-mem-area and make-free respectively allocate memory area to a task and deallocate the area that was previously assigned to a task. In order to ensure than an errant, unproved application task does not deleteriously affect another task or the system, these functions are not accessible to application tasks. Instead the LE and the buffer abstract machine will have the major responsibility for managing the memory in its processor.

The V-function memory, is called by a task, or an abstract machine program, in order to read the contents at a memory location. The O-function, write, is called in order to modify the value at a location.

```
MODULE memory-addressing

        DECLARATIONS

TYPE
        PROC, TASK, WORD = DESIGNATOR

END-TYPE


WORD machineword
boolean b parity
integer offset length address
TASK task
PROC proc
SET-OF TASK s


        END-DECLARATIONS


        PARAMETERS


TASK le (; local executive task)
SET-OF PROC proc-set (; set of processors)
SET-OF TASK tasks (; set of valid tasks)
integer size-le (; number of words occupied by local executive)
integer mem-size (; total number of words of a single memory)
integer max-tasks (; maximum allowable number of tasks)

        END-PARAMETERS


        EXCEPTIONS

MACRO no-proc(proc) = b
     not proc member-of proc-set

MACRO not-a-task(task) = b
     not task member-of tasks

MACRO out-of-bounds(address) = b
     address > mem-size - 1 or address < 0

MACRO task-not-in-proc(proc task) = b
     not task member-of 'task-set(proc)'

MACRO not-authorized-write(proc task address) = b
        not( mem-area-write(proc task)[1] <= address  and
                address <= mem-area-write(proc task)[1] +
                        mem-area-write(proc task)[2] )
```

```
MACRO too-many-tasks(proc) = b
        cardinality('task-set(proc)') >= max-tasks

MACRO task-in-proc(proc task) = b
        task member-of 'task-set(proc)'

MACRO area-not-free(proc base length) = b
        not 'area-free(proc base length)'

MACRO not-a-task(task) = b
        not task member-of tasks


        END-EXCEPTIONS


        FUNCTIONS

 VFUN memory(proc address) = word
        (; memory contents of processor proc at given address)
        EXCEPTIONS
        out-of-bounds(address)
        no-proc(proc)
        END-EXCEPTIONS
        INITIALLY if 0 <= i <= size-le then memory(proc, i) = mem-le(i)
                else if size-le <= i < mem-size then memory(proc,i) = 0
                else undefined

 VFUN area-free(proc,base,length) = b
        (; indicates whether or not the locations in proc from
                base to base+length-1 are free)
        EXCEPTIONS
        out-of-bounds(base)
        out-of-bounds(base+length-1)
        not-proc(proc)
        END-EXCEPTIONS
        INITIALLY if base < size-le then  area-free(proc,base,i) = false
                        else area(proc,base,i) = true

 VFUN mem-area-write(proc,task) = <base,length>
        (; indicates memory range within which task is allowed to write)
        EXCEPTIONS
        no-proc(proc)
        task-not-in-proc(proc, task)
        END-EXCEPTIONS
        INITIALLY if task = le then
                     memory-area-write(proc,le) = <0,size-le - 1>
                  else undefined

 VFUN task-set(proc) = s
        (; indicates set of tasks assigned to processor proc)
        INITIALLY task-set(proc) = le
```

```
OFUN write(proc task address word)
        (; writes word in address of processor proc for task task)
        EXCEPTIONS
        task-not-in-proc(proc,task)
        not-proc(proc)
        not-authorized-write(proc task address)
        END-EXCEPTIONS
        EFFECTS
                memory(proc,address) = word
        END-EFFECTS

OFUN assign-mem-area(proc task base length)
        (; assigns authorized area in proc into which task can write)
        EXCEPTIONS
too-many-tasks(proc)
        task-in-proc(proc task)
        area-not-free(proc base length)
        not-a-task(task)
        END-EXCEPTIONS
        EFFECTS
                task-set(proc) = 'task-set(proc)' union { task }
                forall i,j (base <= i <= j <= length - 1)
                        implies area-free(proc i j) = false
        END-EFFECTS

OFUN make-free(proc task)
        (; deassigns task from proc, causing memory occupied by task
           to be deallocated)
        EXCEPTIONS
        not-proc(proc)
        task-not-in-proc(proc task)
        END-EXCEPTIONS
        EFFECTS
                let base = 'mem-area-write(proc task)[1]'
                let length = 'mem-area-write(proc task)[2]'
                forall i, j (base <= i <= j <= length - 1)
                        implies area-free(i j)
        END-EFFECTS

END-MODULE
```

The Buffer

The buffer module facilitates the transfer of computation results from one processor to another. The module contains a storage area, or buffer, for each triple <proc, taskl, task2> such that:

(1) proc is a processor

(2) taskl is a task currently running in proc

(3) task2 is a task currently running on some processor (possibly proc) that requires computation results from taskl.

The buffer associated with each triple actually consists of two separate storage areas: the even buffer and the odd buffer. On even iterations of the computation for a given task, results are stored in the even buffer; on odd iterations, results are stored in the odd buffer.

The need for such a scheme arises from the kind of synchronization situation illustrated in Figure IX-3. The figure shows a few iterations of computation in processors 1, 2, and 3. The solid horizontal lines represent intervals during which particular tasks are executed. Now suppose that Task B requires for its input, on each iteration, the output of Task A on the previous iteration. Because tasks executing in different processors are only loosely synchronized, Task B may not be executed concurrently in processors 2 and 3. More particularly, B may begin in processor 2 before A completes in processor 1, while B begins in processors

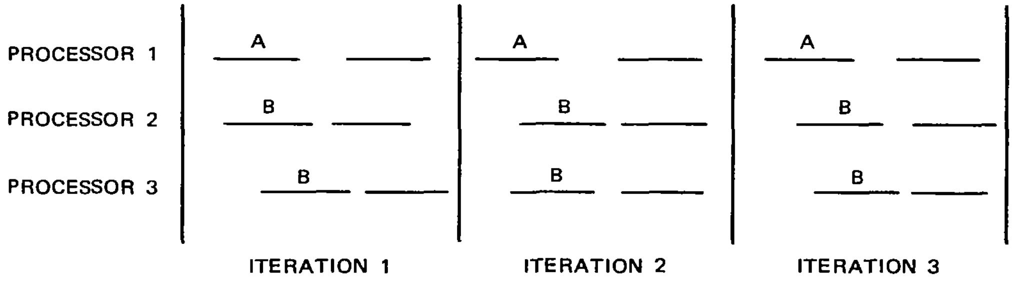


FIGURE IX-3 TIMING DIAGRAM FOR TWO COMMUNICATING PROCESSES

3 _after_ A completes. If a single storage area is used to hold the results of A, the programs running B in processors 2 and 3 will read different input values. Because these inputs are subject to voting, a difficult arises.

The introduction of separate buffers for odd and even iterations allows programs in different processors to read the same data regardless of their relative positions in the iteration frame. In the example situation, the results of the first iteration are placed in the odd buffer and those of the second iteration are placed in the even buffer. During the second iteration frame, both processors running B take their input from the odd buffer.

The specifications for the buffer module are largely self-explanatory. The module's chief function is the OV-function

```
read(procl, proc2, bus, taskl, task2, parity, offset);
```

read is called by the program running task2 in proc2 to obtain input from the program running taskl in procl. If parity is TRUE, the appropriate even-buffer in procl is read; otherwise, the odd buffer is read. The V-functions connected-r and connected-t model the necessary bus switiching. By convention, bus 0 designates the internal connection of a processor to its own memory.

```
MODULE buffer

                DECLARATIONS
        TYPE
                PROC = EXTERNAL DESIGNATOR
                TASK = EXTERNAL DESIGNATOR
                MACHINEWORD = EXTERNAL DESIGNATOR
        END-TYPE

        integer offset, length, bus, number
        boolean b, parity
        PROC proc, proc1, proc2
        TASK task1, task2, task
        MACHINEWORD word

                END-DECLARATIONS

                PARAMETERS

        integer max-buff ;maximum number of buffers allowed in a processor
        integer max-buff-size  ;maximum size of a buffer
        integer numb-busses  ;number of busses in system

                END-PARAMETERS


                EXCEPTIONS

        MACRO no-buffer(proc,task1,task2) = b
                not 'buffs-exist(proc,task1,task2)'

        MACRO out-of-bounds(proc,task1,task2,offset) = b
                offset >= 'buffs-size(proc,task1,task2)'

        MACRO too-many-buffers(proc) = b
          cardinality{<proc,task1,task2> | 'buffs-exist(proc,task1,task2)'}
                        > max-buff

        MACRO buffer-too-long(length) = b
                length > max-buff-size

        MACRO bad-bus(bus) = b
                bus > numb-busses or bus < 0 or (bus = 0 and not proc1 = proc2)

        MACRO same-proc(proc1,proc2) = b
                proc1 = proc2

                END-EXCEPTIONS
```

FUNCTIONS

```
VFUN connected-r(proc) = bus
        (; indicates which bus proc is connected to for receiving data)
        HIDDEN
        INITIALLY undefined

VFUN connected-t(bus) = proc
        (; indicates which proc bus is connected to for transmitting data)
        HIDDEN
        INITIALLY undefined

VFUN buff-mem-odd(proc,task1,task2,offset) = word
        (; stores words in "odd" buffer for transmission from
                task1 to task2)
        HIDDEN
        INITIALLY undefined

VFUN buff-mem-even(proc,task1,task2,offset) = word
        (; stores words in "even" buffer for transmission from
                task1 to task2)
        HIDDEN
        INITIALLY undefined

VFUN buffs-exist(proc,task1,task2) = b
        (; indicates whether buffers exist in proc for the
                transmission of data from task1 to task2)
        INITIALLY undefined

VFUN buffs-size(proc,task1,task2) = length
        (; indicates size of buffers in proc for transmission of data
                from task1 to task2)
        EXCEPTIONS
        no-buffer(proc,task1,task2)
        END-EXCEPTIONS
        INITIALLY undefined

OFUN create-buffers(proc,task1,task2,length)
        (; establisthes buffers of size length in proc in which task1
                will deposit data for task2)
        EXCEPTIONS
        too-many-buffers(proc)
        buffer-too-long(length)
        END-EXCEPTIONS
        EFFECTS
                buffs-exist(proc,task1,task2)
                buffs-size(proc,task1,task2) = length
                forall i (0 <= i <= length) implies
                   buff-mem-odd(proc,task1,task2) = 0
                forall i (0 <= i <= length) implies
                    buff-mem-even(proc,task1,task2) = 0
        END-EFFECTS
```

```
OFUN write(proc,task1,task2,parity,offset,word)
        (; called by task1 to deposit data into the appropriate buffer
                for task2)
        EXCEPTIONS
        no-buffer(proc,task1,task2)
        out-of-bounds(proc,task1,task2,offset)
        END-EXCEPTIONS
        EFFECTS
                if parity then buff-mem-even(proc,task1,task2,offset) = word
                        else buff-mem-odd(proc,task1,task2,offset) = word
        END-EFFECTS


OVFUN read(proc1,proc2,bus,task1,task2,parity,offset) = word
        (; called by task2 running in proc2 to receive data from
                the appropriate buffer deposited by task1 in proc1)
        EXCEPTIONS
        no-buffer(proc1,task1,task2)
        out-of-bounds(proc1,task1,task2,offset)
        bad-bus(bus)
        END-EXCEPTIONS
        EFFECTS
                if not bus = 0 then connected-r(proc2) = bus
                        and connected-t(bus) = proc1
                if parity then word = 'buff-mem-even'(proc1,task1,task2,offset)
                  else word = 'buff-mem-odd'(proc1,task1,task2,offset)
        END-EFFECTS

OFUN delete-buffers(proc,task1,task2)
        (; deletes the buffer in proc for the transmission of data
                from task1 to task2)
        EXCEPTIONS
        no-buffer(proc,task1,task2)
        END-EXCEPTIONS
        EFFECTS
                buffs-exist(proc,task1,task2) = false
                buffs-size(proc,task1,task2) = undefined
                buff-mem-even(proc,task1,task2, offset) = undefined
                buff-mem-odd(proc,task1,task2, offset) = undefined
        END-EFFECTS
                END-FUNCTIONS

                END-MODULE
```

Dispatcher

The primary role of the dispatcher is to store task schedules and dispatch tasks as dictated by the currently applying schedule and by external events.

The dispatcher responds to the passage of time in determining the task to be dispatched. There are two clocks pertinent to this machine: a high speed clock, as represented by the O-function timer, and a slower clock as represented by the O-function clock-tick. Each of these timing O-functions is assumed to be called by a separate independent process that can operate asynchronously with the other system tasks. That is, these clocks are treated like interrupt signals.

The interval between successive clock-ticks is called a frame. The fastest tasks are dispatched once every frame; slower tasks are dispatched every n-th frame, $n > 1$. As previously noted, the dispatcher handles two types of tasks: scheduled and priority. The scheduled tasks run to completion every time they are dispatched. A task calls the O-function job-complete to indicate that is has completed execution. Each task is also given a maximum time for execution, as measured by calls on timer. The V-function max-task-time records the maximum allowed time, and the V-function time-current-task records the remaining execution time. If a task overruns it is undispatched and the next task is dispatched. The information that the dispatcher needs about scheduled tasks is provided by the LE calling the O-function add-regularly-scheduled task. Besides setting the value for max-task-time, this function also makes known the initial status of a task (e.g., entry point, initial value of registers) to the dispatcher. The actual schedule itself is given to the dispatcher by calling the O-function add-regular-schedule. A schedule is a circular list of scheduled tasks, with a pointer identified by next-selected-element.

Priority tasks are only eligible to be dispatched when all of the scheduled tasks have completed their execution in a frame. The parameter gpt, when it appears in a schedule indicates that a priority task is to be dispatched. A clock tick occurring during the execution of a priority task signifies that the status of the currently executing priority task is to be saved, and the next scheduled task is to be dispatched. When a priority task completes its execution, or its execution time exceeds the value of max-task-time another priority task is considered for dispatching. The priority task selected is the task of the highest priority, as reflected by the value of the V-function, priority, such that the time since its last execution exceeds the desired period for that task, as reflected by the value of the V-function period-priority. Information about priority tasks is given to the dispatcher via the O-function add-priority-task.

The dispatcher, using the V-function iter-count, records the number of iterations completed by each task.

```
MODULE dispatcher

        DECLARATIONS

TYPE
        TASK = DESIGNATOR
        TIME = DESIGNATOR
        MACHINEWORD = DESIGNATOR
END_TYPE

integer posint
boolean b
TIME time, time1, time2
TASK task
MACHINEWORD word
CIRCULAR-LIST  task-list
TUPLE-OF MACHINEWORD word-tuple
ONE-OF {regular, priority} kind-of-task

        END-DECLARATIONS


        PARAMETERS

TASK le  (; local executive task)
TASK gpt (; generic priority task-- GPT is the "current-task"
                when a specific priority task is to be
                scheduled)
TASK null-task (; the empty priority task used to fill in when no
                real task is to be scheduled)
TUPLE-OF MACHINEWORD zero-tuple (; tuple consisting of all zero words)
TUPLE-OF MACHINEWORD status-le (; initial status of local executive)
integer pn (; priority level of null-task )
integer max-scheduled-tasks (; maximum number of
                                        regularly scheduled tasks)
integer max-priority-tasks (; maximum number of priority tasks)
integer status-length (; number of machine words
                                        comprising status of any task)
TIME max-time-le (; maximum execution time for local executive)
TIME max-time-null-task (; maximum execution time for null task)

        END-PARAMETERS


        DEFINITIONS

dispatch-next-regular-task
        current-task = currently-selected-element('task-list')
        c = advance-selector('task-list')
        time-current-task =
           'max-task-time(currently-selected-element('task-list'))'
        status-current-task = 'initial-status-task(current-task)'
```

```
dispatch-interrupted-priority-task
        current-task = currently-selected-element('task-list')
        c = advance-selector('task-list')
        time-current-task = 'time-interrupted-task'
        status-current-task = 'status-interrupted-task'

save-status-current-priority-task
        time-interrupted-task = 'time-current-task'
        status-interrupted-task = 'status-current-task'

dispatch-next-priority-task
        let s = { task | task member-of 'priority-task-set' and
                'time-to-next-exec(task)' = 0 }
        exists task1
                task1 member-of s
                forall task2 task2 member-of s implies
                        'priority(task1)' <= 'priority(task2)'
                current-priority-task = task1
                time-current-task = 'max-task-time(task1)'
                status-current-task = 'initial-status-task(task1)'
                time-to-next-exec(task1) = 'period-priority(task1)'

        END-DEFINITIONS


        EXCEPTIONS

MACRO no-task(task) = b
        not task member-of task-set and
        not task member-of priority-task-set

MACRO task-is-gpt(task) = b
        task = gpt

MACRO not-a-priority-task(task) = b
        not task member-of priority-task-set

MACRO task-already-known(task) = b
        task member-of task-set or task member-of priority-task-set

MACRO too-many-regular-tasks = b
        cardinality(task-set) >= max-scheduled-tasks

MACRO tuple-wrong-length(word-tuple) = b
        not length(word-tuple) = status-length
```

```
MACRO too-many-priority-tasks = b
        cardinality(priority-task-set) >= max-priority-tasks

MACRO task-not-regular(task) = b
        not task member-of task-set

MACRO not-current-task(task) = b
        not task = 'current task'

MACRO null-task(task) = b
        task = null-task

MACRO next-task-not-known = b
        not next-selected-task('task-list') member-of 'task-set'

MACRO current-task-not-priority = b
        not 'current-task' = gpt
MACRO next-task-priority = b
        next-selected-element('task-list') = gpt

        END-EXCEPTIONS


        FUNCTIONS

VFUN task-set = s
        (; set of all regular tasks assigned to dispatcher)
        EXCEPTIONS
        END-EXCEPTIONS
        INITIALLY s = {le, gpt}

VFUN task-list = c
        (; circular list of regularly-scheduled tasks)
        EXCEPTIONS
        END-EXCEPTIONS
        INITIALLY c = NEW circular-list(le,gpt)

VFUN max-task-time(task) = time
        (; maximum time allowed for execution of a scheduled task)
        EXCEPTIONS
        no-task(task)
        task-is-gpt(task)
        END-EXCEPTIONS
        INITIALLY max-task-time(task) =
                if task = le then max-time-le-task
                    else if task = null-task then max-time-null-task
                                else undefined

VFUN current-task = task
        (; currently dispatched scheduled task)
        EXCEPTIONS
        END-EXCEPTIONS
        INITIALLY current-task = le
```

```
VFUN time-current-task = time
        (; allowable time remaining for currently dispatched task)
        EXCEPTIONS
        END-EXCEPTIONS
        INITIALLY time = time-le

VFUN status-current-task = word-tuple
        (; status (values of program counter and other registers)
              . of currently dispatched task)
        EXCEPTIONS
        END-EXCEPTIONS
        INITIALLY status-current-task = status-le

VFUN initial-status-task(task) = word-tuple
        (; initial status of each task)
        EXCEPTIONS
        no-task(task)
        END-EXCEPTIONS
        INITIALLY   initial-status-task(le) = status-le
                    initial-status-task(null-task) = zero-tuple

VFUN status-interrupted-task = word-tuple
        (; holds status of interrrupted priority job)
        EXCEPTIONS
        END-EXCEPTIONS
        INITIALLY status-interrrupted-task = zero-tuple

VFUN current-priority-task = task
        (; gives identity of currently executing or
                                    interrupted priority task)
        EXCEPTIONS
        END-EXCEPTIONS
        INITIALLY task = null-task

VFUN time-interrupted-task = time
        (; remaining execution time for interrupted priority task)
        EXCEPTIONS
        END-EXCEPTIONS
        INITIALLY time = 0

VFUN overrun-tasks = < s1, s2 >
        (; indicates tasks which have overrun their allotted times;
         s1 is set of scheduled tasks; s2 is set of priority tasks)
        EXCEPTIONS
        NO-EXCEPTIONS
        INITIALLY s1 = {}
                  s2 = {}

VFUN priority(task) = posint
        (; priority level of priority task--smaller values indicate
                higher priority)
        EXCEPTIONS
        not-a-priority-task(task)
        END-EXCEPTIONS
        INITIALLY priority(null-task) = pn
```

```
VFUN period-priority(task) = time
        (; minimum scheduling frequency for priority task
                measured in clock ticks)
        EXCEPTIONS
        not-a-priority-task(task)
        END-EXCEPTIONS
        INITIALLY time = 0

VFUN time-to-next-exec(task) = time
        (; minimum allowable time to next dispatching
                                            of priority task task)
        EXCEPTIONS
        not-a-priority-task(task)
        END-EXCEPTIONS
        INITIALLY time-to-next-exec(null-task) = 0

VFUN iter-count(task) = posint
        (; indicates the number of iterations completed by task)
        EXCEPTIONS
        no-task(task)
        END-EXCEPTIONS
        INITIALLY iter-count(le) = 0
                iter-count(null-task) = 0

OFUN add-regularly-scheduled-task(task,time,word-tuple)
        (; makes new task known to dispatcher--information concerning
                maximum execution time and initial status are passed)
        EXCEPTIONS
        task-already-known(task)
        too-many-scheduled-tasks
        tuple-wrong-length(word-tuple)
        END-EXCEPTIONS
        EFFECTS
                task-set = 'task-set' union {task}
                max-task-time(task) = time
                initial-status-task(task) = word-tuple
                iter-count(task) = 0
        END-EFFECTS

OFUN delete-scheduled-task(task)
        (; removes regularly scheduled task task)
        EXCEPTIONS
        task-not-regular(task)
        END-EXCEPTIONS
        EFFECTS
                task-set = 'task-set' - {task}
                max-task-time(task) = undefined
                initial-status-task(task) = undefined
                inter-count(task) = undefined
        END-EFFECTS
```

```
OFUN add-regular-schedule(c)
        (; give the schedule of regularly scheduled tasks as the
                circular list c)
        EXCEPTIONS
        END-EXCEPTIONS
        EFFECTS
                task-list = c
        END-EFFECTS

OFUN add-priority-task(task,posint,time1,time2,word-tuple)
        (; makes a new priority task with priority level posint
                maximum execution time time1, minimum time between
                executions time2, and initial status word-tuple
                known to the dispatcher)

        EXCEPTIONS
        task-already-known(task)
        too-many-priority-tasks
        tuple-wrong-length(word-tuple)
        END-EXCEPTIONS
        EFFECTS
                priority-task-set = 'priority-task-set' union {task}
                max-task-time(task) = time1
                initial-status-task(task) = word-tuple
                iter-count(task) = 0
                priority(task) = posint
                period-priority(task) = time2
                time-to-next-exec(task) = 0
        END-EFFECTS

OFUN delete-priority-task(task)
        (; removes priority task task)
        EXCEPTIONS
        not-a-priority-task(task)
        END-EXCEPTIONS
        EFFECTS
                priority-task-set = 'priority-task-set' - {task}
                max-task-time(task) = undefined
                initial-status-task(task) = undefined
                iter-count(task) = undefined
                priority(task) = undefined
                period-priority(task) = undefined
                time-to-next-exec(task) = undefined
        END-EFFECTS

OFUN assign-iter-count(task,posint)
        (; assigns iteration count posint to task)
        EXCEPTIONS
        no-task(task)
        END-EXCEPTIONS
        EFFECTS
                iter-count(task) = posint
        END-EFFECTS
```

```
OFUN job-complete(task)
        (; called by currently dispatched task on completing
           execution; a new task is then dispatched-- if task is a
           priority task, the new task will be as well.  if task is
           a regularly-scheduled task, the new task may be either
           scheduled or priority.  we assume that null-task never
           calls job-complete.)
        EXCEPTIONS
        null-task(task)
        not-current-task(task)
        next-task-not-known
        END-EXCEPTIONS
        EFFECTS
                iter-count(task) = 'iter-count(task)' + 1
                if not 'current-task' = gpt
                   and not next-selected-task('task-list') = gpt
                        then dispatch-next-regular-task
                else  if not 'current-task' = gpt  and
                          next-selected-task = gpt
                        then dispatch-interrupted-priority-task
                else dispatch-next-priority-task
        END-EFFECTS

OFUN clock-tick
        (; signals interruption of priority task and subsequent
            dispatching of regularly-scheduled task)
EXCEPTIONS
        current-task-not-priority
        next-task-priority
        next-task-not-known
        END-EXCEPTIONS
        EFFECTS
                forall task  task member-of 'priority-task-set' and
                                   not 'time-to-next-exec(task)' = 0
                              implies
                                time-to-next-exec(task) =
                                   'time-to-next-exec(task)' - 1
                save-status-current-priority-task
                dispatch-next-scheduled-task
        END-EFFECTS
```

```
OFUN timer
        (; decrements time remaining for current task, logs
           error and dispatches next task if overrun occurs)
        EXCEPTIONS
        next-task-not-known
        END-EXCEPTIONS
        EFFECTS
                time-current-task = 'time-current-task' - 1
                if time-current-task = 0 then
                    if 'current-task' = gpt then
                        overrun-tasks[2] =
                         'overrun-tasks'[2] union {'current-priority
                                                          -task'}
                        else  overrun-tasks[1] =
                            'overrun-tasks'[1] union {'current-task'}
                if not 'current-task' = gpt and
                    not next-selected-element('task-list') = gpt then
                        dispatch-next-scheduled-task
                else if not 'current-task' = gpt
                         and next-selected-element('task-list') = gpt
                                then
                                dispatch-interrupted-priority-task
                else dispatch-next-priority-task
        END-EFFECTS
```

The Reader-Voter

The reader-voter module provides the means of comparing results of different processors working on the same task. In the present design, it is the only mechanism for detecting failures, including those uncovered during diagnosis. The reader-voter is therefore at the heart of SIFT's fault-tolerance machinery.

The reader-voter is also the lowest module in the system in which processor and bus failures are explicitly modeled. The reason is that this module is conceptually the lowest point at which errors are introduced. From the point of view of the global executive, if the reader-voter has not recorded a voting-discrepancy, no fault has occurred--even if certain busses and processors have in reality failed. One must bear in mind, of course, that the reader-voter, like all of SIFT's functions, is actually distributed among the processors of the system--the global executive compares results obtained by reader-voters in all modules and is aware of the possibility of a fault affecting some processor's reader-voter program.

The central focus of the module is the OV-function vote-read(proc task offset). This function is used by the program associated with the task task running in processor proc. The effect of vote-read is to read (via the bus network) the contents of the virtual address <task, offset> of every processor performing the task task, and then vote on the results. If some value receives a simple majority, that value is returned; otherwise, the flag fatal-error is set, and undefined is returned as the value of the call. In either case, unless the vote is unanimous, the flag error-detected is set, and the details of the disagreement are logged.

The outcome of a call on vote-read depends on a number of factors. It clearly depends on whether and which processors executing task are faulty at the time. It similarly depends on which busses used in communicating with these processors are faulty. Naturally, the outcome also depends on what the polled values actually are, right or wrong.

The other V- and O-functions in the module are used to model these considerations. The most important of these are the V-functions proc-bus-assignments, fault, correct-read, and read.

Proc-bus-assignments is a V-function of two arguments, proc and task. For each processor and each task executed within that processor, it stores the information as to what busses are to be used in reading values from the other processors executing the task. This information is represented as a set of PAIRs. Each pair has two parts: a proc part designating a processor, and a bus part designating the bus to be used in reading from that processor. It might be noted that the set pro-bus-assignments (proc task) may contain a pair whose processor component is proc itself--in other words, a processor may wish to read from itself over a bus. A diagnostic routine, for example, might need this capability.

The V-function fault is used to keep track of faults in the hardware. For particular values of its arguments procl, proc2, bus, task, and offset, it returns TRUE or FALSE depending on whether or not a fault exists that impacts a read by procl, using bus bus, of the virtual location <task, offset> in proc2. Note that this V-function is general enough to model a complete breakdown of a proc or bus. If for example, bus 1 is severed, fault returns true on all legitimate combinations of arguments for which bus = 1. Because fault is not intended to be visible outside the module, it is declared HIDDEN. Another HIDDEN V-function, correct-read (task offset), returns the value one would expect to find in the virtual location <task, offset> of a non-faulty processor working on task. Like fault, correct-read is abstract in the sense that it is not actually implemented in the SIFT software. It is defined, rather, in terms of an ideal processor.

The V-function read(procl, proc2, bus, task, offset) delivers the result of a read by procl, over bus bus, of the location <task, offset> in proc2. If a fault condition exists which affects that read (as determined by the V-function fault), an undetermined value is returned. Otherwise, the correct value correct-read (task offset) is returned. Note that the value of read is completely determined by the values of fault and correct-read; read is therefore a DERIVED V-function.

```
MODULE reader-voter

        DECLARATIONS

TYPE
        PROC = DESIGNATOR
        TASK = DESIGNATOR
        MACHINEWORD = DESIGNATOR
        PAIR = STRUCTURE(PROC:  proc, integer:  bus)
END-TYPE

PROC proc, proc1, proc2
SET-OF PROC sp
TASK task, task1, task2
SET-OF TASK st
PAIR pair, pair1
SET-OF PAIR setpairs
MACHINEWORD word, word1
BAG-OF MACHINEWORD votes
boolean b
integer bus, offset

        END-DECLARATIONS


        PARAMETERS

TASK le (; local executive task)
integer max-tasks (; maximum permitted number of tasks in one processor)
integer max-busses (; maximum number of busses)
integer max-offset(task) (; largest allowed offset associated with task)

        END-PARAMETERS


        DEFINITIONS

MACRO majority-opinion(proc task offset) = word
        LET votes =
                BAG {word1 | exists pair
                        pair member-of 'proc-bus-assignments(proc task)'
                        and word = 'read(proc pair.proc pair.bus task
                                                       offset)'      }

        if exists word 1
                        word1 member-of votes
                and     multiplicity(word1, votes) >
                                (1/2)cardinality(votes)
          then word = word1
        else word = undefined
```

```
MACRO dissenting-pairs(proc task offset) = setpairs
        if majority-opinion(proc task offset) = undefined
          then setpairs = 'proc-bus-assignment(proc task)'

        else
         setpairs =
           {pair | pair member-of 'proc-bus-assignments(proc task)'
                and not  'read(proc pair.proc pair. bus task offset)'
                      = majority-opinion(proc task offset)      }

        END-DEFINITIONS


        EXCEPTIONS

MACRO task-not-in-proc(proc task) = b
        not task member-of 'task-set(proc)'

MACRO no-proc-bus-assignment(proc1 proc2 bus task) = b
        not exists pair
                         pair member-of 'proc-bus-assignment(proc1 task)'
                    and  pair.proc = proc2
                    and  pair.bus = bus

MACRO bad-offset(task offset) = b
        offset < : or offset > max-offset(task)

MACRO not-assigned(task) = b
        not exists proc
                         task member-of 'task-set(proc)'

MACRO bad-assignment(proc task setpairs) = b
        exists pair
                 pair member-of setpairs
            and  (pair.bus > maxbusses or
                         exists pair 1
                                 pair1 member-of setpairs
                           and   (pair.bus = pair1.bus
                                         or pair.proc = pair1.proc))

MACRO too-many-tasks(proc) = b
        cardinality( 'task-set(proc)'  ) >= max-tasks

        END-EXCEPTIONS


        FUNCTIONS

VFUN task-set(proc) = st
        (; set of tasks assigned to processor proc)
        INITIALLY st = { le }
```

```
VFUN proc-bus-assignments(proc task) = setpairs
         (; for each proc and task, yields set of PAIRS--one pair
            for each other processor working on that task; the first
            component of each pair names the processor, the second gives
            the bus assignment for reading from that processor)
        EXCEPTIONS
        task-not-in-proc(proc task)
        END-EXCEPTIONS
        INITIALLY undefined

VFUN fault(proc1 proc2 bus task offset) = b
        (; indicates whether or not a fault exists that impacts a read
           by proc1 using bus bus of the memory of proc2 at the
           location associated with task, offset)
        HIDDEN
        EXCEPTIONS
        no-proc-bus-assignment(proc1 proc2 bus task)
        bad-offset(task offset)
END-EXCEPTIONS
        INITIALLY false

VFUN correct-read(task ofset) = word
        (; result that one would expect from a non-faulty module)
        HIDDEN
        EXCEPTIONS
        not-assigned(task)
        bad-offset(task offset)
        END-EXCEPTIONS
        INITIALLY undefined

VFUN read(proc1 proc2 bus task offset) = word
        (; result of proc1's reading of location associated with task
           and offset in proc2 using bus bus)
        DERIVED
        DERIVATION
                if fault(proc1 proc2 bus task offset) then
                                                  word = undetermined
                else word = correct-read(task offset)

OVFUN vote-read(proc task offset) = word
        (; returns majority vote on value associated with task and
           offset.  if vote is not unanimous, disagreements are logged.
           if no majority exists, returns undefined and sets
           fatal-error flag)
        EXCEPTIONS
        task-not-in-proc(proc task)
        bad-offset(task offset)
        END-EXCEPTIONS
        EFFECTS
                word = majority-opinion(proc task offset)
                if word = undefined then fatal-error(proc) = true
                disagreement-set(proc) =
                                dissenting-pairs(proc task offset)
        END-EFFECTS
```

```
OFUN error-detected(proc) = b
        (; flag indicating that disagreement exists)
        DERIVATION
                not disagreement-set(proc) = {}

VFUN fatal-error(proc) = b
        (; flag that indicates lack of a majority)
        INITIALLY false

OFUN assign-task(proc task setpairs)
        (; assigns new task to proc- setpairs indicates the other
           processors working on that task and the busses to be used
           in reading from them)
        EXCEPTIONS
        bad-assignment(proc task setpairs)
        too-many-tasks(proc)
        END-EXCEPTIONS
        EFFECTS
                task-set(proc) = 'task-set(proc)' Union {task}
                proc-bus-assignments(proc,task) = setpairs
        END-EFFECTS

OFUN delete-task(proc task)
        (; deassigns task to proc)
        EXCEPTIONS
        task-not-in-proc(proc task)
        END-EXCEPTIONS
        EFFECTS
                task-set(proc) = 'task-set(proc)' - {task}
                proc-bus-assignments(proc task) = setpairs
        END-EFFECTS

OFUN cause-fault(proc1 proc2 bus task offset)
        (; produces a fault that affects reads by proc1 over bus bus
           of location in proc2 associated with task, offset)
        HIDDEN
        EXCEPTIONS
        bad-offset(task offset)
        END-EXCEPTIONS
        EFFECTS
                fault(proc1 proc2 bus task offset) = true
        END-EFFECTS
```

```
OFUN change-correct-read(task offset word)
        (; updates correct-read to give correct result for current
           iteration)
        HIDDEN
        EXCEPTIONS
        not-assigned(task)
        bad-offset(offset)
        END-EXCEPTIONS
        EFFECTS
                corect-read(task offset) = word
        END-EFFECTS

        END-FUNCTIONS

END-MODULE
```

REFERENCE

1. D. L. Parnas, "A Technique for Module Specification with Examples," Comm. ACM, Vol. 15, No. 5, pp. 199-218 (May 1972).

APPENDIX A

MARKOV PROCESSES

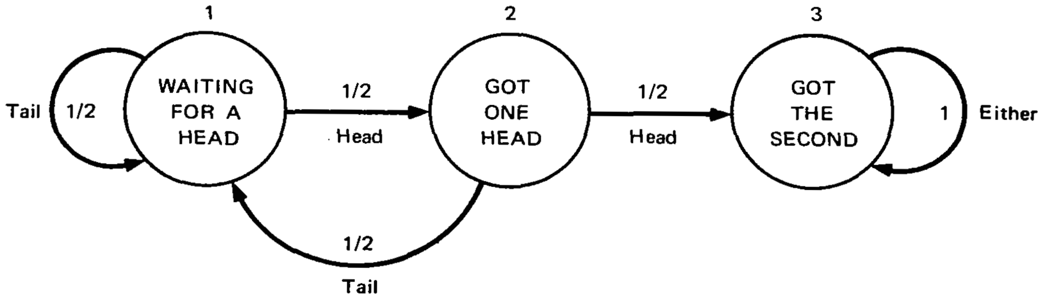
APPENDIX A

MARKOV PROCESSES

There is a simple, elegant, and powerful theory for handling models of the type considered in this report--providing that the following condition holds. The probability P_{ij} of making any state transition is independent of the manner in which state i was reached. (The transition probabilities are history independent). In this case the model is said to have the Markov property and to define a Markov process. Note that abnormal events of the transient or spontaneous failure type have this character, but failures in equipment that "wears out" like an automobile do not.

Markov processes can be treated as either discrete time or continuous time processes. In the former case, time is assumed to proceed in discontinuous "ticks" where one state transition must occur at each tick. For example, consider the following model for a coin flipping experiment in which the object is to obtain two "heads" in a row.

The state diagram is:



and the whole process can be described by the transition matrix of P_{ij}

$$P_{ij} = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix} = P$$

Given the probabilities of occupancy of a particular state at time n, say

$$p^{(n)} = (p_1, p_2, p_3)$$

then $p^{(n+1)}$ is given by $p^{(n)}P$. Or, given $p^{(0)}$, then $p^{(n)} = p^{(0)}p^n$. For example, with

$$p^{(0)} = (1\ 0\ 0) \quad .$$

One easily gets

$$p^{(1)} = \left(\frac{1}{2}\ \frac{1}{2}\ 0\right)$$

$$p^{(2)} = \left(\frac{1}{2}\ \frac{1}{4}\ \frac{1}{4}\right)$$

$$p^{(3)} = \left(\frac{3}{8}\ \frac{1}{4}\ \frac{3}{8}\right)$$

and so forth. The general solution for $p_i^{(n)}$ is known to be of the form

$$\text{Det} \begin{bmatrix} \left(\frac{1}{2} - \lambda\right) & \frac{1}{2} & 0 \\ \frac{1}{2} & (-\lambda) & \frac{1}{2} \\ 0 & 0 & (1 - \lambda) \end{bmatrix} = 0$$

yielding values $\lambda = 1, \frac{1}{4}(1 + \sqrt{5}), \frac{1}{4}(1 - \sqrt{5})$.

A closed form expression for the probability of occupancy of each state may now be easily obtained using the initial probabilities for $p^{(0)}$, $p^{(1)}$ and $p^{(2)}$. For example, the probability of being state 3 after n steps (tosses) is

$$p_3^{(n)} = 1 - \frac{4}{\sqrt{5}} \left[\left(\frac{1 + \sqrt{5}}{4} \right)^{n+2} - \left(\frac{1 - \sqrt{5}}{4} \right)^{n+2} \right]$$

We show the above example in some detail for comparison with the continuous-time formulation below.

A continuous-time Markov process may be derived as a limiting case of a discrete-time process in which the ticks of time become infinitesimal; however, we take a slightly different approach. Since we are less interested in particular transition probabilities than we are in probabilities of state occupancy, we derive an expression for the probability of being in a state q at time t via another limiting argument.

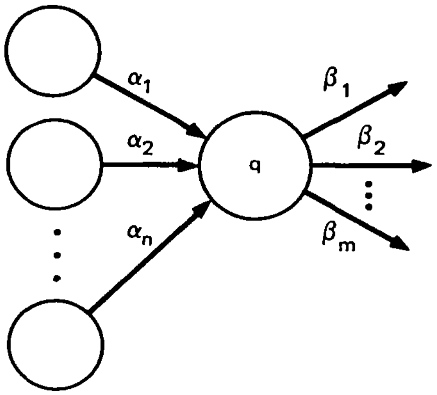
For this purpose we temporarily assume that the probability of transition from any state i to another state j is proportional to the time spent in state i for sufficiently small times. That is

$$P_{ij} = \rho \, \Delta t \quad , \quad \Delta t \rightarrow 0$$

where ρ is a constant with dimensions time^{-1} and value $0 \leq \rho < \infty$.

This assumption is necessarily true for uniformly distributed random stochastic events that occur at an average rate ρ independent of past history (the Markov property).

For any state q



we can then write

$$P_q(t + \Delta t) = \sum_{\substack{i=1 \\ i \neq q}}^{n} \alpha_i P_i(t)\ \Delta t + P_q(t) \left[1 - \Delta t \sum_{\substack{i=1 \\ i \neq q}}^{m} \beta_i \right]$$

or taking the limit, $\Delta t \rightarrow 0$

$$P_q'(t) = \sum_{i=1}^{n} \alpha_i P_i(t) - \sum_{i=1}^{m} \beta_i P_q(t) \qquad (i \neq q)$$

The above system of linear differential equations, together with an initial vector of state occupation probabilities, say $P_i = (1,0,0,0 \ldots 0)$, completely determines the state of the system for all time.

To observe the correspondence between this formulation and the discrete time case (and also to greatly facilitate solution of the system) one can take the Laplace transform of each equation. With transform variable S and $\beta_q = \sum_1^m \beta_i$ the above becomes

$$(S + \beta_q)\ P_q^*(S) = \sum_1^n \alpha_i P_i^*(S) \qquad (i \neq q)$$

or more neatly as the matrix equation $\overset{*}{P}(S) \times M = \overset{*}{P}(0)$, as follows.

$$\begin{bmatrix} P_1^*(S) \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ P_n^*(S) \end{bmatrix} \times \begin{bmatrix} (S + \beta_1) & - & \alpha_{12} - \alpha_{13} & - & \ldots\ \alpha_{1n} \\ -\alpha_{21} & & (S + \beta_2) - \alpha_{23} & - & \ldots\ \alpha_{2n} \\ \cdot & & & & \\ \cdot & & & & \\ \cdot & & & & \\ \cdot & & & & \\ -\alpha_{n1} & - & \alpha_{n2} \cdots & & (S + \beta_n) \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \cdot \\ \cdot \\ \cdot \\ 0 \end{bmatrix}$$

Observe that:

1. The differential equations above are similar but not identical to the Chapman-Kolmogorov equations which describe transition probabilities. Our equations describe occupation probabilities.

2. The transform technique moves us into a purely algebraic domain where approximations and limits, may be made before inverting to obtain solutions in the time-domain.

3. The general solution of the system of equations is given (as in the discrete case) by $P_q(t) = \Sigma\, a_i e^{\lambda_i t}$ where λ_i are the roots of the polynomial equation Det. M = 0.

4. Two limiting cases of behavior are immediately apparent:

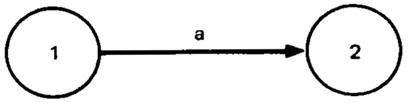
   a. As $t \rightarrow \infty$ $P_q(t) \rightarrow a_1 e^{\lambda_1 t}$ where λ_1 is the numerically largest eigenvalue.

   b. Since $P_q(S) = N_q(S)/D(S)$, a ratio of polynomials, we have that as $S \rightarrow \infty$ $P_q(S) \rightarrow A\, S^{-(k+1)}$ k + 0, 1, 2 . . . where A is a constant. Therefore as $t \rightarrow 0$, $P_q(t) \rightarrow \frac{A}{k!}\ t^k$

The latter limit theorem is one of several kinds of argument that can be used to deduce general features of a solution without actually obtaining it.

Examples:

A.



$$M = \begin{bmatrix} (S + a) & 0 \\ -a & S \end{bmatrix} \qquad \text{Det } M = S(S + a) \qquad \text{roots } 1, -a$$

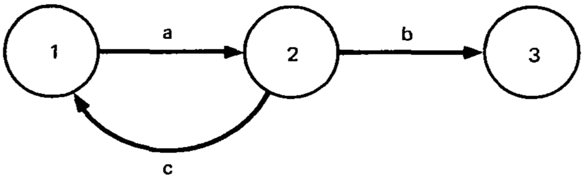
$$P_1^*(S) = \begin{bmatrix} 1 & 0 \\ 0 & S \end{bmatrix} \qquad \text{Det } M = \frac{1}{S + a}$$

$$P_2^*(S) = \begin{bmatrix} (S + a) & 1 \\ -a & 0 \end{bmatrix} \qquad \text{Det } M = \frac{a}{S(S + a)} = \frac{1}{S} - \frac{1}{S + a}$$

$$P_1(t) = e^{-at} \quad , \quad P_2(t) = 1 - e^{-at}$$

$P_1(t) + P_2(t) \equiv 1.$ (This is the simple exponential occupation distribution for state 1)

B.



$$M = \begin{bmatrix} (S + a) & -c & 0 \\ -a & (S + b + c) & 0 \\ 0 & -b & S \end{bmatrix} \text{Det } M = S(S + a)\ (S + b + c) - acS$$

$$\overset{*}{P}_3(S) = \frac{ab}{S(S + a)(S + b + c) - acS} \approx \frac{ab}{S^3} \quad \text{as } S \rightarrow \infty$$

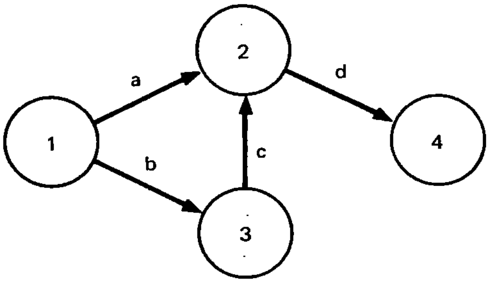
$$P_3(t) \approx \frac{1}{2}\, abt^2 \quad \text{as } t \rightarrow 0$$

There are opportunitis for automation of the solution process. Where the transition rates are numeric quantities, the explicit solutions are easily obtained by machine using linear equation solvers and eigenvalue routines. One can also use algebraic manipulation programs to obtain the parameterized expressions $P^*(S)$. For state graphs without cycles, the individual $P_i^*(S)$ are easily obtained by a "chain rule."

$$P_q^*(S) = \sum_{\text{input}} \alpha_i P_i^*(S) / \left(S + \sum_{\text{out}} \beta \right)$$

For example, no consideration of the associated matrix M was required in the system below.

c.



$$P_1^*(S) = 1 \cdot \frac{1}{(S + a + b)}$$

$$P_3^*(S) = bP_1^*(S)/(S + c)$$

$$P_2^*(S) = [aP_1^*(S) + cP_3^*(S)]/(S + d)$$

$$P_4^*(S) = dP_2^*(S)/S$$

Furthermore, questions concerning the "dependency of $P_2(t)$ on the value of a″ can be answered by taking the appropriate partial derivitives in the S domain. To illustrate

$$\frac{\partial}{\partial a} P_2^*(S) = P_1^*(S)?(S + d) = \frac{1}{(S + a + b)(S + d)}$$

or

$$\frac{\partial}{\partial a} P_2(t) \sim t \qquad \text{as } t \to 0$$

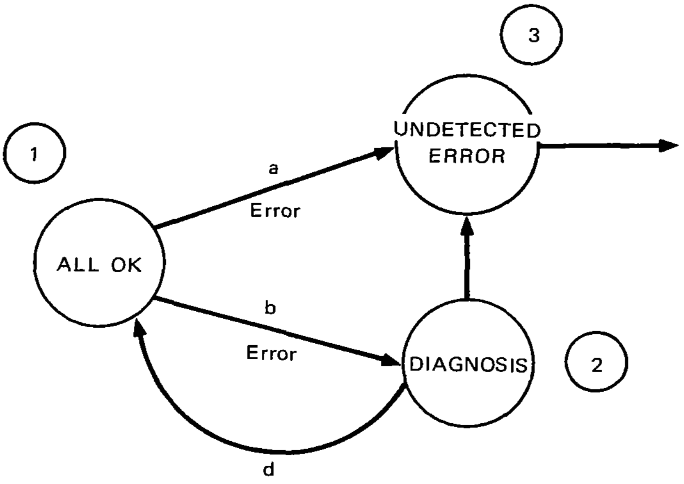
4. Incorporation of Non-Markov Behavior

The pure Markov model is an excellent approximation to the situation where all state transitions in the model correspond to stochastic (usually abnormal) events such as component failure or the onset of a transient. However, there are three other types of behavior that we would like to be able to handle and it seems particularly important to try to incorporate these events into a pure Marko\ model so that our powerful analysis techniques will still be applicable:

a. Processes of fixed duration (such as the running of a program) that occur either (1) periodically in time or (2) in response to another event.

b. Processes with nonexponential probability density functions, such as transients which may have a narrow distribution of duration times.

The two types of process in a. above might be called deterministic in the sense that they involve behavior that is definitely history-dependent. First consider the following situation:



Suppose there is a diagnostic check that is very brief in duration and is executed periodically with a low duty cycle. Starting in state 1 we have two possibilities. If an error occurs during a period when the diagnostic check is <u>not</u> running then we will go to state 3. If, on the other hand, the diagnosis routine happens to be operative when the error occurs then we find ourselves in state 2 where the error may or may not be detected. If it is, we return (in this simplified situation) to state 1 otherwise we go to state 3. Now suppose that the diagnostic routine runs for a period τ_1 and is invoked with period $\tau_2 >> \tau_1$. Further suppose that the actual error rate is ρ and that $\rho < 1/\tau_2$ (errors occur less frequently than the diagnosis period). Under these circumstances it is an excellent approximation to assume that errors are <u>completely uncorrelated</u> with the initiation of the diagnostic test. Therefore we may assign to the transition rates a and b the values

$$a = \rho \left(1 - \frac{\tau_1}{\tau_2}\right)$$

$$b = \rho \frac{\tau_1}{\tau_2}.$$

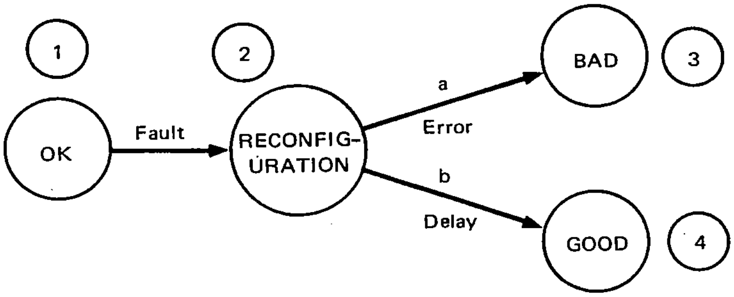
where, of course, $a + b = \rho$. Moreover, if the probability that error detection actually occurs when reaching state 2 is p, then the rates

$$c = p/\tau_1 \quad , \quad d = (1 - p)/\tau_1$$

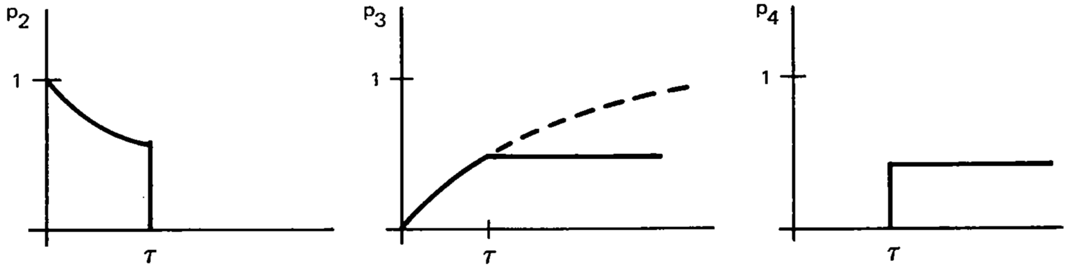
are a good approximation to actual transition probabilities. That is, with $p = 1$ one would expect on the average one transition from 2 to 3 in the τ_1, and this, by definition gives the corresponding transition rate C. Generally, the above treatment must be quite satisfactory so long as $\rho \ll 1/\tau_2 \ll 1/\tau_1$, as will frequently be the case since typical values for these quantities are

$$\begin{aligned} \rho &\sim 10^{-3} & \text{hour}^{-1} \\ 1/\tau_2 &\sim 10^{3} & \text{hour}^{-1} \\ 1/\tau_1 &\sim 10^{7} & \text{hour}^{-1} \end{aligned}$$

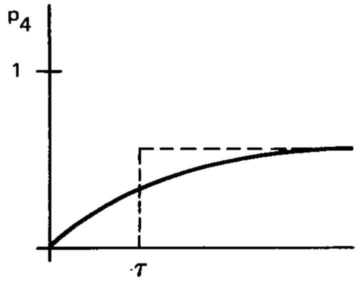
The second form of deterministic behavior we need to adequately handle is the case in which fixed delays occur. Consider the following situation:



Here we assume that the event of a fault when in state 1 causes the system to enter a state 2 in which a reconfiguration program is started up. This program runs for a fixed time τ. If another fault or error occurs during the running of the program, we enter an unsatisfactory state 3. Otherwise after time τ we enter a satisfactory state 4. The actual occupation probabilities for states 2, 3, and 4 are as shown below:

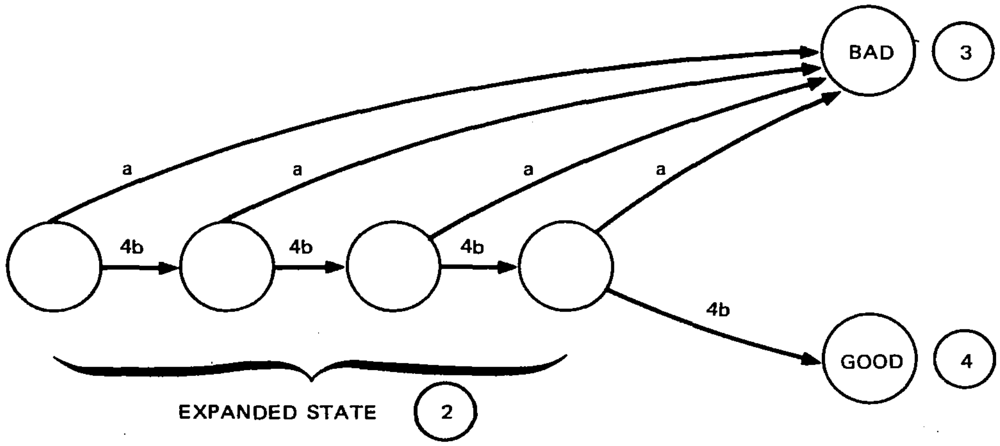


An attempt to approximate this situation by assigning a constant transition rate to b is not a very good strategy because p_4 would then appear as shown below:

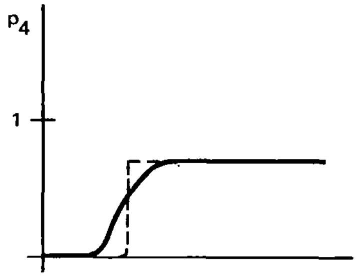


which would assign a relatively large probability to a transition from 2 to 4 during the period when this is not possible.

An artifice for improving the approximation is to replace state 2 by a chain of states, a sort of probabilistic delay line:



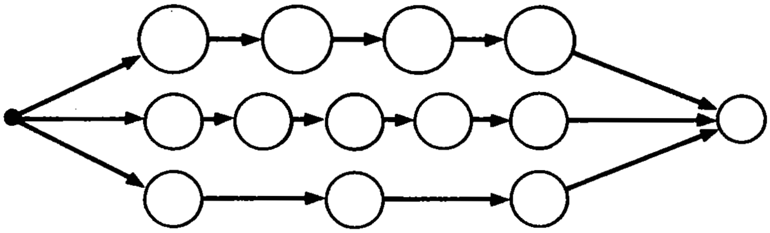
now the easily computed transform of the probability to be in state 4 is of the form $1/S(\beta/S + \beta)^n$ and in the time domain p_4 now appears as follows:



which is a much closer approximation to the desired behavior.

General Probability Distribution Function

In general, if we wish to approximate some probability distribution function (say behavior of a particular type of transient), it is always possible to obtain an arbitrarily close approximation by replacing the state having the odd behavior with a series-parallel combination of states having real positive transition rates as depicted below:



Construction of these approximations is much like the problem of synthesizing passive electrical networks having prescribed transfer characteristics. The "best" approximation to a given probability function for a fixed number of states would lead to the use of complex rather than real transition rates. This is an interesting possibility to consider in future research. To effectively employ such techniques we would need some theorems that say when it is safe to use an approximation that would lead to nonphysical occupation probabilities for some of the fictitious states involved in the synthesis.

1. Report No. NASA CR-3011	2. Government Accession No.	3. Recipient's Catalog No.

4. Title and Subtitle DESIGN STUDY OF SOFTWARE-IMPLEMENTED FAULT-TOLERANCE (SIFT) COMPUTER	5. Report Date June 1982
	6. Performing Organization Code
7. Author(s) J. H. Wensley, J. Goldberg, M. W. Green, W. H. Kautz, K. N. Levitt, M. E. Mills, R. E. Shostak, P. M. Whiting-O'Keefe, and H. M. Zeidler	8. Performing Organization Report No.
	10. Work Unit No.
9. Performing Organization Name and Address SRI International 333 Ravenswood Avenue Menlo Park, California 94025	11. Contract or Grant No. NAS1-13792
	13. Type of Report and Period Covered Contractor Report
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Washington, DC 20546	14. Sponsoring Agency Code

15. Supplementary Notes

Langley Technical Monitor: Nicholas D. Murray
Final Report

16. Abstract

This paper reports on a continuing effort to design a flyable SIFT computer that can demonstrate the feasibility of an integrated function, fault-tolerant computer in commercial aviation. The goals of the work reported in this paper are:

(1) to develop the SIFT design concept to a point at which its potential reliability may be evaluated with reasonable accuracy;

(2) to investigate alternate strategies for physical implementation, using available or specially designed components;

(3) to prove the correctness of the hardware and software designs; and

(4) to model the system and evaluate its effectiveness from a fault-tolerance point of view.

17. Key Words (Suggested by Author(s)) SIFT Fault tolerant Ultrareliability Reconfigurable computer system Multicomputer system	18. Distribution Statement Unclassified - Unlimited Subject Category 62

19. Security Classif. (of this report) Unclassified	20. Security Classif. (of this page) Unclassified	21. No. of Pages 252	22. Price A12

For sale by the National Technical Information Service, Springfield, Virginia 22161