N84-18953

NASA Contractor Report 172146

Development and Analysis of the Software Implemented Fault-Tolerance (SIFT) Computer

Jack Goldberg
Milton W. Green
William H. Kautz
Karl N. Levitt
P. Michael Melliar-Smith
Richard L. Schwartz
Charles B. Weinstock

SRI International
Menlo Park, California 94025

Contract NAS1-15428

February 1984

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

Table of Contents

List of Figures

List of Tables

I. Summary of Results

This chapter presents an overview of the SIFT computer system and its development. Successive sections describe technical goals, design approaches and solutions, development history and significant technical results.

A. GOALS OF THE SIFT DEVELOPMENT

This section describes the goals of the SIFT development and the primary criteria that guided its design.

1. Introduction

SIFT (Software-Implemented Fault Tolerance) is an experimental computer system designed to provide extremely reliable computing service for critical functions in advanced air transports. An example of such critical use is the control of dynamically unstable aircraft, designed for high energy efficiency. The SIFT design is intended to assure correct execution of aircraft control programs despite the occurrence of computer hardware malfunctions. The design does not address other sources of system unreliability, such as incorrect flight-control programs and faulty sensors, actuators, and communication channels, but techniques for program correctness were employed that have significant potential for the goal of highly reliable aircraft software.

The SIFT development was intended to apply techniques of fault-tolerant computer architecture to a very demanding aircraft application. Early in the effort, it became clear that the problem of estimating the reliability of machines designed for this application was beyond the state of the art, due to (1) the extremely low acceptable probability of system failure, which made life-testing totally impractical, and (2) the potentially high complexity of fault-tolerant systems, which made logical analysis very difficult. These characteristics will inevitably cast doubt on any reliability predictions for fault-tolerant systems, because the model used for the prediction either might not truly represent the real system, or it might be so highly simplified that the important parameters might be effectively unmeasurable. For this reason, project goals were extended to include development of new techniques for analyzing the reliability of fault-tolerant computer systems.

Fortunately, recent results in the theory and practice of program verification indicated that it is possible to design a fault-tolerant system to be provable, i.e., so that one can prove mathematically that the design is consistent with its formal specification. Applying this technique, one could verify that a given reliability model truly represents a proposed computer design. The use of program verification techniques also happened to be well suited to the architecture's software-intensive approach, which was selected for other reasons, such as compatibility with maximum use of standard hardware.

In the course of the work, techniques for verification of fault-tolerant computer designs were developed and successfully applied. We are encouraged to believe that the goal of designing provably fault-tolerant computer systems is achievable, and we submit that these techniques constitute at least as significant a contribution to reliable system technology as the SIFT architecture itself.

2. Design Criteria

The performance goals for SIFT were chosen to reflect the computing needs of advanced air transports. There were two main assumptions:

(1) Centralizing aircraft computing functions will help to avoid the wasteful addition of separate computers for each new increase in aircraft functionality, and

(2) Advanced air transports will require new levels of computing performance, e.g., for stability augmentation in fuel-efficient airframes, and new levels of reliability, resulting from the use of computers to control flight-critical functions.

The first assumption implies that the computer system should have good general-purpose characteristics and that it be capable of serving multiple aircraft processes concurrently. We note the recent trend towards a proliferation of computers on aircraft, in the form of micro-computers, embedded in aircraft subsystems. These tend to be cost-effective, but problems of resource sharing, system fault tolerance, and software compatibility have not been solved and represent an obstacle to system effectiveness.

The second assumption, which concerns both performance and reliability, had several consequences. In order to achieve adequate performance, the computer system must have relatively high computing speed and large memory capacity, compared with current single-purpose aircraft computers. In the initial feasibility study [25], data were gathered on possible future applications in order to determine ranges of future requirements for computer speed and memory capacity. The reliability goal was not so readily determined. The general guideline of the Federal Aviation Administration is that a catastrophic failure should be extremely unlikely, but this objective is not given a quantitative form. Based on a review of the literature on socially accepted risk levels, and informal discussions with NASA personnel familiar with industry practice, the reliability goal was taken as the requirement that the computer system should have a probability of failing to meet its critical computational requirements during a ten-hour flight of less than 10^9. This level of reliability is unprecedented, and represents a severe technological challenge. It was noted, however, that not all aircraft computations would require the same level of reliability.

While reliability for critical functions was the paramount objective, several other reliability properties were seen to have economic significance. First, any improvement in the computer system's capability for self-test and diagnosis, a natural side-benefit of fault-tolerant design, would be beneficial in reducing the cost of maintenance. Second, it is very advantageous to be able to dispatch an aircraft following a landing in a multi-leg journey, even though some failure may have occurred (of course, the reliability of the total flight must meet requirements based on total flight duration). Numerical goals for mean-time-to-repair and dispatch probability were not established.

To summarize, the SIFT computer system was developed to test the feasibility of using modern techniques for fault-tolerant design to meet the severe reliability and performance requirements of advanced air transports. The technical challenge implied by this goal was that several necessary techniques of design and analysis had not previously been brought together in a practical demonstration vehicle of the given scale of performance and reliability.

B. DESIGN APPROACHES AND SOLUTIONS

General aspects of the SIFT system will be discussed in nine sections: (1) basic redundancy issues, (2) reconfiguration, (3) the SIFT approach, (4) a hierarchical design view, (5) verification and testing, (6) proof of correctness, (7) software methodology, (8) operational support equipment, and (9) test environment.

1. Basic Redundancy Issues

In considering the goal of correct computation with a system failure rate of less than 10^{-10} per hour (10^{-9} per ten hours), the first question is whether this rate can be achieved by a conventionally designed computer, using the highest standards of construction. Given the need for several hundred integrated-circuit devices (at current integration levels), and a device failure rate of 10^{-7} permanent failures per hour for quality devices, this approach is easily seen to be hopeless, even without considering transient device failures and failures due to interconnections, power supplies, and other system components. The use of redundancy is clearly essential, but the proper form of redundancy is by no means obvious, since, in poor designs, the increased failure rate due to the added components may outweigh the benefits derived from the added fault-tolerant functionality.

Given the decision to use redundancy to improve reliability, one of the most fundamental issues is the choice of the level of complexity of the basic unit of redundancy. For example, should redundancy be applied at the level of a gate, a register, a central processing unit, or an entire computer? In general, application of redundancy requires the introduction of additional components, e.g., for selecting or combining redundant elements. Since these additional components introduce new sources of failures, it is necessary to balance the loss of resources (under error conditions) that results from using too large a unit of redundancy against the introduction of error resulting from the use of too small a unit of redundancy (which tends to increase the total level of redundancy).

In our preliminary study of alternatives, we recognized a strong trend toward increased device complexity in computer technology,. In order to be able to keep our design scheme relevant to several future technology stages, it was decided to take an entire computer (CPU, memory, I/O, etc.) as the major unit of redundancy. The same choice has been made in numerous subsequent research and development efforts.

2. Reconfiguration

In considering possible forms of redundancy, perhaps the simplest is the use of multiple identical channels, with some form of error masking at the system output interface. A typical form of error masking is majority voting, e.g., in which the output is determined by two-out-of-three, or three-out-of-five agreement (in general, for n channels, the output is defined as equal to the value presented identically by m or more channels, where m is the next integer greater than n/2, and is undefined if there is no such value). In this simple scheme, the system will fail when half the number of channels or more have failed. The fault tolerance capability of this scheme can be improved, at the cost of increased complexity, if faulty channels are removed from the system when they are recognized. For example, a five channel majority-voting system can degrade in steps so as to tolerate the loss of a total of three channels, since at each step, a single faulty processor will be out-voted by at least two nonfaulty processors; without the recognition and removal of faulty channels, a given channel system could survive the loss of only two channels. This advantage will be realized only if the reconfiguration scheme can be implemented with low complexity. The SIFT program essentially was based on the belief that the potential benefits of the recognition-and-removal scheme (more conventionally called reconfiguration) could be achieved in practice.

The basic paradigm for the reconfiguration approach is as follows:

- *Fault Isolation*: Partition and interconnect system elements so as to minimize the propagation of faults in the system.

- *Error Masking*: Combine the results of redundant computations in a way that prevents erroneous results from appearing at the system output.

- *Fault Detection and Diagnosis*: Detect evidence of a fault, and identify the responsible unit of redundancy.

- *Reconfiguration*: Reconfigure the system so that the faulty unit is logically removed from the set of computations that may affect system output.

- *Restoration*: Restore system data to a proper state, as needed.

This is essentially the paradigm for fault tolerance used in SIFT, in the FTMP computer of C.S. Draper Laboratory, and in other experimental machines (notably the pioneering JPL-STAR computer of the late 1960s).

3. The SIFT Approach

The point of departure for the SIFT development was in the approach used to implement the paradigm in contemporary hardware and software. The chosen approach was motivated by the following views:

- *Maximum Use of Standard Hardware*: Since component reliability tends to increase with maturity of production, increased intrinsic reliability should result from the use of standard components at all system levels.

- *Minimum Dependency on Shared Facilities*: Facilities such as power supplies, clocks and communication lines, which are usually shared by multiple elements in conventional computers, are points of vulnerability, and should either be replicated and not shared, or should be specially protected.

- *High-Level Programming*: Use of high-level programming languages in implementing system software is preferred, in order to enhance reliability and modifiability.

- *Design for Provability*: The design should be kept very simple, in order to allow design verification using current methods of proof of correctness.

These views strongly motivated the design of the SIFT architecture, which is based on the following primary features:

- The unit of hardware redundancy and reconfiguration is a whole computer, of conventional design.

- The only shared facilities are a multiple-path intercomputer communication network and a fault-tolerant primary power supply; clocks, secondary power supplies and aircraft I/O channels are provided in each computer.

- The primary function for fault detection and fault masking is majority voting, applied to task outputs.

- All steps of the fault-tolerance paradigm are implemented in programs written in a high-level programming language; the hardware functions are limited to computation, inter-processor communication, input-output and fault isolation (detailed descriptions of the hardware facilities that support these functions are presented in Chapter II).

Important secondary concepts in SIFT deal with multitasking, redundancy management, reconfiguration, and latent-fault detection. These are addressed as follows:

- Multitasking is provided by preassigning tasks to time segments in a frame-structured schedule.

- Flexible redundancy management is provided by tables that define the assignment of tasks to processors.

- Reconfiguration is provided by modifying the task assignment tables.

- Reconfiguration management is provided by a replicated task (called the Global Executive), whose state data are the task assignment table; the effect is to provide exactly the same treatment and protection to the reconfiguration function as to all other software functions.

- Latent fault-detection is accomplished by a continuous background task that exercises the hardware in a nonoutput mode.

There are other functions of a supporting nature, i.e., interprocessor communication, processor synchronization, and consistent replication of data, that might logically be considered as tertiary features, since they do not contribute to system fault tolerance. Nevertheless, they have a crucial system role and were indeed the source of considerable difficulty in design and debugging. Briefly, the technical issues for these functions are as follows:

- *Interprocessor Communication*:

  SIFT was originally designed to use a multiple-bus network; interprocessor paths within the network would be constructed at the demand of an individual processor, whenever it needs data that are located in another processor. In this scheme (described in [34]), the multiple busses are under continuous contention, and, since the processor demands are not precisely synchronized, each entry of a datum to each of the three or five copies of a program creates a separate service call on the bus system.

  Unfortunately, some worst-case conditions in busses and processors could cause abnormally long delays in resolving contentions for bus service. Providing adequate time to ensure safe communication in the event that these conditions occur would reduce bus data rate to an unacceptable level. This problem led to a change of interconnection schemes and protocols.

  The resulting design employs nonshared links among all processor pairs and calls for data to be transferred at the initiative of any originator of data, on a broadcast basis. The result is a much higher data-transfer rate than for the demand scheme, even without abnormal fault conditions.

- *Processor Synchronization*:

  Since voting in SIFT is performed only on task outputs, synchronization need only be accomplished on a task basis. In order to reduce common-mode failure, each SIFT processor has its own clock. The processors may be synchronized via adjustments to a time-of-day register, which is accessible by the CPU, and for the intended applications, only relative synchronization among the processors is needed. A new algorithm was developed to synchronize multiple processors in the presence of both clock drifts and clock failures.

In developing the algorithm, a certain anomalous behavior was seen to be possible, i.e., a given processor time might be seen as higher or lower than its time value by other processors (due to transmission ambiguities), with the result that slow processors might be adjusted to go yet slower, and fast processors to go yet faster. The algorithm developed deals satisfactorily with this anomaly.

- *Consistent Replication*:

  In SIFT, fault detection is accomplished by comparing the outputs of supposedly identical results of redundant computations at the time they are presented for voting. This is a valid test only if those processes received identical inputs. Most data in SIFT are distributed in triples or quintuples, in which a single error is always correctly interpretable. In certain cases, only a single version of data is present, e.g., at an input from a sensor. In such cases, identical copies of the data should be distributed as inputs to all redundant processes. This operation has the possible anomaly (associated with several realistic fault modes) that two processors may obtain different values, which may confuse subsequent fault-detection operations. For a single fault of this kind, at least four processors are required. A special algorithm, called Interactive Consistency, was developed and generalized for an arbitrary number of faults.

Detailed descriptions of the software facilities that support these functions are given in Chapter III.

4. A Hierarchical Design View

The several design features discussed may be viewed in a hierarchical framework, as in Figure I-1 (a similar hierarchical view will be presented in discussing the validation of the SIFT reliability model). In this presentation, assumptions are made at each level that are realized by functions at the lower levels. The *application* level is the view defined by the aircraft system engineer; it deals with aircraft functions, such as aircraft state sensing, control and actuation, as implemented in computer programs. The computer system engineer seeks the view of the second level, in which an *ideal computer* executes, without error, programs that implement the functions of the *application* level.

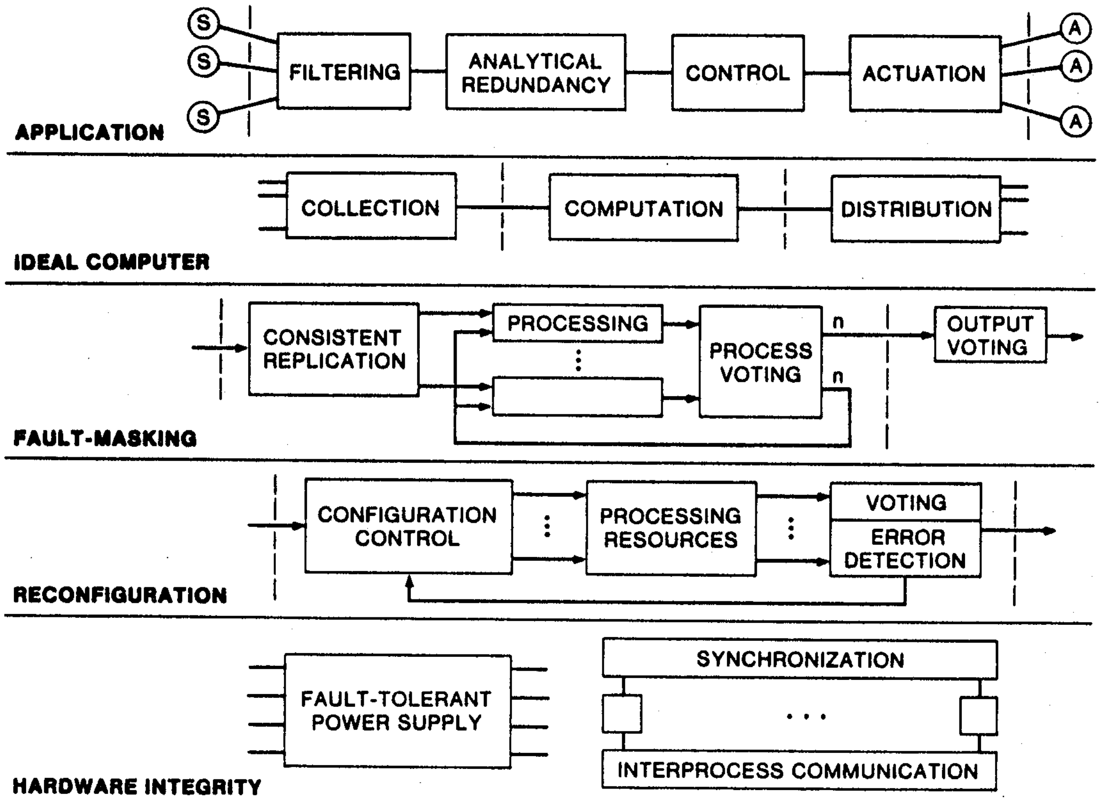


FIGURE 1-1 SIFT VIRTUAL-MACHINE HIERARCHY

At the *fault masking* level, redundant processors receive consistently replicated data and use voting to mask errors resulting from processor faults, in order to give the effect of the *ideal computer* assumed at the higher level. At the *reconfiguration* level, faulty processors are eliminated from the voting process in order to avoid the compromising of the voting function at the higher level that would result from multiple faults.

Finally, at the *hardware integrity* level, basic functions such as fault-tolerant power supply, processor synchronization, and interprocessor communication are provided to allow safe, uninterrupted communication among multiple processors.

SIFT may thus be seen as a basically fully-distributed computing system, which, by virtue of system software for synchronization and fault tolerance, acts as a single reliable computer for aircraft applications.

5. Verification and Testing

As mentioned previously, reliability estimation for extremely reliable fault-tolerant systems is not a well-developed art, due to the impracticality of life testing and the complexity of logical analysis. A strategy was developed for the reliability analysis of SIFT that had two main parts:

(1) Proof that the design was a true implementation of an abstract reliability model, and

(2) Measurement of reliability performance factors necessary to evaluate the model.

This strategy is illustrated schematically in Figure I-2

The reliability model used for SIFT is a Markov model of the processor set, in which each state represents some combination of good, faulty-active and faulty-inactive (reconfigured) processors. Various state transitions correspond to the occurrence of random hardware failures and to actions carried out by the system to accommodate those failures. The model thus constitutes a very high-level description of SIFT behavior. A simplified version, representing permanent failure modes only, is illustrated in Figure I-3 (the full model described in Chapter IV reflects transient behavior).

As shown in the scheme of Figure I-2, the design of the system is examined for consistency with the reliability model in order to verify correctness of the design. In the SIFT development, that examination, which attempts to establish that consistency with mathematical rigor, constructs a logical chain that proceeds from a high-level system model (composed of an external performance model and a reliability model), down through formal specifications and into the programs that comprise the executive system. For completeness, proof of the validity of assumptions about the hardware functionality would also be necessary.

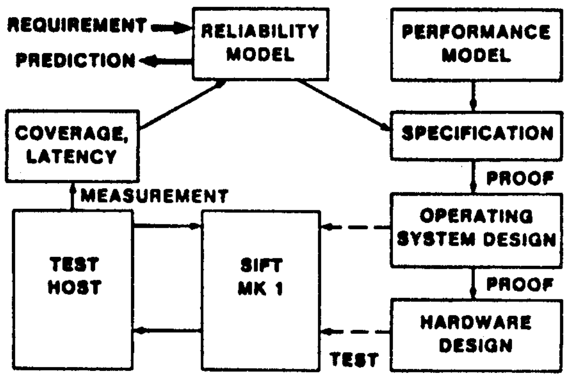


FIGURE I-2 SIFT VALIDATION SCHEME

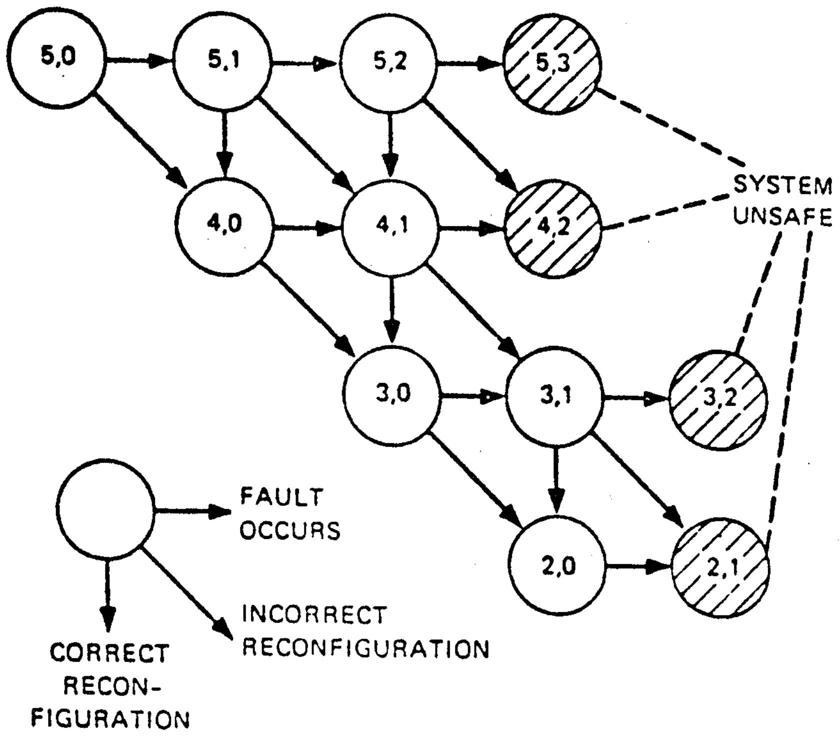


FIGURE I-3 A PARTIAL VIEW OF THE RELIABILITY MODEL

Given a validated reliability model, use of it for prediction requires quantitative estimates of its state transition rates. Since it is impractical to derive these analytically, an experimental approach is proposed. Tests would be conducted on a physical machine, in which faults are simulated or injected (nondestructively). The results would be analyzed statistically and estimates of transition rates are derived.

The result would be a reliability prediction based on a validated model of the real system. Detailed descriptions of the model and of its application are given in Chapter IV. A detailed discussion of the proposed testing plan is given in Appendix A.

6. Proof Of Correctness

Given the major role of software in SIFT, methods for proving program correctness can be used to verify a major portion of the SIFT design. As mentioned in the preceding section, the approach most suitable to SIFT is to generate a hierarchy of abstract models that progressively, through continuously increasing detail, form a logical bridge between the Markov and input-output models on the one hand, and a set of detailed specifications of the executive software on the other. The program code is then proven correct with respect to the specifications. The model hierarchy was developed specially for SIFT, while the proof of correctness work, and the mechanical tools used, were developed on a separate NASA contract, NAS1-15528. A detailed description of the proof analysis is given in Chapter IV.

7. Software Methodology

The commitment to the use of software for crucial fault-tolerant functions opened the possibility of using modern software-design methods to support reliable design and ease of understanding and proof. The SIFT software development benefited from the availability of HDM (Hierarchical Development Methodology), a mathematically formal approach to structured programming that was developed at SRI in the context of research on secure operating systems. In HDM, a system developer uses a specification language (SPECIAL) to give abstract definitions of module interfaces, and to define hierarchies of modules. HDM is supported by several tools for construction and verification.

Considerable effort was applied to keep the executive software simple and efficient (it comprises about 800 lines of code). The design is very modular, and, to some extent, hierarchical. The external behavior of each module is specified in the SPECIAL specification language. (A summary of the language is given in Appendix C and the actual specifications are given in Appendix D). These specifications provide an unambiguous definition of the modules and give a firm formal basis for proving both that the design is consistent with the high level model of SIFT functionality and that the code is consistent with the specification. The software tools used to perform these proofs, the STP system, were constructed without a parser for the SPECIAL language. Thus, for the proofs, the specifications were expressed in the LISP-like internal representation of the STP system. The modules were implemented mostly in Pascal; about 20% is written in assembly language, in order to deal with machine functions, primarily those dealing with interprocessor communication.

SIFT is one of the first two uses of HDM for an operational system (the other is KSOS, a secure operating system developed by Ford Aerospace Systems). Aside from its use in clarifying software function, it helped to focus programmer attention on design issues, thus helping to avoid both conceptual system design errors and coding errors.

8. Operational Support Equipment

The SIFT power supply system was designed as a two-level system, with conventional secondary power conditioning in each computer, and a central supply for primary power. The latter clearly must employ redundancy due to its uniqueness and criticality. The need for a special design provided the opportunity to include tolerance for partial loss of aircraft primary power. A common configuration for aircraft power is to have four power busses, in which one or two busses may fail, with significant frequency. The design for the SIFT primary power supply tolerates any combination of up to two failures among the four internal power modules and the four input lines. An additional requirement on the power supply was that certain levels of power should be delivered during total interruption of input power under two circumstances: (1) for accidental interruptions during flight of several minutes, full power should be delivered to all processors, and (2) for planned or accidental interruptions on the ground of up to 24 hours, power should be provided to the memory units of all processors. The amount of power in the latter mode was kept to a relatively low value by the use of CMOS memory devices. Design of the SIFT fault-tolerant power supply was a significant engineering effort, but it did not require development of new design principles. A detailed description of the power supply design is given in Chapter II.

A second item of operational support equipment was the aircraft input-output channel. In anticipation of future test environments, it was decided to use the Military Standard 1553A protocol, which provides for bidirectional communication (in this case, with both sensors and actuators). In the absence of commercial products for this function at the time, a special design was developed.

Due to the great variety of possible requirements and equipments, the SIFT development program did not address the problem of communicating redundant output results to redundant-input aircraft actuators. Consequently, each processor is provided with an independent 1553A channel, with the intention that, in a given system, some external means will be employed to combine the multiple outputs, e.g., by voting at the actuators, using available information about which processors are deemed to be fault-free.

9. Test Environment

Facilities for testing are provided by a general-purpose host computer, which serves both for program loading and for exercising and observing the SIFT computer system's fault-tolerant behavior. These facilities have the following general characteristics:

- *Compiling*: A Pascal compiler was developed for the SIFT cpu (a Bendix BD930) by Bendix Corporation. The complexity of the compiler made it unfeasible to operate it in the selected host computer (a Data General Eclipse), so it was built to run on the large research computer available at SRI (a DEC KL-10). A link was provided to transfer compiled code between the KL-10 and the Eclipse computer.

- *Program Transfer and Observation*: Special circuits were provided for communication between SIFT and the host computer both for the loading of programs and for two-way communication between the test computer and arbitrarily selectable SIFT registers. The latter facilities provide a powerful test capability, enabling the experimenter to inject erroneous data at any time and place in the computation, and to observe the consequential behavior.

- *Aircraft Simulation*: In order to test the capability of SIFT to meet its computational requirements under fault conditions, means were provided to have SIFT carry out simulated aircraft computations. For this purpose, communication was provided between the host computer and the 1553A input-output channels of each SIFT computer. This took the form of a specially designed multiplexer that provided two-way communication between the test computer memory and the concurrently operating I-O channels. A particular aircraft simulation was designed, based on the Microwave Landing System demonstration conducted by Bendix. Equations of motion of the USAF T39 aircraft were programmed for simulation of aircraft motion within the test computer, and a simple flight-control program was written for execution in SIFT. These facilities make it possible to carry out fault-injection experiments within the context of a simulated flight-control operation.

The test environment described is somewhat fragmented and limited in power and flexibility, but it does provide basic facilities for program development, performance measurements, and fault-injection experimentation. Considerably greater power will be required to carry out test and evaluation measurements to the depth needed for future design practice.

C. DEVELOPMENT HISTORY

This section summarizes the history of the SIFT development, including the several stages of definition, design, construction, and integration, and the contributions of the several participating organizations.

The SIFT development proceeded in the following stages:

Preliminary Study (1972). SRI studied possible aircraft requirements and computer technologies for the 1980-1990 time period [25]. Using available literature and the results of earlier SRI studies ([5, 9, 10]), several architectures were distinguished that had the potential for meeting these requirements and making appropriate use of expected technology [32]. The reliability requirement of 10^{-9} per hour was distinguished and large-scale-integrated semiconductor technology was recognized as the most promising technology. Three architectures were recommended: SIFT (conceived during the study), a published scheme by A. Hopkins [11] (later developed by C. S. Draper Laboratory as the FTMP computer), and a scheme named Bus Checker System (conceived during the study).

Feasibility Study (1975). NASA initiated two parallel computer developments: one, based on the SIFT concept, emphasized software implementation of fault-tolerance logic, and the other, based on the Hopkins scheme, emphasized hardware implementation. Both developments assumed the same computational and reliability objectives. SRI undertook a study of architecture, software design, and validation methodology for the SIFT concept [35]. The recent development of program-proving techniques led to the goal of a formally provable design. A strategy for testing was also developed.

Subcontractor Selection (1976). NASA authorized a program of machine development, based on the results of the SIFT feasibility study. SRI invited major manufacturers of aircraft computers to bid on the design and construction of a SIFT-based architecture. Several valid bids were received. The bid by Bendix Flight Systems Division was selected.

System Development (1977). Detailed engineering design was undertaken by Bendix for the SIFT computer, the test environment, and the software development system. SRI undertook the design of the fault-tolerant software system, including the use of the HDM specification methodology. SRI and Bendix jointly examined various existing flight-control programs that might provide realistic demonstration of an aircraft application under simulated fault conditions.

A design was proposed by Bendix and approved by SRI. The design implemented the original SRI concept of demand-initiated data transfer over a multiple shared-bus interprocessor network. Midway through construction, SRI discovered a weakness in the demand-initiated bus design and recommended a change to a broadcast-bus design. The change was approved by NASA and carried out by Bendix. (A discussion of the technical issues is given in Chapter II).

Several subsystems were developed concurrently (on individual subcontracts): SCI Incorporated developed the interface between the SIFT processor 1553A channels and the test computer; Bendix developed (at its own expense) a Pascal-BD930 compiler that was made available to the project; and August Systems Incorporated designed and developed a fault-tolerant power supply to serve as the central supply of primary power.

Work on formal verification of the SIFT design was conducted using verification tools developed in other SRI work. Initially the work employed a program-proving system built by Robert Boyer and J Moore [1]. The final effort employed a deductive system built by Robert Shostak [28]. This work continued through the end of the project, and is continuing under separate NASA support.

System Integration (1980). The several major and minor system hardware components were assembled and integrated by Bendix. Concurrently, SRI assembled software components, including the Bendix Pascal compiler, the SIFT executive, a demonstration flight-control program for SIFT written by SRI on the basis of the Bendix Microwave Landing System (MLS) demonstration (using the USAF T-39 aircraft), and a T-39 aircraft simulator for the test computer, written by SRI on the basis of a dynamic model supplied by Bendix. The SRI integration effort employed a DEC-KL time-shared computer.

The SIFT hardware system and test environment were shipped to SRI for software integration and testing. The integration proved to be very difficult. Contributing was the fact that full integration of the software system itself had to be done on the SIFT processors, due to the impracticality of simulating asynchronous multiprocessor activity on the KL research computer. Unfortunately, there were several very subtle hardware faults that evaded prior discovery at the hardware integration step, and these faults seriously confused the software-hardware integration effort. In retrospect, it is clear that inadequate facilities were provided in the SIFT design to support debugging.

After a considerable expenditure of effort by SRI, aided by Bendix engineers, NASA recommended the organization of a "Tiger Team" effort to attempt to achieve a correctly working system. A team was composed of several members of the SRI project, two of the original Bendix designers, and Mr. John Wensley, of August Systems, Inc., the original designer of the SIFT system. The team worked intensively for five weeks, and was completely successful in uncovering and solving the evident hardware and software problems. The result was a system that carried out all intended SIFT fault-tolerance functions within its test environment, including the demonstration program derived from the T39 Microwave Landing System.

Unfortunately, insufficient effort was available to carry out the planned program of systematic test-injection and fault-response observation. During this period, SRI achieved a successful automatic proof of correctness of the major chain of fault tolerance, from a high-level specification of fault-masking to detailed Pascal code. Part of the effort for this result was supported on a companion project at SRI on System Performance Proving (Contract NAS1-15528).

D. RESULTS

The SIFT computer has been built, debugged, operated in all its intended modes, and delivered to NASA Langley Research Center for inclusion in its AIRLAB facility for the testing of advanced aircraft electronic systems. The analysis of its reliability falls short of the intended goals, in that, due to the greater than expected debugging effort, insufficient resources remained to carry out the scheduled fault injection and recovery tests. Goals for the proof of correctness of design also were not completely realized, but the major proof chain was successfully verified by an automatic theorem prover.

We offer the following view of the achievements and shortcomings of the SIFT development.

ACHIEVEMENTS

- A new architecture for aircraft computers intended for flight-critical applications has been demonstrated, and has achieved acceptance as a contribution to the state of the art.

- The use of program verification techniques for verifying the correctness of design for fault-tolerant computers was developed and demonstrated successfully.

- Several fundamental problems in fault-tolerant computer design were uncovered and solved with full generality, specifically,

    Synchronization of fault-prone clocks

    Consistent replication of data among fault-prone machines.

  A result that apparently has not been known to many designers of fault-tolerant computer systems is that three clocks are insufficient for correct clock synchronization in the presence of faults unless special algorithms that affix digital signatures to messages are used.

- Technical results have been published in professional journals and reported in numerous technical conferences [4, 6, 7, 8, 20, 21, 23, 24, 30, 31, 33, 34]. SIFT is now widely known in the technical community.

- Many technical assumptions on which the SIFT architecture was based appear to have been validated, e.g.,

  The use of a whole computer as the unit of redundancy was very consistent with the recent development of single-chip computers,

  The use of software to implement fault tolerance algorithms is compatible with performance requirements, although some functions, such as voting, would profit from implementation in microcode or equivalent, and

  Other benefits of software implementation were realized, including ease of design modification and portability.

- The computer and its test environment were successfully built, demonstrated, and delivered.

SHORTCOMINGS

- The amount of effort required to complete the development and integration exceeded expectations.
- The planned fault-injection testing was not conducted (but the basic functions required for such testing were demonstrated).
- The flight-control demonstration programs were much simpler than hoped (due to the effort it would have required to rewrite the real flight-control programs that were available).
- Execution of the demonstration programs is slower than real-time (due to insufficient power in the host computer)
- A major redesign was required for the interprocessor communication subsystem (due to certain critical worst-case conditions that were overlooked in the original design)
- Insufficient facilities were provided in the design to aid in debugging the prototype. Examples of useful facilities are: some use of error-detecting codes (not for final use, but for debugging and maintenance checkout), special registers for observing internal state, and facilities for forcing the synchronization of clocks during checkout of higher-level functions.

E. ACKNOWLEDGMENTS

Major contributors to the SIFT project at SRI included John Wensley (the originator of the SIFT architectural concept), Michael Melliar-Smith, Charles Weinstock, William Kautz, Karl Levitt, Richard Schwartz, and Jack Goldberg (project leader). Significant technical contributions were made by Leslie Lamport and Robert Shostak. Carol Sand and Spencer Floyd provided invaluable support in project management. Contributors at Bendix included Kurt Moses, David Tawfik, Jules Lenart, John McGough, Don Porawski, James Sokolich, and Phil Forman. We gratefully acknowledge the very substantial help, guidance, support and encouragement, over the entire program duration, of Nicholas Murray, Project Monitor, and Billy Dove, Branch Head, of the Aircraft Electronic Systems Branch, Flight Electronics Division, NASA-Langley Research Center, and the constructive interest of the Branch members. We appreciate the special efforts of Charles Meissner, Brian Lupton, and Dan Palumbo in the final phase of system debugging and delivery. The entire report was composed by Mary Oakley.

II. SIFT Hardware

This chapter describes the hardware design of the SIFT computer system and the SIFT fault-tolerant power supply. SIFT computer system hardware provides computation, inter-processor data transfer, and fault isolation, but not fault tolerance as such. The latter function is provided by software functions, and is described in the following chapter.

A. SIFT HARDWARE DESIGN

1. Introduction

This section describes the hardware architecture of the SIFT computing system*. The system was designed and built by Bendix Flight Systems Division under subcontract to SRI International for NASA-Langley Research Center.

Midway in the system development, certain worst-case conditions were discovered in the interprocessor communication subsystem that would have required substantial limitations on the rate of data transfer for safe operation. A new design was, therefore, developed and successfully implemented. A comparison of the two designs is given in the final paragraphs of this section.

2. General Structure

SIFT employs a multiprocessor architecture that achieves fault isolation by physical and electrical separation among processing units, and fault tolerance by replicating computing tasks among the units. Error detection and system reconfiguration are performed by software to maintain the operational integrity of the computer system.

The increased power of multiprocessor architectures can be nullified by an interconnecting bus system that is slowed by contention for port or bus services [9]. The SIFT system is a multiprocessor computer array that utilizes dedicated ports and busses for all interprocessor data transmissions, thus avoiding delays due to contention that occur in shared-bus systems.

Computing is carried out by high-speed Bendix BDX-930 processors. Each processor main memory contains 30K words; each word is 16 bits long. This memory holds the SIFT executive program and the application programs. Each processor communicates with the other processors via dedicated, buffered bit-serial busses. Synchronization task

*The text is taken, with slight revision, from a report prepared by K. Moses of Bendix, Flight Systems Division.

dispatching and fault tolerance are achieved by the cooperative action of the individual executive programs.

3. Interconnection System

The basic structure of the SIFT computer system is designed to drastically reduce interprocessor communication delays. A variable number of processors (up to 8 in the current prototype) are connected to each other by dedicated links, so that each processor can immediately broadcast the results of its computations to all the other processors (Figure II-1).

A block diagram of the SIFT computer system is shown in Figure II-2. The computations and broadcasts are carried out in an iterative sequence for a real-time avionics application. Computations carried out by the BDX-930 processor are temporarily stored in the scratchpad memory data file. This data file can be accessed by the broadcast shift register, receiver, 1553 data link, and the CPU in that order of priority. This sequence has three phases, which control the activities of the system components.

1. *Load Phase*--The processor computes its assigned tasks, loads resultant data into its local data file, loads the associated destination address into its transaction file, and loads the starting transaction address into the transaction pointer to start the broadcast. The *broadcast phase* of operation is then initiated. It should be noted here that the *broadcast* and *receiver* phases (described below) function independently of the processors and do not detract from the power or speed of the CPUs that make up the SIFT computer system.

2. *Broadcast Phase*--The broadcast sequencer broadcasts a data word (from data file) along with the associated destination address (from transaction file) at a maximum rate of one data word every fifteen μs. This rate was selected to ensure that the receiver had sufficient time to detect the data word and store it in its local data file. The broadcast sequence continues until end-of-file (EOF) is reached in the transaction file. The control register shown in II-2 contains the EOF indicator. The flow diagram for this sequence of events is shown in Figure II-3.

   End-of-file (EOF) is reset by loading the transaction pointer with the starting address. The delay timer is then started. It times out in 14.75 μsec, resulting in a transmission rate of up to one data word approximately every fifteen μs. A 16-bit data file word (Figure II-4) is then loaded into the transmitter, and its receipt is acknowledged.

   The 25-bit serial word is then concurrently broadcast to all other processors in the system. Following transmittal acknowledgment and two NOP's, the

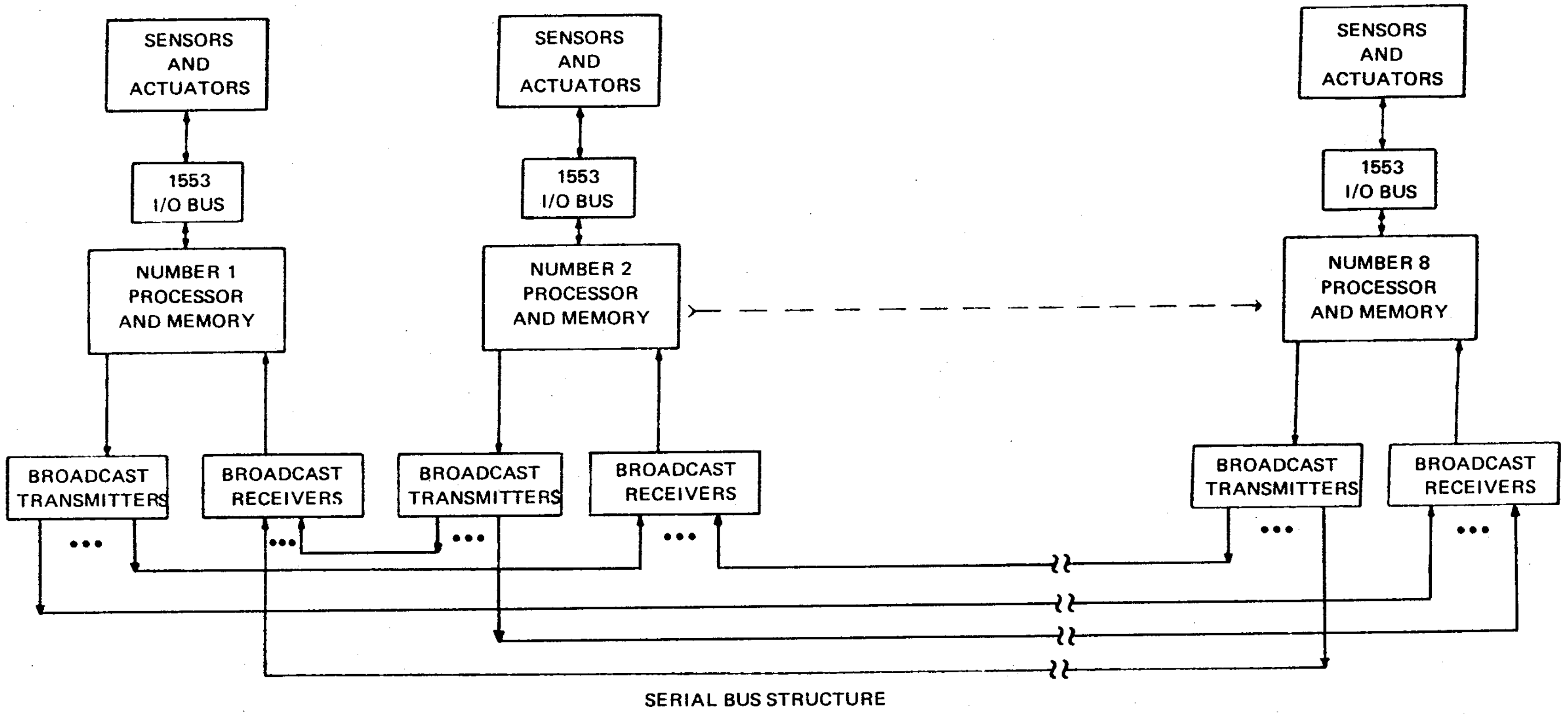


FIGURE II-1 SIFT ARCHITECTURE

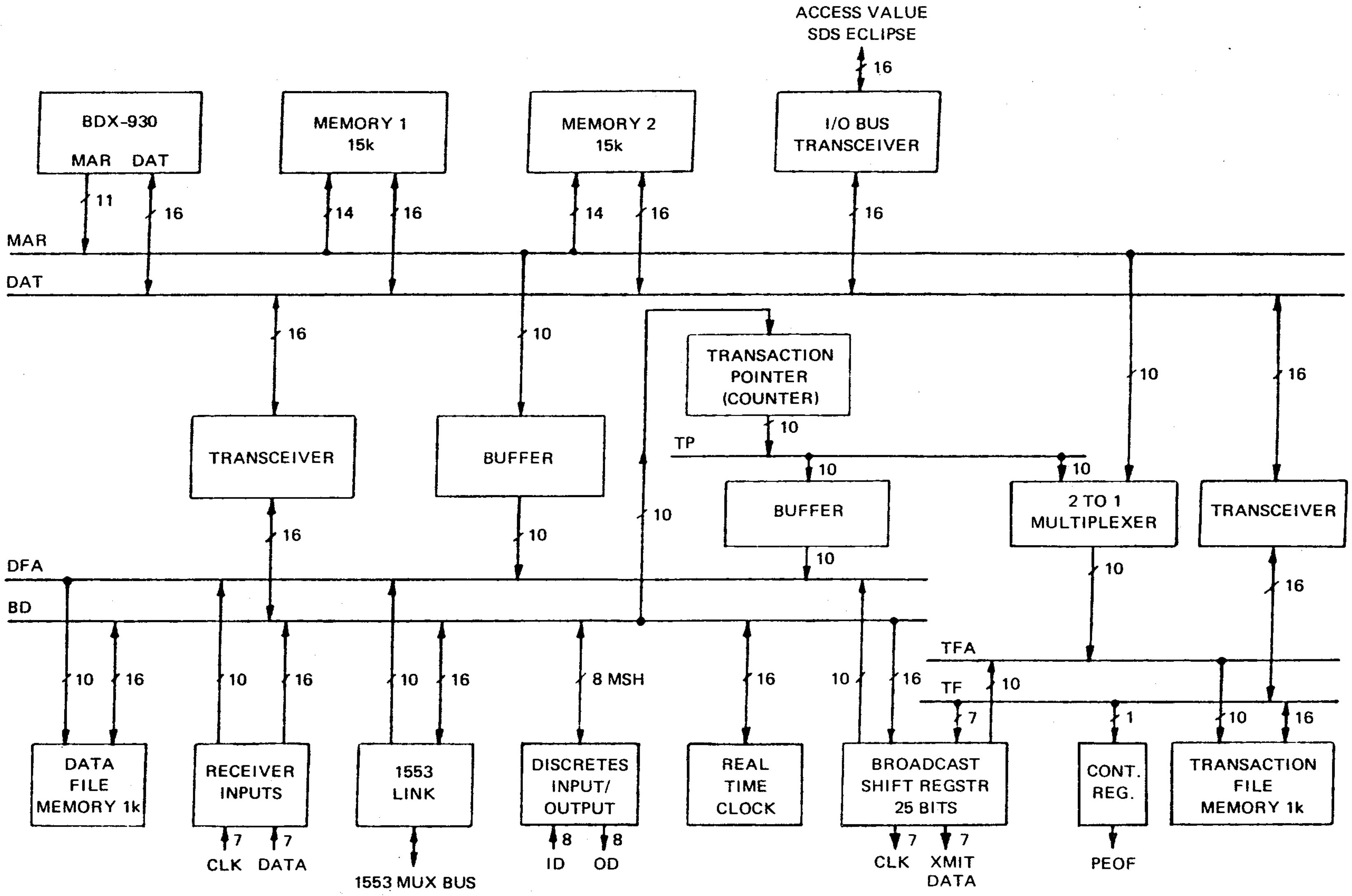


FIGURE II-2 SIFT PROCESSOR DESIGN

EOF is updated and the transaction pointer is advanced to the next transaction if additional words remain. After time-out and the execution of a NOP, the broadcasts are either terminated or this sequence of events is repeated.

3. *Receiver Phase*--The 25-bit serial word is transmitted in synchronism with a 4 MHz clock over busses that are dedicated to each destination processor. The transmitted word is stored momentarily in dedicated receivers in the destination processors. Here, receiver sequencers scan the receivers for full registers, then steer the data words to the local data file locations indicated by the destination addresses (Figure II-5).

   Each receiving processor receives the same data words and stores these data words at the same relative location in its local data file. The flow diagram for this sequence of events is shown in Figure II-6. The maximum time it could take to load a particular data word occurs when (1) the scanner has to scan a total of eight registers, and (2) the CPU, transmitter, data link, and six other receivers have prior access to the data file. Under these conditions, it would take 9.12 μsec to load the designated word into the data file. If the designated word were located in the first register scanned and if there were no other contenders for data file access only 0.8745 μsec would be required to transfer the word to the data file.

The preceding discussion on the broadcast and receiver sequencers has shown that the time to complete one transaction (i.e., from the loading of a data file word and its destination address into the transmitter of processor i to the storing of that word in the data file of processor j) under the worst-case conditions is equal to 18.253 μsec The minimum time to complete this transaction is 8.6235 μsec.

4. CPU

Each processor contains a Bendix central processing unit (CPU), the BDX 930 (Figure II-7). The BDX-930 is a microprogrammable, parallel, binary digital processor with a 16-bit word length. The CPU performs 16-bit parallel arithmetic. An instruction pipeline organization provides concurrent fetch, decode, and execute operations to maximize execution speed. A request/response system is used to lengthen those micro-orders that interface with the slower speeds of tape units and other I/O devices.

The use of high-performance Schottky transistor-transistor logic elements permits very fast internal clocking rates -- as high as 16 megahertz (62.5-nsec period). This produces a CPU cycle time of 250 nanoseconds and an average operation rate of 942 KOPS. Interregister ADD is executed in 500 nanoseconds; firmware-based MULTIPLY is executed in 5.1 ms.

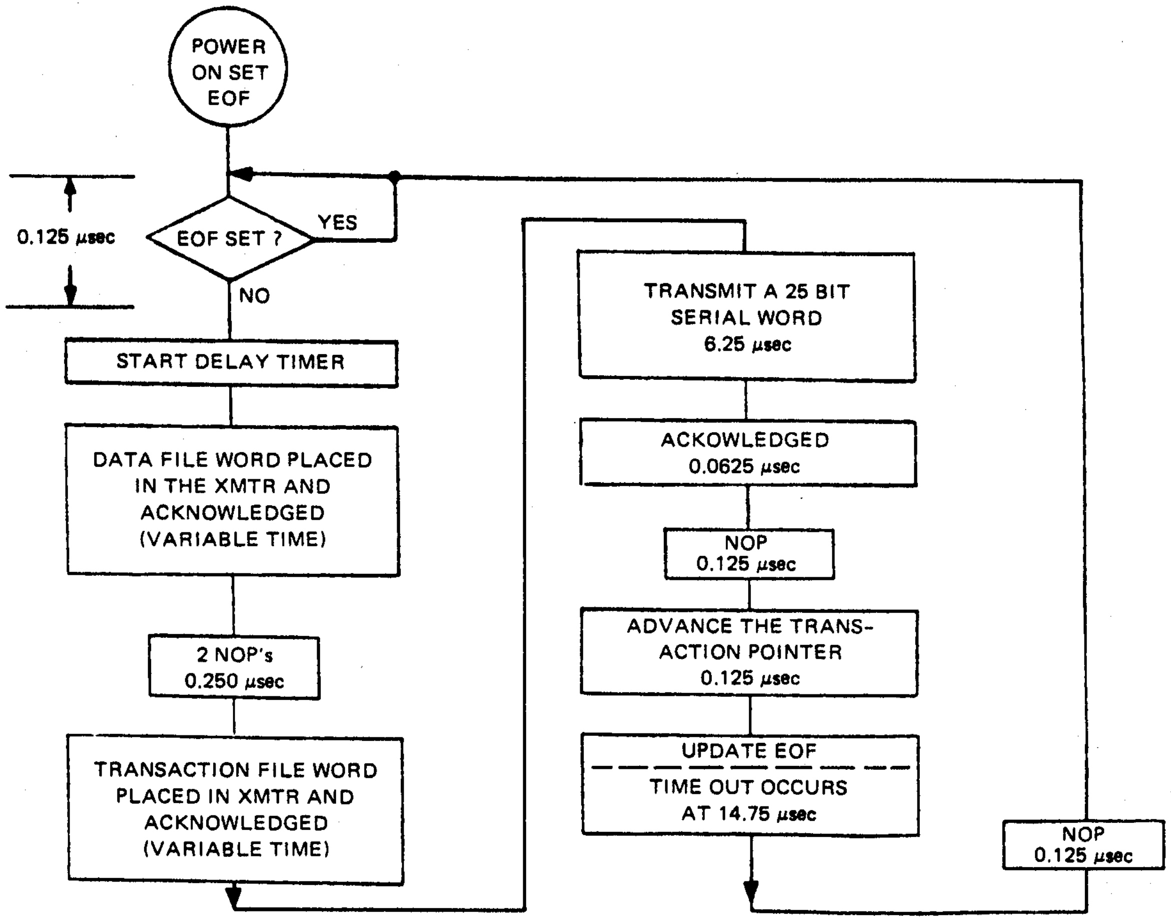

POWER ON SET EOF
0.125 µsec
EOF SET ?
YES
NO
START DELAY TIMER
DATA FILE WORD PLACED IN THE XMTR AND ACKNOWLEDGED (VARIABLE TIME)
2 NOP's 0.250 µsec
TRANSACTION FILE WORD PLACED IN XMTR AND ACKNOWLEDGED (VARIABLE TIME)
TRANSMIT A 25 BIT SERIAL WORD 6.25 µsec
ACKOWLEDGED 0.0625 µsec
NOP 0.125 µsec
ADVANCE THE TRANS-ACTION POINTER 0.125 µsec
UPDATE EOF
TIME OUT OCCURS AT 14.75 µsec
NOP 0.125 µsec

FIGURE II-3 BROADCAST SEQUENCER

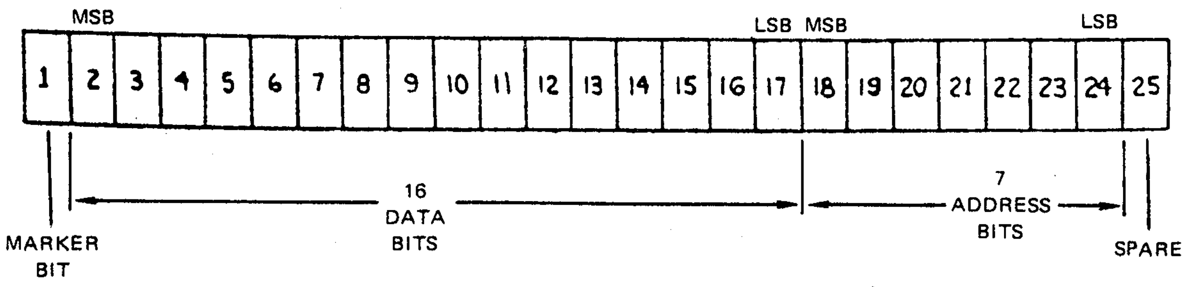


FIGURE II-4 SERIAL WORD FORMAT

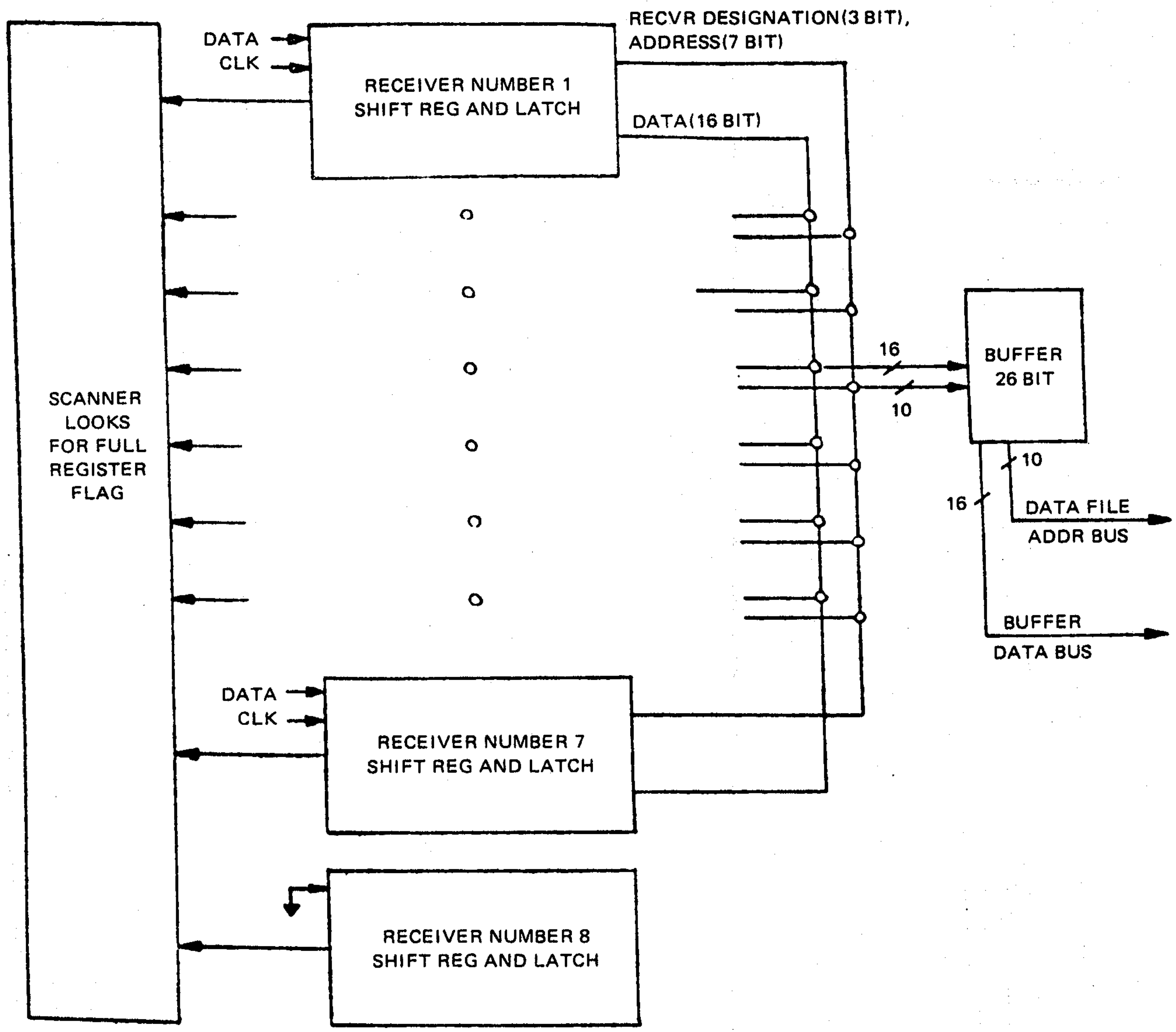


FIGURE II-5 RECEIVER MODULE

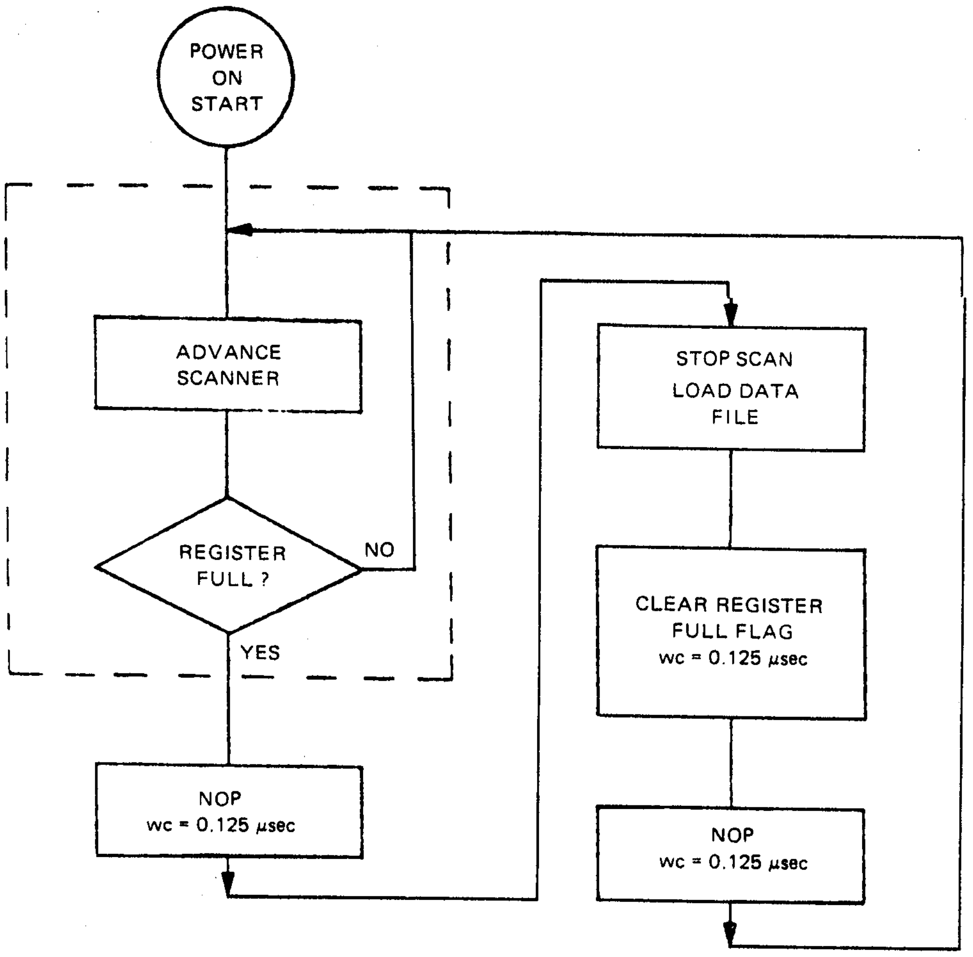


FIGURE II-6 RECEIVER SEQUENCER

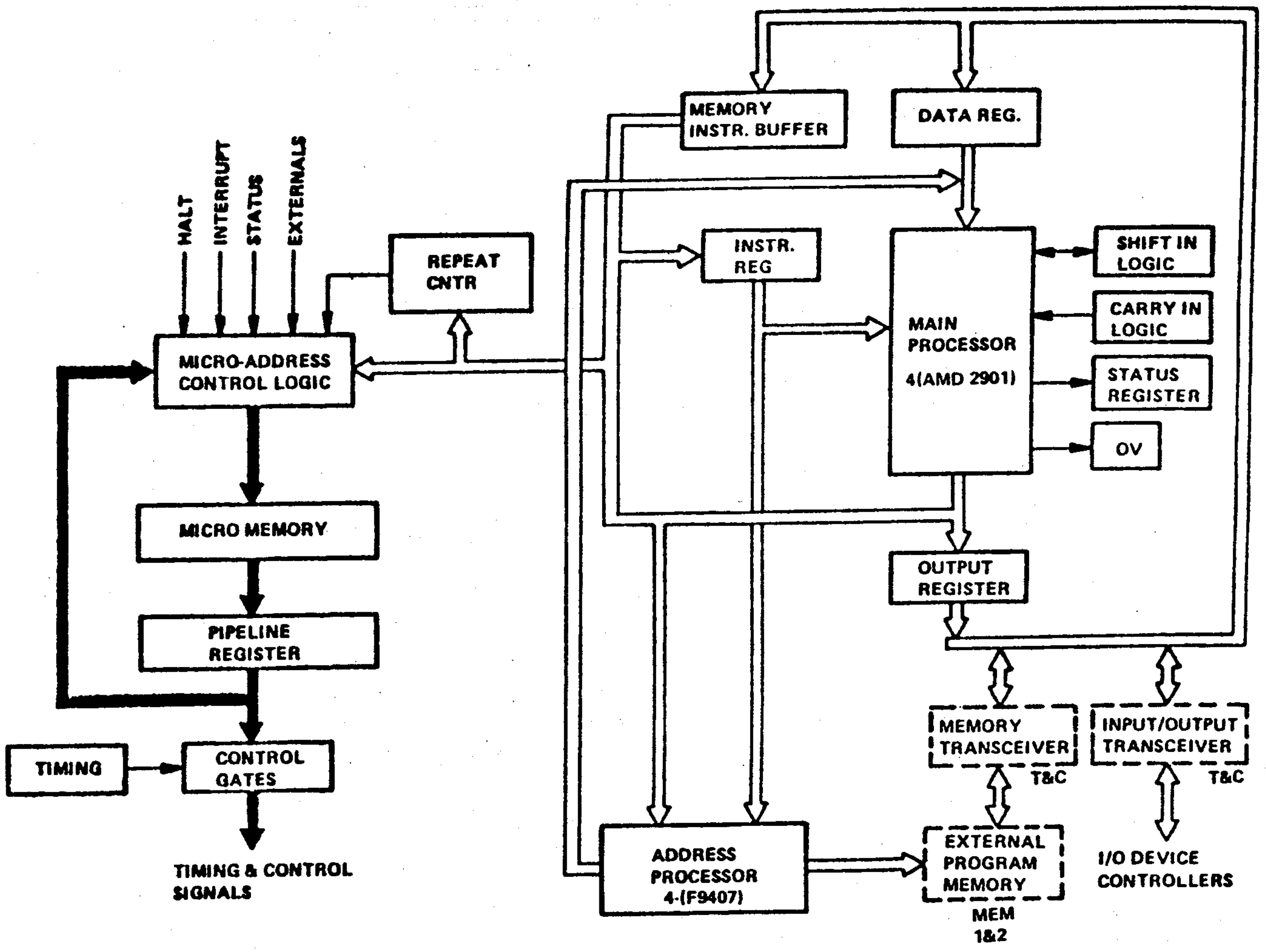


FIGURE II-7 BDX-930 PROCESSOR BLOCK DIAGRAM

The BDX-930 consists of 86 microcircuits mounted on one printed circuit board.

5. Memory

Memory addresses are logically subdivided into mapped segments as shown in Figure II-8. Each processor's main memory and stack contain 30K words; each word is 16 bits long. This memory holds the SIFT executive program, the application programs, the transaction and data files used by the interprocessor broadcast communication mechanism, and the control stack.

The significant results of each processor's computations are temporarily stored in a scratch-pad memory data file. Each data file contains 1K data words (each 16 bits long).

High speed interprocessor communication is provided by separate processor/bus interface elements that control the bit-serial transmission and reception of data words. The memory destination of each transmission is provided by commands stored in the transaction file in each processor. Each transaction file contains 1K words (each 16 bits long).

6. Discrete Functions

A reserved block of 8 addresses is used to address 12 discrete functions that are firmware or hardware implemented (Figure II-9). These functions are called by executive programs to perform various input-output, inter-processor communication, and synchronization functions. The implemented functions include: read processor identity number; set EOF; read real-time-clock; write (set) real-time-clock; read 1553A registers; write 1553A registers; and load transaction pointer.

7. External I/O

External I/O information is transferred by MIL-STD-1553A serial links. Time-division-multiplex controllers govern the data flow between aircraft actuators, sensors, avionics modules, and the BDX-930 processor. There is one controller for each BDX-930 processor and 1553A bus. Each 1553A controller and bus can support up to 32 remote terminals with associated actuators, sensors, or avionics modules. A detailed description of the 1553A controller design is given in Appendix E.

8. Test Control

In order to validate the design to the confidence level required by the specifications, hardware tests, system tests, and software validation tests may be performed using the Test Control System (termed Software Development System (SDS) in Bendix documentation). The Test Control System is capable of performing all the operations of the Bendix BDX-930 Access Panel. The operations are performed under software control

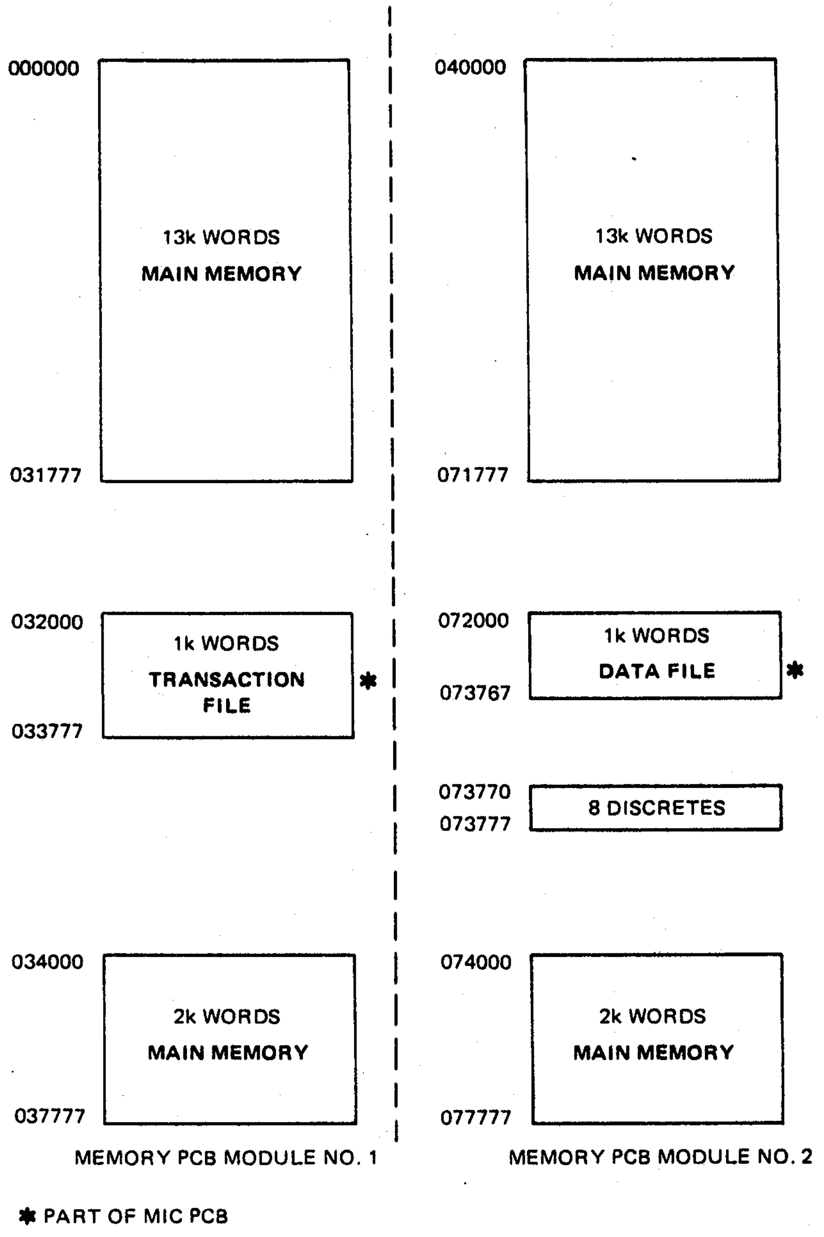


FIGURE II-8 MEMORY MAP

	READ	WRITE
073770	READ PROCESSOR ID	SET EOF
073771	READ 1553A STATUS REGISTER	LOAD TRANSACTION POINTER
073772	UNUSED	
073773	READ REAL TIME CLOCK	WRITE REAL TIME CLOCK
073774	READ 1553A CMD REGISTER	WRITE 1553A TEST
073775	READ 1553A CMD REGISTER	WRITE 1553A CMD REGISTER
073776	READ 1553A T REGISTER	WRITE 1553A T REGISTER
073777	READ 1553A T REGISTER	WRITE 1553A ADDRESS REGISTER

FIGURE II-9 DISCRETES

executed on an Eclipse S/230 Data General computer (referred to in this report as the Host Computer). A maximum of eight Bendix BDX-930 computers can be controlled with the current SDS interface.

The interface between the Eclipse S/230 and the Bendix BDX-930 consists of two modules. The first module is a standard 15" x 15" card that plugs into any unused I/O slot of the S/230. The second module is rack mounted in a chassis that connects with the first module through a standard DB 4192 paddleboard connector, which is wired to the unused I/O slot of the S/230 computer backplane. The Bendix BDX-930 computers are connected to the rear of the second module through their respective access panel cables (see Figure II-10).

9. Comparative Analysis of the Original and Final SIFT Bus Systems

Of primary importance in the design of an ultrareliable fault-tolerant computer system is the means used for achieving fault isolation. In the initial design, faults were isolated to a specific processor or bus. This was accomplished by using a specially designed redundant bus system to interconnect the processing units. Each processor and its associated memory formed a processing module, and each of the modules was connected to a multiple bus system. A block diagram of this system is shown in Figure II-11.

Communication between processors was achieved by serial linkages set up by the bus controllers. This process was characterized by a request/response interaction between processors and controllers (see Figure II-12) and functioned in the following manner. A processor P_a requiring data from another processor's memory M_j would issue a request for a specific bus, e.g., B_k. When that bus became free, the request would be acknowledged, and the bus would wait for memory M_j to detect a request for data.

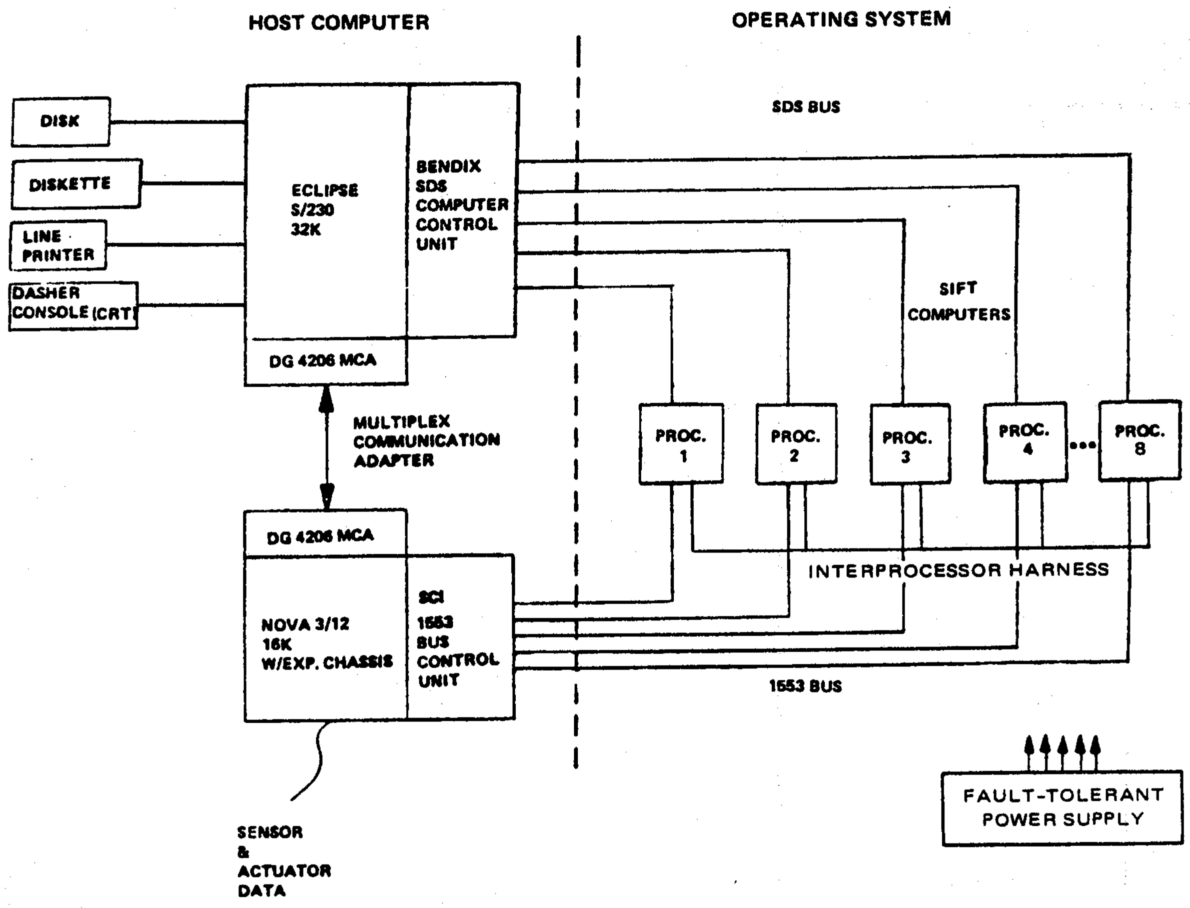


FIGURE II-10 TEST CONTROL SYSTEM

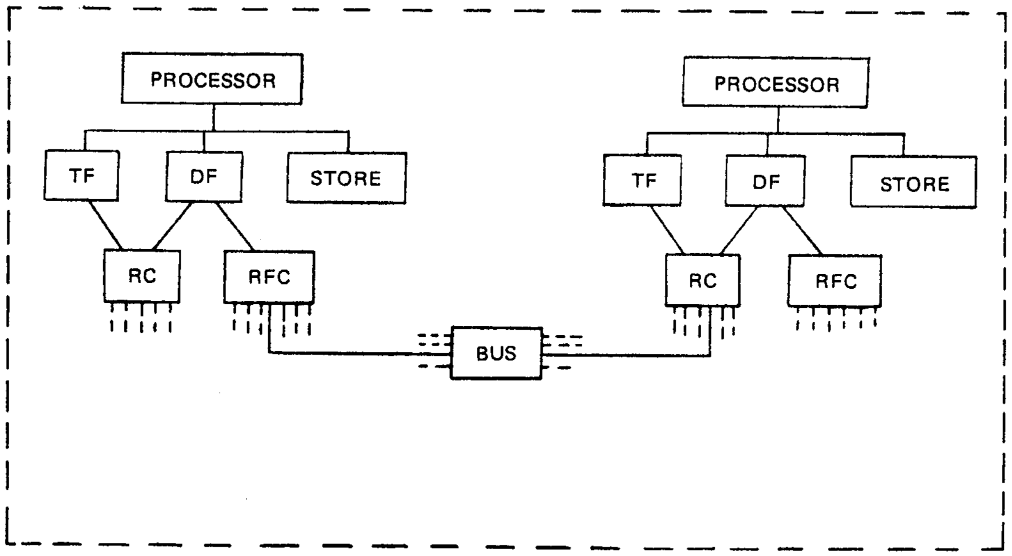


FIGURE II-11 ORIGINAL DEMAND-BUS DESIGN

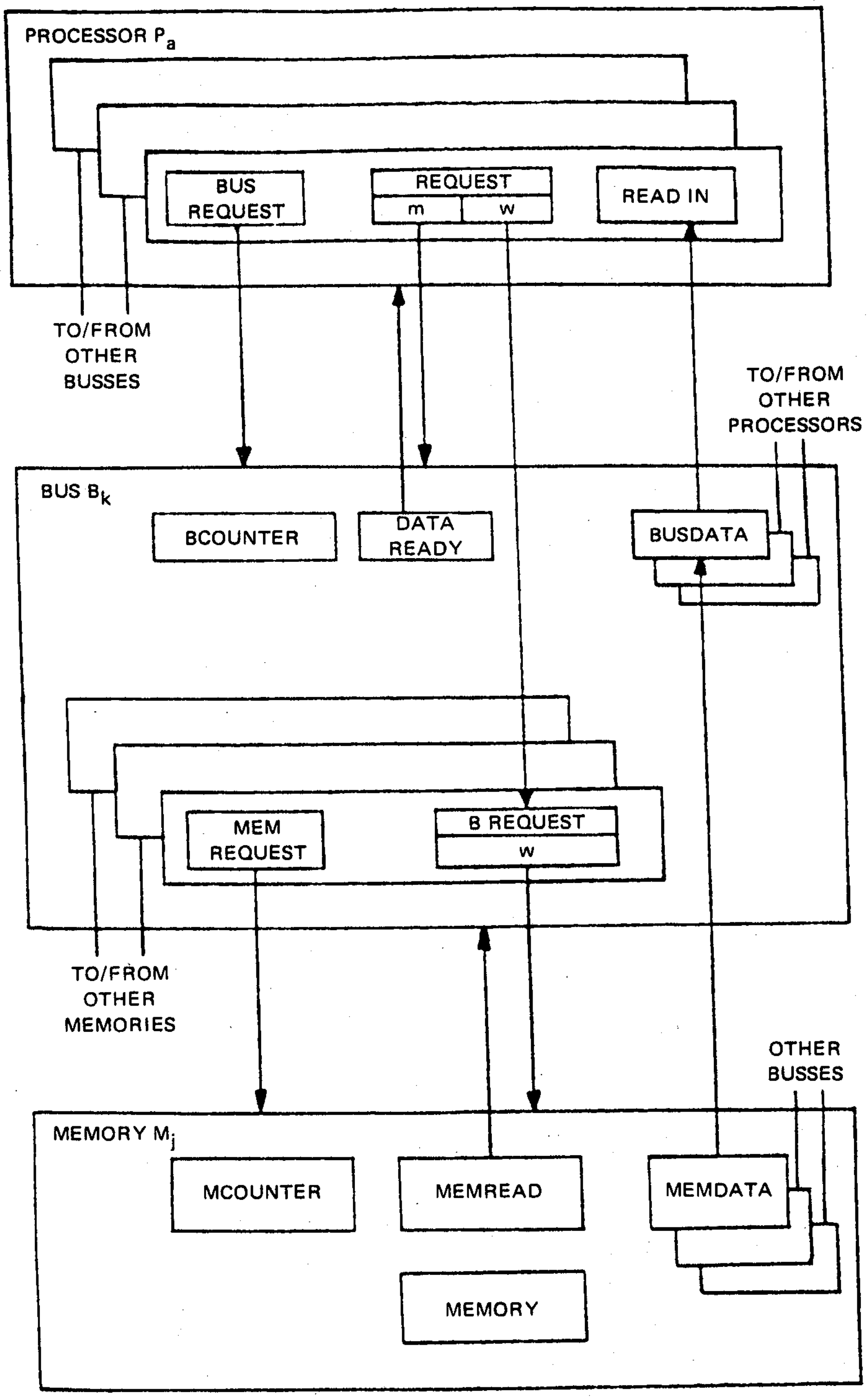


FIGURE II-12 INTERPROCESSOR COMMUNICATION IN THE DEMAND-BUS DESIGN (SUPERSEDED)

Memory M_j would then access and transfer the data to bus B_k, which would then release M_j, and transfer the data to P_a, which would then release B_k.

Of primary concern in evaluating this design is the time taken to complete a transaction for different conditions. The system evaluated consisted of 8 processors and 8 interconnecting busses. Table II-1 gives a breakdown of the time taken to complete a transaction in the absence of any bus-contention or memory-contention delays.

Table II-1
INTERCONNECTING BUS SYSTEM

INTERCONNECTING BUS SYSTEM (8 PROCESSORS, 8 BUSSES)
- NO BUS OR MEMORY CONTENTION DELAYS -

The sequence of events and the maximum time each requires are as follows:

1.	memory identification setup	0.937 µs
2.	bus controller cycle	5.321 µs
3.	memory cycle	10.664 µs
4.	memory data storage	1.312 µs
	total time =	18.234 µs

It is evident that the time to complete a transaction will be longest when a total of 8 processors are contending for a particular bus and a total of 8 busses are contending for a particular memory. This worst-case transaction time is

8 [time for 7 busses to access a particular memory]

+7 [time for processor to acquire bus, bus to access memory, memory data to be transmitted and bus to be released]

+ [time for designated processor to acquire bus, bus to access memory, memory data to be transmitted to, and stored in, processor]

The time for 7 busses to access a particular memory is

7 [memory cycle - memory request scan + scanner advance]

$=7\ [10.664\ \mu\text{sec} - 1.0\ \mu\text{sec} + 0.1875\ \mu\text{sec}]$

$= 68.9605\ \mu\text{sec}$

The time for a processor to acquire a bus, a bus to access a memory, memory data to be transmitted and the bus to be released is

[memory cycle - memory release - memory request scan + bus controller cycle ; scanner advance]

$= [10.664\ \mu\text{sec} - 0.249\ \mu\text{sec} - 1.0\ \mu\text{sec} + 5.792\ \mu\text{sec} + 0.1875\ \mu\text{sec}]$

$= 15.3945\ \mu\text{sec}$

The time for a designated processor to acquire a bus, a bus to access a memory, memory data to be transmitted to, and stored in the processor is equal to

[memory identification setup + bus controller cycle + memory cycle + memory data storage - memory request scan + scanner advance]

$= [0.937\ \mu\text{sec} + 5.321\ \mu\text{sec} + 10.664\ \mu\text{sec} + 1.312\ \mu\text{sec} - 1.0\ \mu\text{sec} + 0.1875\ \mu\text{sec}]$

$= 17.4215\ \mu\text{sec}$

Thus, the worst-case transaction time is

$8(68.9605) + 7(15.3945) + 17.4215$

$= 676.867\ \mu\text{sec}.$

It is evident that, under these worst-case conditions, i.e., maximum contention at the memory and at the bus controller, the transaction time is intolerable.

In order to reduce this time delay to an acceptable value, a broadcast bus-design concept was investigated and adopted. There are many advantages to be attained by implementing the broadcast-bus design. This design (see Figure II-13) provides dedicated busses and receivers for all data transmissions so that there are no delays due to contention for a particular bus and memory. A processor neither requests data nor requests that data be transmitted over a particular bus. When a value has been calculated in the course of one task that has been designated to be needed by other tasks, it is broadcast to all other processors. This occurs in the following manner: The data file is loaded with the values to be broadcast and the transaction file is loaded with the addresses to be broadcast to. The transaction pointer is then set to start the broadcast. The broadcast sequencer broadcasts a data word from the data file along with the associated destination address from the transaction file. Data words are transmitted at a rate of one data word every fifteen ms. This rate was selected to ensure that the receiver had sufficient time to detect the data word and store it in its local data file. The broadcast sequence continues until end-of-file (EOF) is reached in the transaction file. The flow diagram for this sequence of events is given in Figure II-3 and

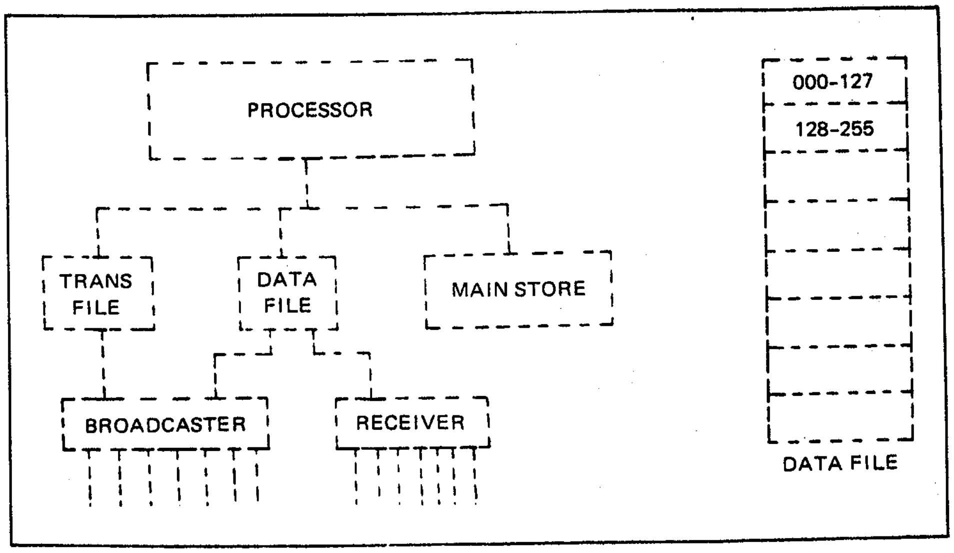


FIGURE II-13 FINAL (BROADCAST) DESIGN FOR INTERPROCESSOR COMMUNICATION

discussed in Section 3*.

The 25-bit serial word is then concurrently broadcast to all other processors in the system. Following transmittal acknowledgment and two NOPs, the EOF is updated, and the transaction pointer is advanced to the next transaction if additional data words remain. After time out and the execution of a NOP, the broadcasts are either terminated or this sequence of events is repeated.

In 6.25μsec the 25-bit serial word is transmitted, in synchronism with a 4 MHz clock, over busses that are dedicated to each destination processor. The transmitted word is stored momentarily in dedicated receivers in the destination processors. Here, receiver sequencers scan the receivers for full registers, then steer the data words to the local data file locations indicated by the destination addresses. Each receiving processor receives the same data words and stores these data words at the same relative location in its local data file. The flow diagram for this sequence of events is shown in Figure II-6.

*Following Figure II-3, the worst case time for placing the data file word in the transmitter would occur if the data link had prior access to the data file. In this case the time would be 1.4365 μsec. If the transmitter received immediate access to the data file, the time would be 0.6245 μsec. Following two NOPs (No-Operation Cycles) requiring 0.250μsec, the 7-bit transaction file word is placed in the transmitter, and its receipt is acknowledged. The maximum time for this action to take place results when the 930 had initial access and equals 1.1865 μsec. In the absence of any contention for the data file, the time is 0.6245 μsec. The total time for loading the transmitter, i.e., placing the data file word the transaction file words in the transmitter, thus ranges from 1.4990 to 2.8730.

The maximum time to load a particular data word occurs when the scanner has to scan a total of 8 registers and the 930, the transmitter, the data link, and six other receivers all have prior access to the data file. Under these conditions, it takes 9.12 μsec to load the designated word into the data file (see Table II-2).

Table II-2
TIME TO COMPLETE SPECIFIED OPERATION
IN RECEIVER SEQUENCER LOOP

SEQUENCER OPERATION	TIME IN MICROSECONDS							
	PASS 1	PASS 2	PASS 3	PASS 4	PASS 5	PASS 6	PASS 7	PASS 8
SCANNER ADVANCE	0.1875	0.1875	0.1875	0.1875	0.1875	0.1875	0.125	0.1875
NOP	0.125	0.125	0.125	0.125	0.125	0.125	N/A	0.125
LOAD DATA FILE	1.686	0.8115	0.562	0.562	0.562	0.562	N/A	0.562
CLEAR REGISTER FULL FLAG	0.125	0.125	0.125	0.125	0.125	0.125	N/A	N/A
NOP	0.125	0.125	0.125	0.125	0.125	0.125	N/A	N/A
SUBTOTALS	2.2485	1.374	1.1245	1.1245	1.1245	1.1245	0.125	0.8745

NOTE: Total worst case time = 9.12 μsec
N/A = Not applicable

If the designated word were located in the first register scanned and if there were no other contenders for data file access, only 0.875 μsec is required to transfer the word to the data file.

Table II-2 gives a detailed breakdown of the 9.12 μsec required to load a particular word into the data file under worst-case conditions. As indicated, it requires eight partial or complete passes through the loop. During the first pass it is assumed that the 930 and the transmitter have prior access to the data file and that, during the second pass, the data link is accessing the file. The time for the seventh pass is minimal since it assumes that this register does not contain a full word.

In summary, the time required for the transmission of a word from one processor to another is the sum of the times of three sequential funcions: loading the source-processor transmitter (1.499μsec to 2.873μsec), transmission (6.250μsec), and reception at the destination processor (0.875μsec to 9.120μsec). The time required under worst-case conditions is 18.243μsec, and under best-case conditions is 8.624μsec. These times can be compared, respectively, to the 676.867 μsec and 18.234 μsec obtained for the contention-bus system.

In summary, the broadcast-bus design concept has resulted in the achievement of acceptable transaction times and, in addition, has simplified the logic design and reduced the required number of interface signals.

B. THE SIFT FAULT-TOLERANT POWER SUPPLY

1. Introduction

The SIFT computers receive power for their operation from a two-level power supply system. Each computer contains its own power supply, which converts 28 vdc power into the various voltages needed for the digital circuit boards. The low-power CMOS memory is supplied separately, preserving its information during times when the processor is not operating. The set of SIFT computers obtain power from a central source, the Fault-Tolerant Power Supply (FTPS), which derives its primary power (117 vac) from the main aircraft power system.

The FTPS has two functions: (1) reliable supply of power to the SIFT computers during normal operation, and (2) reliable supply of power to the memory supplies only, under two special circumstances: (a) temporary total loss of aircraft power (a few minutes), and (b) ground testing (up to twenty-four hours).

Contemporary aircraft employ several configurations of primary power. The FTPS is designed to receive inputs from a set of four input power lines and to tolerate the following faults or losses:

- Loss of power on any one or two power input lines with no loss in function
- Electrical failures in any one or two internal power units
- Any electrical failure in the internal battery system (except that this will cause loss of the power back-up function)
- Short-circuit load conditions in any number of outputs, one at a time, with no loss of service to the remaining loads.

Achievement of these objectives required careful design. The greatest design complexity arose from the nonlinear and nonconsistent behavior of batteries, even those of the highest production quality.

The FTPS was designed and built by August Systems, Inc. A thorough test procedure was designed by SRI, and lengthy tests were conducted by SRI and August Systems during its development.

The remainder of this section reviews the FTPS design and the procedure developed for its testing.

2. Brief Description of the FTPS

The FTPS consists of four parallel August Power Supplies (APSs) driving a common load, which consists of eight SIFT processors (see Figure II-14). Each of the four APSs consists of a tandem connection of a Primary Power Supply (PPS), a Battery Backup Unit (BBU), and a Diode Interconnection Network (DIN).

The PPS is a Lambda LGS-EE-28-OV-R supply capable of delivering nominal 28 volts (i.e., 24v to 30v), 25 amps (at 40°C), from a 105-130vac source, 47-440Hz. It is protected internally against excessive input voltage or output current. (See the manufacturer's literature for additional specifications.)

The BBU consists of a Lambda 41-volt charging supply, a series cascade of 16 GATES, 2.2-volt, lead-acid batteries (0800-0004 5AH "X"); a battery charging circuit and indicator light; a circuit breaker (CB); and a pass element and regulator (PER) unit. The PER switches in the battery supply when the PPS supply voltage falls below a preset threshold value and switches out the batteries when the output voltage drops below another threshold.

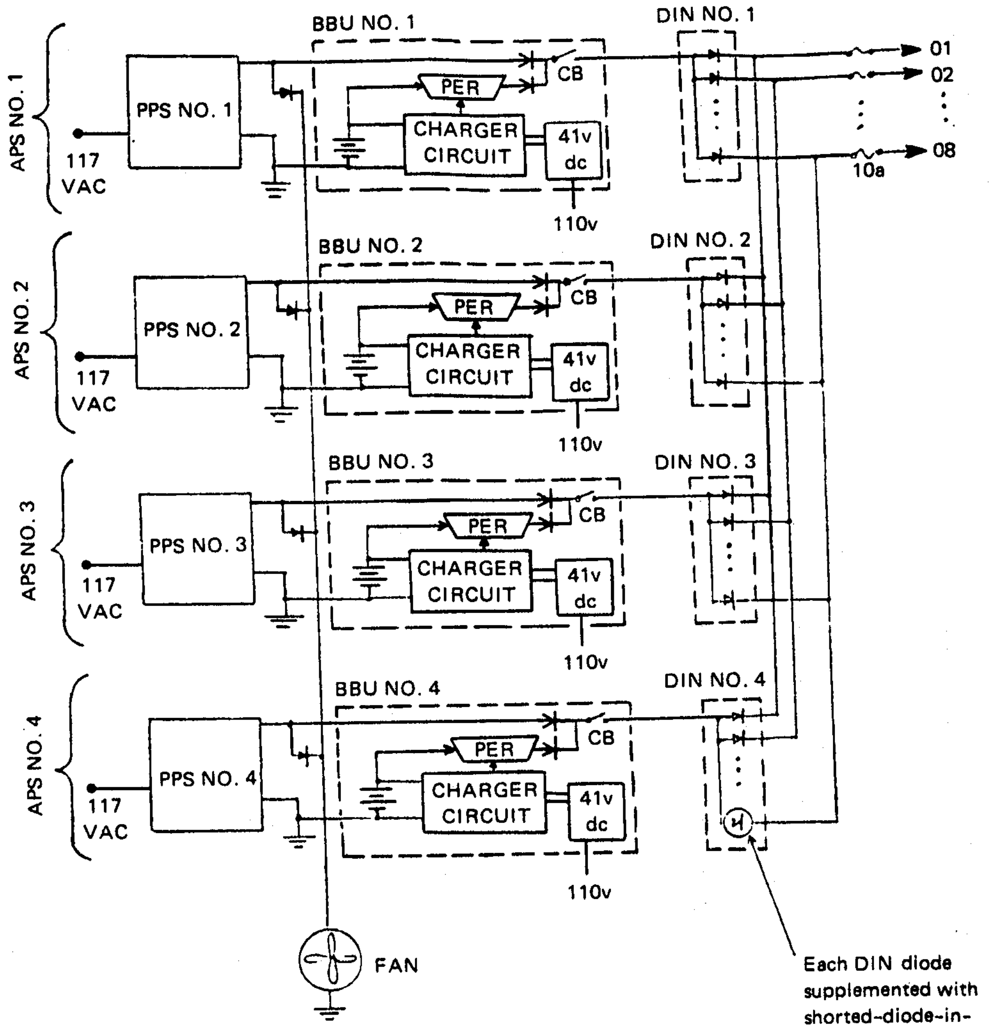
The DIN is a diode distribution network in which corresponding DIN output terminals of the four APSs are bused together to drive the eight 28-volt inputs of the SIFT processors.

In addition each output line contains a 10a fuse, and a common air-circulating fan is supplied from the PPSs through a 4-diode network. In the laboratory version, the fan represents a single point of failure; in an aircraft installation, cooling air would be derived from the aircraft itself and the fan would not be used.

3. Testing Strategy

The Fault Tolerant Power Supply (FTPS) for SIFT presents a unique challenge in testing. Because of the redundancy built into its design, an overall functional test, even under stress conditions, is not sufficient to verify that the entire FTPS is free of faults. Rather, it is necessary to check that, in the presence of a single fault in one member of every redundant set, the set is still able to perform its intended function without degraded performance. Protection against certain double faults has also been provided in the design and must be similarly tested. Fault isolation must be checked; namely, it must be verified that a fault in any one member of a redundant set does not, in itself, prevent proper continued operation in another, fault-free member. Similarly, short circuit at the power input to any one SIFT processor must not cause interruption of power to the other processors.

The test procedure described in the next section verifies both the performance and the fault-tolerance of the FTPS. It is based upon a full failure modes and effects analysis



PER = Pass Element and Regulator

FIGURE II-14 SCHEMATIC DIAGRAM OF FAULT-TOLERANT POWER SUPPLY

(FMEA), at the level of the circuit elements, for the entire FTPS except the BBU charging and PER, which are regarded as single functional units not further decomposed.

Performance is tested against the specification of constant and interference-free output voltage and current under full-load conditions (all processors operating), assuming adequate input ac power and assuming a normal operating environment. Quantitatively, these specifications are:

Input: 117 $\pm$ 10% vac rms, 400 Hz.

Output: 28 $\pm$ 4 vdc, nominal 25a; ripple $\leq$ 1.5 v-peak-to-peak, 1Hz to 10kHz.

Fault tolerance is provided through redundancy and fault isolation. Redundancy occurs in the FTPS at three levels:

- Level 1: The FTPS consists of four APSs, operating in parallel.
- Level 2: Within each APS, the required output power can be supplied from two sources: either the PPS or (for a limited time) the backup battery in the BBU. (Switchover is automatic.)
- Level 3: Within each APS, overvoltage/overcurrent protection is provided in both the PPS and the BBU.

Fault isolation is achieved by a high degree of mechanical isolation and by electrical isolation through the power diodes in the BBU and DIN of each APS. These diodes prevent propagation of faults from a defective APS to a good one. They also protect against loss of power to good processors in the event that one of the eight fan-in connections to the fuses develops a dead short. Protection is provided against most double faults, except under certain conditions deemed to be extremely unlikely.

All redundant units and isolation features must be checked individually during the course of testing, to the maximum extent possible without danger of damage to the hardware.

Protection has been provided in the design of each APS for the following specific failure conditions. Each of these conditions must be checked during the testing for adequate fault protection:

PPS:

-Steady overvoltage or undervoltage, including zero voltage
-Hum or high-frequency interference (400 Hz or higher), due to defective filtering or unintended oscillation
-Transient interference or erratically varying output voltage
-Circuit breaker breaking too slowly, at too high load current, or not at all

-Poor output regulation to variations in line voltage (117 v normal)
-Poor output regulation to variations in load current (0-25 a)
-Output impedance low or zero.

BBU:

-Single diode is shorted or its back resistance is too low
-Any circuit node shorted to ground
-Any single diode open or forward resistance too high
-Any single circuit connection open
-Battery charge control circuit defective: charge rate too high or low, or circuit not switching battery in and out at the correct voltages or otherwise erratic
-Main regular circuit defective: switchover not occurring at correct voltages or erratic; battery drain excessive
-Battery weak or dead (high internal impedance, cell open or shorted, low output voltage, or not accepting charge); battery charge indicator circuit or light inoperative.

DIN:

-Any single diode shorted or back resistance too low
-Any circuit node shorted to ground
-Any single diode open or forward resistance too high
-Any single connection open
-Any single shorted-diode detector circuit, or its corresponding light, inoperative
-Any single power diode not capable of handling full load current without overheating
-Short between any two output terminals.

In addition, a few global failure conditions need to be checked for in the FTPS as a whole:

FTPS:

- Reduced dc output voltage or current from any one or two APSs

- Transient interference from any one APS

- Interruption of primary 117 vac power to the FTPS for a period of 3-5 minutes

- Short circuit to ground at any one of the eight outputs.

4. Conclusions

A detailed test procedure based on the set of failure assumptions listed above was developed and applied. Several subtle failure conditions were uncovered through its application.

The detailed test procedure, the results of the testing, and the details of FTPS circuit design, circuit operation, calibration, and maintenance have been submitted separately to NASA.

III. SIFT Software

This chapter describes the organization and design of the SIFT executive system, which is responsible for all fault-tolerance functions in the SIFT computer system. It also presents formal specifications for all constituent program modules and an example of the Pascal code for a "voter" module (documentation of the full code has been submitted to NASA). A guide to the creation of application programs is given in Appendix F.

A. ORGANIZATION AND DESIGN

1. Introduction

This section describes the organization, design and detailed functional description of the SIFT executive system. The discussion is preceded by an abstract description of system processing and communication facilities as seen by the executive system.

2. Processing and Inter-process Communication

The following is a brief description of SIFT processing and inter-process communication facilities as seen by the SIFT executive software system.

a. Processing Structure

Computing in SIFT consists of iterations of well-ordered, pre-planned sequences of task-processing, voting and inter-process communication. Each task instance consists of a reading of inputs, program execution and broadcasting of outputs, all within the same iteration. There are no global data and no data retained from one iteration to the next; rather, data are passed from task to task (more specifically, the inter-task data consist of the results of votes over sets of redundant task outputs). There is no built-in logical regulation of inter-process communication; rather, all data transfers and task initiation must be scheduled in advance.

b. Inter-process Communication

Although SIFT is physically a distributed-processing system, all transfers of information are pre-scheduled and coordinated by synchronized clocks. Transfer of information between processors is therefore more suitably viewed as program-controlled transfers between buffer registers within a single computer than as logically synchronized transfers of data between asynchronously operating processors.

Physical transfer of data between processors is accomplished by programmed calls to special machine functions, which serve to broadcast data from the originating processor to all other processors. Broadcasting is undirectional, without acknowledgment, and is initiated by each originating processor(s) according to its own schedule. Since each processor's schedule is a part of a preplanned overall schedule, and since all non-faulty processors are synchronized to within a controlled amount of time-skew, inter-processor communication is well-regulated on a global basis.

In each processor, broadcasting makes use of two special 1024 word blocks of memory, known as the *data file* and the *transaction file.* The data file is further broken down into eight blocks of 128 words, each known as *data buffers.* The data broadcast by processors are received and stored in these data buffers, a separate data buffer being used for information from each processor. Aside from the host processor, data can be placed in a particular data buffer by only one other processor.

The transaction file is used to control the broadcast. The broadcasting processor stores in the transaction file a sequence of destination addresses and sets the transaction pointer to the start of the sequence in the data file. Setting the transaction pointer initiates the broadcast. The last entry in the sequence is marked with an end-of-file flag, which stops the broadcast. All processors receive all broadcast data, but all processors do not necessarily run all tasks. Hence, only a subset of them may actually use the data; the data received by the other processors is voted but not used.

3. The SIFT operating system

The SIFT operating system provides the fault-tolerant environment in which the application tasks are executed. Its functions generally fall into one of these classes:

Scheduling	The operating system must cause application tasks to be executed at the proper time.
Synchronization	The operating system must keep the processors moderately synchronized, sufficiently to assure unambiguous inter-task communication.
Consistency	The operating system must provide identical copies of unique input data to all good processors.
Communication	Results of tasks must be transmitted to all processors in a timely manner.
Fault masking	Erroneous data, generated by a faulty processor, must not be allowed to propagate, and the source of erroneous data should be identified.
Reconfiguration	The operating system must be able to detect a faulty unit and logically remove it from the system, in order to avoid the possible simultaneous occurrence of faults in more than one active processor.

Input/output	The operating system must see to it that sensor data are received from, and actuator data are transmitted to, the interface to the aircraft (according to MIL-STD 1553A convention).

Each of these functions will be addressed in the following sections.

a. The SIFT Hierarchy

The hiding of fault-tolerance mechanisms from the application tasks is one of the benefits of the SIFT design. This is accomplished through the use of a hierarchical structure, exhibited as Figure III-1. The SIFT hardware itself is at the lowest level of this hierarchy. This level supports a set of modules that together implement the *SIFT virtual machine*, which provides the fault-tolerant facilities that are local to each processor. These facilities include error detection, buffer (or data file) management, voting, and local task scheduling. The SIFT virtual machine, in turn, supports more global fault-tolerant facilities responsible for synchronization and reconfiguration. These facilities are part of the upper levels of the operating system, but the programs that implement them are treated by the lower levels in exactly the same way as the application tasks. The highest level of the hierarchy is a module consisting of sets of precomputed task distributions and schedules for all possible combinations of faulty units. This module provides a framework for performing reconfiguration to achieve fault-tolerance.

b. Design for Provability

At the high level of reliability required for the SIFT system, the validity of the implementing software itself becomes a serious issue. An overriding concern during the design of the SIFT operating system has been that it be mathematically provable. Thus, every opportunity has been taken to simplify the system to the greatest extent possible, so as to make formal proof feasible.

Our approach to proof (discussed in detail in Chapter IV, Validation and Verification) is to develop an intuitively clear and complete model of the SIFT system and to determine that the model expresses the necessary reliability properties. We must then prove that the model accurately describes the SIFT system. To do so, we develop another model that is slightly more complex, and prove that the properties of the simpler model are implied by those of the more complex model. We continue this process until the level of the actual code of the SIFT system is reached. At this point we prove that the code implies the properties of the next higher model. To achieve a complete proof, it would be necessary, ultimately, to verify that the hardware correctly interprets the software, according to the machine instructions assumed in the software design.

The model immediately above the SIFT software constitutes a set of precise specifications on SIFT executive functions. These specifications are written in the SPECIAL language, which is a component of the SRI-developed software methodology HDM (Hierarchical Development Methodology) [27]. The SPECIAL language is summarized in Appendix C, and a full set of specifications is included in Appendix D.

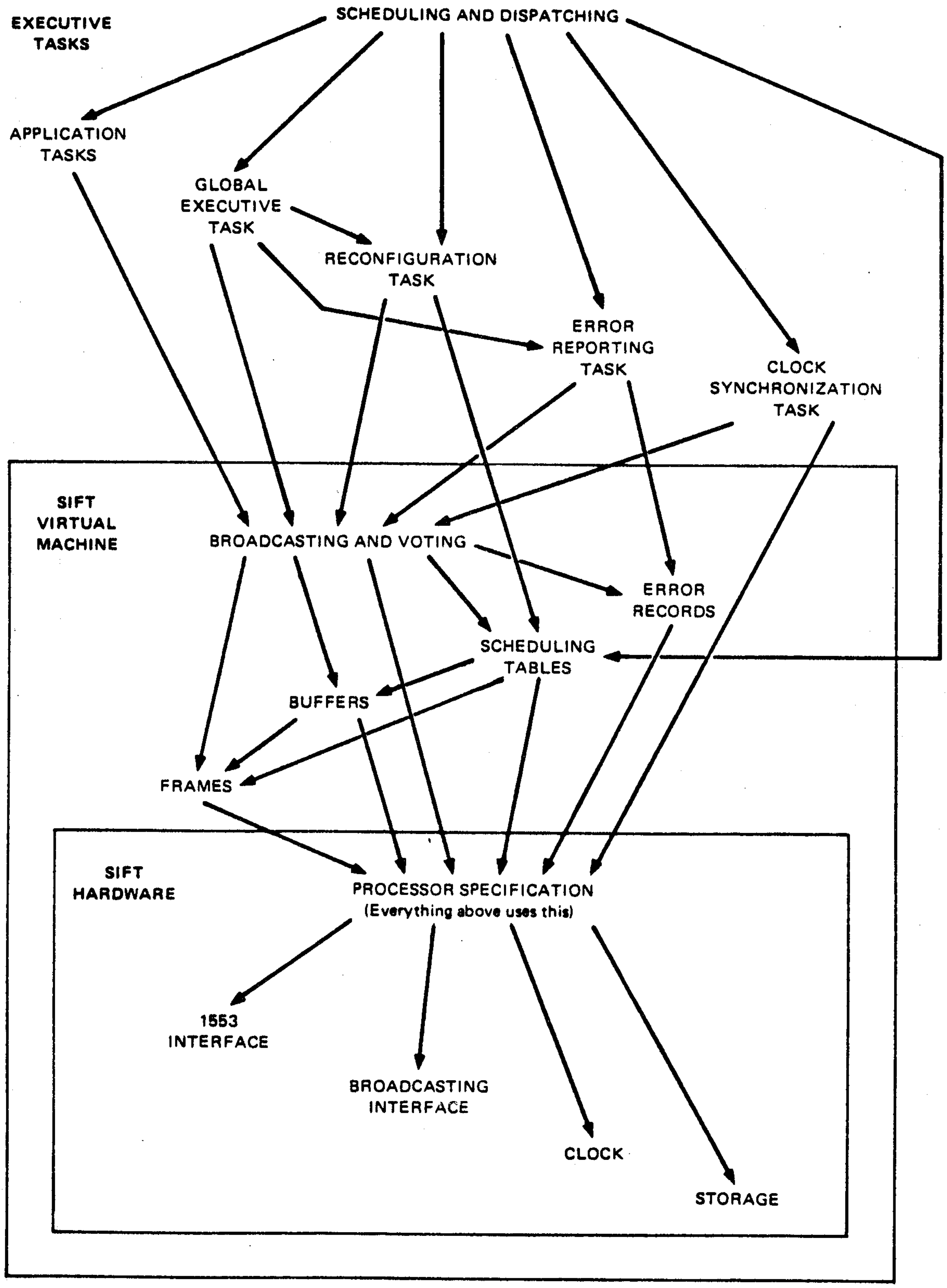


FIGURE III-1 THE SIFT SPECIFICATION HIERARCHY

To ease the proof, the SIFT system was written in a subset of PASCAL. In general it is easier to prove a program written in a high-level language than to prove one written in assembly language. While a program written in a high-level language usually is not as efficient as one written in assembly language, recent advances in compiler technology have reduced this penalty to acceptable levels. The PASCAL for the Bendix 930 is a compiler that takes advantage of these advances. Additionally, the penalty can be reduced by carefully writing the code, keeping in mind what the resulting object code is likely to be. The resulting PASCAL code should be provable and fast. (It may also be, at least in some cases, inelegant, i.e., verbose.) One version of the PASCAL code for the voter module appears at the end of this chapter.

c. Organization of Processing and Communication in SIFT*

The version of SIFT delivered for use in AIRLAB employs a very restricted form of process creation and interprocess communication, compared with the version described in the paper printed in the October 1978 Proceeding of the IEEE [Wensley]. The 1978 version incorporated features such as

scheduling: priority based, preemptive and periodic
task length: arbitrary
allocation of tasks to processors: dynamic
task replication and voting: transparent to the application engineer
iteration rates: multiple

A design was developed for an operating system that would support these features, but it was too complex to be verified by proof of correctness, given the present state of the art. In order to permit formal verifiability, the flexibility of processing and communication aimed for in the 1978 design was severly curtailed. The corresponding features of the baseline AIRLAB design are:

scheduling: static, preplanned, non-preemptive
task length: discrete segments
allocation of tasks to processors: static task replication, scheduling and voting: manual construction by the
    application designer of schedule tables, assignments of task
    replicates to processors and voting tables
task iteration rates: a single rate.

The resulting design places a heavier burden on the application programmer and tends to be less efficient in utilizing hardware resources. It is, however, more determinate, and hence more readily analyzable for completeness and consistency. Elimination of mechanisms for dynamic process creation, multilevel priorities, multiple iteration rates, etc., also tends to reduce intrinsic vulnerability to faults and design errors.

*We are indebted to Daniel L. Palumbo and Ricky W. Butler for the comparison presented here.

d. Task Structuring and Communication

This section will define and explain the use of tasks as a means of organizing processing and inter-process communication.

The basic element of computation on the SIFT computer is the *task*. There are *application tasks* and *executive tasks*. Examples of application tasks include (for the flight-control application) the yaw damper, the pitch inner loop, the roll inner loop, and others. Clock synchronization is an example of an executive task.

Computation in SIFT is conducted in regular time segments called *frames* and *subframes*. The lengths of the segments are system parameters. In the present design, the subframe length is set at 3.2 milliseconds. A fixed number of subframes combine to form a *frame*. In the design, there are 31 subframes to the frame, which is thus approximately 100 ms long. The number of subframes per frame is also a system parameter. Each task runs at a regular rate, the fastest tasks running once a frame, and others running every other frame or with an even longer period. Tasks that run once a frame are scheduled to a specific subframe, while slower tasks fill in the empty subframes.

A SIFT task is highly structured. Each iteration of a task starts from the initial state of the task, with no data carried forward from the previous iteration of the task. The task then obtains input data from the postvote buffers. These inputs may include data from other tasks, data from the previous iteration of this task, and aircraft sensor inputs through interactive consistency. The task must not obtain any input from any source that is not voted. The task then executes and generates its result values, which are distributed to other tasks by the broadcast mechanism. Broadcasting from a given processor is performed by placing the result values in the processor's Data File and activating the broadcast transmitter to transmit those values to the data files of all of the other processors. These results in the data files are subsequently voted and the consensus results placed in postvote buffers, where they can be used by other tasks.

Normal tasks are scheduled so that their execution requires one or more subframes, during which their results are broadcast. These subframes may differ for different processors. The voting of the results is also a scheduled operation, and the subframe during which the voting occurs may also be different on different processors. The schedules are constrained so that:

- No voting of a task's results can be scheduled before the subframe following the last execution subframe for that task on any processor.
- No task that needs to use the voted results can be scheduled to run on a processor before the vote occurs on that processor.

For urgent tasks, transport delay (the total time required for sensor sampling, computation and actuator drive) can be a critical design criterion. Such tasks on SIFT must be scheduled so that replications of the task execute in the same subframe on all processors. The results can then be voted at the start of the next subframe and will be available for use by further urgent tasks during that next subframe.

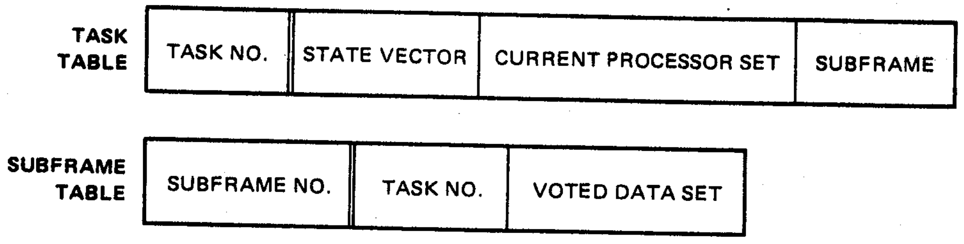


FIGURE III-2 SCHEDULE TABLES

e. The Scheduler

To accomplish the necessary replication, execution and intercommunication of tasks, a schedule must be composed that explicitly:

allocates task replicates to processors
schedules all task events so as to assure proper frequency of occurrence
and satisfaction of inter-task delays, and
designates data elements to be voted, and the subframe in which they are voted.

This schedule may be conceived as an N x M table of task events, where N is the number of processors in a working configuration and M is the number of subframes in a frame. For a given processor, the schedule is a list of votes and task activations in successive subframes.

This information is employed by the Scheduler function. It is organized in two sets of tables: The Task Table, and The Subframe Table, as shown in Figure III-2. The task table is organized by the task identification number, and contains:

- A *state vector* pointer for the task. This is where a state is saved when a task is interrupted to run another task.

- A list of the processors that the task is presently running on.

- The subframe in which the task should run.

In the *subframe table*, each subframe has a set of entries; an entry contains:

- The data to be voted upon during that subframe.
- The task to run at that subframe.

The information in these tables enable the scheduler to operate very efficiently. Aside from the power-up code, the scheduler is the only portion of the SIFT operating system that is interrupt-driven. At the start of every subframe, a clock interrupt occurs, and control is transferred to the scheduler. The scheduler suspends the currently executing task, orders the voter to vote data as scheduled, and then starts (or resumes) the task scheduled for this subframe. The lowest-level task is a diagnostic task that is always ready to run. Figure III-3 shows the detail of a subframe. Specific guidelines on schedule construction are given in Appendix F.

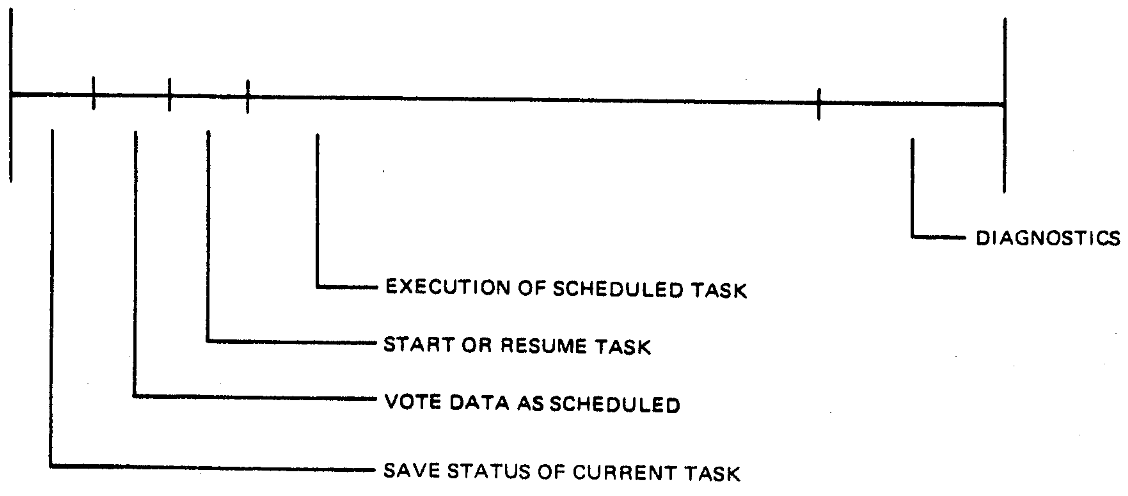


FIGURE III-3 SUBFRAME STRUCTURE

f. The Voter

The purpose of the voter is to perform a majority voting operation on task outputs and detect and report erroneous outputs. The voting function is applied to multiple instances of data in the data-file buffer, and the results are placed into the appropriate slot of the postvote buffer. The tasks get their actual values from the postvote buffer, and have no direct interaction with the voting process. All votes are done on the basis of an exact match; that is, all inputs that are accepted as contributions to the output of the voter must be identical. Processors providing values that are not identical to the output are reported to the error handler (Subsection g).

processor	1	2	3	4	5	6
subframe						
1	Sensors		Sensors		Sensors	
2		Replicate		Replicate		Replicate
3	MLS	MLS	MLS			
4	Guidance	Guidance	Guidance	Fuel	Fuel	Fuel
5		Pitch	Pitch	Pitch	Pitch	Pitch
6	Roll	Roll	Roll	Roll	Roll	
7	Yaw	Yaw	Yaw	Nav	Nav	Nav
8	Thrust	Thrust	Thrust	Thrust	Thrust	
9		Display	Display	Display	Display	Display
10	Error	Error	Error	Error	Error	Error
11		Replicate	Replicate	Replicate	Replicate	
12	Global	Global	Global	Global	Global	
13	Reconfig	Reconfig	Reconfig	Reconfig	Reconfig	Reconfig
14	Clock	Clock	Clock	Clock	Clock	Clock

FIGURE III-4 STRUCTURE OF A TYPICAL FRAME

In the unlikely event that the voter cannot find a majority, errors are recorded for all processors and a "no vote" value is placed in the postvote buffer. This value may be preset as desired by the application tasks, and is used without change by the SIFT operating system. (the interpretation of this "no vote" signal, and the action to be taken, is application dependent, and is not built into SIFT). Figure III-5 shows the relationship of the postvote buffer with the data file and the transaction file.

g. The Error Handler

The SIFT error handler keeps track of errors detected on the local processor. These errors fall into two distinct classes:

1. Errors detected as the result of receiving bad data from some other processor.

2. Errors detected as the result of some local malfunction.

It is not always easy to distinguish between these two cases. Consider, for example, a voting error that occurs because processor X's value doesn't agree with that of processors Y and Z. It may seem obvious that this is an error belonging to the first class above, but if the disagreement is the result of an undetected parity error in the data file of the local processor, then the error clearly belongs to the second class. Unfortunately, in this case we can tell only that the values disagreed but not why they disagreed. Each processor has a *processor error table*, which it uses to keep track of the number of errors it has detected involving other processors. Ultimately data will be used by the SIFT executive to help determine what reconfiguration actions to take (Subsection h, below).

There are certain errors, however, that may result only from a local malfunction. These can be grouped loosely under the title *abnormal task termination* errors. Such errors can be the result of a task raising an exception condition, or the scheduler determining that a task has not been completed within the allotted time period (possibly the result of a runaway loop). All errors of this nature may be recorded for further diagnosis, either by on-board diagnostic processes or by ground support equipment.

h. Fault Diagnosis and Reconfiguration

Fault diagnosis and reconfiguration are accomplished by the error reporter, the global executive, and the reconfiguration task. The first and last of these run on every processor. The global executive runs as any other replicated task.

The Error Reporter

The *error reporter* runs on every processor. Its function is to analyze the local error table to determine which processors are perceived to be malfunctioning. The error reporter considers a processor to be malfunctioning whenever its count of errors in the error table exceeds some threshold value. This threshold is a system parameter, and has

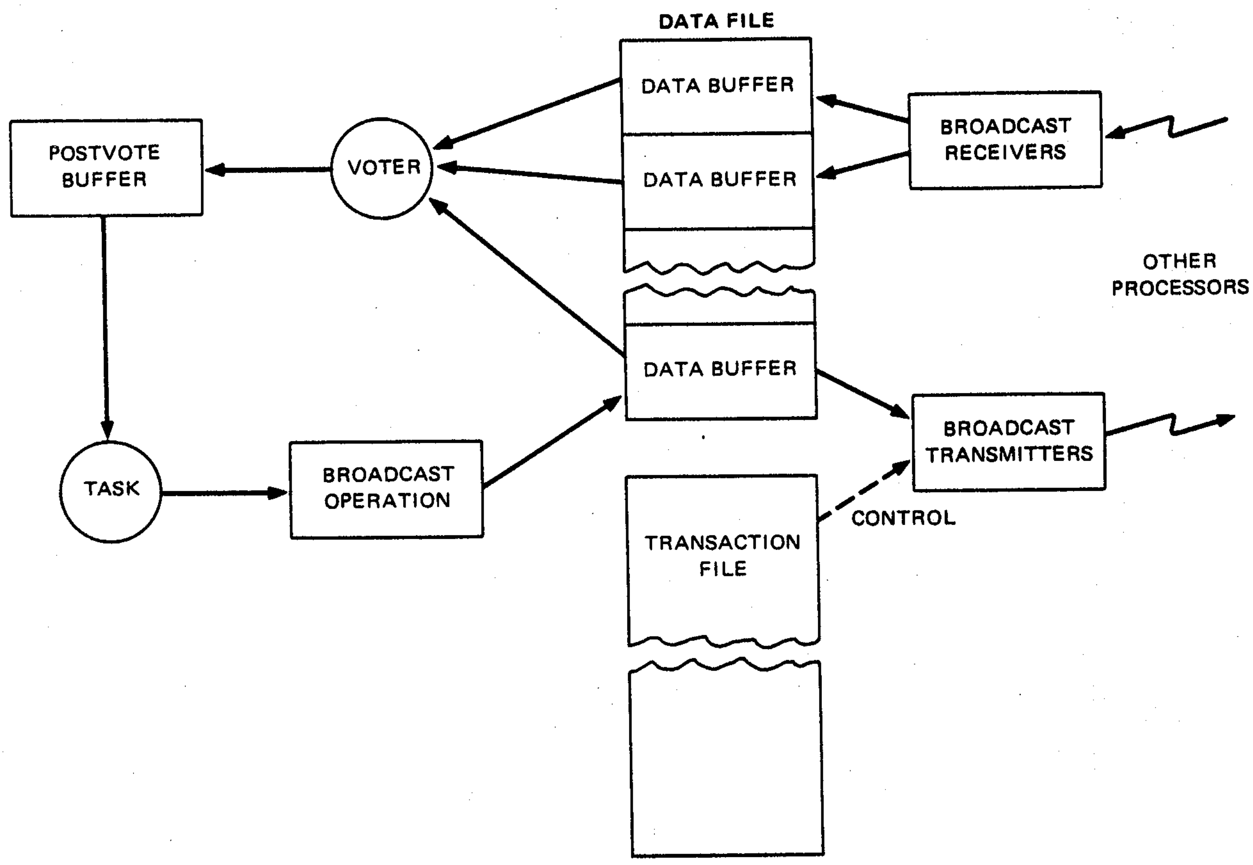


FIGURE III-5 SYSTEM DATA FLOW

been set at the level of two errors in the present design. A better value, or perhaps a more complex decision procedure, will be determined through experience with the operational system. The results of the error reporter's analysis are broadcast in the same way as any other task-computed data. Since not all processors run all tasks, each is likely to have a slightly different error table, hence, the error reports issued by different processors are not guaranteed to match exactly. Therefore, these error reports are themselves subject to analysis; that analysis is done by the *global executive* task.

The Global Executive

The *global executive*, a task that runs on multiple processors, analyzes the reports issued by the error reporter and determines whether to modify the SIFT system configuration. The decision is based on the following rule:

> Whenever the reports from two or more processors identify another processor as malfunctioning, the decision is made to reconfigure the system so that the processor is logically removed from the system. The report from a processor is never used to determine whether or not that processor is malfunctioning.

This rule needs some explanation. At first it seems unreasonable to ignore a report from the processor under consideration. Indeed, it is certain that a processor that calls itself faulty is in fact malfunctioning, but it is precisely because it is malfunctioning that other processors must ignore it, in order to avoid confusion. For example, the malfunctioning processor could conceivably send different reports to different copies of the global executive. If only one other processor has noticed the malfunction, some instances of the global executive will decide that a reconfiguration is needed, while others will decide that it is not. Since the decision of the global executive is voted upon by the reconfiguration task, an error would be attributed (incorrectly) to the processor that noticed the malfunction. This would configure out a correctly functioning processor, and eventually the system could fail.

The output of the global executive is a list of processors to reconfigure out of the system. This list is broadcast to the reconfiguration task.

The Reconfiguration Task

The reconfiguration task receives inputs from the global executive and carries out the orders. Precomputed task and buffer tables exist for each possible number of functioning processors. Thus, for an eight-processor SIFT, there are eight sets of tables. Given a particular combination of working processors, each processor determines its *virtual processor number*. The processor uses this number to determine which tasks it should be running and to build the appropriate schedules. The act of reconfiguration involves: (1) mapping the real processor numbers into virtual processor numbers, (2) determining which set of tables to use, and (3) building the schedule tables.

The relationship of the tasks concerned with diagnosis and reconfiguration is shown in Figure III-6.

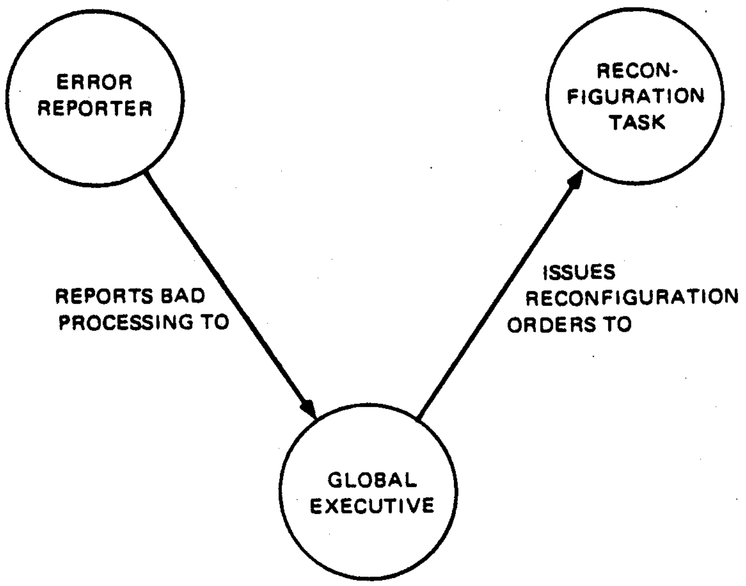


FIGURE III-6 TASKS CONCERNED WITH RECONFIGURATION

i. Clock Synchronization

The design of the SIFT system, as of other reliable systems that use voting, requires a degree of synchronization between processors. Exact-match voting requires identical outputs from the replicated programs on working processors, which implies that those replicated programs obtain identical inputs, a condition that itself requires an element of synchronization between those replicated programs. In SIFT, as in other designs, this synchronization is obtained by providing each processor with its own clock, by reference to which the various programs are scheduled, and by ensuring that the clocks of the various processors remain synchronized to within a tolerance. Designs that depend on bit-by-bit voting at the instruction execution level require quite tight clock synchronization, to within, say, 50 nanoseconds, but the buffering and software voting approach of SIFT allows synchronization at the task level, thus permitting a relatively large skew between clocks without risk; in the present case 50 microseconds is acceptable.

Even within the more relaxed skew constraints of SIFT, it is not possible to synchronize the clocks initially and expect to obtain a rate of drift low enough that the synchronization constraints can be satisfied for the required flight duration. To maintain a skew of less than 50 nanoseconds over a 10 hour flight implies a relative rate of drift of less than about 1.4×10^{-9} seconds/second. While such rates of drift are not impossible to obtain, they require clock technology of a type quite inappropriate for SIFT, whose clocks actually have individual drifts of about 10^{-5}. This is a factor of almost 10^{-4} more than can be accepted without continuous resynchronization.

The mechanism by which the clocks are resynchronized could be itself a cause of failure unless the design is such that a fault in any one clock cannot cause the other clocks to deviate from each other by more than the prescribed limits. An approach to fault-tolerant clock synchronization is discussed in the following section.

Fault-Tolerant Clock Synchronization Algorithms

Existing systems use a relatively simple algorithm to resynchronize clocks, which is widely purported to be "obviously" sound even in the presence of a faulty clock. This algorithm, for three clocks, requires that each processor observe the three clocks and reset its clock to the median clock of the three. This algorithm is deemed "obviously" sound since, with a single faulty clock, the median clock is either a good clock, in which case synchronizing to it is acceptable, or else it is a value between two good clocks, in which case synchronizing to that value is also acceptable. Daly, et al. [2], give an algorithm for four clocks, which is designed to be immune to transient spikes but which also claims to mask any form of behavior of a single faulty clock.

Both of these existing algorithms are at risk to a "malicious" clock that gives different time values to the other clocks. Consider three clocks, A, B, and C, of which A runs slightly faster than B and C has failed. If C reports to A a time value ahead that of A, and to B a time value behind that of B, then A and B will both see three clock values of which one is ahead of them, one is behind them, and their own clock value is the median. Thus A and B will both decide that no change in their clocks is required and will gradually drift out of synchronization. A similar failure mode exists for the Daly algorithm.

Even where the clock values are distributed as clock pulses on buses that lead to all other clocks, so that the same clock pulse is sensed by all other clocks, this type of fault could exist, caused by a degraded pulse generator and slightly different thresholds in the pulse receivers.

Appendix B describes three possible clock synchronization algorithms and proves their fault tolerance. The particular clock synchronization algorithm (interactive convergence) selected for SIFT requires four clocks to mask any single faulty clock, or 3N+1 clocks to mask any N simultaneous clocks. The algorithm is:

- Each processor obtains the values of all the other clocks remaining in the configuration, comparing those values to the value of its own clock to find the apparent skew between those clocks and its own
- The processor regards the skew of its own clock as zero
- The skews are examined to find any that are greater than some threshold. Such skews are set to zero
- The arithmetic mean of the skews is calculated and that mean is used to correct the processor's own clock.

```
If

N is the total number of clocks remaining in
  the configuration
G is the number of good clocks
F is the number of failed clocks
T is the time between resynchronizations
t is the time required to resynchronize
r is the maximum rate of drift of a good clock
e is the maximum reading error in obtaining the time
  value of a clock
E is the bound on the maximum clock skew (δ in Appendix B)
Q is the threshold for rejection of a clock skew (Δ in Appendix B)
```

then it can be shown that the maximum clock skew can be bounded by E provided that

$$E > \frac{(3\ F\ r(T + t) + 2Ne + 2Grt)}{(N - 3F)}$$

and the threshold Q satisfies

$$E + e + r(T+t) < Q < \frac{G(E-2e-2rt) - Fr(T+t)}{2F} .$$

Given the expected SIFT parameters of
N=4, F=1, G=3, T=1 ms, r=1 μs, e=5 μs

we obtain E > 43 μs. Bounds for Q, given several values of E (all values in microseconds) are:

for $E = 45$	$51 < Q < 52$
$E = 50$	$56 < Q < 59$
$E = 55$	$61 < Q < 67$
$E = 60$	$66 < Q < 74$

The most uncertain and critical parameter is e. Should e turn out to be 10 microseconds, we would obtain $E > 83$ μs. Bounds for Q, given several values for E (all values in microseconds) are:

$E = 85$	$96 < Q < 97$
$E = 90$	$101 < Q < 104$
$E = 95$	$106 < Q < 112$
$E = 100$	$111 < Q < 119$

Interactive Consistency

The three channel, majority-voting structure of SIFT depends on an assumed initial condition that at least two of the three channels have guaranteed the correct results. Thereafter the algorithm of having three independent channels voting and recalculating, with, at most, one faulty channel, suffices to maintain that condition of least two correct results. We must, however, ensure the assumed initial condition.

Consider an item of information entering the system from an unreplicated source. To obtain the three channel operation, this item of information must be replicated from its single source to each of three channels. This operation may be affected by a fault such that one or more of the channels obtains erroneous information. There are several possible outcomes:

- The three channels all obtain the same information, which may be correct or erroneous

- Two of the channels obtain one value, but a different value is given to the third channel; the next voting will allow all three channels to obtain the same value, though not necessarily the correct value

- All three channels are given different values; now the majority vote is unable to select any one of these, and some default value must be used.

Clearly if the information is a composite of several component values and the three channels each have plausible but different values, voting on each component independently might find agreement between different pairs of processors for each component. This would result in a 'nonsense' value but would not give an indication of the real lack of consensus.

Even in a broadcast system using common buses (unlike SIFT) it is possible for a single faulty source to deliver different values to the three channels. For example, a fault may cause a marginally degraded output pulse, which is recognized by some channels and not by others. This is sufficient to cause all the problems discussed here.

Special care must be taken where one of the three channels is the single source of the information. Figure III-7 shows a typical case of a three-channel system, in which channel A generates a single value to be replicated on the three channels. Attached to each arrow is the value transmitted. Consider the possibility that channel A is faulty so that the values generated by it are arbitrary. In the illustrated case, the faulty channel always sends value Y to channel B and value Z to channel C.

Note that channels B and C will each be able to obtain a majority among their inputs, but they will obtain different values.

It might be thought that this problem could be resolved by a slight increase in sophistication, perhaps by careful analysis of the voter error reports or by a further exchange of information between channels. However, Pease, et al. [24], demonstrate that, for a three-channel system, no such algorithm can exist and that four channels are required to mask all faults of this type. More generally they demonstrate that, to mask N such faults simultaneously, 3N+1 channels are required, and they describe the algorithms required.

For a four-channel system, the algorithm of Pease, et al., requires that the single source and the three channels to which the information is first distributed must be independent, so that a single fault cannot affect both the single source and any of the three channels. The sequence of actions is:

1. The source, possibly a processor, generates or obtains the information

2. The source distributes the information to three other processors

3. The three other processors distribute their copies of the information to whoever requires it

4. A user of the information, receiving three copies of it, performs a majority vote. If a majority exists, and only one fault is present, every working processor will obtain the same majority. If all three values are different, a default value must be used, and every working processor will also fail to find a majority and will use the default.

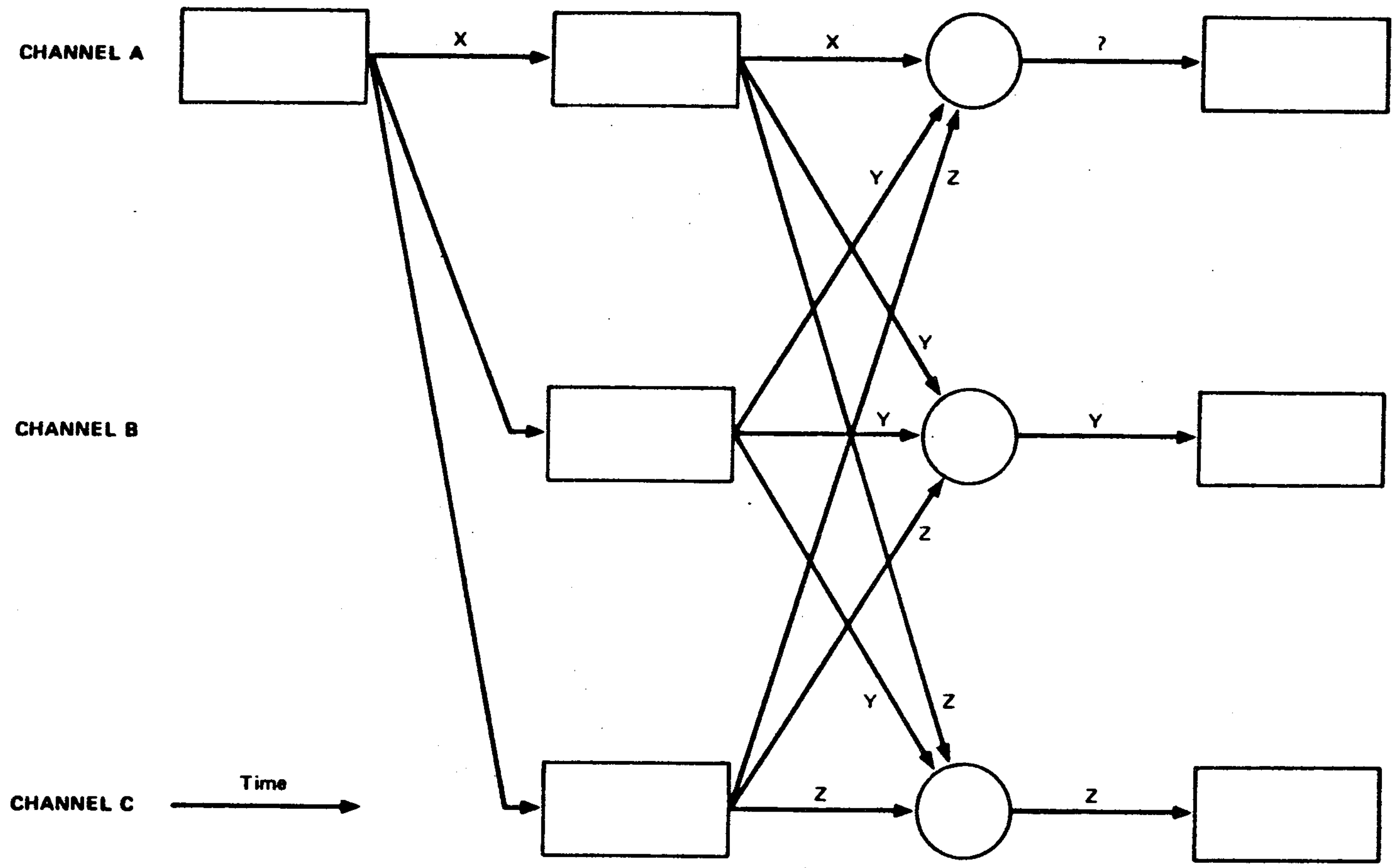


FIGURE III-7 LOSS OF INTERACTIVE CONSISTENCY IN A THREE-CHANNEL SYSTEM

For the extension of this algorithm to cover more than one fault, and for the proofs, see Pease, et al. [24].

Consider now the input of a sensor value from a replicated sensor. The sequence of actions is:

1. The value from each of the replicated sensors is read by a processor, probably a different processor for each sensor

2. Each of the processors distributes its sensor value to at least three other processors, and these then redistribute their values to the processors that are to use the sensor

3. A user of the sensor now has three copies of the value for each of the sensors; using the standard interactive consistency algorithm above, the three copies are voted to obtain a single value for each of the replicated sensors, a value that is known to be consistent for every working processor

4. The user of the sensor may now examine the sensor values for plausibility, filter them, average them, or perform whatever other processing is required by the application; none of this application-dependent merging of the values from replicated sensors should be performed before interactive consistency has been assured for each sensor value individually.

If the sensor itself were capable of recording its value at appropriate intervals and subsequently making that value available to all processors, it might be possible to avoid requiring a single processor to read and redistribute that value. However, the sensor would be required to remain synchronized with the processors, and that synchronization requires knowledge of the adaptive reconfiguration. Thus, the sensor would rapidly become quite complex and would require a processor with its own local executive. A much simpler design results from using the SIFT processors to read and staticize sensor values.

In addition to the input of sensor values, there are three other places in the SIFT system in which information from a single source might be replicated across a multichannel system. These are:

1. The error reports to the global executive, discussed in that context

2. The resynchronization of the processors' clocks, discussed in that context

3. Information transfer from an unreplicated application program to a replicated application program, currently thought to be unlikely and not provided for at present in SIFT.

B. SIFT EXECUTIVE SOFTWARE SPECIFICATIONS

This section is an introduction and guide to Appendix D, which presents formal specifications for all of the SIFT executive software, except for the Interactive Consistency and Clock Synchronization functions. The specifications are written in the language SPECIAL, which is a component of the SRI-developed software methodology HDM (Hierarchical Development Methodology) [27]. The present version of SPECIAL is incapable of representing time and multiprocessing, which are essential issues in the consistency and synchronization functions.

A brief tutorial on SPECIAL, which is extracted from a recent National Bureau of Standards report [19], is presented in Appendix C. Perusal of that appendix will aid in understanding the following discussion.

1. Introduction

The specification of SIFT consists of two parts: the specifications of the SIFT models and the specifications of the SIFT PASCAL program, which actually implements the SIFT system. The code specifications are the last of a hierarchy of models describing the operation of the SIFT system and hence are related to the SIFT models as well as the PASCAL program. These specifications serve to link the SIFT models to the running program.

In order to facilitate proof of the consistency between the PASCAL program and the code specifications, the specifications are very detailed and closely follow the form and organization of the PASCAL code. In addition to describing each of the components of the SIFT code, the code specifications describe the assumptions of the upper SIFT models, which are required to prove that the code will work as specified. These constraints are imposed primarily on the schedule tables.

Appendix D assumes an understanding of the motivation and basic algorithms of SIFT. An acquaintance with the Hierarchical Development Methodology (HDM) and SPECIAL is helpful but not required. This SIFT specification was written in a variant of SPECIAL developed as part of a PASCAL code verification system. The specification is written as a series of paragraphs with names such as TYPES and PARAMETERS. Comments are enclosed in "$(...)" and serve only as informal descriptions of the formal specification. The data objects of SIFT are called VFUNs in the specification but otherwise are identical to the variables of the SIFT program.

The code specification is not a complete description of the SIFT program. The SIFT system consists of a number of processors all working concurrently in approximate synchronization. The specification given in Appendix D considers only one of those processors and contains no description of how the processors communicate or how they maintain synchronization. Any part of the SIFT program that explicitly involves the passage of time is outside the realm of the specification.

The next sections will refer in detail to the SPECIAL specification of SIFT executive software found in Appendix D.

2. The Type Declarations

The TYPES paragraph at the top of the specification declares the data types used in the SIFT implementation as well as some types used purely for specification purposes. These data types are very similar in meaning to data types used in most programming languages such as PASCAL.

The data types are broken into sections for each major data component of SIFT. The first data component is the schedule table whose data type is called SCHED_ARRAY. An array of arrays, it has a component for each processor, each configuration, and each subframe. The second component is the datafile through which the processors communicate. The datafile type is called the DATAFILE_ARRAY and has components for each taskname and each element of the results produced by each task. The POLL_ARRAY type describes an entry for each configuration, each processor, and each task and contains a boolean value indicating whether that task is run by that processor in that configuration. The ERROR_ARRAY type describes an array containing the error count for each processor. The INPUT_ARRAY has an element array for each task.

Some data types are used for specification purposes and for internal processing in the implementation. SET_OF_INT describes a set of integers in the usual mathematical sense of a set. TASK_ARRAY is an array of integers indexed by task name. BOOL_ARRAY is an array of boolean values.

3. The Parameter Declarations

The parameters of the specification correspond to the constant values of the implementation. The actual values of these objects are not specified, but, instead, sufficient constraints are placed on the possible values so that the proof will succeed. The parameters and their meanings are given below:

frame_size	The number of subframes in a frame
max_processors	The maximum number of processors in any configuration
my_processor	The physical number of this processor
max_activities	The maximum number of activities allowed in any subframe
max_elems	The maximum number of values any task can produce

max_tasks	The maximum number of tasks in the system
bottom_val	The special value returned when a task does not run and when no majority is found in voting
err_threshold	The number of error reports before a processor is considered faulty
vote,dummy_vote, execute	The possible activities of a subframe
reconfig	The name of the reconfiguration task
global_exec	The name of the global executive task
error_report	The name of the error reporting task
null_task	The name of the null or maintenance task
sched_table	The schedule table for each processor, configuration, and subframe, giving a set of activities to perform
poll	Determines whether a processor in a given configuration runs a particular task
inputs	Gives the names of the tasks that will produce results used by a particular task
i_c	Indicates which tasks are interactive-consistency tasks
result_size	Gives the number of values returned by each task
error_i_c_tasks	The names of the error-reporting, interactive-consistency tasks.

4. The Definitions

Definitions are used as a convenient way to name a concept in the specification for easy reference. They are equivalent to pure mathematical functions in that they take a sequence of arguments and produce a determined result. The only definition in the specifications defines the concept of a majority of a set of values. The values being voted on are found in the datafile corresponding to each processor that ran the task as indicated by the POLL. The definition compares the set of all processors that ran the task to the set of all processors that agree on the resulting value. If the second set is larger than half the first set, a majority value has been found.

5. The Parameter Invariants

This section of the specifications contains the constraints on the parameter values mentioned in the above section on parameters. Some of the constraints seem very obvious, but they must be made explicit in order for the mechanical verification effort to succeed. Each constraint is described below (the hyphens provide for human-readable separations of phrases, while meeting machine requirements for continuous symbol strings in the names of variables):

1. frame_size, max_processors, max_activities, max_elems, and max_tasks should all be positive values.

2. 1 <= my_processor <= max_processors The processor number of this processor must be a legitimate processor number.

3. The activities vote, dummy_vote, and execute should be positive numbers and each should be different.

4. The tasks reconfig, global_exec, null_task, the error reporting tasks, and the error-reporting, interactive-consistency tasks should all be different tasks, that is, not equal to each other.

5. An execute that uses values must follow the vote or dummy_vote on those values.

6. There is never scheduled both a vote and a dummy_vote on the results of a task during a subframe.

7. The results of an execute are not voted on during the same subframe they are produced.

8. Reconfiguration is always the last task run in the subframe. This is because reconfiguration completely changes the current schedule so it is not possible to continue the old schedule.

9. No vote is scheduled during the same subframe as an error report. That is because a vote might change the error count that the error-reporting tasks broadcast.

10. The result of an execute is only voted on once.

11. No task is executed more than once in any subframe.

12. The only activities scheduled are vote, dummy_vote, and schedule.

13. No other activities are scheduled after the null task is started to fill out the subframe.

14. The global executive only takes as input itself (the previous execution) and the error-reporting and interactive-consistency tasks.

15. The error-reporting and interactive-consistency tasks broadcast their inputs.

16. The global executive considers a processor to be no longer working if it was not previously working or if a majority of the other processors have declared it to be bad.

6. The Specification Functions

The specification functions serve two purposes. They define the names and types of the variables that make up the state of the system, and they define the operations that modify the state during execution. The state variables correspond to the global PASCAL variables of the SIFT implementation and the operations of the specification correspond to the FUNCTIONs and PROCEDUREs of the implementation. The variables are described first, followed by the operations.

1. Subframe contains the current subframe number which ranges from 0 to one less than the maximum number of subframes.

2. Config contains the current configuration. This is the number of processors that are currently considered to be working.

3. Input is an array of values waiting to be input to particular tasks. They are the result of voting on the outputs of other tasks.

4. Datafile is the broadcast area where results of tasks are placed and automatically broadcast.

5. Errors is the number of errors counted for each processor due to nonagreement in voting.

6. Real_to_virt is a mapping from a real physical processor number to the processor number used in the schedule tables for this configuration.

7. Virt_to_real is a reverse mapping of real_to_virt.

The main procedure of the implementation is Dispatcher, which is invoked at the start of each subframe due to a clock interrupt and which, in turn, invokes each of the activities of that subframe. The specifications of the dispatcher describe how each of the state variables described above is changed according to the schedule tables. The change to each state variable is described in sequence:

1. The subframe number is incremented by one, indicating that another subframe has elapsed. If the end of the frame is reached, the subframe number is reset to zero

2. Config is changed only if a reconfiguration was done during this subframe. Its new value corresponds to the number of processors reported working by the global executive

3. Input is changed to reflect the updated values found by voting on values produced by tasks in other subframes. For each entry in the input table, if no vote was done in this subframe on the value, then the value remains unchanged. If there was a vote, the new input value corresponds to the majority value computed by the vote

4. The entries in the datafile are updated when tasks broadcast their results. If a task did not run during this subframe, then its old values from the previous subframe remain. If the task did run, its output values are placed in the datafile

5. If any voting found less than total agreement, then the error counts were incremented by the number of nonagreements found

6. If a reconfiguration was done, the variables real_to_virt and virt_to_real were correctly updated to reflect the new processor configuration.

The dispatcher calls a variety of other routines to perform the activities. The Vote_activity routine conducts a vote by collecting the values to be voted on and calling the three way voter VOTE3. Dummy_vote just replaces the old values by the special value. The global executive counts up the error reports and decides which processors are running. The reconfiguration task uses the results of the global executive to reconfigure the system. Since the actual application tasks running on the SIFT computer are not known for this proof exercise, a general routine representing all of the possible application tasks is included.

C. PASCAL CODE FOR THE VOTER

The following is a PASCAL code for the voter, presented as an example of the software that makes up the SIFT operating system. The actual voter may differ in slight details. The code has been designed without loops, in order to provide for the fastest possible execution of this important function.

```
var errors:array[0..7] of integer;
        p1,p2,p3,p4,p5,v1,v2,v3,v4,v5:integer;

procedure fail;
        begin
        (* All returned values are wrong, so report all
           processors involved. This could be done
           in line, but it would take too much room.
           The minor additional time that it takes to
           call the subroutine is probably worthwhile. *)
        errors[p1]:=errors[p1]+1;
        errors[p2]:=errors[p2]+1;
        errors[p3]:=errors[p3]+1;
        errors[p4]:=errors[p4]+1;
        errors[p5]:=errors[p5]+1
        end;

function vote5(default:integer): integer;
        begin
        (* This is the five way voter.  It assumes that
           V1 .. V5 is initialized with the 5 values to
           be voted, and P1 .. P5 has the corresponding
           processors.  Default is returned in the case
           that there is no majority value.  The
           procedure is basically a simple IF tree
           (pruned where possible) to achieve the
           quickest possible vote. *)
        if v1=v2 then
            if v1=v3 then
                begin
                if v1<>v4 then errors[p4]:=errors[p4]+1;
                if v1<>v5 then errors[p5]:=errors[p5]+1;
                vote5 := v1
                end
            else if v2=v4 then
                begin
                errors[p3]:=errors[p3]+1;
                if v1<>v5 then errors[p5]:=errors[p5]+1;
                vote5 := v1
                end
            else if v1=v5 then
```

```
          begin
          errors[p3]:=errors[p3]+1;
          errors[p4]:=errors[p4]+1;
          vote5 := v1
          end
      else if v3=v4 then
               if v3=v5 then
                   begin
                   errors[p1]:=errors[p1]+1;
                   errors[p2]:=errors[p2]+1;
                   vote5 := v3
                   end
                else
                   begin
                   fail;
                   vote5:=default
                   end
      else
               begin
               fail;
               vote5:=default
               end
  else if v1=v3 then
          if v1=v4 then
                   begin
                   errors[p2]:=errors[p2]+1;
                   if v1<>v5 then
                           errors[p5]:=errors[p5]+1;
                   vote5 := v1
                   end
          else if v1=v5 then
                   begin
                   errors[p2]:=errors[p2]+1;
                   errors[p4]:=errors[p4]+1;
                   vote5 := v1
                   end
          else if v2=v4 then
                   if v2=v5 then
                           begin
                           errors[p1]:=errors[p1]+1;
                           errors[p3]:=errors[p3]+1;
                           vote5 := v2
                           end
                   else
                           begin
```

```
                    fail;
                    vote5:=default
                    end
          else
                 begin
                 fail;
                 vote5:=default
                 end
     else if v4=v5 then
          if v2=v4 then
                 begin
                 errors[p1]:=errors[p1]+1;
                 if v2<>v3 then
                        errors[p3]:=errors[p3]+1;
                 vote5 := v2
                 end
          else if v1=v5 then
                 begin
                 errors[p2]:=errors[p2]+1;
                 errors[p3]:=errors[p3]+1;
                 vote5 := v1
                 end
          else if v3=v5 then
                 begin
                 errors[p1]:=errors[p1]+1;
                 errors[p2]:=errors[p2]+1;
                 vote5 := v3
                 end
          else
                 begin
                 fail;
                 vote5:=default
                 end
     else if v2=v5 then
          if v2=v3 then
                 begin
                 errors[p1]:=errors[p1]+1;
                 errors[p4]:=errors[p4]+1;
                 vote5 := v2
                 end
          else
                 begin
                 fail;
                 vote5:=default
                 end
```

```
        else if v2=v3 then
                if v2=v4 then
                        begin
                        errors[p1]:=errors[p1]+1;
                        errors[p5]:=errors[p5]+1;
                        vote5 := v2
                        end
                else
                        begin
                        fail;
                        vote5:=default
                        end
        else
                begin
                fail;
                vote5:=default
                end
        end;

function vote3(default:integer):integer;
        begin
        (* This is the 3 way voter.  It assumes that
           V1 .. V3 contains the 3 values to be
           voted, and that P1 .. P3 contains the
           processors. *)
        if v1=v2 then
                begin
                if v1<>v3 then errors[p3]:=errors[p3]+1;
                vote3:=v1
                end
        else if v1=v3 then
                begin
                errors[p2]:=errors[p2]+1;
                vote3:=v1
                end
        else if v2=v3 then
                begin
                errors[p1]:=errors[p1]+1;
                vote3:=v2
                end
        else
                begin
                errors[p1]:=errors[p1]+1;
                errors[p2]:=errors[p2]+1;
                errors[p3]:=errors[p3]+1;
```

```
          vote3:=default
          end
end.
```

IV. Validation and Verification

This chapter presents results on the analysis of SIFT system reliability. The first part is concerned with the structure and implications of a Markov-type reliability model of fault-tolerant behavior under permanent and transient fault conditions. Standard analysis methods are used, but special issues arise dealing with the complexities of transient behavior.

The second part is concerned with the validation of the reliability model. This is accomplished by verifying that the actual design and implementation of the SIFT system are consistent with the model. The verification consists of a formal (i.e., mathematical) proof of correctness, using techniques hitherto employed in program proving. The present application to computer system verification is novel, and is intended to have general applicability to reconfigurable fault-tolerant computer systems.

A. THE MARKOV RELIABILITY MODEL

A sufficiently catastrophic sequence of component failures will cause any system to fail. The SIFT system is designed to be immune to certain likely sequences of failures. To guarantee that SIFT meets its reliability goals, we must show that the probability of a more catastrophic sequence of failures is sufficiently small.

The reliability goal of the SIFT system is to achieve a high probability of survival for a short period of time--e.g., a ten-hour flight--rather than a large mean time before failure (MTBF). For a flight of duration *T*, survival will occur unless certain combinations of fault events occur within the interval *T* or have already occurred prior to the interval *T* and were undetected by the initial checkout of the system. Operationally, faults of the latter type are indistinguishable from faults that occur during the interval *T*.

To estimate the probability of system failure, we have developed two finite-state Markov-like reliability models in which the state transitions are caused by the events of fault occurrence, fault detection, and fault "handling." The combined probability of all event sequences that lead to a failed state is the system failure probability. A design goal for SIFT is to achieve a failure rate of less than 10^{-9} for a ten-hour period.

For the reliability model, we assume that hardware fault events and electrical transient fault events are uncorrelated and exponentially distributed in time (constant failure rates). These assumptions are believed to be accurate for hardware faults because the physical design of the system prevents fault propagation between functional units (processors and buses) and because a multiple fault within one functional unit is no more serious than a single fault. The first model assumes that all faults are permanent (for the duration of the flight), so it does not consider transient faults. The second model extends this analysis to include the effects of transient faults. Failure rates for hardware have been estimated on the basis of acti-e component counts, using typical reliability figures for similar hardware.

For the first model we let the state of the system be represented in the reliability model as a triple of integers (h,d,f) with $f \geq d \geq h$, where such a state represents a situation in which f faults of individual processors have occurred, d of those faults have been detected, and h of these detected faults have been "tolerated" by reconfiguration. There are three types of possible state transition.

1. $(h,d,f) \rightarrow (h,d,f+1)$, representing a fault in a processor

2. $(h,d,f) \rightarrow (h,d+1,f), d<f$, representing the detection of a fault

3. $(h,d,f) \rightarrow (h+1,d,f) h<d$, representing the tolerating of a detected fault.

This is illustrated in Figure IV-1.

The first two types of transition-processor fault and fault detection, represented in Figure IV-1 by arrows--are assumed to have constant probabilities per unit time. However, the third type of transition--failure tolerating, represented in Figure IV-1 by vertical arrows--represents the completion of a reallocation procedure. We assume that this transition must occur within some fixed length of time τ.

A state (h,d,f) with $h<d$ represents a situation in which the system is reconfiguring. To make the system immune to an additional fault while in this state is a difficult problem, since it means that the procedure to reconfigure around a fault must work despite an additional, undetected fault. Rather than assuming that this problem could be solved, we took the approach of trying to ensure that the time τ that the system remains in such a state is small enough to make it highly unlikely for an additional fault to occur before reconfiguration is completed. We therefore made the pessimistic assumption that a processor fault that occurs while the system is reconfiguring will cause a system failure. Such failures are represented by the "double-fault" transitions indicated by asterisks in Figure IV-1. In this module, which assumes three way voting, we assume that each of these transitions results in a system failure.

We have calculated the probability of system failure through a double fault transition, and also through reaching a state with fewer than two nonfaulty processors, for which we say that the system has failed because it has "run out of spares."* A brief summary of these failure probabilities for a five processor system is shown in Table IV-1.

The first model indicates that the exhaustion of spares was not a significant failure mode

*The probability of system failure because of multiple *undetected* faults has not been computed precisely, but is expected to be comparable to the double fault values.

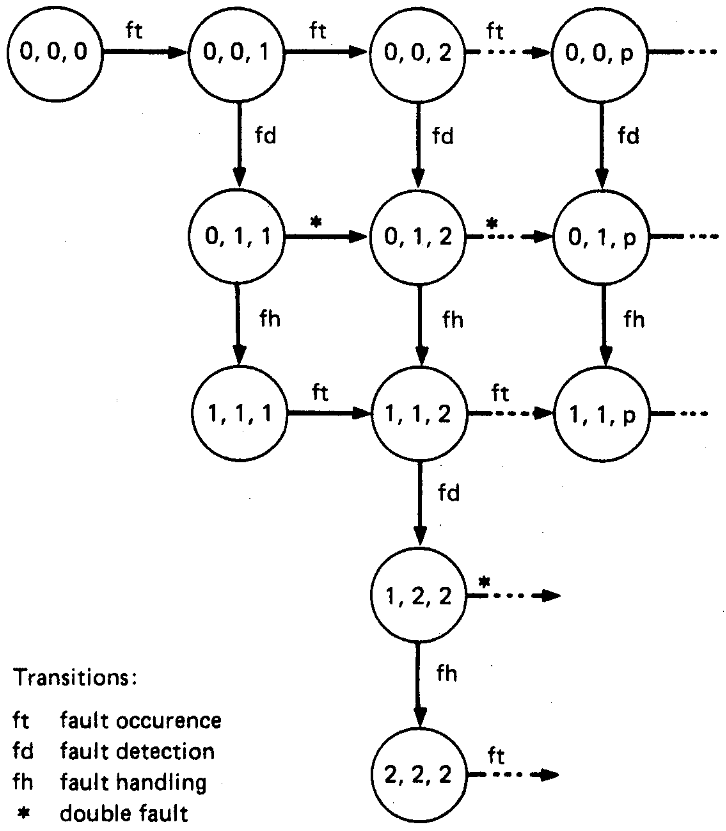


FIGURE IV-1 INITIAL RELIABILITY MODEL

Table IV-1

FAILURE PROBABILITY FOR VARIOUS CAUSES

FAILURE CAUSE	FAILURE PROBABILITY
Exhaustion of spares	5×10^{-12}
Double Fault (T = 100 ms)	7×10^{-11}
Double Fault (T = 1s)	7×10^{-10}

T = 10 hours

in comparison to the occurrence of a second fault before completion of the reconfiguration for a first fault, but superficial calculation indicates that coincidences between transient faults, or between transient faults and reconfigurations from solid faults, might also be significant failure modes. Consequently, a second reliability model, considering both solid and transient faults, has been constructed. This model will be used as a high-level specification of SIFT fault-tolerant behavior for the purpose of verifying the SIFT design.

Since the old model did not consider transient faults, the global executive must either regard transient faults as solid, with a risk of exhaustion of spares, or else must be proven able to recognize and ignore transient faults, which presents an almost impossible proof. A Markov model that represents both solid and transient faults, with appropriate probabilities for correct and incorrect recognition of them, allows the global executive to be designed in a more natural and effective manner while still remaining provable.

The second Markov model consists essentially of the addition of transient errors to the previous Markov model for SIFT. The previous model consisted of a two-dimensional matrix of states, and the addition of transient errors converts this into three dimensions. The three indices into the state matrix are:

1. The number of processors that have been removed from the configuration by reconfiguration

2. The number of processors that have developed solid faults

3. The number of processors that have developed transient faults, and for which the error symptoms of those transient faults are still under consideration by the global executive.

Figure IV-2 shows an example of projecting the model into the plane of the first two indices. The resulting model is clearly very similar to the previous model with the addition of the transitions caused by the incorrect diagnosis of a transient-fault symptom resulting in the conclusion that the transient fault was solid; this, of course, results in the reconfiguration of a working processor out of the system.

Figure IV-3 shows a set of states and transitions resulting from transient errors for one of the states of Figure IV-2. This rather complex diagram results from considering up to three successive transient faults.

Note the transitions from (a,b,1) to (d) and from (d) to (a,b,2). The first of these transitions represents the rate of occurrence of transient faults, while the second represents the rate at which the errors due to transient faults are masked by voting and removed from the system. Typically, errors due to transient faults are masked at the end of the iteration in which the fault occurs. Before the error is masked, the system is vulnerable to further faults, solid or transient. Once the masking has occurred and the fault no longer constitutes a direct risk to the system, the symptoms of the fault remain to be diagnosed by the global executive. To facilitate the analytic reduction of this model, discussed below, the possibility of a solid fault occurring while in state d is neglected, and thus no transition for such a fault is depicted. This has a negligible effect on the results because of the relatively slow rate of occurrence of solid faults.

The transition (d) to (k) represents the rate of occurrence of a second transient fault before the errors due to the first are fully masked, and that from (k) to (r) represents a third transient fault occurring before even the first is masked out. In contrast, the transition (a,b,1) to (m) represents a second transient fault occurring after the first has been masked but before its symptoms have been considered by the global executive. Thus states (d), (m), and (t) are states with errors due to one transient fault to hazard the system, states (k) and (s) are states with errors due to two transient faults, and state (r) has errors due to three transient faults and, thus, is a state with a high risk of system failure. States (a,b,0), (a,b,1), (a,b,2), and (a,b,3) are states with no unmasked errors due to transient faults. In all cases the effect of solid faults must be considered with the errors due to transient faults to determine the risk of system failure, and thus the failure-rate transitions are not shown on Figure IV-3.

Consider next the transition (a,b,1) to (e) and the states (e), (f), and (g). The transition (a,b,1) to (e) represents the rate at which the global executive is able to diagnose the symptoms of a single fault and to reconfigure the system accordingly. States (e), (f), and (g) are used only to distinguish the possible decisions made by the global executive, and thus do not represent states of significant occupancy in the Markov model. This is achieved by attaching a very fast transition rate, called "fast," to each of them. The ratio of their transition rates determines the probabilities of the various decisions. Transitions (e) to (f) and (e) to (g), respectively, represent the probability that the global executive considers a transient-fault symptom or a solid-fault symptom. The model

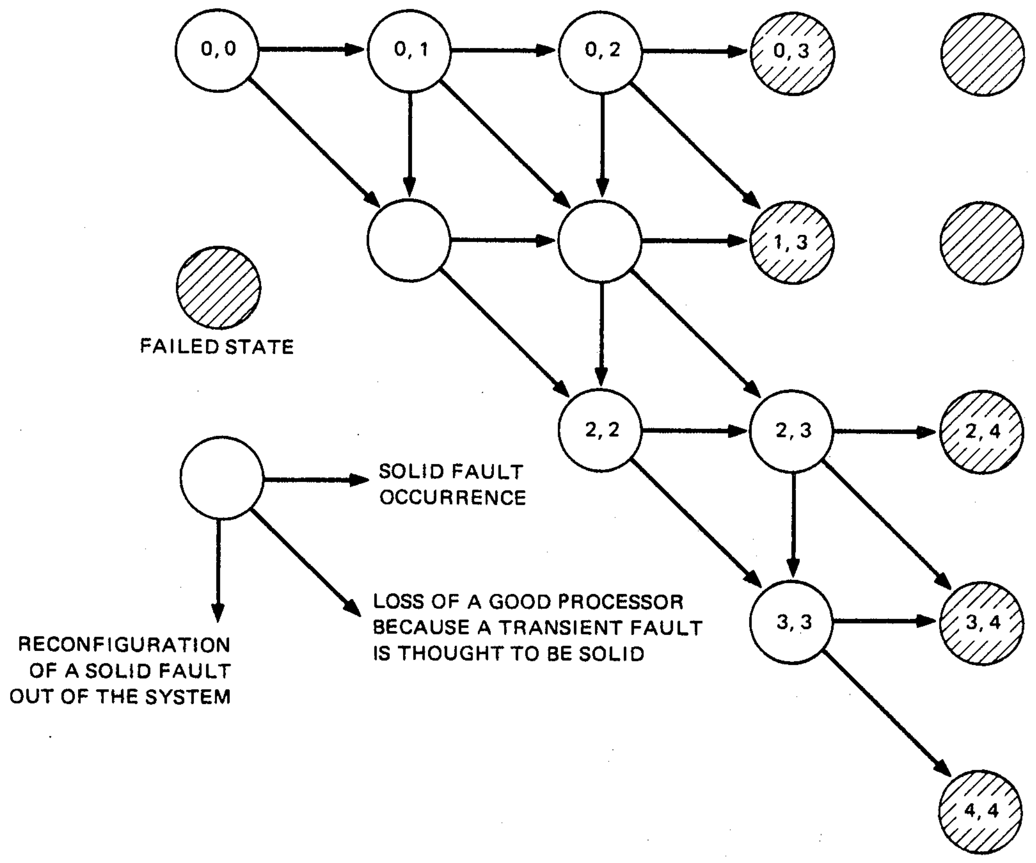

0, 0
0, 1
0, 2
0, 3
1, 3
2, 2
2, 3
2, 4
3, 3
3, 4
4, 4
FAILED STATE
SOLID FAULT OCCURRENCE
LOSS OF A GOOD PROCESSOR BECAUSE A TRANSIENT FAULT IS THOUGHT TO BE SOLID
RECONFIGURATION OF A SOLID FAULT OUT OF THE SYSTEM

FIGURE IV-2 THE STATES OF THE NEW MARKOV MODEL FOR A 5 PROCESSOR SYSTEM, PROJECTED INTO THE PLANE OF THE FIRST TWO INDICES, AND THUS NOT SHOWING TRANSIENT ERROR STATES.

The first index denotes the number of processors that have been removed from the system by reconfiguration, and the second index denotes the number of processors suffering from solid faults.

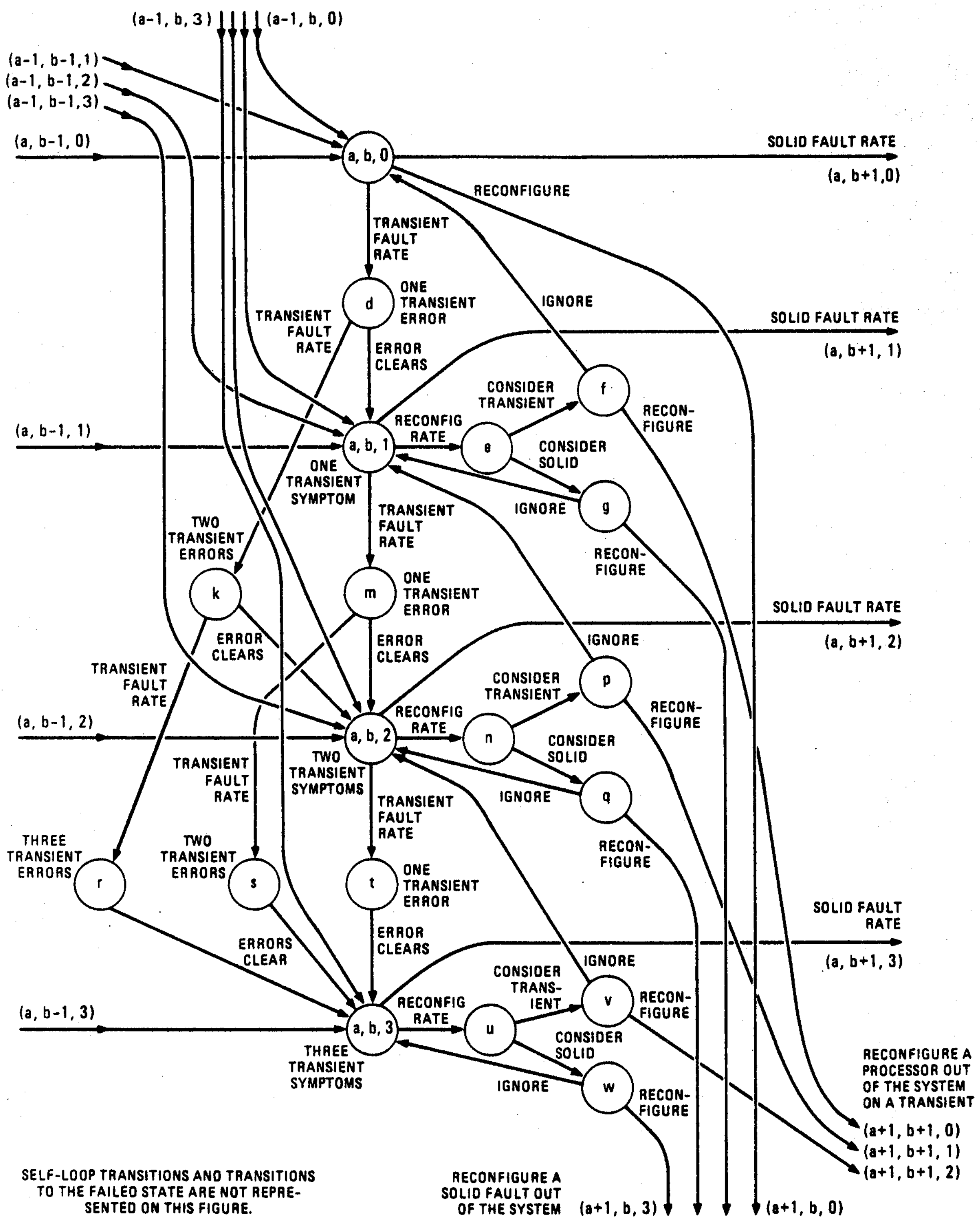


FIGURE IV-3 A SET OF STATES AND TRANSITIONS RESULTING FROM TRANSIENT ERRORS FOR ONE OF THE STATES OF FIGURE IV-2

assumes that fault symptoms are considered singly and that every outstanding fault has an equal chance of consideration.

For each fault considered, the global executive may correctly recognize the nature of the fault, or it may mistake a transient fault for a solid fault or vice versa. The model allows the user to define the probability of correctly recognizing a fault as a function of the number of faults outstanding. It is to be expected that a design for the global executive that reduces the risk of interpreting a transient fault as solid, with the resulting loss of a processor, will correspondingly increase the risk that a solid-fault symptom will be regarded as transient and will thus be ignored for a longer time.

If the global executive considers a transient fault and correctly identifies it, the transition is (f) to (a,b,0). If the global executive considers a solid fault and correctly identifies it, reconfiguring the system, the transition is (g) to (a+1,b,2), while, if the executive incorrectly regards it as a transient fault and ignores it, the transition is (g) to (a,b,1). While the global executive is considering the fault symptoms, further faults may occur. These are represented by (a,b,1) to (a,b+1,1) for a solid fault, and by (a,b,1) to (m) for a transient fault.

In practice the states of Figure IV-3 are more than are needed for the analysis of the system. They can be reduced to a set of equivalent states and transitions shown in Figure IV-4. The validity of this transformation is based on the large ratio of fault rate to reconfiguration rate for the various fault types.

Table IV-2 shows the rates for transitions in Figure IV-3, while Table IV-3 shows the corresponding rates for the reduced system of Figure IV-4. Table IV-4 gives the meanings of the symbols used in the rate tables.

Sample Results from the Model

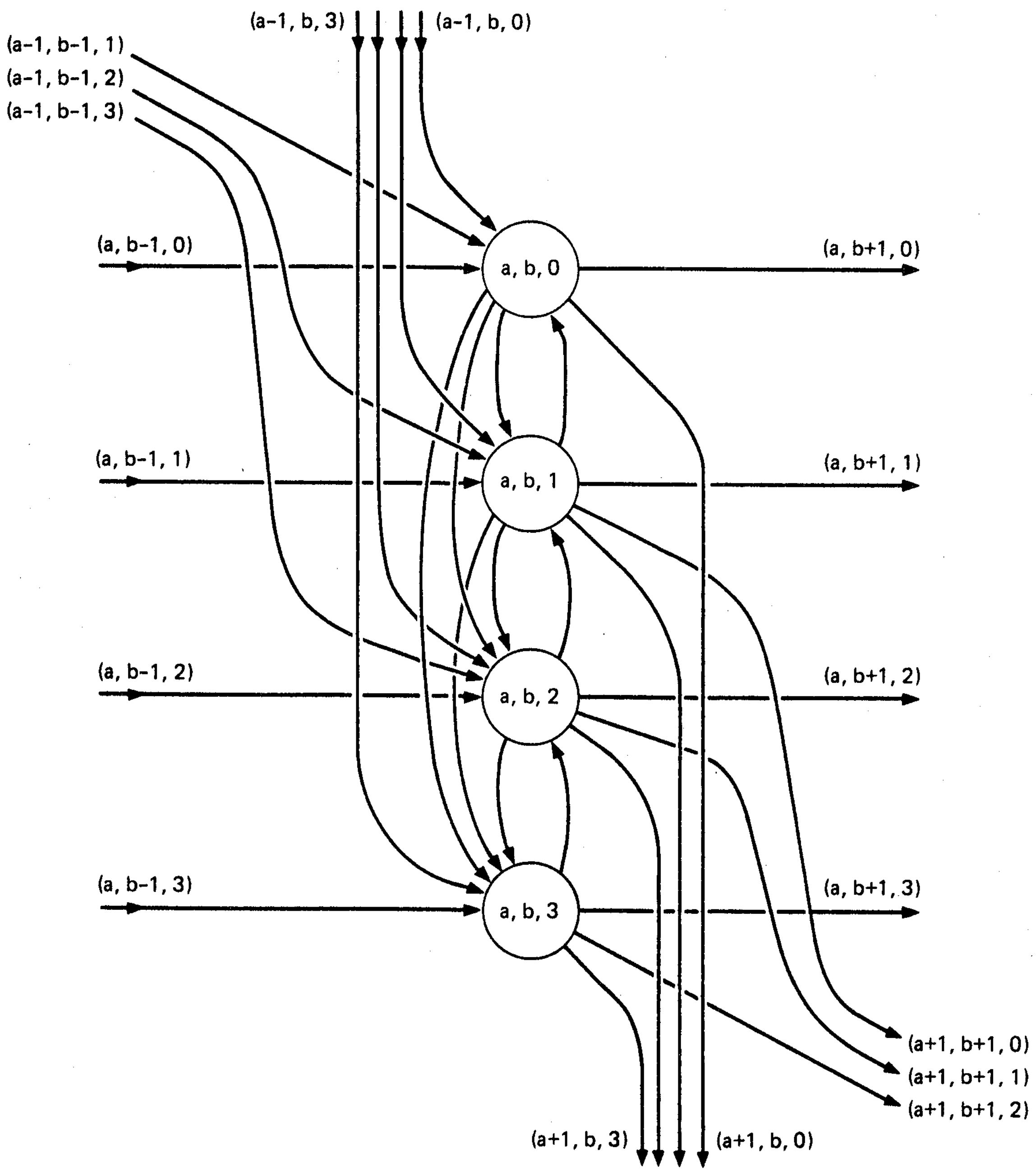
These sample results from the model are *not* intended to predict the actual reliability of SIFT. The parameters fed into the model for these sample results cannot be regarded as "considered representations" of SIFT component reliability or performance, and, thus, one should not regard the results as predictive of SIFT reliability. By appropriate (or inappropriate) choice of parameters, any reliability whatever could have been obtained from the model.

These sample results are, however, useful for

- Illustrating the capabilities of the model
- Demonstrating the sensitivity of system reliability to changes in the various parameters

Table IV-2: The transition rates between the states of Figure IV-3

a,b,0	=> a,b+1,0	$(N\text{-}b)\lambda$
a,b,0	=> d	$(N\text{-}b)\mu$
a,b,0	=> failed	if $F<2(b\text{-}a)$ then fast else O
d	=>a,b,1	ϕ
d	=> k	$(N\text{-}b\text{-}1)\mu$
d	=> failed	if $F<2(b\text{-}a)+2$ then fast else O
a,b,1	=> a,b+1,1	$(N\text{-}b)\lambda$
a,b,1	=> e	θ
a,b,1	=> m	$(N\text{-}b)\mu$
a,b,1	=> failed	if $F<2(b\text{-}a)$ then fast else O
e	=> f	(a/(b-a+1))fast
e	=> g	((b-a)/(b-a+1))fast
f	=> a,b,0	(1-Q(b-a+1))fast
f	=> a+1,b+1,0	(Q(b-a+1))fast
g	=> a,b,1	(P(b-a;1))fast
g	=> a+1,b,0	(1-P(b-a+1))fast
k	=> a,b,2	ϕ
k	=> r	$(N\text{-}b\text{-}2)\mu$
k	=> failed	is $F<2(b\text{-}a)+4$ then fast else O
m	=> a,b,2	ϕ
m	=> s	$(N\text{-}b\text{-}1)\mu$
m	=> failed	if $F<2(b\text{-}a)+2$ then fast else O
a,b,2	=> a,b+1,2	$(N\text{-}b)\lambda$
a,b,2	=> n	θ
a,b,2	=> t	$(N\text{-}b)\mu$
a,b,2	=> failed	if $F<2(b\text{-}a)$ then fast else O
n	=> p	(2/(b-a+2))fast
n	=> q	((b-a)/(b-a+2))fast
p	=> a,b,1	(1-Q(b-a+2))fast
p	=> a+1,b+1,1	(Q(b-a+2))fast
q	=> a+1,b,2	(P(b-a+2))fast
q	=> a,b,2	(1-P(b-a+2))fast
r	=> a,b,3	ϕ
r	=> failed	if $F<2(b\text{-}a)+6$ then fast else O
s	=> a,b,3	ϕ
s	=> failed	if $F<2(b\text{-}a)+4$ then fast else O
t	=> a,b,3	ϕ
t	=> failed	if $F<2(b\text{-}a)+2$ then fast else O
a,b,3	=> a,b+1,3	$(N\text{-}b)\lambda$
a,b,3	=> u	θ
a,b,3	=> failed	if $F<2(b\text{-}a)$ then fast else O
u	=> v	(3/(b-a+3))fast
u	=> w	((b-a)/(b-a+3))fast
v	=> a,b,2	(1-Q(b-a+3))fast
v	=> a+1,b+1,2	(Q(b-a+3))fast
w	=> a+1,b,3	(P(b-a+3))fast
w	=> a,b,3	(1-P(b+a+3))fast



SELF-LOOP TRANSITIONS AND TRANSITIONS
TO THE FAILED STATE ARE NOT REPRESENTED
ON THIS FIGURE.

FIGURE IV-4 A REDUCED SET OF EQUIVALENT STATES AND TRANSITIONS

Table IV-3: A set of transition rates appropriate to the reduced set of states shown in Figure IV-4.

a,b,c	=>	failed	if $F<2(b\text{-}a)$ then fast
			elseif $F<2(b\text{-}a)+2$ then $(N\text{-}b)\mu$
			elseif $F<2(b\text{-}a)+4$ then $(N\text{-}b)(N\text{-}b\text{-}1)(\mu*2)/\phi$
			else $(N\text{-}b)(N\text{-}b\text{-}1)(N\text{-}b\text{-}2)(\mu*3)/(\phi*2)$

a,b,c	=>	a,b,c+1	$(N\text{-}b)\mu$
a,b,c	=>	a,b,c+2	$(N\text{-}b)(N\text{-}b\text{-}1)\mu$
a,b,c	=>	a,b+1,c	$(N\text{-}b)\lambda$
a,b,c	=>	a+1,b,c	$P(b\text{-}a+c)\lambda*((b\text{-}a)/(b\text{-}a+c))$
a,b,c	=>	a,b,c-1	$(1\text{-}Q(b\text{-}a+c))\theta*(c/(b\text{-}a+c))$
a,b,c	=>	a+1,b+1,c-1	$Q(b\text{-}a+c)\theta*(c/(b\text{-}a+c))$

Table IV-4: The meanings of symbols used in the rates of Tables IV-2 and IV-3

Symbol	Meaning
N	The number of processors in the configuration

F	if $(N\text{-}a)\geq 5$ then 5
	elseif $(N\text{-}a)\geq 3$ then 3
	else 1

λ	The rate of solid fault occurrence in a single processor
μ	The rate of transient fault occurrence in a single processor
θ	The rate of reconfiguration
ϕ	The rate of masking of transient errors
fast	$\gg \max(\theta,\phi)$
P(x)	The probability of correctly diagnosing a solid fault in the presence of x solid fault or transient fault symptoms
Q(x)	The probability of incorrectly diagnosing a transient fault, and of regarding it as a solid fault, in the presence of x solid or transient fault symptoms.

- Showing target values for component reliability if a SIFT-like system is to meet its reliability goals.

To exercise the model, we have assumed a "standard" set of parameters for the SIFT system. Many of the results exhibited here involve varying one or more of these parameters; the graphs explicitly indicate the parameters that have been varied. Other parameters will remain at their standard values, which are:

Number of Processors	5
Number of Processors Voting	5 on global executive 3 on flight control tasks
Mission Duration	10 hours
Rate of Solid Faults	$2.0\text{x}10^{-4}$/hour
Rate of Transient Faults	$2.0\text{x}10^{-3}$/hour
Rate of Intermittent Faults	0.0/hour
Reconfiguration Time	10^{-4} hour

Again, these are illustrative values, and no representation is made that these are the correct values for SIFT.

Figure IV-5 shows how the reliability of the example system depends on the number of processors in the system.

Results are given for both (1) the case where all critical tasks are replicated three ways and three-fold voting is used to mask errors, and (2) the case where the global executive uses five-fold replication and voting. The difference between the reliability of these two cases is largely due to transient faults closely spaced in time, where the second fault occurs before the first fault can be fully masked, thus exposing the system to a risk of failure. It can be seen that, with inadequate replication and masking, there is a limit to the level of reliability attainable, and that the addition of further processors may degrade rather than improve reliability.

It is possible to design most of the flight-control application programs to be self stabilizing and thus relatively invulnerable to transient disturbances, particularly with the substantial inertia of commercial transport aircraft. Consequently, three way voting should suffice for flight-control programs. No such method has been found for the global executive. Certain portions of the flight-control application have characteristics similar to the global executive, particularly the flight-phase logic, and may need similar treatment.

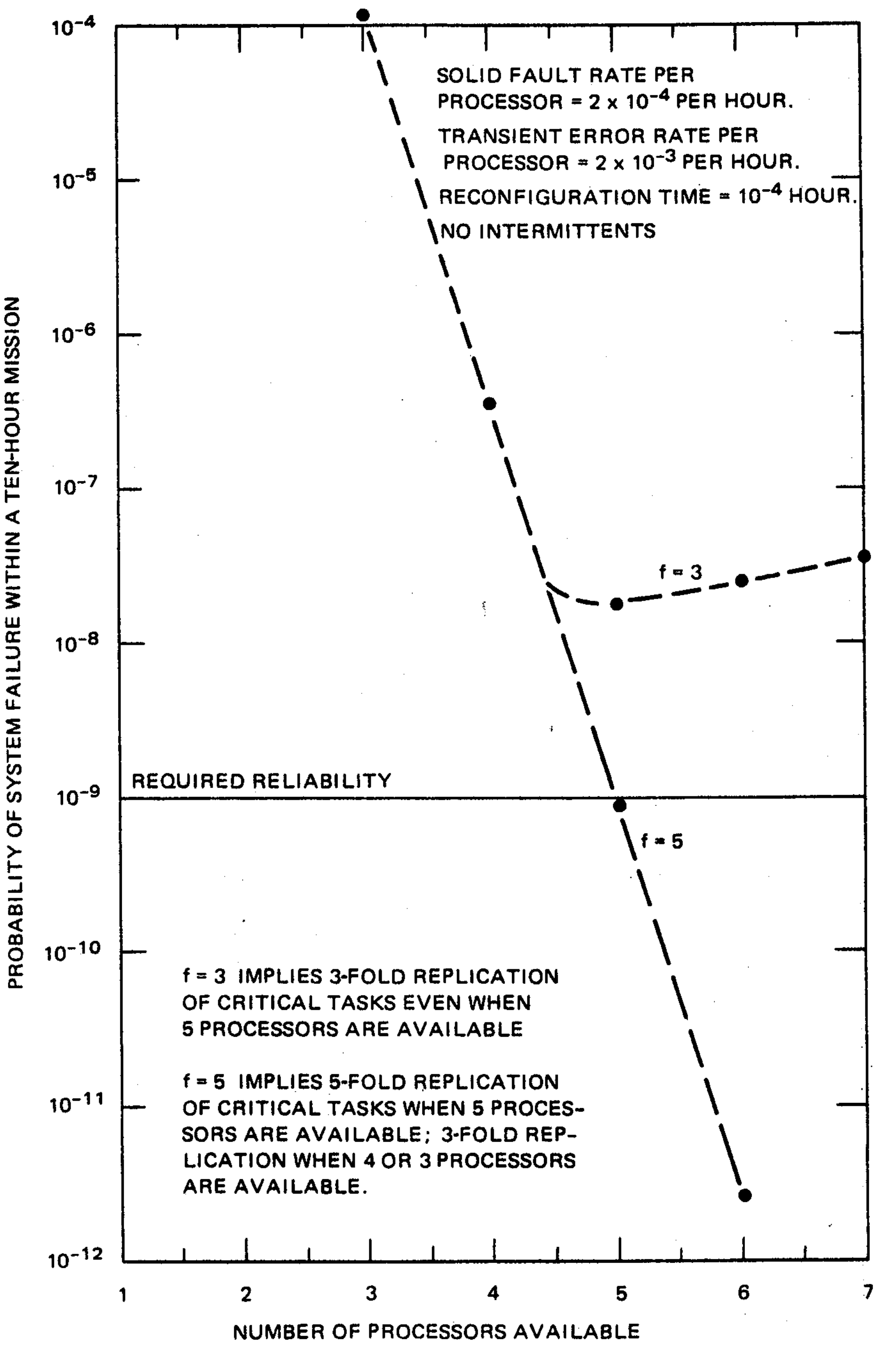


FIGURE IV-5 PROBABILITY OF SYSTEM FAILURE WITHIN A 10-HOUR MISSION VS NUMBER OF PROCESSORS

Figure IV-6 shows the probability of system failure during missions of various durations. The figure shows this probability both for the reference case where the global executive uses five-way voting, and also for three-way voting on the global executive. Again, the effect on reliability of transient errors closely spaced in time is evident. Without five-way voting on critical tasks, the required reliability is not achieved even for quite short flight durations.

The three-processor, 40-second mission case was included for its relevance to the requirements on automatic landing systems, such as those included in current aircraft.

Figure IV-7 shows the dependence of reliability on the solid fault rate. It is assumed, for these results, that the transient fault rate is ten times the solid-fault rate. It is evident that the dependence of reliability on fault rate is very steep and that the assumed processor reliability of 2.0x10 E-4/hour is just barely adequate with five processors and five-way voting for critical tasks such as the global executive. However, substantial improvements in reliability can be expected from improved VLSI technology over the next few years. Such improvements in processor reliability will have an impressive effect on system reliability.

Figure IV-8 investigates the sensitivity of system reliability to the transient-fault rate. Very little solid information is available about the rate of transient faults under realistic conditions. The dependence of reliability on transient-fault rate is really a dependence on the ratio between the transient-fault rate and the error-masking rate. Thus a faster fault masking time would have the same effect as a corresponding reduction in transient fault rate.

Figure IV-9 displays the effect on reliability of erroneous diagnosis of fault symptoms by the global executive. To the right of the graph are two curves that show the effect of mistaking a solid fault as a transient, and thus of ignoring it. The fault, being solid, will, of course, continue to generate errors, and will thus be subjected to further attempts at diagnosis, which may correctly recognize it as a solid fault. The effect is thus one of lengthening the reconfiguration time. It can be seen that, with five-way voting of critical tasks, the system has ample error-masking capacity and is not very sensitive to reconfiguration time or to mistaken diagnoses that lengthen the reconfiguration time. In contrast, a three-way voted system, with less error masking, is quite sensitive to the effect of such mistakes.

To the left of Figure IV-9 are two curves that show the effect of mistakenly regarding a transient fault as solid, and thus of discarding a working processor prematurely. It can be seen that, with five-way voting of critical tasks, where exhaustion of spare processors is a primary failure mode, the effect of such mistakes is significant, and a very high level of discrimination between transient and solid faults is required. For a three-way voted system, the primary failure mode is the coincidence of transient faults rather than exhaustion of spares, and thus a moderate rate of mistakenly regarding a transient fault as solid is not significant.

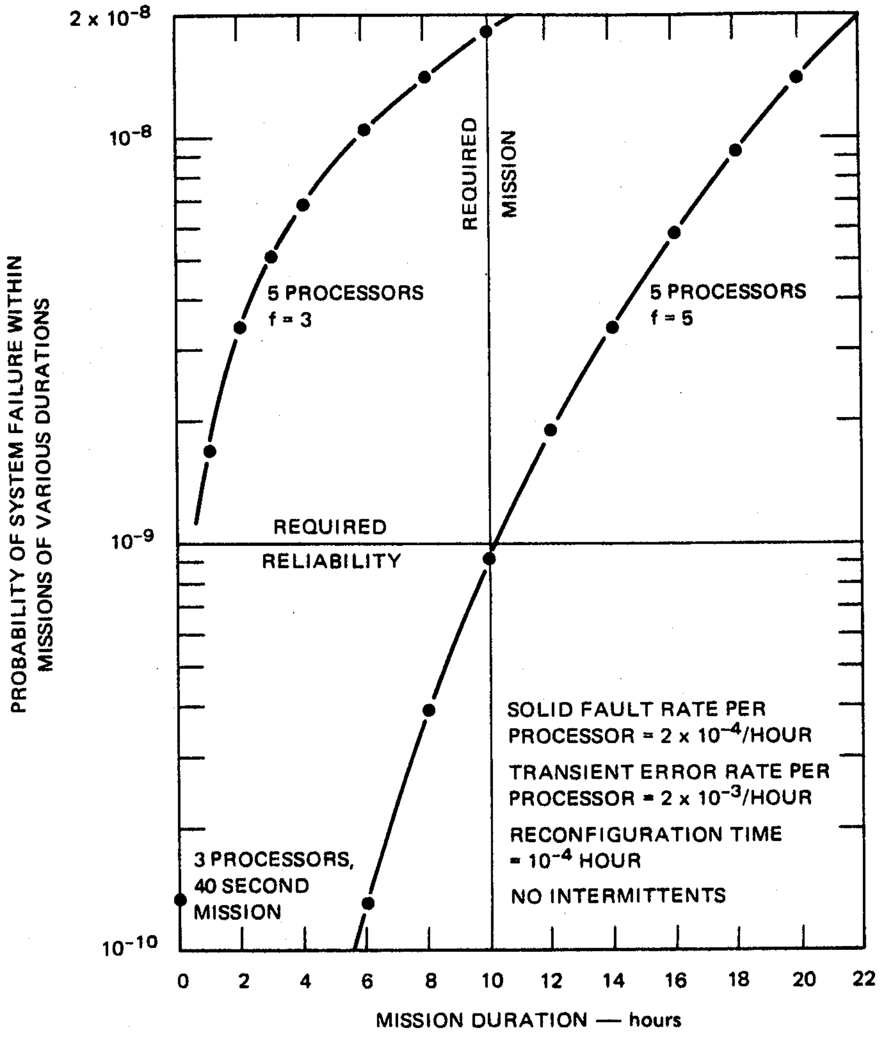


FIGURE IV-6 PROBABILITY OF SYSTEM FAILURE WITHIN MISSIONS OF VARIOUS DURATIONS

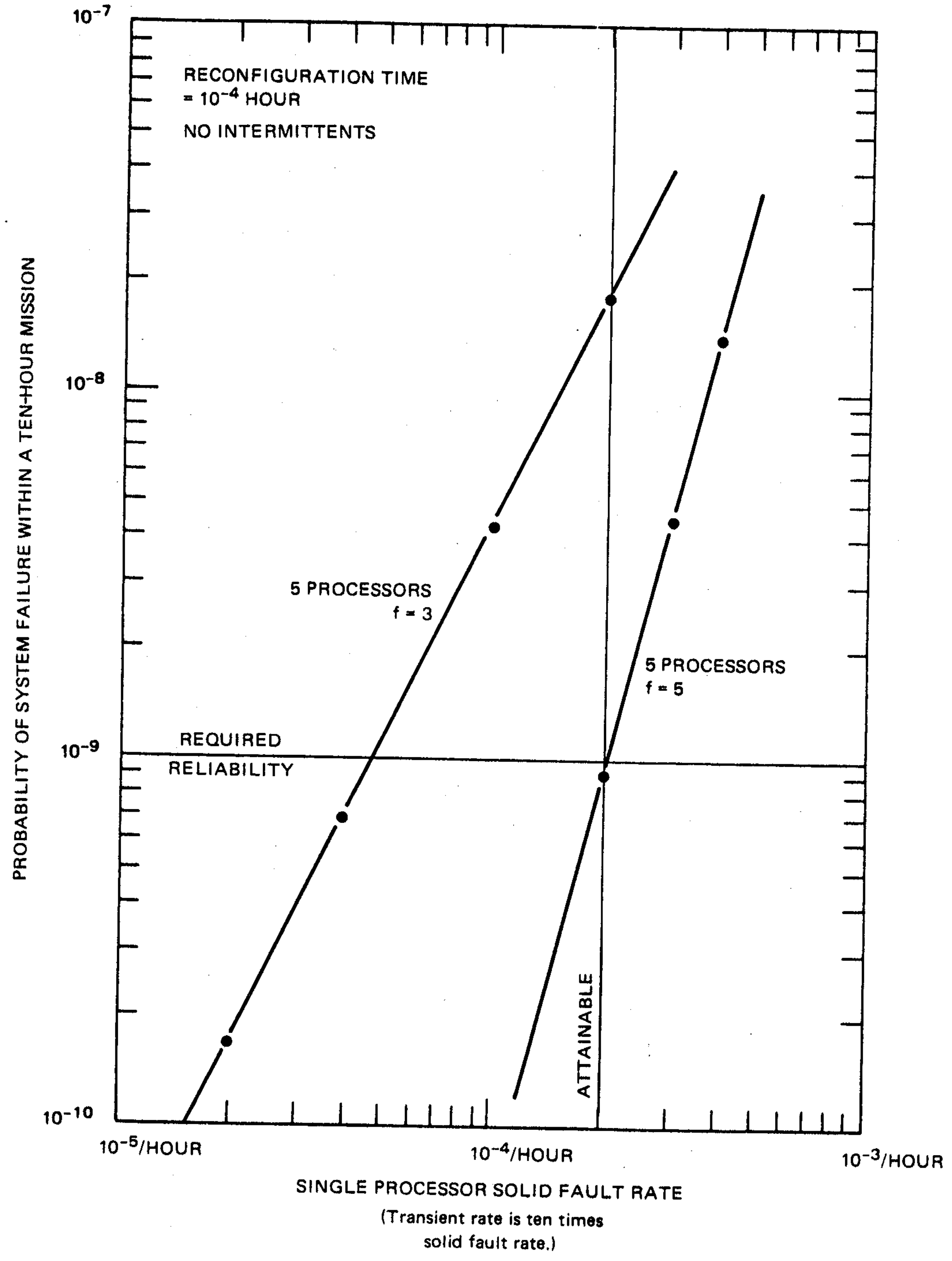


FIGURE IV-7 PROBABILITY OF SYSTEM FAILURE WITHIN A TEN-HOUR MISSION VS SOLID FAULT RATE

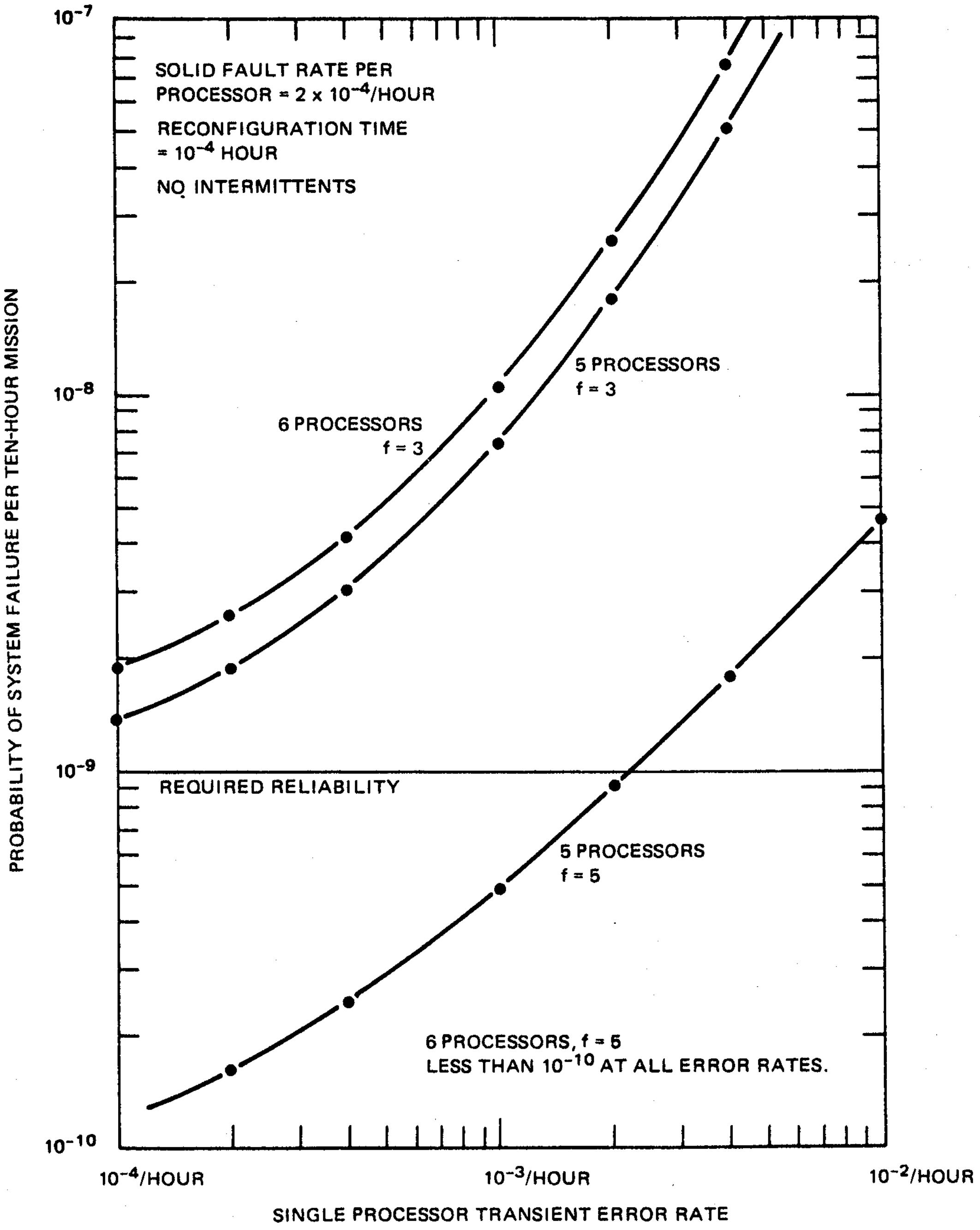


FIGURE IV-8 PROBABILITY OF SYSTEM FAULURE PER TEN-HOUR MISSION VS TRANSIENT ERROR RATE

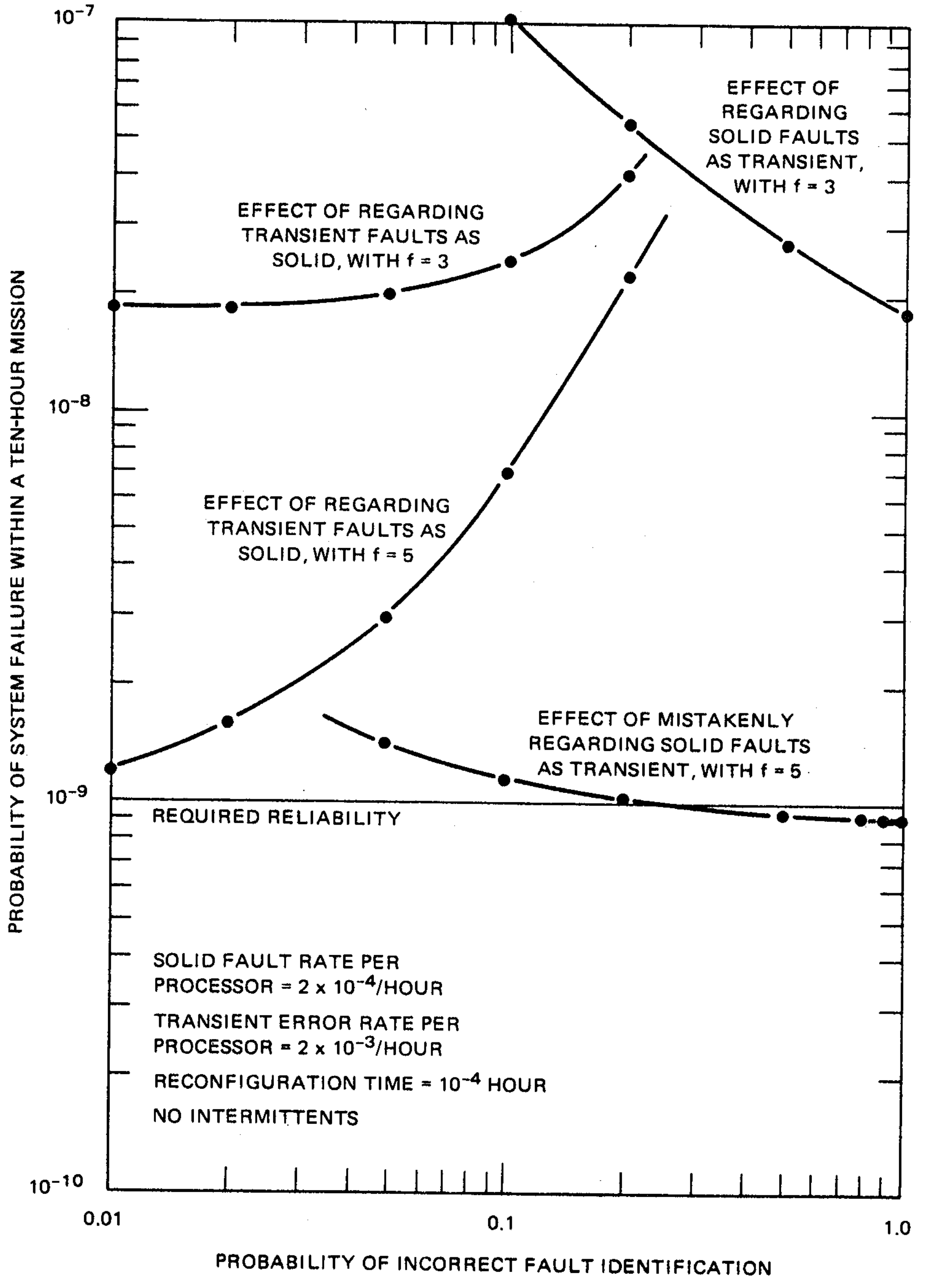


FIGURE IV-9 PROBABILITY OF SYSTEM FAILURE WITH A TEN-HOUR MISSION VS PROBABILITY OF INCORRECT FAULT IDENTIFICATION

These results have a significant effect on the design of the global executive. In a system using five-way voting for critical tasks, the global executive should be reluctant to diagnose a fault as solid while there is any chance that it is transient. Such an executive will wait until the fault has persisted for perhaps as much as 200ms, and many error reports have been received, before concluding that the fault must be solid and that the processor must be discarded. A system using only three-way voting must react much more quickly to faults because of its more limited error-masking capacity. Consequently, the design for the global executive must be willing to accept a rather higher rate of mistakenly regarding transient faults as solid.

Figure IV-10 analyzes the probability that failed processors in a flight-control system will prevent departure of a flight until the failed processors are replaced or repaired. These results are compared with results for FTMP published in Hopkins [12]. Dispatch failure probability is particularly important for longer international flights involving destinations at which full sets of spare parts may not be available.

It can be seen that, with five processors required for dispatch, a system containing only five processors would incur a significant risk of dispatch failure. But inclusion of a sixth processor reduces the risk to acceptable levels, and a seventh processor should provide ample margin for even the most protracted international routes. These results are generally comparable to those for FTMP when allowance is made for the higher number of processors required for dispatch using FTMP.

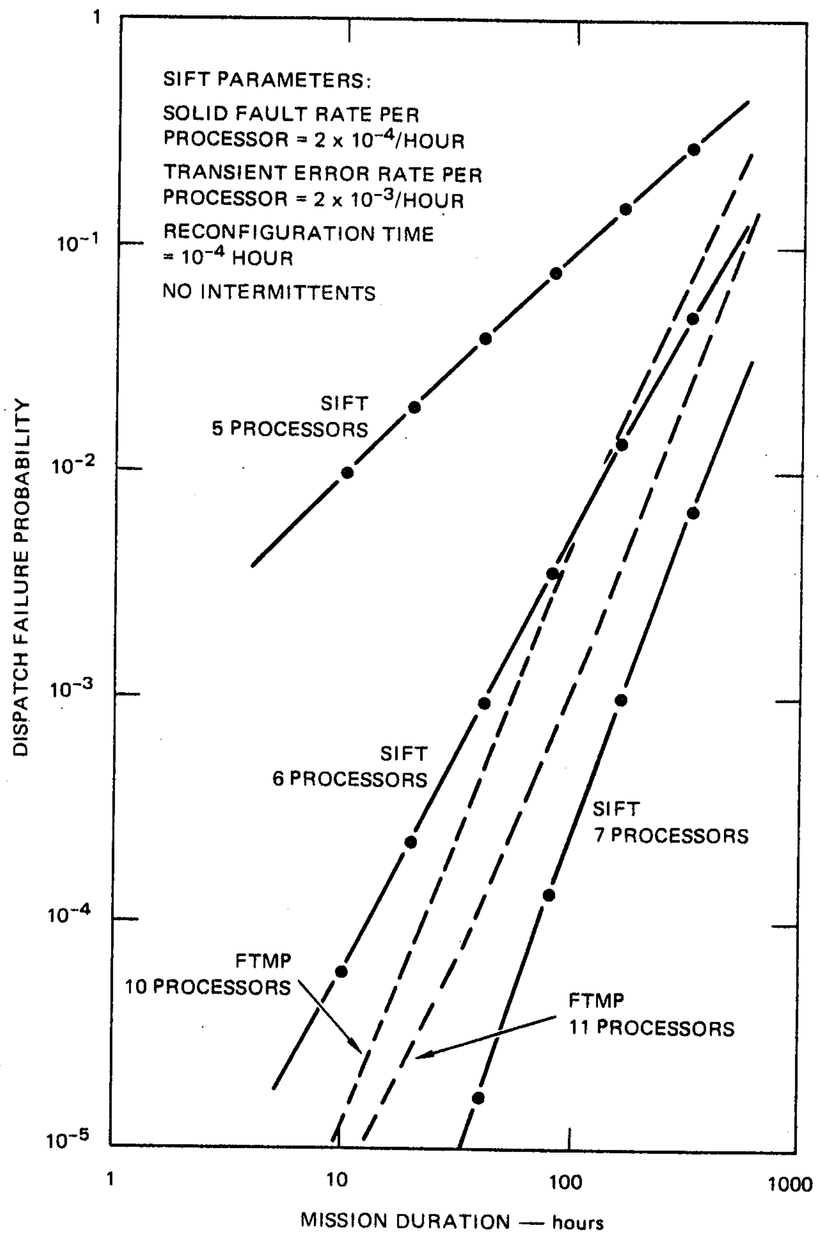


FIGURE IV-10 DISPATCH FAILURE PROBABILITY (5 processors required for SIFT; 8 processors required for FTMP; FTMP from IEEE report.)

B. The Hierarchical Specification and Mechanical Verification of the SIFT Design

This section describes the formal specification and proof methodology employed to demonstrate that the SIFT computer system meets its requirements. The hierarchy of design specifications is shown, from very abstract descriptions of system function down to the implementation. The most abstract design specifications are simple and easy to understand, almost all details of the realization having been abstracted out, and can be used to ensure that the system functions reliably and as intended. A succession of lower-level specifications refines these specifications into more detailed, and more complex, views of the system design, culminating in the Pascal implementation. The section describes the rigorous mechanical proof that the abstract specifications are satisfied by the actual implementation.

1. The Role of Formal Proof

The extreme reliability requirement on SIFT imposes a very severe problem in substantiating the achievement of that level of reliability. At the required reliability, a mere observation, even of a large number of systems, will be ineffective. Further, a SIFT system must be able to recover successfully from several million faults for every allowable system failure, and must, therefore, be able to recover from quite improbable and unforeseen faults and even combinations of faults. Thus validation by fault injection, while necessary, is unlikely to convince us that SIFT meets its reliability requirements.

The justification that SIFT meets the reliability requirement must be based on an extrapolation from fault rates that are easier to measure, such as those for an individual processor. For SIFT, this extrapolation takes the form of a discrete Markov analysis, with the numbers of working and faulty processors defining the states and the fault and reconfiguration rates defining the transitions. The validity of this extrapolation depends on a number of assumptions, and, at the desired level of reliability, even "minor" violations of the assumptions can have significant effects on the reliability achieved. Thus the assumptions must, themselves, be quite rigorously substantiated if the claimed reliability is to be believed. For instance, one important assumption of the Markov analysis is that the occurrence of faults is well described by a Poisson model with complete independence between processors. Much of the electronic and mechanical design of SIFT is intended to maintain this independence.

The validity of the Markov analysis depends also on the assumption that the states and the transitions of the Markov model correspond accurately to the actual system, and that the states in which system failure is possible are correctly identified. But this correspondence is far from obvious, for the actual system has very many states with many complex transitions between them, and the correspondence must be maintained even when one or more of the processors has suffered a fault. In SIFT, this correspondence is based on a predicate **system safe** indicating that the replication of each of the tasks is sufficient so that the voting can mask the effects of the faults present in the system. The validation of SIFT now consists of two parts. The first of these is a demonstration that, so long as **system safe** is true, the system performs the desired flight-control function, even though one or more processors may be faulty. This is a correctness property for the function performed by the system. The second is a demonstration that the Markov analysis computes an upper bound on the probability that **system safe** becomes false. This is a correctness property for the probabilistic reliability analysis of the system. Because even a very small defect in the demonstrations could allow failures at an unacceptable rate, these demonstrations must be performed with the rigor of mathematical proof. In this paper we consider only the first of these parts. An outline of the probabilistic reliability analysis is given in Wensley [32].

The necessity for formal mathematical proof to ensure that SIFT meets the desired functional and reliability requirements presents two major issues:

- How does one define the criteria sufficient to ensure the correct functioning of the system?
- How does one prove that the criteria are satisfied by the actual system?

The first issue is crucial if the formal verification effort is to have any practical significance. One must have confidence, even as a noncomputer scientist, that the formal specifications stating what is meant by the correct functioning of the system in fact reflect the *intended behavior.* That a formal specification expresses what the system designer intuitively means must, in the end, be determined by inspection. A formal specification must therefore be *believable* if rigorous mathematical correspondence to the specification is to ensure the desired effect. The larger and more complex the system, the more acute the problem becomes. Specifications reflecting the detailed behavior of the system allow the most straightforward formal verification effort, but it is difficult to ensure that low-level specifications embody what is meant by the proper functioning of the system. Very high-level specifications, abstracting from the details of the system, are necessary if we are to state the overall functional and fault-tolerance properties of the system in a way that can be understood and believed. The problem then becomes one of reconciling the very high-level specifications with the detailed transformations performed by the programs of the actual system.

In order to state high-level system specifications that can be shown to be consistent with the actual program, one must formulate not just a single specification of the system, but a *hierarchy* of specifications. Our approach is to state a tiered set of system specifications, as illustrated in Figure IV-11.

Each level L_i in the hierarchy specifies an abstract view of the system in terms of a set of primitive predicates P_i and functions F_i. The specification for the model is given by a set of axioms, characterizing those properties of the model appropriate for that level of system abstraction. At each level in the hierarchy, a specification L_i can be seen as an *abstraction* of the previous level L_{i+1}. Correspondence between successive levels is done by expressing each primitive function and predicate of higher-level L_i in terms of the functions and predicates of the lower-level L_{i+1}. With this mapping, one must then prove that each property derivable from the higher-level specification can be proved from the lower-level specification. The mapping between levels need not be complete; the mapping itself may be given as a set of axioms, saying only enough about the correspondence to derive the necessary axioms of the higher level as theorems from the axioms of the lower-level. It is required only that the mapping axioms be *consistent*, i.e., that there exist a complete functional mapping between levels that satisifies the mapping axioms. By demonstrating the correspondence between successive levels L_i and L_{i+1}, one can conclude by induction that any property provable from the highest-level specification is also provable from the lowest-level specification. Thus, any analysis of the system based on a higher level specification in the hierarchy is valid and could have been performed on the lowest-level system specification.

Within the hierarchy, the lowest-level specification of the system is the actual SIFT system executed by the hardware, while the highest-level specification reflects the intended overall function performed by the fault-tolerant system. The higher-level specifications represent, in effect, *system requirements*, stating properties to be possessed without defining method of attainment. As one moves down the hierarchy, each lower-level specification successively introduces additional mechanism in the design specification to achieve the fault-tolerance and expresses a more detailed and operational view of system transformation. Between successive specification levels, one can perform *incremental design verification*, proving that the more detailed design specification at the lower level supports the abstracted view at the higher level. By gradually introducing the algorithms used to achieve fault-tolerance, one can verify each aspect of the design at the highest level of abstraction containing the necessary concepts.

As an example, one can prove that replication and majority voting serve to mask faults, using a specification of the system as a single (and therefore synchronous) global object. Having proven this paradigm with respect to that specification level, one can then define a lower-level specification of the system as a distributed asynchronous system with a broadcast communication interface. It is then required to exhibit a mapping from the distributed system view to the global system view at the higher specification level. Demonstration that each axiom of the global-state specification is provable from the axioms defining the distributed-state specification will ensure that any theorems about fault masking in the global-system view are valid for the distributed-system view as well. Thus the paradigm of fault masking through task replication is introduced and validated prior to introducing techniques for fault isolation through distribution of resources.

2. Mechanized Specification and Verification

Attempting to formally characterize and justify the design of any real system is complex and tedious. Without mechanical aids for constructing formal specifications and rigorously enforcing sound proofs, this task would be completely impractical and would not produce a credible result. Our early experience in formulating formal "paper" specifications and giving informal mathematical arguments of correctness was fraught with specification ambiguity and oversights in the informal correctness proofs. In response to this, and our desire to mechanize the style of specification and verification employed in our previous "paper" attempts, a new mechanical verification system was designed and implemented.

STP [29] is an implemented system supporting specification and verification of theories. As implemented, STP did not contain a parser for SPECIAL, and thus, for this verification the specifications were expressed in the LISP-like internal representation of STP. The logic of STP is an extension of a multisorted (strongly-typed) first-order logic. The logic includes type parameterization and type hierarchies. STP support includes syntactic type checking and proof components as part of an interactive environment for developing and managing theories in the logic. At the core of the system is a fast, complete decision procedure [28] for a quantifier-free theory of (Presburger-like) arithmetic. The user of the system can introduce new types and function symbols, with the semantics specified through a set of first-order axioms. By providing aid to the theorem prover in the form of selection of appropriate instances of axioms and lemmas, the user raises the level of competence of the prover to the full first-order theory specified. A fundamental characteristic of the system is that the user need know no details of the theorem prover itself; the system forms a complete mechanization of a simply-characterized theory. As a result of a successful proof attempt using STP, one obtains the sequence of axioms and intermediate lemmas, together with their necessary instantiations, which lead to the theorem. The system automatically keeps track of which formulas have been proved and which have not, so that the user is not forced to prove lemmas in advance of use. The system also monitors the incremental introduction and modification of specifications to monitor soundness.

3. An Outline of the Specification Hierarchy

Figure IV-12 shows an outline of the various specifications and analyses that are used in the justification of the reliability of SIFT. Before the individual specifications are described in detail, we give a description of their intent and interaction. On the right of the figure is a hierarchy of specifications of the correct functional behavior of SIFT, while on the left is a set of analyses that yield the probability of that correct behavior. The models at the bottom of the figure describe the hardware of SIFT, upon which the more abstract analysis is based.

The IO Specification, the most abstract functional description of the system, asserts that, *in a safe configuration*, the result of a task computation will be the effect of applying its designated mathematical function to the results of its designated set of input tasks, and that this result will be obtained within a real-time constraint. Each task of the system is defined to have been performed correctly, with no specification of how this is achieved. The model has no concept of processor (thus no representation of replication of tasks or voting on results), and of course no representation of asynchrony among processors. The specification of this model contains only 8 axioms and is intended to be understandable by an informed aircraft flight control-engineer.

The Replication Specification elaborates upon the IO Specification by introducing the concept of processor, and can therefore describe the replication of tasks and their allocation to processors, voting on the results of these replicated tasks, and reconfiguring to accommodate faulty processors. The specification defines the results of a task instance on a *working* processor based on voted inputs, without defining any schedule of execution or processor communication. This model is expressed in terms of a global system state and system time.

The Activity Specification develops the design into a fully distributed system in which each processor has access only to local information. Each processor has a local clock and a broadcast communication interface and buffers. The asynchrony among processors and its effect upon communication is modeled. The specification explicitly defines each processor's independent information about the configuration and the appropriate schedule of activities. The schedule of activities defines the sequence of task executions and votes necessary to generate task results within the required computation window. The Activity Specification is the lowest level description of the complete multiprocessor SIFT *system.*

The PrePost Specification consists of specifications for the operating system for a single processor. The specification, in terms of pre-condition/post-condition pairs, facilitates the use of sequential proof techniques to prove properties of the Pascal-based operating system as a sequential program. These specifications are very close to the Pascal programs, and essentially require the programs to "do what they do".

The various programs that form the SIFT executive are written in Pascal and form the *Pascal Implementation*, from which is derived by compilation the *BDX930 Implementation.* This is the lowest level specification of the SIFT software.

The functional behavior described by the *I/O Model* is assured only so long as the predicate **system safe** remains true. The analyses shown on the left of Figure IV-12 provide the probability that **system safe** will remain true and hence that the desired functional behavior will continue.

In the remainder of the paper, we present details of the specifications comprising the SIFT design hierarchy. *Unless otherwise noted, all specifications and mappings are taken from actual system specifications and completed proofs.* For pedagogical purposes, we have used a syntactic transliteration of the actual form of the specifications. The STP system forced all user interaction to use a LISP-like prefix notation; we have transformed this into more common mathematical notation.

The mechanical proof of consistency between the various levels of specification and further details of its derivation are contained in [21].

4. Input/Output Specification

The Input/Output Specification of SIFT, the highest level specifying functional behavior, defines the input/output characteristics of tasks performed by SIFT. The specification defines the configuration of system tasks and expresses the flow of information between tasks. Based on an abstract notion of time, which may be interpreted as subframe time, we refer to iterations of a task taking place during various time intervals. The time interval for a particular iteration of a task is referred to as its *execution window*, having a begining time and an ending time. Each task is defined to use as inputs the values produced by its input tasks and produces one or more outputs during its execution window. Based on a high-level predicate specifying whether a task is *safe* during a particular iteration of a task, the specification defines that a task which is safe during an iteration will produce exactly one output value, computed as a function of its input values. Provided that the entire system is safe throughout some interval (i.e., that all tasks are safe for that interval), we can prove by induction that all tasks will compute correct functions of their intended inputs. This defines at a high level what it means for SIFT to function correctly.

Conspicuously absent from this model is any notion that a task is replicated and computed on a set of processors. At a lower level, we shall explain that the value the I/O specification defines as resulting from a given task iteration will actually be the outcome of a majority vote of processors assigned to compute the task. The *task safety* predicate taken as primitive in the I/O specification, specifying when a task can be relied upon to produce correct results, will be defined at a lower level to be a function of the amount of task replications and the number of working processors.

Briefly, the model is organized as follows. Each task a in *Tasks* the set of all executive and application tasks, computes a (mathematical) function, denoted by **function**(a), of its input values. The function **apply**($f, \overline{v}$) takes as parameters a functional value and an argument list and produces the result of applying the function to the argument list.[1] **Inputs**(a) denotes the set of tasks providing inputs to a. For task $b \in$ **Inputs**(a), the input to an iteration of a is provided by the most recently completed iteration of b prior to the execution window of that iteration of a. A derived function b **to** i **of** a denotes the iteration of b providing input to the i-th iteration of a. Because all tasks iterate once per

[1]This is in fact a trick to use a first-order encoding of functional value domains.

frame, one can prove (as indeed we do) that b **to** i **of** a is equal to i or $i-1$, that is, that the input task is either "executed" in the same frame as the task or in the previous frame. During each iteration i of a task a, **Result**(a,i) denotes the set of output values which are produced. In order to map task iterations to subframe time, the function i **of** a is used to denote the time interval $[t_1, t_2]$ comprising the execution window of the i-th iteration of a. The functions **beg**(i **of** a) and **end**(i **of** a) are used to denote the begining and end of the execution window, respectively.

The overall structure of task configurations within the I/O model is illustrated in Figure IV-13. For a task such that the predicate **task** a **safe during** i is true, a will produce exactly one output value during its execution window. The output(s) of a task which is not safe during its iteration is unspecified. Because the configuration of tasks is different for different phases of the flight, not all tasks necessarily compute each iteration. A predicate a **on during** i determines whether **Result**(a,i) is expected to compute a function of its inputs or to return a special $\perp$ element as its value.

Within the I/O specification, the interactive consistency algorithm is defined as a special form of task. For such a task a, satisfying the predicate **i/c**(a), its associated mathematical function **function**(a) is defined to be the identity function. Recall from our discussion in Section 4 the interactive consistency algorithm is used in order for multiple processors reading unreplicated (and possibly unstable) input to reach agreement on an input value. As we explain below, a safe interactive consistency task will *always* produce a single output value.

Based on these primitive functions and predicates, the I/O specification contains eight axioms, expressing constraints on when task iterations are to take place and that safe tasks compute functions of their designated inputs. We do not illustrate the entire set of axioms here. The axioms related to the scheduling of task iterations are straightforward. They express basic requirements that successive iterations of a task are properly ordered in time and that the execution window of a task b must precede the execution window of a task a to which it provides input.

The major axiom defining the Input/Output behavior of a task is the following:

$$\begin{array}{l}
a\ \textbf{on during}\ i \quad \wedge \\
\textbf{task}\ a\ \textbf{safe during}\ i \quad \wedge \\
\forall b \in \textbf{Inputs}(a) \\
\qquad |\textbf{Result}(b, b\ \textbf{to}\ i\ \textbf{of}\ a)| = 1 \\
\quad \supset \\
\textbf{Result}(a,i) = \\
\qquad \left\{ \textbf{apply}\left(\textbf{function}(a), \left\{ < v, t > \middle| \begin{array}{l} t \in \textbf{Inputs}(a) \quad \wedge \\ v \in \textbf{Result}(t, t\ \textbf{to}\ i\ \textbf{of}\ a) \end{array} \right\} \right) \right\}
\end{array}$$

This axiom defines that any iteration of a task a, such that (1) a is both **on** and **safe** and (2) each task b providing input to the i-th iteration of a returns exactly one output value during its corresponding iteration (the notation $|s|$ denotes the cardinality of set s), will return exactly one output during its iteration (i.e., that **Result**(a, i) will be a singleton set). The value produced will be that resulting from applying its designated function **function**(a) to the set of (tagged) values produced by its input tasks. The set of input values is specified as a set of pairs $< v, t >$, where, for each task t in the input set, v is the value in the (singleton) set **Result**$(t, t$ **to** i **of** $a)$. Thus, provided a is safe and its input is stable, it will correctly compute an output value. This is the main statement of functional correctness of the system that is demonstrated by the proof effort.

In the case of interactive consistency tasks, one additional axiom governs its input/-output characteristics:

$(\textbf{i/c}(a) \quad \wedge \quad \textbf{i/c task } a \textbf{ safe during } i) \quad \supset \quad |\textbf{Result}(a, i)| = 1$

This defines that an interactive consistency task which is safe during its iteration will always produce a single value as output. By the previous axiom, if its input task is **safe** and thus provides a single output, the interactive consistency task will perform its associated function (in this case the identity function) on the input. Even if the input task is not **safe**, however, the current axiom defines that *some* single output value will be produced. This is the main correctness criterion for the interactive consistency algorithm. We did not carry out a mechanical proof of this axiom – a hand proof can be found in Melliar-Smith and Schwartz [20].

These are the major axioms of the I/O specification. In the next section, we present the next lower-level specification and show how the primitives and stated axioms of the I/O specification are supported at the next level.

5. The Replication Specification

The Replication Specification, at the next lower level, introduces the notion that tasks are replicated and executed by some number of processors. Based on a high-level concept of each processor communicating its results to all other processors, a specification of the majority voting performed by each processor is given. Also defined (but not proven) is the information flow through which error reports from individual processors are provided to the global executive. This information is used by the global executive in order to diagnose processor faults and remove, from the configuration, processors deemed to have solid faults.

The concept of task scheduling has been refined to define not only the execution window for task execution but also the set of processors assigned to execute the task. The function **poll for** i **of** a denotes the set of processors assigned to compute the i-th iteration of task a. The I/O model primitive predicate a **on during** i is derived within the Replication model as:

a **on during** $i \quad \equiv \quad \exists p \in$ **poll for** i **of** a

With the concept of processor computation occuring in the Replication model, the **task safe** predicate appearing as primitive within the I/O model can be derived within the Replication model in terms of working processors. The Replication model includes a variable S, which denotes the set of "safe" processors at any given time. $S^{[t_1,t_2]}$ denotes the set of processors safe during the interval $[t_1, t_2]$. At the Activity model level, we will define a processor being "safe" as a rather complex function of having correctly functioning hardware, being in the correct configuration, and having a clock within some skew of other processor clocks. Of course this set will not have an implementation counterpart, since the implementation will never have perfect information concerning the set of correctly functioning processors.

A derived concept at this level is that of a task iteration's *data window.*
The **DWindow for** b **to** i **of** a is defined to be the time interval
[**beg** (b **to** i **of** a) **of** b, **end** (i **of** a)].
Based on this function, we define **DWindow for** i **of** a to be the interval extending from the begining of the execution window of the earliest input task to a and extending to the end of the execution of i **of** a.

Using these concepts of data window and the set of working processors, we can now derive the **task safe** predicate of the I/O model as follows:
task a **safe during** i

$\equiv$

$2 \times \left|\textbf{poll for } i \textbf{ of } a \bigcap \mathbf{S}^{\textbf{DWindow for } i \textbf{ of } a}\right| > |\textbf{poll for } i \textbf{ of } a|$

$\vee \quad \sim a$ **on during** i

The definition states that a task a is safe either if a majority of the processors assigned to compute the task are working for the data window of the task or if the task is not **on during** i. It is necessary that the processors are in the working set S for the entire data window of the task in order that we can be assured (in mapping to the next lower-level specification) that the processor will not corrupt its input data prior to its use. We omit discussion of the conditions necessary to define the safety of interactive consistency tasks.

With the concept that a processor computes an iteration of a task comes the function **Result**(a, i) **on** p which denotes the set of outputs produced by processor p for the i-th iteration of task a. In a manner left unspecified by this level, processor p communicates its results to all other system processors. The function **Result**(a, i) **on** p **in** q denotes the value that processor q has reportedly received from processor p for the i-th iteration of a. The relationship between **Result on** and **Result on in** is defined by the following axiom:

$q \in \textbf{poll for } j \textbf{ of } b \bigcap \mathbf{S}^{\textbf{DWindow for } j \textbf{ of } b}$

$\supset$

Result(b, j) **on** $q =$

$$\left\{ v \;\middle|\; \begin{array}{l} \exists p \;\; p \in \mathbf{S}^{j\ \mathbf{of}\ b} \quad \wedge \\ \qquad v = \mathbf{Result}(b,j)\ \mathbf{on}\ q\ \mathbf{in}\ p \end{array} \right\}$$

This defines that, for a processor q in the **poll** set that is **safe** for the **DWindow**, the **Result** set **on** q is equal to the set of values that processors **safe** for the execution window have reportedly received from q. More intuitively, this states that the output of a working processor in the poll is the set of values reportedly received by working processors.

The function **Result**(a,i) **in** q is used to define the result of processor q voting on the output of the i-th iteration of a based on the results communicated to it.

The overall structure of the Replication model is illustrated in Figure IV-14. The task structure shown is a refinement of the task configuration illustrated in Figure IV-13.

As we shall show shortly, the I/O primitive **Result**(a,i) for a safe task iteration will be derived as the value a majority of assigned processors obtained by their voting. All processors are required to report the results of each task computation to all processors, and all processors are required to vote on all received values. Rather than a task producing a set of output values as in the I/O model, in the Replication model, a task produces a set of *sequences* of values. This reflects the fact that conceptual values in the system actually consist of a sequence of "machine words". Processor voting is scheduled (as specified at the next level of specification) on a word by word basis. We define voting via the following axiom:

$$\begin{array}{l} p \in \mathbf{S}^{j\ \mathbf{of}\ b} \quad \wedge \\ 1 \leq y \leq \ \mathbf{result\ size}(b) \\ \quad \supset \\ (\mathbf{Result}(b,j)\ \mathbf{in}\ p)[y] = \\ \qquad \mathbf{majority}\left(\left\{ < v,q > \;\middle|\; \begin{array}{l} q \in \mathbf{poll\ for}\ j\ \mathbf{of}\ b \quad \wedge \\ v = (\mathbf{Result}(b,j)\ \mathbf{on}\ q\ \mathbf{in}\ p)[y] \end{array} \right\}\right) \end{array}$$

For a safe processor p, a vote on a defined value position y, the y-th element in **Result**(b,j) **in** p is defined to be equal to the **majority** of first components in the set of value-processor pairs $< v,q >$, where q is in the **poll** set and v is the y-th component of the **result on in** value in processor p. This represents an encoding of majority value in the bag of all values in p reportedly received from processors in the **poll** set for task b.

The main execution axiom of the Replication Specification is now given as follows:

$$\begin{array}{l} p \in \mathbf{poll\ for}\ i\ \mathbf{of}\ a \bigcap \mathbf{S}^{\mathbf{DWindow\ for}\ i\ \mathbf{of}\ a} \\ \quad \supset \\ \mathbf{Result}(a,i)\ \mathbf{on}\ p = \\ \qquad \left\{ \mathbf{apply}\left(\mathbf{function}(a), \left\{ < v,t > \;\middle|\; \begin{array}{l} t \in \mathbf{Inputs}(a) \quad \wedge \\ v \in \mathbf{Result}(t, t\ \mathbf{to}\ i\ \mathbf{of}\ a)\ \mathbf{in}\ p \end{array} \right\}\right)\right\} \end{array}$$

This axiom, quite similar to its counterpart in the I/O model, defines that a working processor p in the **poll** set for the i-th iteration of task a, will compute the proper function of its locally-voted input values. Note that, unlike its I/O axiom counterpart, this is purely a local specification of the actions of a single, working processor operating on locally-computed information – still with respect to a synchronous system.

We are now in a position to define the mapping up to the I/O concept of **Result**(a, i). This is given by the following axiom:

Result$(a, i) =$

$$\left\{ v \;\middle|\; \begin{array}{l} \exists p \;\; p \in \mathbf{S}^{i \text{ of } a} \quad \wedge \\ \quad v = \textbf{Result}(a, i) \textbf{ in } p \end{array} \right\}$$

This expresses the set **Result**(a, i) as consisting of the set of values that safe processors obtained as a result of voting.

We omit discussion of the other axioms of the Replication Specification. In order to show that the I/O Specification is a valid abstraction of the Replication Specification, we must prove that the I/O axioms follow as theorems from the Replication axioms and the mappings.

The proof of the main Execute axiom of the I/O Specification required that each safe processor voting be shown to obtain the same voted value, assuming from the antecedent of the I/O Execute axiom that the task is safe and that there is only one value of the **Result** of each input task. This implies that each safe processor applies the correct mathematical function to the same set of input values and thus every safe processor produces the same correct output value. But our I/O assumption of **task safe** asserts that a majority of the processors computing the task are safe; therefore, the majority of computed values must be the correct value.

The proof of the main I/O Execute axiom from the Replication axioms required approximately 22 proofs, with an average of 5 premises necessary per proof, and 106 instantiations of axioms and lemmas overall.

6. The Activity Specification

This level of specification defines a completely local view of the behavior of a single processor in the SIFT system. The *fully distributed* nature of the SIFT system is specified at this level: each processor has an independent concept of time, configuration, and schedule. Also at this level is a more explicit model of the activities and data structures carry out the transformations specified at the Replication level. Whereas the Replication level defines the executed and voted values for each execution window of a task, the Activity level defines a schedule of *execute* and *vote* activities to realize this within the execution window, as shown in Figure IV-16.

Within the Activity model is the first indication that the SIFT system is not synchronous; the subframes on the various processors start and finish at slightly different real times. Two functions, **start**(t, p) and **finish**(t, p) map subframe time on processor p to real times at which the subframe starts and finishes, as shown in Figure IV-15. "Real-time" is represented in the specification as a discrete domain, which can be thought of as "clock ticks," to allow induction. A short **overhead** interval occurs between the finish of one subframe and the start of the next. Because of clock skew and transport delay within SIFT, the processors will not be exactly synchronized, but, for the system to function correctly, it is necessary that the clocks remain within a specified tolerance, **max skew**, of each other. This is the responsibility of the clock synchronization task, a part of each processor's Local Executive, using an algorithm whose proof is given in Appendix B. The required synchronization is expressed by:

$$\begin{array}{l}\textbf{clock safe}(p,t) \quad \wedge \quad \textbf{clock safe}(q,t) \\ \quad \supset \\ \textbf{finish}(t,p) + \textbf{broadcast delay} \leq \textbf{start}(t+1,q) \quad \vee \\ \textbf{finish}(t,q) + \textbf{broadcast delay} \leq \textbf{start}(t+1,p)\end{array}$$

As we discussed earlier, SIFT is carefully designed so that the distributed system is *effectively synchronous*. Within the limits given above, asynchronism caused by processor clock skew has no external effect. In the case of the broadcasting of the results of a task, for example, our specifications define the value at the destination only after the latest time at which the broadcast could have been completed, given the maximum processor skew. It is necessary to prove that no access to these data is attempted before that time, in order to map this asynchronous system up to the higher-level, synchronous Replication and I/O models.

The state of each processor is specified using two state-selector functions, corresponding to two data structures of the SIFT operating system: a data file connected via a broadcast interface to all system processors, and an input file into which voted values are placed and from which a task retrieves its input values. In the Activity specification, the function **datafile in** p **for** a **on** q **at** rt denotes the value in the datafile in processor p at real-time rt for the result of task a on processor q. The function **input in** p **for** a **at** rt denotes the value in the input file in processor p at real-time rt for the voted result of task a.

As we mentioned earlier, each processor has an independent opinion of the configuration it is expected to use in scheduling activities. At the start of a subframetime t, processor q uses as the appropriate configuration **config**(t, q) the value in a configuration subfield of **input in** q **for** GE() **at start**(t, q), where GE() denotes the replicated Global Executive task. For configuration c, the function **sched**(c, t, q) denotes the sequence of activities scheduled for subframetime t on processor q. An activity is either $< execute, a >$, specifying the execution of task a or $< vote, a, y >$ specifying a vote on element y of the output of task a. Figure IV-6 illustrates the interaction between the data structures and scheduled activities.

The effect of an *execute* is specified by the following axiom:

$$
\begin{array}{l}
p, q \in \mathbf{W}^t \quad \wedge \\
< \mathit{execute}, a > \in \ \textbf{sched}(\ \textbf{config}(t,q), t, q) \\
\quad \supset \\
\textbf{datafile in } p \textbf{ for } a \textbf{ on } q \textbf{ at } (\ \textbf{finish}(t,q) + \ \textbf{broadcast delay}()) = \\
\quad \textbf{apply}\left(\textbf{function}(a), \left\{ < v, b > \middle| \begin{array}{l} b \in \textbf{Inputs}(a) \quad \wedge \\ v = \textbf{input in } q \textbf{ for } b \textbf{ at start}(t+1, q) \end{array} \right\}\right)
\end{array}
$$

The set W^t denotes the set of *correctly functioning* (working) processors during subframetime t. The antecedent of the axiom defines that processors p and q are working during subframe t and that an execute activity for a is among the activities scheduled for processor q, according to its perceived configuration. The consequent specifies that the datafile in each working processor p for a on q at the finish of that subframe plus the broadcast delay, *according to q's clock*, is equal to the correct function applied to the set of input values present in the input file at the *start of the next subframe.* Several explanations are in order. The hardware broadcast interface connecting processor q's datafile to all processor datafiles is asynchronous and can be initiated at any time during the subframe, with respect to q's clock. In the event of an execute and a broadcast by processor q sometime during subframe t, the earliest moment at which the entry for a on q can be guaranteed is the finish of the subframe plus the maximum broadcast delay. Thus the value is only defined at this moment in time, and with respect to the broadcasting processor's clock. It was necesary to demonstrate that, with respect to receiving processor p's clock, the information is present by **start**$(t+1, p)$. Given the set of specified schedule constraints, it was shown that the information is present in all loosely synchronized processors prior to the first moment at which access can occur.

One might notice that an execute activity scheduled during subframe time t causes the datafile at the start of time $t+1$ to contain the result of applying the appropriate function to the arguments present at the start of time $t+1$. This rather noncomputational definition is due to the possibility of one subframe containing a vote on an input value and subsequent use in an execute. The effect of this sequence can be characterized by stating that the execution uses as inputs the values defined after the end of the subframe. In mapping this to the computation performed by the implementation, it was necessary to prove that schedule constraints allow this to be achieved by sequentially performing the activity sequence scheduled for the subframe.

The axiom defining a vote activity scheduled for the subframe is the following:

$$
\begin{array}{l}
p \in \mathbf{W}^t \quad \wedge \\
< \mathit{vote}, a, y > \in \ \textbf{sched}(\ \textbf{config}(t,p), t, p) \\
\quad \supset \\
(\textbf{input in } p \textbf{ for } a \textbf{ at start}(t+1, p))[y] = \\
\quad \textbf{majority}\left(\left\{ < d, q > \middle| \begin{array}{l} q \in \textbf{poll by } p \textbf{ for } a \textbf{ at } t \quad \wedge \\ d = (\textbf{datafile in } p \textbf{ for } a \textbf{ on } q \textbf{ at start}(t,p))[y] \end{array} \right\}\right)
\end{array}
$$

Given a working processor p scheduled to perform a vote on the y-th component of a during subframe t, the input file in p at the start of the following subframe is defined to be the majority of datafile values present in the datafile at the start of subframe t. The function **poll by**p **for** a **at** t denotes the set of processors determined by p at the time of the vote to have executed the last iteration of task a. This is defined as a rather complex function of p's view of the system configuration at the start of the subframe and of the schedule table. We do not give the definition here.

These axioms constitute the primary axioms defining the Activity specification. There are in all approximately 40 axioms defining the introduced functions and predicates of the model and constraining the composition of the schedule table.

In terms of the functions of the Activity model, we can now define the mappings to the function symbols of the Replication model. The function **Result on in** of the Replication level is derived with the following axiom:

$$v = \mathbf{Result}(a, i)\ \mathbf{on}\ p\ \mathbf{in}\ q$$
$$\equiv$$
$$\forall y, t\ \mathbf{beg}(i\ \mathbf{of}\ a) \leq t < \mathbf{end}(i\ \mathbf{of}\ a) \quad \wedge$$
$$1 \leq y \leq \mathbf{result\ size}(a) \quad \wedge$$
$$< vote, k, y > \in \mathbf{sched}(\ \mathbf{config}(t, q), t, q)$$
$$\supset$$
$$v[y] = (\mathbf{datafile\ in}\ q\ \mathbf{for}\ a\ \mathbf{on}\ p\ \mathbf{at\ start}(t, s))[y]$$

Briefly, the mapping axiom defines each component y of the **Result on in** value to be the value present in the datafile at the time during the execution window when a vote activity is scheduled for element y. Thus, the concept of value reportedly received by processor q from processor p is defined as the value used at the time of a scheduled vote on q.

In an analogous manner, the mapping up to the Replication **Result in** voted value is defined by the following axiom:

$$\mathbf{start\ frame}(\ \mathbf{frame}(t)) = i \times \mathbf{frame\ size}() \quad \wedge$$
$$1 \leq y \leq \mathbf{result\ size}(a) \quad \wedge$$
$$< vote, k, y > \in \mathbf{sched}(\ \mathbf{config}(t, p), t, p)$$
$$\supset$$
$$(\mathbf{Result}(a, i)\ \mathbf{in}\ p)[y] = (\mathbf{input\ in}\ p\ \mathbf{for}\ a\ \mathbf{at\ start}(t + 1, p))[y]$$

Briefly stated once again, each y-th component of **Result in** for processor p is defined to be the value in the input file in p at the start of a subframe following a vote scheduled on element y during a subframe corresponding to the i-th iteration of task a. Intuitively, the voted value is the value in the input file following a scheduled vote. Schedule constraints allow only one vote to be scheduled on a given element during an execution window.

The **poll for of** concept of a global poll set found in the Replication level is mapped up from the Activity level with the following axiom.

$$\textbf{pollfor } i \textbf{ of } a = \left\{ \begin{array}{l} q \mid \exists p \exists t \exists y \\ \quad \textbf{start frame}(\ \textbf{frame}(t)) = i \times \textbf{framesize}() \quad \wedge \\ \quad 1 \le y \le \textbf{result size}(a) \quad \wedge \\ \quad < vote, a, y > \in \textbf{sched}(\ \textbf{config}(t,p), t, p) \quad \wedge \\ \quad p \in \mathbf{S}^{i \textbf{ of } a} \quad \wedge \\ \quad q \in \textbf{poll by } p \textbf{ for } a \textbf{ at } t \end{array} \right\}$$

The global concept of **poll for** *i* **of** *a* is derived as the set of all processors included in **poll by** *p* **for** *a* **at** *t* at the time of a scheduled vote (of any element) on a processor *p* safe for the execution window.

Finally, the last mapping to be illustrated is the derivation of the set of safe processors, as used in the Replication model. This is defined by the following mapping axiom:

$$\mathbf{S}^t = \left\{ \begin{array}{l} p \mid p \in W^t \quad \wedge \\ \quad \textbf{clock safe}(p,t) \quad \wedge \\ \quad \textbf{task } GE() \textbf{ safe during last}(t, GE()) \quad \wedge \\ \quad \textbf{Result}(GE(), \textbf{last}(t, GE())) = \\ \qquad \textbf{input in } p \textbf{ for } GE() \textbf{ at start}(t,p) \end{array} \right\}$$

The above definition represents a precise statement of a processor that is correctly functioning, has a view of the last Global Executive output reflecting the consensus, and whose clock is close enough to other safe processors to properly communicate. The interaction between processor safety and the output of the Global Executive is worthy of further explanation. The definition does not require the processor to have been safe during previous subframes; this allows transient faults to have affected the processor in the past. The only requirements expressed are (1) that the Global Executive task have had sufficient replication to remain safe (effectively since system start-up), (2) that the configuration (contained within the output of the Global Executive) for the current subframe be unaffected, and (3) that clock safety be recovered despite any transients affecting the clock in the past.

The proof of the relationship between the Replication Specification and the Broadcast Specification was quite challenging. The proof involved showing that the distributed system has, as a valid abstraction, the synchronous, global characterization expressed in the Replication Specification. This required that the axioms and schedule constraints imply consistency of configuration and schedule within a single processor and between processors during an execution window. It was necessary to show that vote and execute activities, replicated on different processors and running during different subframes within the frame, use the same information for input. Furthermore, the proof required that the various processors, operating independently and asynchronously with only local information, communicate with each other without mutual interference; that the task schedules guarantee that results are always available in other processors when required, and are never accessed during broadcast. The derivation of the Replication axioms involved 56 proofs, with an average of 7 premises each, and 410 instantiations of axioms and lemmas overall.

7. PrePost and Imperative Levels

The PrePost specification intended to form a bridge between the Activity Specification and the Pascal programs of the operating system. It is expressed in terms of preconditions and postconditions for operating system operations and the specifications are very close to the Pascal programs, essntially requiring the programs to "do what they do".

The Activity level represents a specification for each processor in the distributed, multiprocessor system. In contrast, the PrePost level, very similar in abstraction to the Activity level, defines the behavior of a single, independent processor. The model employs the data structure abstractions present in the actual Pascal operating system implementation and is intended to facilitate a connnection between the multiprocessor system specification and the proof of the Pascal operating system executing on a single processor.

At the program level of abstraction, even conceptually simple properties require very complex specification and tedious verification. Because of the difficulty inherent in mapping between design specifications and an imperative implementation model, we deliberately limited the conceptual jump between the two levels. Having proved all considered aspects of the design correct at higher levels in the hierarchy, the only conceptual jump between the lowest level design specification and the implementation was the change in specification medium; the PrePost specification expresses that the "code does what it does." A traditional verification condition generation paradigm [13] was employed to prove precondition-/postcondition procedure characterizations from the Pascal procedures, each treated as a sequential program. We explain only enough of the model and its specification for the reader to glean an overall understanding of the nature of the specification.

Within the PrePost model, the state of a processor is specified as a pair $< p,t >$, where p is a processor id and t is a subframe time. Accessor functions **proc** and **time** map states into component processor and time components (respectively). For a state pair $< p,t >$, the function $\mathbf{next}(< p,t >) =< p,t+1 >$. Within the PrePost model, each data structure of the Pascal program is declared as an explicit function of the state. At the program-level, the datafile is implemented as a two-dimensional array of type **`array [proc,task] of array[Integer] of Integer`**, mapping a processor id and task name into the array of Integer values currently in the datafile. The input file is a program structure declared of type **`array[task,Integer] of Integer`**, task name and element number into an Integer value. Similarly, the schedule table is implemented as an array of type **`array[proc, config, subframe, activity-index] of activity`**, defining for each processor, configuration, subframe, and activity index, which activity is to be performed. The schedule table is a constant data structure present in each processor and thus not a function of the state.

The following PrePost axiom defines the semantics of the Execute activity:
$\mathbf{proc}(siftstate) \in \mathbf{W}^{\mathbf{time}(siftstate)} \quad \wedge$

$$
\begin{array}{l}
\exists j\ 1 \leq j \leq \textbf{ max activities}() \quad \wedge \\
\quad < \textit{execute}, a > = \textbf{sched table}()[\ \textbf{real to virt}(\textit{siftstate})[\ \textbf{proc}(\textit{siftstate})], \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{pconfig}(\textit{siftstate}), \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{subframe}(\textit{siftstate}), \\
\qquad\qquad\qquad\qquad\qquad\qquad j\,] \qquad \wedge \\
\forall b \forall j \forall y\ 1 \leq y \leq \textbf{result size}()[b] \quad \wedge \\
\qquad \textbf{p.inputs}[a, j] = b \neq \textbf{null task}() \\
\qquad\quad \supset \quad \textbf{inp}[j, y] = \textbf{input}(\ \textbf{next}(\textit{siftstate}))[b, y] \\
\supset \\
\textbf{datafile}(\ \textbf{next}(\textit{siftstate}))[\ \textbf{proc}(\textit{siftstate}), a] = \textbf{task results}(a, \textbf{inp})
\end{array}
$$

The antecedent of the axiom defines the case where the processor component of the state is correctly functioning for the current subframe, some activity of the schedule for the current configuration and subframe specifies an Execute for task a, and the auxiliary array variable **inp** contains the value in the input data structure in the state **next**(*siftstate*), for each input task b indicated by the array **p.inputs**. The array **real to virt**, shown here as an explicit function of the state, maps a real processor id into a logical processor id, in terms of which the schedule table is defined. Assuming the antecedent holds, the axiom then defines the datafile in *the executing processor* in state **next**(*siftstate*) to contain the results of applying the appropriate mathematical function to the input array **inp**. As we discussed in the previous section, it is required to prove during code verification that sequential execution of the schedule activities will satisfy this noncomputational specification of effect. A mapping axiom defines that the value corresponding to the processor's own entry in the datafile of a safe processor will be in all other datafiles by the start of the next subframe.

In order to apply sequential verification techniques to the Pascal program executing on the processor, it is necessary to make the state $< p, t >$ of the processor and the dependence upon a correctly functioning processor implicit. The sequential proof, in effect, considers execution on a properly functioning Pascal machine satisfying the axiomatic specification of Pascal. Furthermore, the **next**(*siftstate*) transition is taken to be one iteration of the Pascal **dispatcher** procedure, called once per subframe by a clock interrupt to execute the scheduled activity sequence. This "metatheoretic" jump is the only departure from our formal notion of hierarchy and is made as a concession to allow traditional code verification tools to form the last link in the proof. The validity of this jump is dependent upon a proof that the dispatcher in fact is allowed to execute as a sequential program, with no clock interrupts before completion and with no interference between internal and external data structure access. The former assumption was demonstrated by a timing analysis of the actual Bendix 930 code and the latter by the non-interference proof at the Activity Specification level.

The following precondition/postcondition characterization of the dispatcher is produced and verified for the actual dispatcher procedure:

$$\begin{array}{l}
\exists j\ 1 \leq j \leq \textbf{max activities}() \quad \wedge \\
\quad < execute, a > = \textbf{sched table}()[\ \textbf{real to virt}(myproc), \\
\qquad\qquad\qquad\qquad\qquad\qquad pconfig, \\
\qquad\qquad\qquad\qquad\qquad\qquad subframe, \\
\qquad\qquad\qquad\qquad\qquad\qquad j\ ] \qquad \wedge \\
\forall b \forall j \forall y\ 1 \leq y \leq \textbf{result size}()[b] \quad \wedge \\
\qquad \textbf{pinputs}[a, j] = b \neq \textbf{null task}() \\
\qquad\quad \supset \quad \textbf{inp}[j, y] = \textbf{input}[b, y]
\end{array}$$

$$\left\{\textbf{dispatcher}\right\}$$

$$\textbf{datafile}[myproc, a] = \textbf{task results}(a, \textbf{inp})$$

Hoare sentences like the above, asserting properties of a sequential dispatch procedure and its effect on the Pascal data structures, were proven consistent with the actual implementation.

The code proof required demonstration of approximately 40 verification conditions and was carried out by Dwight Hare and Karl Levitt using a version of the SPECIAL code verification system. The design proof between the Activity and PrePost specifications required 17 proofs, with an average of 9 premises each, and 148 instantiations overall.

8. Conclusions and Further Work

Our proof has demonstrated that the Pascal implementation of the SIFT distributed system satisfies the execution axioms of the I/O Specification. That the axioms of the I/O Specification characterize "correct" system operation remains a subjective judgement. The soundness of the axiomatic specifications is demonstrated by the existence of an imperative model at the lowest level of the hierarchy, relative to interpretations for all unimplemented function and predicate symbols (such as **W**, the set of working processors). Also assumed is the correct implementation of the Pascal machine, realized by the Pascal compiler and the Bendix BDX930 hardware.

The proof of the fault tolerant clock syncronization algrithm was performed independently, without mechanical support, and is given in Appendix B. The mechanical proof given here, the proof of correspondence to the I/O Specification, encompasses scheduling, rating, and interprocessor communication. Yet to be performed is the proof of orrespondence to the probabilistic reliability analysis, encompassing error diagnosis and reconfiguration. We expect to perform this remaining proof over the next year.

The process of formal specification and verification of SIFT resulted in the discovery of four design errors – errors that would have been difficult or impossible to detect by testing. Early specification efforts uncovered the insufficiency of three clocks for fault-tolerant clock synchronization (see Appendix B). The formal proof revealed that tasks not scheduled to execute did not regenerate their default result value every iteration, thus exposing that result to the accumulation of errors from transient faults.

A conclusion of our work is the importance of *design verification* prior to implementation verification. The highest-level design specifications for the SIFT system could not have been expressed in terms of specifications of individual Pascal programs.

The STP system used for specification and mechanical proof was developed concurrently with the proof effort, with its approach heavily influenced by our ongoing experience in attempting the proofs. The success of the man-machine symbiosis depended upon the user being able to express naturally his understanding of the proof in guiding the proof. Under other sponsorship, SRI is currently developing a new specification language for HDM, including parameterized theories, specification of state-modifying operations, and Hoare sentences, and are constructing an enhanced STP verification system.

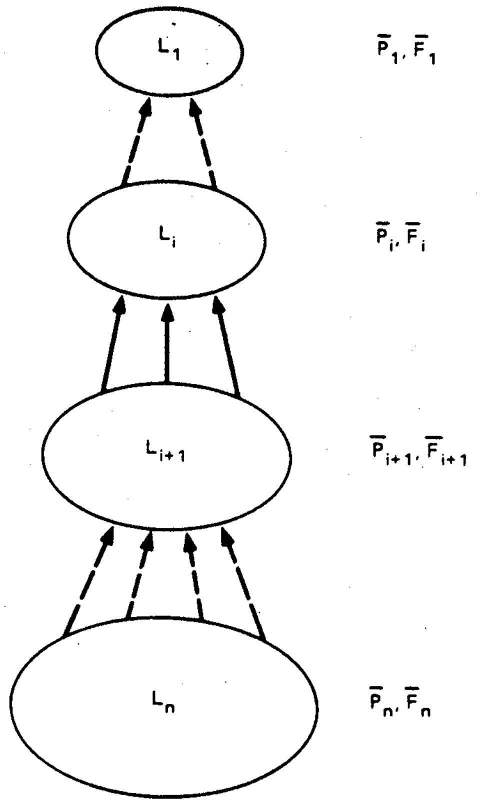

L_1
$\overline{P}_1, \overline{F}_1$
L_i
$\overline{P}_i, \overline{F}_i$
L_{i+1}
$\overline{P}_{i+1}, \overline{F}_{i+1}$
L_n
$\overline{P}_n, \overline{F}_n$

FIGURE IV-11 A HIERARCHY OF SPECIFICATIONS

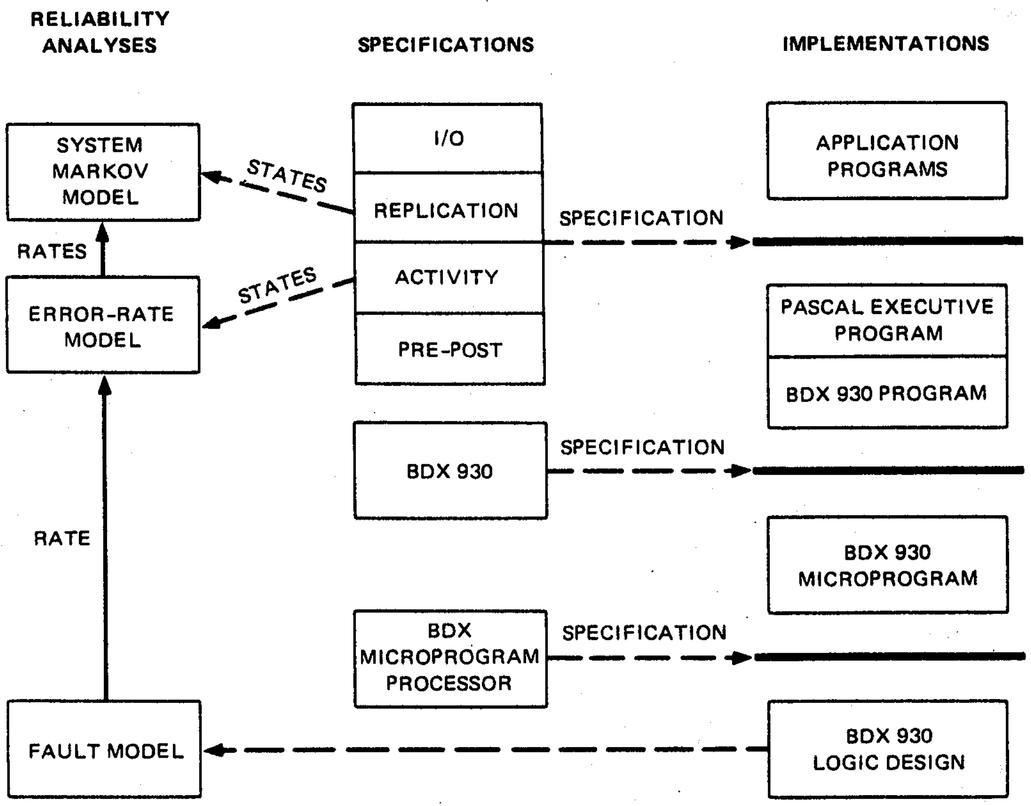


FIGURE IV-12 THE HIERARCHY OF SPECIFICATIONS AND ANALYSES USED TO SUBSTANTIATE THE RELIABILITY OF SIFT

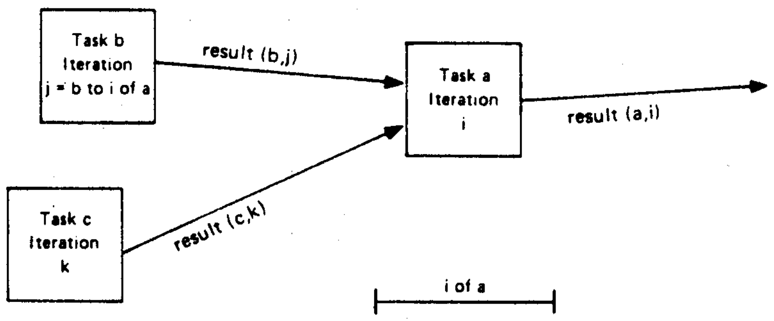


FIGURE IV-13 THREE TASKS IN THE I/O SPECIFICATION

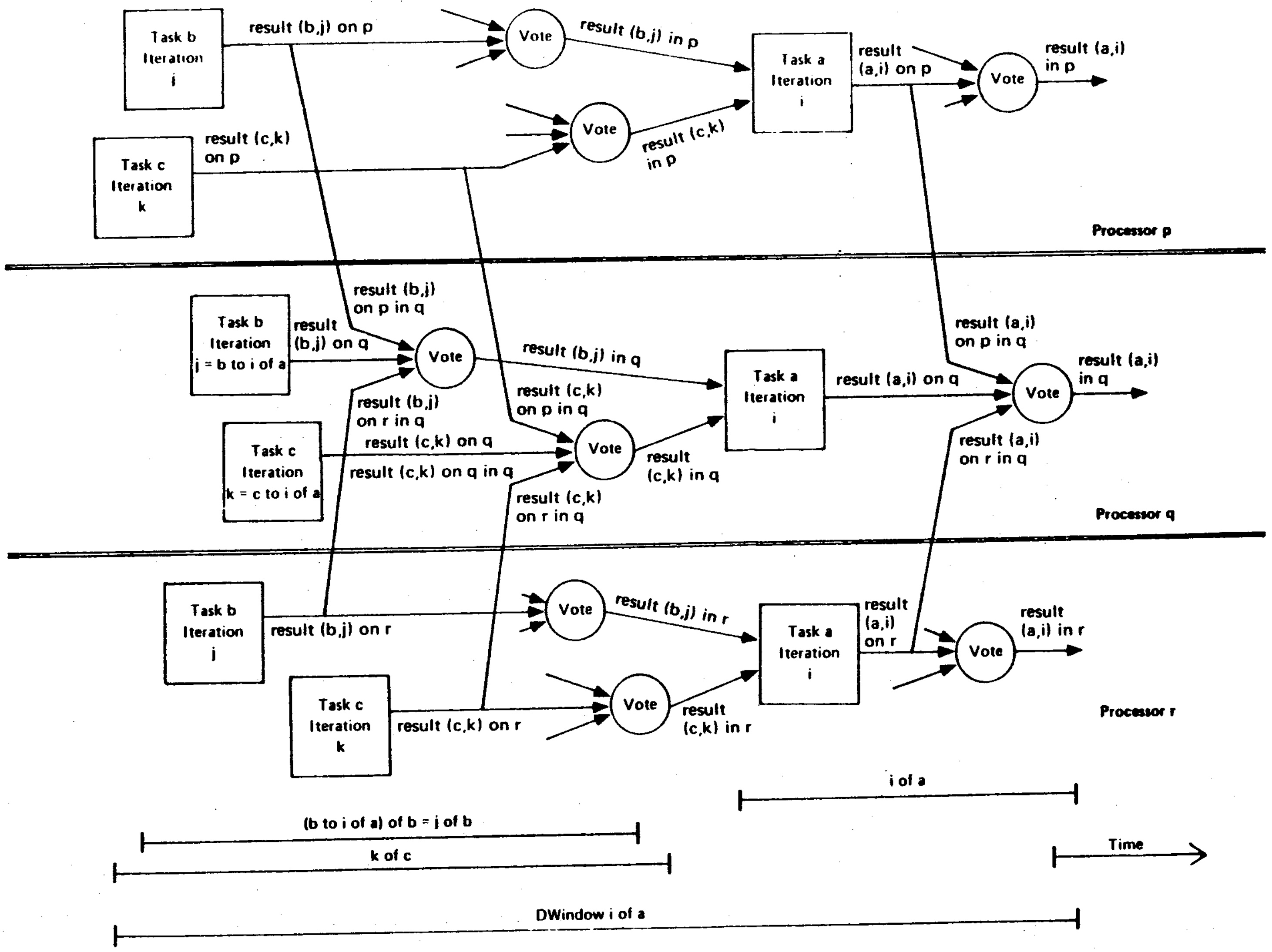


FIGURE IV-14 THREE TASKS IN THE REPLICATION SPECIFICATION

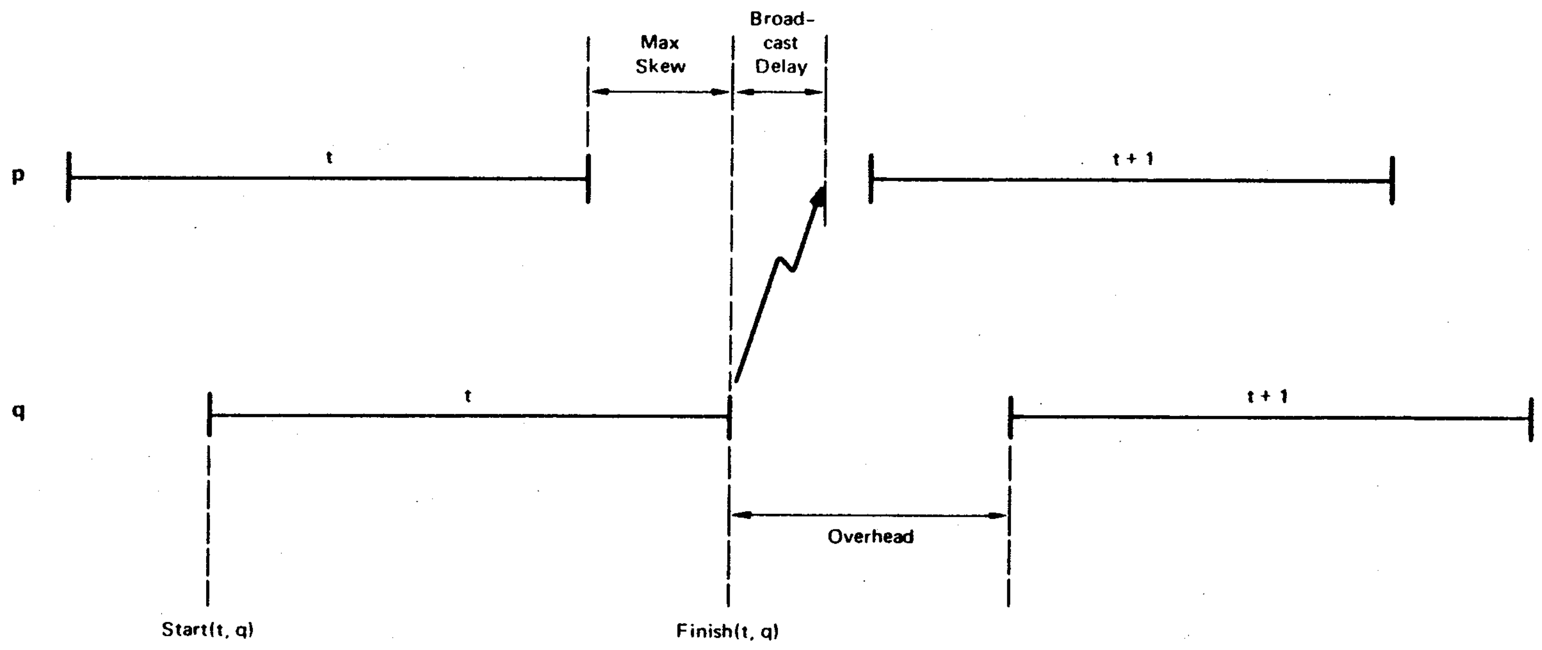


FIGURE IV-15 THE TIMING RELATIONSHIPS BETWEEN SUBFRAMES ON ASYNCHRONOUS PROCESSORS

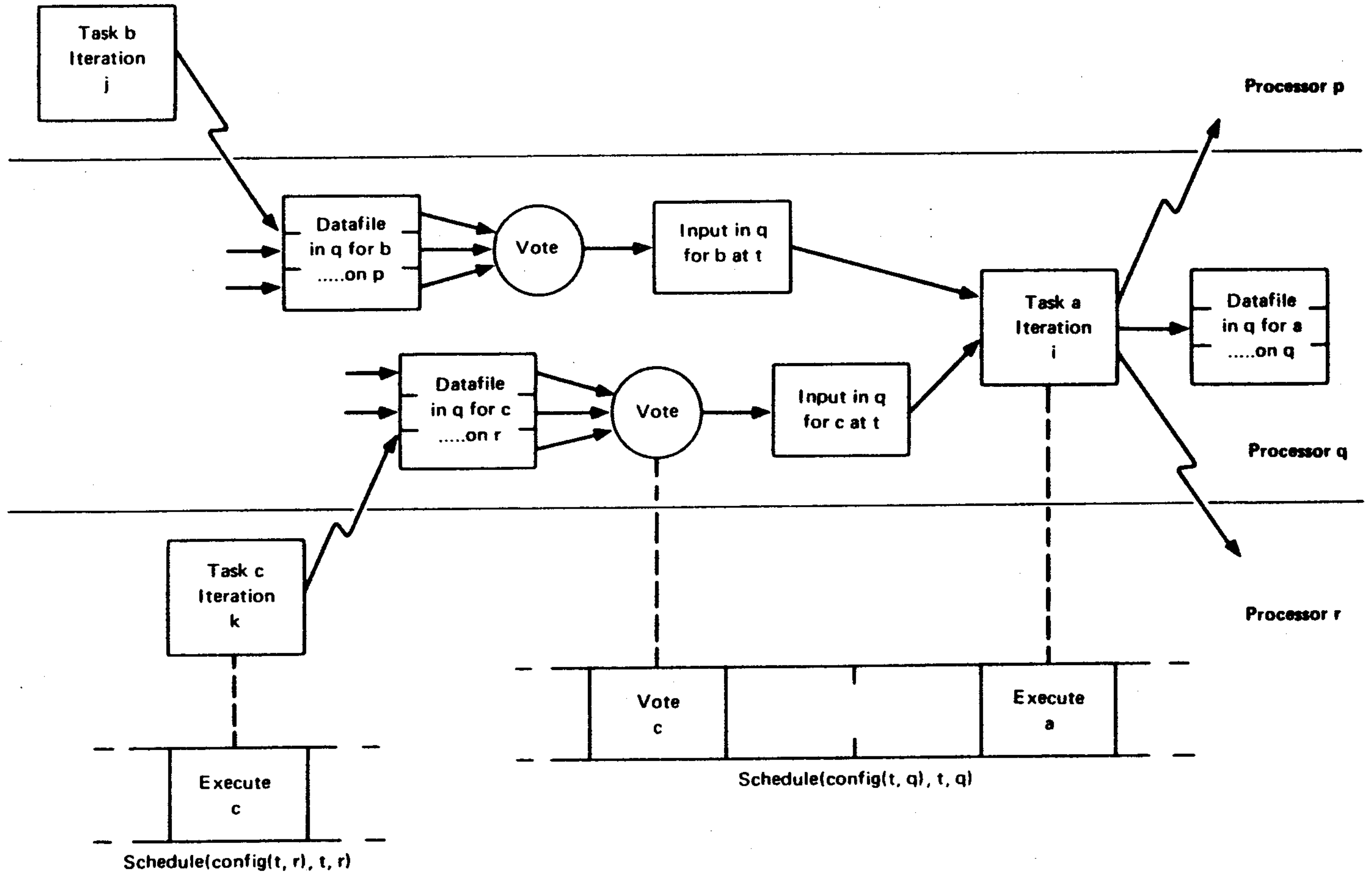


FIGURE IV-16 A PARTIAL VIEW OF THREE TASKS IN THE ACTIVITY SPECIFICATION

Appendix A. SIFT Test Plan

1 Test Philosophy and Goals

This section describes a strategy and plan for testing the performance and fault-tolerance characteristics of SIFT. In the validation and verification plan developed for SIFT, testing is not intended to verify correctness of the SIFT design but rather to measure quantitative characteristics, such as recovery time and performance under fault conditions, and to verify that a given version of the machine was correctly constructed.

The approach described was developed concurrently with the SIFT architecture, and guided the design of the SIFT test facility. The test plan was not actually implemented, due to the greater than anticipated effort required for system integration and debugging. Despite the lack of experience with its application, we believe that it offers a general and effective technique, not only for SIFT but for a broad class of fault-tolerant, general-purpose computer systems.

Test Philosophy

The purpose of the SIFT test is to help provide a very high level of confidence that the actual SIFT computer is performing according to its specifications with respect to both fault-free performance and survivability in the presence of faults. In the broad sense, this confidence is supported by the design approach adopted for the computer, by the care taken in its design and construction, and by the proofs that the programs indeed solve the problems for which they were written. The test of an actual SIFT computer must enhance this confidence by demonstrating, both to the casual observer and to the most critical test operator, that SIFT actually functions as intended in a representative operating environment of input data, programs, and failures.

The anticipated test will not be completely exhaustive; indeed, exhaustive testing over all input conditions, computations, modes, and possible faults is not possible within a test of limited duration. Nevertheless, successful operation of SIFT over a period of several weeks, including simulation of a variety of both flight conditions and faults, can provide a very convincing demonstration that all reasonable sources of degrading performance have been anticipated and dealt with in the design and development of the machine and its software.

The particularly high reliability requirements and the reconfiguration concept of SIFT are more demanding of performance than the computational requirements themselves. In addition, fault simulation and thorough monitoring of the internal status of SIFT will certainly be easier to do under ground laboratory conditions than in flight. Consequently, those aspects of SIFT concerned with reliability and accommodation to faults will necessarily receive the greatest emphasis in testing.

In principle, the test of any system may be conducted on any of several levels. For example, one might verify that all gates and intergate connections in a system are behaving in accordance with the design, by applying system inputs that cause each individual gate to be flexed through all of its distinguishable input combinations, and that create sensitized paths through the rest of the system so that any erroneous gate signals will manifest themselves at system outputs. (ICs and many kinds of circuit boards are normally tested in this way.) At a higher level, one might verify that all individual instructions are executed properly on a representative sample of data. Alternatively, to jump to the highest level, one could run benchmark programs to measure the throughput, speed, accuracy, and reliability of the machine, under a wide variety of input-data conditions, without ever concerning oneself about individual gates or instructions.

In practice, none of these approaches is sufficient by itself. In order to achieve high confidence that the system is performing according to its specifications, a well-balanced mixture of tests corresponding to various system levels is required. Such a multilevel testing philosophy is particularly important in a computer such as SIFT whose reliability requirements are exceptionally high. Fortunately, whereas a thorough analysis of fault modes at different levels would be impractical in many computers intended for general-purpose application, the basically simple structure of the units composing SIFT, plus the limited and known program repertoire, combine to make the task feasible and practical.

The highest-level specification of SIFT is expressed in terms of the reliability model, which relates overall SIFT reliability to the failure rates of individual hardware units (mainly processors), through other relevant variables such as the reconfiguration status, diagnostic detection and recovery times, and transient error probability. Part of the function of the testing will be to determine some of these rates and times empirically within a realistic application environment. Certain assumptions made in setting up the model may also be checked by actual experiment.

This same application environment will also provide the opportunity for a performance test, most of which may be run in the absence of faults.

At the lowest level, the test will verify that the actual SIFT hardware correctly executes the machine instructions and the communication functions called for at the lowest specification level (actual programs). This part of the validation will verify that the system is a true implementation of the design.

Another function of the testing will be to verify, through extensive demonstration, that certain fundamental principles of the SIFT concept are actually achieved. For example, it will be shown that all processors operating with identical inputs and initiated with identical memory contents produce identical outputs.

Finally, the testing program will provide an opportunity to optimize empirically certain diagnostic parameters, such as the number of detected errors received before reconfiguration is initiated, and to include actual diagnostic tests for subtle, long-latency faults that may escape identification during fault analysis.

General Strategy and Goals

This test philosophy for SIFT will be implemented by conducting the tests in a sequence of three stages of increasing difficulty. It is assumed at the start that debugging of the computer has been practically (if imperfectly) completed. The three test phases are as follows:

I Verification of function and performance of SIFT under fault-free conditions, for a variety of programs and input-data patterns.

II Verification of its capability to accommodate faults of various types, under the same program and input conditions and a representative set of faults (short-term survivability).

III Verification of dependence of reliability of SIFT on a variety of design assumptions and parameters, under a broader class of faults, inputs, and (perhaps) programs (long-term survivability).

The primary goal of the SIFT testing effort is to carry out these three test phases.

Secondary goals, as indicated above, are:

- To aid in debugging
- To aid in optimizing certain design parameters that are difficult to determine without operational experience.

All tests on SIFT will be carried out with the aid of a System Test Facility (STF), which consists of a Data General Eclipse computer, a suitable complement of peripherals, a set of testing programs, and some miscellaneous hardware for fault insertion.

2 Test Strategy and Approach

Test Parameters

The life cycle of a *permanent* fault is depicted in Figure A-1 and may be described as follows:

Fault occurrence (FO), followed by a *latency interval*, and terminating in

Fault detection (FD), which occurs repeatedly during an *alert interval*, ending at the rth detection with

Fault location (FL), when the *reconfiguration interval* starts, leading finally to a condition of

Fault handled (FH).

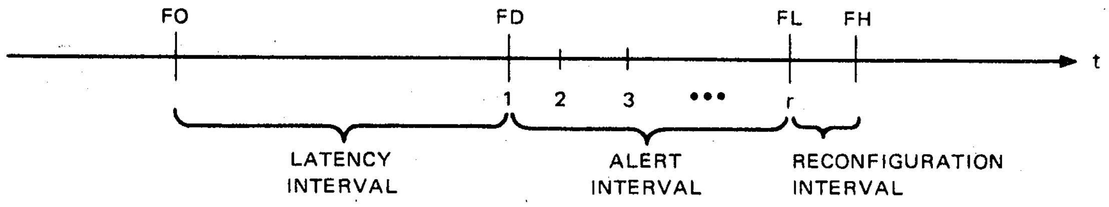


FIGURE A-1 LIFE CYCLE OF A PERMANENT FAULT

In general, the first FD causes voters to disregard outputs from the questionable unit, when generating consensus, but to continue to operate these units and to increment error counts as if they were still functional. When the count for a unit reaches a preset value r (2, 3, or 4), reconfiguration takes place.

For a *transient* fault, only the first two steps (FO and FD) pertain. That is, if fewer than r FDs occur within a reasonable time T_d, starting at the first FD, then the detected error is assumed to be due to a transient, rather than a permanent, fault, the questionable unit is reinstated, and voting returns to normal.

For an *undetected* fault, only the event FO takes place, of course.

The duration T_L of the latency interval is determined by the fault coverage (actually error coverage) of the application and diagnostic programs and the rates at which they are being run. The duration T_A of the alert interval is a function of these same factors, plus the integer r and the time T_d. The time for reconfiguration is a small constant, and is determined by the running times of the reconfiguration programs in the global and local executives -- not more than a few milliseconds.

The variables T_L and T_A, and especially the former, can be expected to be distributed approximately exponentially, a consequence of the nearly Poisson distribution of naturally occurring faults. The presumed shapes of the distributions enter into the SIFT reliability model as assumptions, and their average values appear as parameters in the model for calculating overall SIFT reliability. One of the specific goals of SIFT testing is to verify the shapes and average values of these distributions.

The above description applies to a single isolated fault, which is the most common type and the type that SIFT is designed to accommodate in a deterministic manner, one at a time. A more difficult and presumably less likely situation occurs when two or more of the fault cycles shown in Figure A-1 overlap one another in time. This overlap may occur in a variety of ways. If too many faults occur too close together in time, in too many different processors, the fault tolerance capability of SIFT is exceeded, and failure occurs in the form of either an output error or system collapse. Such a condition could conceivably occur if one or more latent faults lurked in the computer without manifesting during a long period of operation, only to manifest coincidentally with another fault, perhaps correlated with the first one.

Prior analysis has dealt with identifying these extreme situations, estimating their relative likelihoods of occurrence, and determining how SIFT should deal with them. For present purposes, it is sufficient to note that any such multiple overlapping faults, to the extent that they may be described at all, must be identified individually as potential faults by analysis of the circuitry, and injected during the tests to verify that they are properly accommodated. Examples of even very extreme conditions may be created to test the limits of SIFT's capability for fault accommodation.

Fault Diagnosis

Fault diagnosis is concerned with the detection, location, and handling of faults in a system. In the case of SIFT, the term will be reserved for these three steps of fault tolerance when they are carried out during *normal* operation, apart from debugging, check-out, and laboratory testing. Since faults in SIFT are accommodated by reconfiguration (rather than masking or repair) and since reconfiguration is an inherent part of SIFT and not a separable function, diagnosis refers here to the method and means for detecting random faults during normal operation, and tracing them to the defective processor. There is no need in SIFT to localize them any more finely, of course.

The detailed design of the fault-diagnosis programs in SIFT need not be carried out fully before testing is planned. Two of the features of these programs are important, however, and must be clearly understood.

First, diagnosis will be accomplished by a mix of three types of programs, as follows:

Type A: Regular application-task programs run under normal operating conditions. Input data is derived from sensors only. SIFT outputs (actuator and display signals) may or may not be blocked, depending on whether the corresponding control and display functions are needed at the moment. The error detection and location information derived from voting is active in both cases.

Type B: Regular application-task programs that have been "stretched" internally to force entry into computation modes normally encountered only infrequently, and, where possible, to make heavier and more varied use of certain subunits of the hardware and callable subprograms. Dummy sense data are provided to flex the programs through a wide range of input conditions. SIFT outputs are blocked, but the detection and location information from voting is active, as for Type A.

Type C: One or more specially prepared diagnostic programs, created specifically to ferret out elusive and hard-to-detect faults. These may be partially self-diagnostic--that is, they may be operable only within single processors, independent of the others.

It is not known at the present time to what extent Type C programs will be needed, what their particular form should be, and what is the appropriate mix of the three types of programs.

Second, none of these diagnostic programs is intended to detect faults in the 1553A links that drive the actuator outputs. Such faults appear to SIFT as changes in the signals provided to the actuators, just as if the actuators themselves were defective. This limitation of coverage is inherent in the design of any fault-tolerant multiprocessor, and is a consequence of the fact that, at some point, the protected signals and data within the redundant system must interface with the nonredundant environment. In the case of SIFT, tolerance to actuator failures will normally be provided by multiple, separately driven actuators, or by associated fault-tolerant actuator circuitry external to SIFT itself.

Proof-related testing

As indicated above, the SIFT hardware will be tested by a balanced combination of proof-related and fault-injection testing. These two aspects will now be discussed individually.

In order to support the analytic validation effort of SIFT, the test procedure includes portions at three levels of the proof hierarchy.

At the top level, overall performance will be verified by benchmark runs, using actual executive and task programs running in full complement. These will be run, first, in the absence of simulated faults. For these (and all subsequent) runs, SIFT will be interfaced to a program environment in the STF consisting of a simulated aircraft capable of operating in a variety of flight modes, as well as a simulated flight sequence with optional operator (pilot) intervention, aerodynamic disturbances, and navigation inputs.

At the bottom level, every individual BDX930 instruction used in SIFT programs will be validated independently against a formal ISP specification of that instruction. Each instruction validation will be carried out by executing, on SIFT, an STF-controlled test sequence that conforms to the formal specification. This part of the testing is an extension of the self-test program supplied with the Bendix 930.

At intermediate levels, three other tests of SIFT will be carried out. The *isolation* property will be verified by monitoring SIFT response to a large number of simulated faults during the execution of a variety of task and executive programs, to demonstrate that the set of nonfaulty processors is immune to erroneous signals produced by a faulty processor. The *identity* property of SIFT will be verified by running the benchmark programs in parallel on all processors, with clocks well synchronized, in order to demonstrate that nonfaulty processors having identical input data compute identical outputs (even with respect to timing). The *consensus* property will be verified by detailed monitoring of processor outputs during the creation of extreme error conditions, particularly those affecting synchronization. This will demonstrate the validity of the voting and Interactive Consistency implementations, and especially the property that the Interactive Consistency subprograms and voters produce identical outputs for both a minority of *1* and a minority > 1 errant processors. This last part is included to permit most of the fault simulation tests to proceed with faults injected into only *one* processor rather than two or more.

Fault Injection Testing

The goals of this portion of the testing are to demonstrate the immunity of SIFT outputs to hardware faults, including timing discrepancies, occurring in a minority of processors, and to identify any faults that have particularly long latencies.

In the broadest sense there is no need to be concerned in SIFT with the number, type, or frequency of faults occurring within a single processor, except as those faults manifest themselves at the interface between the faulty processor and the nonfaulty processors. This isolation property could be persuasively demonstrated by a test in which the offending processor is replaced by a signal source capable of broadcasting, to the other processors, a rich variety of data sequences representative of faulty conditions, as suggested by the diagram in Figure A-2.

For the signal source, one might use a random-noise generator, were it not for the fact that the time required to exhaust all possible meaningful sequences (or even a significant fraction of them) is many orders of magnitude too great to be accommodated in a test program of limited duration. Consequently, random-signal generation at the processor interface level cannot be relied upon for testing SIFT.

Instead, means must be found to generate only those processor output errors that correspond to a representative class of faults and operating conditions. There appear to be three possibilities:

1. The faulty processor could be replaced by an emulating computer within which faults are simulated.

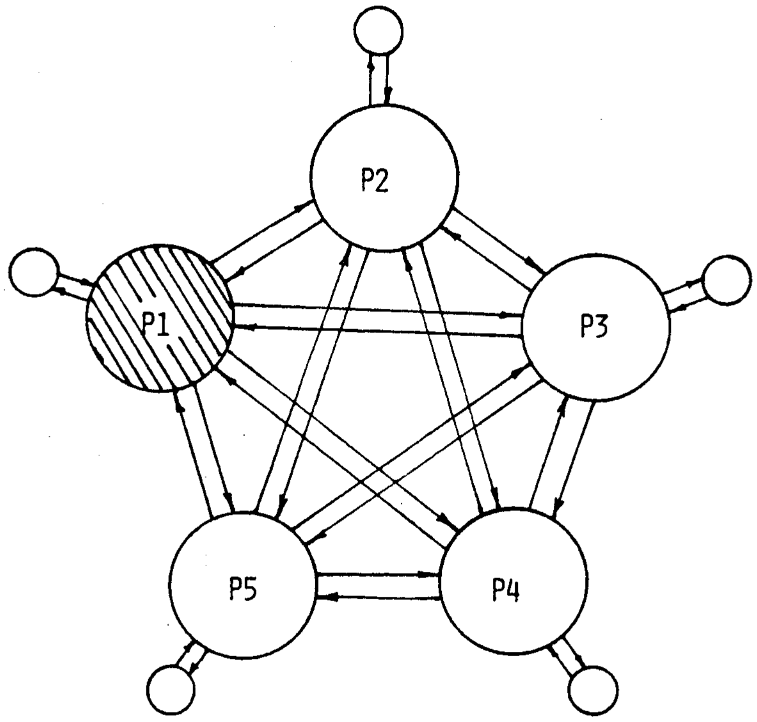


FIGURE A-2 FIVE-PROCESSOR SIFT CONFIGURATION

2. Faults could be simulated by means of software, within an actual SIFT processor.

3. Faults could be injected directly into the hardware of an actual SIFT processor.

Option 1 provides the broadest fault coverage of the three, but the relatively slow speed of the emulation slows down the testing considerably, even if a large, high-speed computer were employed for the emulation. The second option restricts the fault class somewhat to faults that can be simulated by changing bits in memory and registers, but it is fast (nearly real-time) and requires a minimum of external equipment. The third option restricts the fault class even more (though in a different way). Since the fault injection must be done manually, it is tedious and slow. There is also the danger of accidental permanent damage to the computer, due to overloads, burn-outs, exposure to the environment, or mechanical damage. However, the corresponding tests would be a more dramatic demonstration of SIFT's fault-tolerant feature.

For the testing of SIFT we have chosen the second and third options, with primary reliance on option 2: fault injection by software simulation.

Some faults may arise naturally during the testing of SIFT, but their expected frequency is so low that they are virtually useless for testing purposes. They affect the design and execution of the test plan only in that provision must be made to distinguish any that do occur from faults that are injected. Actually, faults left over from debugging will probably exceed the number of naturally occurring faults. The distinction is somewhat moot, since, in most cases (design errors excluded), it will never be known whether they occurred during fabrication or arose later.

Manual Injection of Faults During Testing

To a casual observer, manually injected faults will surely provide the most convincing evidence that fault tolerance has been achieved, even though many faults of interest cannot be so injected. It is planned that most of the following faults will be injected manually:

Components removed

Connectors disconnected

Boards extracted

Certain open circuits on boards

Certain short circuits on boards

Components replaced by others with different values

Signals injected at test points (e.g., noise, transients, 400 cycle hum, etc.)

Special adjustable circuits (for changing clock frequency, power-supply voltage, etc.)

Special hardware has been constructed for manual fault injection:

- 3 *board extenders* provide physical accessibility to board circuitry. Series switches and tie points are provided on all extended lines except power connections.

- 10 *IC-extenders* (DIP jumpers) provide electrical accessibility to board circuitry*.

*All IC's on the PI, T&C, MIC and CPU boards of processor #7 are mounted in IC sockets, so that these ICs may be extended for test purposes.

- 2 identical *test boards* consist of fault emulation logic circuits, mini-switches, and sockets, for creating various fault conditions in a SIFT circuit.

Faults are injected by direct connection into the signal lines of a selected IC that has been removed and placed in an appropriate socket on the test board. The DIP jumpers are used to connect the IC under test to its socket on the SIFT board. Faults are injected through mini-switches on the test board that are in-line between the IC and the DIP jumper.

Figure A-3 shows the logic diagrams for the fault-emulation logic. It includes three miniswitches that can be used to create various faulty logic conditions. Connection points for tying into this logic are located on the test board and labeled with letters. The letters correspond to the connection points listed in Table A-1, which shows the conditions that may be created to emulate a fault and which connection points and switch positions (where applicable) to use.

Fault Simulation

The broader class of simulated faults may be injected with the aid of a program, resident mainly in the STF, that has the capability of introducing errors into programs, data, and control information within the individual units of SIFT. Many such simulated faults correspond exactly to gate-level faults, and require changing only one or a few memory or register bits at a time. Others correspond to higher-level faults in procedures and programs or involve the modification of entire words.

It will be necessary during testing to augment the program library of the processors with a small auxiliary program for momentarily interrupting normal program execution so that errors may be injected. Another auxiliary program will be needed to make processor status information readily accessible for monitoring purposes. Requirements on these and other special testing programs, as well as the diagnostic programs, are described in Section 4.

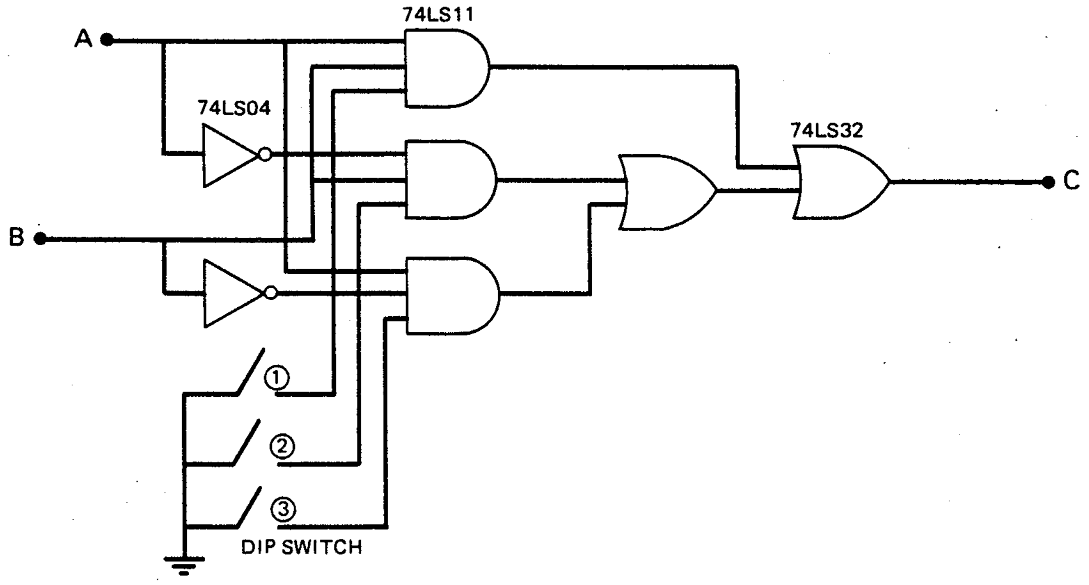
The class of faults to be injected through simulation include at least the following, singly and in combination:

- Transient and permanent faults
- Both correlated and uncorrelated multiple faults
- Faults at most or all system levels, for example:

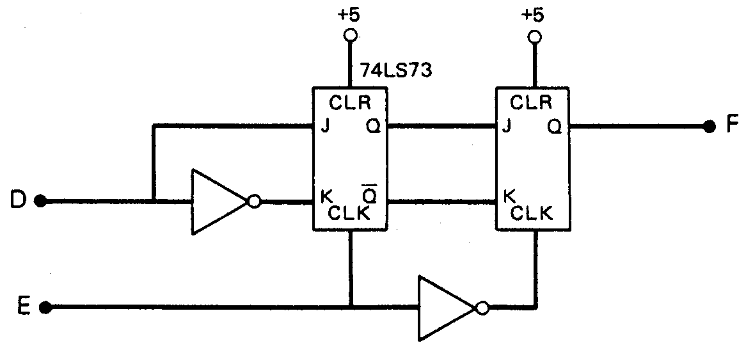
  Lowest level: opens, shorts, delays, electrical loading, environmental effects, defective memory cells, etc.

  Intermediate level: errors in instructions, clock drift, reduced computational

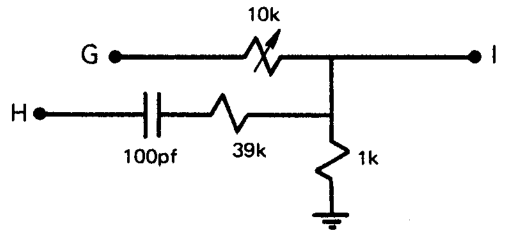
TWO OF THE FOLLOWING CIRCUIT:



ONE OF THE FOLLOWING CIRCIUT:



TWO OF THE FOLLOWING CIRCUIT:



THREE OF THE FOLLOWING CIRCUIT:

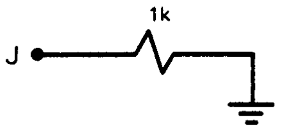


FIGURE A-3 TEST BOARD CIRCUITS

Table A-1
TEST BOARD CONNECTION POINTS FOR FAULT INJECTION

CONDITION	INPUT(S)	OUTPUT	SWITCH POSITIONS	OTHER CONNECTIONS
Stuck "0"	A	C	1, 2, and 3 closed	Tie B to gnd
Stuck "1"	A	C	1, 2, and 3 open	Tie B to gnd
Inverted bit	A	C	1 and 3 closed 2 open	Tie B to gnd
Logical OR	A, B	C	1, 2, and 3 open	
Logical AND	A, B	C	2 and 3 closed 1 open	
Delay of one half clock cycle	D, E (clock)	F	n/a	
Noise	G, H (noise generator)	I	n/a	
Short to gnd	J	J	n/a	

accuracy, incorrect decisions, flags and error indicators, erroneous data, etc.

High level: program execution and calling errors, improper sequencing , misreconfiguration, incorrect voting, time-outs, errors corresponding to violation of proof conditions, etc.

Fault Analysis

An analysis of all SIFT hardware units was conducted in order to determine, at least categorically, the classes of faults to which SIFT is susceptible and for which fault tolerance must be provided. Special emphasis was placed on those potential faults that could lie in interface hardware between processors, and those that correspond to software errors related to communications between processors--namely, those for which there is a risk that a faulty processor might interfere with the correct operation of a nonfaulty processor, or make it appear that its own fault lies within a nonfaulty processor.

An attempt was made to trace the errors resulting from every gate-level fault to the point where each error is manifested as one or more erroneous bits in an addressable memory location. This analysis assumed a range of normally occurring input conditions for stimulating the gate or storage element in question, and for creating a sensitized path from that gate or storage element to the memory location. This process was carried to completion for virtually all faults. In a few cases the input conditions were found to be so special that they could not be assumed to occur within a reasonable time during random input testing or normal operation; these involved priority circuitry of the data file, and a few spots in the microcode of the CPU. These faults require special Type C diagnostics. In the case of a defective bus output driver in the broadcast unit, a fault in one processor may appear as a fault in another. In a few other cases faults were found to be undetectable, because they fell in redundant or protective circuitry: timeout units in the transmitter and receiver sequencers, and in the power-on and power-off circuitry. These faults cannot be detected except by manual disassembly. Finally, a few others occur in memory or circuit functions that were provided for in the design of the processor but were not used by the operating programs: a multiple-word broadcast mode, some of the 930 instructions, portions of the main memory, and one of the data-transfer modes of the 1553A link.

Stress Tests

Stress tests are planned within the testing program in order to determine certain limits of system functioning under extreme fault, error, and operating conditions. Even though these situations are impossible or very unlikely to occur during normal operations, the information obtained from them will be used to provide points of calibration in the relation that expresses the fault tolerance as a function of the number, variety, and timing of fault conditions. The stress tests may also identify unexpected weaknesses or susceptibilities in SIFT.

Typical stress fault conditions will

- Introduce transient faults at an increasing rate
- Increase the number of faults introduced at one time (correlated or uncorrelated)
- Accelerate the effective clock drift rate (by lengthening the clock resynchronization period)
- Force very frequent reassignment, rescheduling, and reconfiguration
- Increase the application task load to fit more tightly within the computation cycle (frame)
- Reduce the initial number of available processors to three
- Use dummy sensor input data that correspond to extreme aircraft motion and aerodynamic changes
- Reduce power-supply voltage.

All of these conditions except the last may be imposed through software only.

Symmetry Test

The first step in Phase I of SIFT testing will be to carry out a symmetry test of all processors and all boards, in order to verify, under full-load conditions, that

- All processors are functional in all slots
- All boards are functional in all processors
- All board pairs are functional in all processors.

This test is necessary to provide assurance that the multiple symmetries inherent in the SIFT architecture are implemented in the actual hardware. These conditions were not satisfied during the debugging--that is, certain processors did not work in particular slots, and, more commonly, some of the boards and board pairs appeared to be incompatible with certain processors or with one another. These problems, most or all of which were subsequently diagnosed and repaired, were due to marginal and/or noisy signal levels and to construction defects that introduced subtle timing problems.

To this end, a compact series of seven tests was designed to satisfy the three conditions listed above. It may be seen by inspection of the arrays in Table A-2, that all possible assignments of each processor to a different slot, and each board to a different processor, are included except that (a) the 1553A boards are not moved from their home processors (*one* board type may be safely considered to be part of the processor), and (b) the two memory boards, which do not communicate with one another, are kept together. Every pair of differing boards of the same two types (except the two memory boards, as just indicated) is operated in a common processor during exactly one of the seven tests.

3 Test Procedure and Requirements

Phase I--Fault-Free Performance Tests

The goal of this first series of tests is to demonstrate that the SIFT computer is capable of carrying out a typical set of aircraft-control tasks, as well as the local and executive program functions.

The first step in this phase will be to conduct the symmetry test described above.

For the Phase I test proper, the STF provides a simulated aircraft in the form of a program on the Eclipse that realizes the generalized equations of aircraft motion within an aerodynamic environment. The inputs to this program are actuator signals generated by SIFT, as well as mock aircraft-control commands and mode-selection commands from the test operator. The outputs to this program consist of sensor inputs to SIFT, as well as signals to the test operators for monitoring aircraft response. (Eventually, this will include a dynamic display.) We have also provided for environmental influences such as wind gusts, downdrafts, and aircraft malfunctions (e.g., one engine out).

The only faults in this phase of testing will arise either randomly or because of incomplete initial checkout and debugging.

The entire SIFT computer will be run with a full program load and a variety of applied inputs, in order to provide high assurance that it is operating correctly and at the planned performance level, in the absence of faults.

Phase II--Short-Term Survivability Tests

The goal of the short-term survivability tests is to demonstrate the capability of SIFT to recover successfully from various types of faults when running selected programs and using selected input data patterns. The Phase II demonstration is intended to be almost entirely qualitative; that is, the selection of faults, programs, and input data will be representative but not exhaustive, and will not be applied in the large variety to be employed in the quantitative tests in Phase III.

Each individual test in this series will be carried out in the following sequence of steps:

Table A-2

PERMUTATION TESTS

P	S	930	T&C	MEM 1 & 2	BR	MIC	PI	1553
1	1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3	3
4	4	4	4	4	4	4	4	4
5	5	5	5	5	5	5	5	5
6	6	6	6	6	6	6	6	6
7*	7	7	7	7	7	7	7	7

TEST No. 1

P	S	930	T&C	MEM 1 & 2	BR	MIC	PI	1553
1	4	7	6	5	4	3	2	1
2	1	1	7	6	5	4	3	2
3	5	2	1	7	6	5	4	3
4	2	3	2	1	7	6	5	4
5	6	4	3	2	1	7	6	5
6	3	5	4	3	2	1	7	6
7*	7	6	5	4	3	2	1	7

TEST No. 2

1	5	6	4	2	7	5	3	1
2	3	7	5	3	1	6	4	2
3	1	1	6	4	2	7	5	3
4	6	2	7	5	3	1	6	4
5	4	3	1	6	4	2	7	5
6	2	4	2	7	5	3	1	6
7*	7	5	3	1	6	4	2	7

TEST No. 3

1	2	5	2	6	3	7	4	1
2	4	6	3	7	4	1	5	2
3	6	7	4	1	5	2	6	3
4	1	1	5	2	6	3	7	4
5	3	2	6	3	7	4	1	5
6	5	3	7	4	1	5	2	6
7*	7	4	1	5	2	6	3	7

TEST No. 4

1	3	4	7	3	6	2	5	1
2	6	5	1	4	7	3	6	2
3	2	6	2	5	1	4	7	3
4	5	7	3	6	2	5	1	4
5	1	1	4	7	3	6	2	5
6	4	2	5	1	4	7	9	6
7*	7	3	6	2	5	1	4	7

TEST No. 5

1	6	3	5	8	2	4	6	1
2	5	4	6	1	3	5	7	2
3	4	5	7	2	4	6	1	3
4	3	6	1	3	5	7	2	4
5	2	7	2	4	6	1	3	5
6	1	1	3	5	7	2	4	6
7*	7	2	4	6	1	3	5	7

TEST No. 6

1	1	2	3	4	5	6	7	1
2	2	3	4	5	6	7	1	2
3	3	4	5	6	7	1	2	3
4	4	5	6	7	1	2	3	4
5	5	6	7	1	2	3	4	5
6	6	7	1	2	3	4	5	6
7*	7	1	2	3	4	5	6	7

TEST No. 7

P = Processor Number
S = Slot Number
* Socketed Processor

Numbers within each 7 x 7 square give **board number** of each type (column) to be inserted in each processor (row).

1. Initialize SIFT.

2. Select a representative fault, either by the test operator or by systematic or random selection from the stored catalog of candidate faults. Inject the fault into the initialized SIFT, either manually or by simulation, as appropriate.

3. Select the program or program mix in a similar fashion, and so instruct SIFT.

4. Select the input data patterns similarly, and prepare the corresponding files or programs in the STF.

5. Issue the start signal to SIFT, and run the program.

6. The program will stop when a certain preselected waiting time (e.g., one second) has elapsed after proper reconfiguration, or a longer time (e.g., one minute) if no proper reconfiguration took place.

The system is then ready for the next individual test.

This same sequence applies to the injection of multiple faults, except that longer waiting times will be allowed to provide for multiple reconfigurations and the increased likelihood of unpredictable after-effects.

In all cases, the STF will monitor the internal status of SIFT for later analysis, and, in particular, to make certain that reconfiguration was carried out properly--that is, that the units retired from use included the one (or more) into which the fault(s) had been injected. The most informative monitoring is expected to occur at a high level, even though most of the faults are injected at a low level.

Phase III--Long-Term Survivability Tests

The main goals of the long-term survivability tests are to

- Gather data necessary for estimating SIFT reliability, using the Markov reliability model

- Check certain of the assumptions on which the reliability model is based

- Determine quantitative limits on SIFT's capability for recovering from extreme fault conditions.

A secondary goal is to collect data that will aid in optimizing design decisions concerned with

- Reconfiguration strategy

- Diagnostic program mix
- Consistency algorithms for clocks and for sensor input data
- Duration of time-out intervals.

Phase III may be regarded as the quantitative counterpart of Phase II.

These goals will be achieved by repeating many times the same types of tests employed in Phase II, except that (a) the particular faults injected, the input data and the programs being run will vary over much wider ranges, and (b) only simulated faults will be injected. Detailed records will be kept of the times at which various fault-related events take place (FO, FD, FL, and FH), and all other important changes in system status between these times.

At the close of each individual fault test, an STF program will digest and compress the time and event data in these records for permanent storage and later statistical analysis.

Statistical Analysis

This analysis, to be carried out later by another STF program, calculates the frequency distribution, the corresponding averages, and the time constant for

- Latency interval
- Alert interval
- Reconfiguration interval
- Time from FO to system failure (on stress tests).

These calculations will be made for the tests as a whole, as well as for separate portions corresponding to particular fault classes, programs, program mixes, and families of input data.

The first of these, the latency interval, is by far the most important since it is the main parameter that must be entered into the reliability model for determination of overall SIFT reliability. It is also used for expressing the fault (or error) coverages of the diagnostic programs. The analysis will also provide information on the relative frequencies of incorrect to correct reconfigurations, of effectively undetectable faults to detectable faults, and of multiple-processor to single-processor fault location (in cases where single-processor reconfiguration would have been sufficient).

4 SIFT Software Needed for Diagnosis and Testing

This section presents the rationale and the requirements for special SIFT software needed for internal diagnosis and for testing (designated Type B and Type C diagnostics). (Type A programs are the regular task programs.)

Type B Diagnostics

This diagnostic function is achieved by running the flight control and global executive tasks using dummy, randomized inputs while all I/O connections to the 1553A link and all local executive action are blocked.

To this end, additional SIFT software is required (a) to generate pseudorandom numbers from a continually changing seed common to all working processors, and (b) to replace each input variable by the current instance of the pseudorandom number (or, in some cases, by the sum of the two, or another simple function*) as each task is called in diagnostic mode by the scheduler. Blocking of the I/O will be carried out by the scheduler, by simply eliminating the calls to those tasks that initiate 1553A transmission and reception.

In order for the Type B diagnostic programs to be effective, it is essential that the randomization of inputs not throw any of the programs into running conditions that would cause different processors to produce different results. This might occur in computations dependent upon the processor identification number (PID) or the value of the real-time clock (RTC); it could also occur if any program were thrown into a loop, which might be broken at an unpredictable point by a time out.

The flight-control program is completely deterministic and time-constrained, even on mode control, whenever random signals are applied to all of its inputs. Furthermore, no part of this program is dependent upon processor number or the value of the real-time clock. Thus, the flight-control program may be operated in Type B diagnostic mode without difficulty.

The operating system programs are similarly deterministic under random stimulation of their inputs, except possibly for nonreplicated tasks--that is, those based on unvoted data: (a) clock synchronization, (b) error reporting, and (c) reconfiguration. Diagnostic versions of these three tasks must be created by forcing them into a replicated mode, namely, by providing them with dummy, randomized input data common to all processors, then broadcasting the results and voting on the broadcast values, just as if the task were replicated. Output values are blocked from further action. Specifically:

*In most cases these pseudorandom values will be subjected first to a simple, fixed transformation, in order to make them more representative of actual values.

- Synchronization:

  Inputs: A set of dummy clock values, provided by the pseudorandom generator in each processor. (Each processor uses the same set of dummy values.)

  Output: A replacement value for one of the clocks.

  Condition: Since the processor ID is used on this calculation, it too must be replaced by a pseudorandom value.

- Error reporting:

  Inputs: A set of dummy voter outputs, provided by the pseudorandom generator. (Same for all processors.)

  Outputs: Processor error report vector.

  Condition: Same as above.

- Reconfiguration:

  Inputs: A dummy reconfiguration request and task list, both randomized. (Same for all processors.)

  Outputs: Schedule information from the precomputed tables.

  Conditions: Only one or a few entries from the schedule tables are to be generated as outputs on each execution of the diagnostic program, under control of a randomized table address.

For the presently assumed diagnostic strategy, all tasks (including the clock synchronization, error reporting, and reconfiguration tasks) perform during diagnostic executions just as they would normally, whenever faults are discovered, notwithstanding the fact that some of the diagnostic tests create conditions that are far outside the range of normal functioning. For instance, a processor is presently deconfigured when a fault is discovered in a portion of its circuitry or memory that may not be used at all during the execution of nondiagnostic tasks. This policy appears to be sound, but it should be reviewed later after additional experience with SIFT.

Type C Diagnostics

This function requires that a special SIFT program be included to detect all known faults that are not necessarily detected within a reasonable time by regular task programs (Type A) or the above diagnostic programs (Type B).

Candidate faults in this group are as follows:

- Highly time-dependent faults, such as can occur in the data-file priority logic, associated interrupt logic, the 16 MHz oscillator, and any portions of the 930 that happen to be employed only during execution of the clock-synchronization task.

- Faults in protection circuitry, such as timeout units and the power-on control logic.

- Highly data-dependent faults, such as those that may affect infrequently used portions of the schedule tables (e.g., for the case of a small number of surviving processors) and those that affect possible but unlikely combinations of error reports.

- Faults in circuitry not used in Type A or B programs, such as those in certain portions of the main and microprogram memories associated with rarely used instructions.

- A few residual faults occurring in particular gates activated only under very special control conditions. (A few of these have been found by hand analysis of the processor logic; others might be uncovered by a full-scale logic simulation [not planned]; still others may turn up during testing as long-latency faults or faults effectively undetected.)

- Faults in redundant circuitry, such as those in gates that have been paralleled to increase their fanout capability, and in paralleled filter capacitors.

Specific test conditions have been defined for all of these fault classes except the last. For this last class, diagnosis is impossible without extensive circuit modifications, which we feel to be undesirable and unnecessary.

All of the above tests are to be collected in the form of a special diagnostic task program, to be run in replicated mode, just like other replicated tasks, and at a rate to be determined.

Some of the Type C diagnostics require small modifications to the operating system itself in order to provide branching or temporary storage of control conditions not otherwise needed.

SIFT Status Monitoring

The status of SIFT must be checked at certain points during the testing cycle in order to (a) measure the times at which error detection and reconfiguration take place, so that the frequency distributions of fault latency times may be calculated for various fault classes and running conditions; (b) distinguish naturally occurring faults from those deliberately injected, and distinguish any programming or design errors from fault-induced errors; and (c) provide information needed to trace the propagation of errors due to faults that have long latencies or are undetected during testing.

The Eclipse MONITR package includes the capability for normally initiated halting and for reading out the contents of arbitrary memory locations from the halted SIFT. However, the MONITR has no provision for halting under SIFT-generated conditions. Test points have been provided for this purpose on the front panel of the processor, but this means was provided only as a backup; indeed, to use it for triggering Eclipse interrupts appears to be very awkward. Instead, we recommend a small internal SIFT program, to be used only during testing, which stores locally selected status information, in an otherwise unused portion of main memory, for later readout by the Eclipse during a halt. It will also be necessary to provide flagged branches in the error reporting, reconfiguration, and (provisionally) clock-synchronization task programs, to cause this status recording to be initiated whenever the processor is in the test mode (test flag *set*).

The status information to be recorded, and the times of recording, are as follows:

- Which processors are "up" and which schedules are in current use: on initialization and whenever there is any change in the schedules in use.

- Error report, and the data-file address of the apparently erroneous data words: whenever an error is detected.

- Real-time clock value: before and after each fault insertion, fault detection and reconfiguration, and after each clock synchronization.

- Frame* and subframe numbers: in all cases.

The frame and subframe numbers, in conjunction with knowledge of which schedules are in use, determine indirectly which program was running when the error detection, reconfiguration, or synchronization event occurred.

The above functions are performed separately in each processor. Comparison and reduction of the data retrieved from the various processors after halting are carried out in the Eclipse, after the test results are displayed and before they are archived on disc.

Fault Injection through Simulation

This function requires that one or more selected bits of one or more words in main memory, data file, transaction file, discretes, or the accessible registers in the 930 be

*The frame count should be chosen so that it is unique over a period of at least three minutes.

modified on command from the Eclipse. Modification consists of replacing the selected bit by a *0*, by a *1*, or by the binary complement of its current value.

This function might be carried out entirely from the Eclipse, by halting, reading out of selected word, changing the selected bit, replacing the changed word and restarting. However, its implementation is best controlled directly from the SIFT end, for the following reasons:

- The time at which the fault is injected is most properly a function of the SIFT cycle time (e.g., frame number) rather than Eclipse time. Obviously SIFT time is easier to access directly from SIFT.

- The simulation of permanent and intermittent faults by modifying registers or memory contents requires that the fault be reinserted after each occasion in which the register or memory may have been written over during the program executions. This reinjection frequency is quite high for some of the 930 accumulators, which may be written into many times in a single subframe. Again, reinjection is much easier from SIFT than from the Eclipse*.

For these reasons, it will be necessary to provide, in the SIFT software, a flag ("test mode") and branches to a special fault-injection procedure. This procedure will read, from preassigned locations in the main SIFT memory, the fault-insertion information, which has been preloaded from the Eclipse. This information specifies which bits of which words are to be modified and how, and at what frequency. The procedure will then carry out the recovery bit modifications.

In general, the conversion of transient into permanent or intermittent faults will be handled (1) by the means just described for repeated transient injection, or (2) by hardware fault insertion, or (3) by assuming that the modified memory cell has not been rechanged during program execution, and then checking it at the end of the test. Method (2) will be used for most nontransient faults in the 930 accumulator and address registers. Method (3) is deemed to be sufficient for most faults in read-only portions of the main memory.

Clock Synchronization

A test of the clock-synchronization program may be done (1) by injecting carefully selected, incorrect values into the RTC, and then monitoring the consequences, or (2) by

*Maintenance of RTC synchronization during halting presented a number of problems, since the RTC continued to run in a halted processor. The original plan was to read and store each RTC value immediately after halting, then to fetch and reinsert this value just before restarting. However, this method introduced skews which appeared to be greater than tolerable. Consequently, the hardware was modified to freeze the RTC during halting.

simply reducing the frequency at which the synchronization program is executed, relying upon the normal rate of clock drift to create the erroneous clock values. The second alternative is simpler, and appears to be sufficient. The clock resynchronization interval has been parameterized so that it can be readily changed from the Eclipse during testing.

Overhead for Diagnostics and Testing

The requirements stated in the above sections assume that sufficient main-memory space and sufficient free subframes are available in SIFT to implement them. At the date of this writing these conditions appear to be met.

Appendix B. Synchronizing Clocks in the Presence of Faults*

1. Introduction

In a fault-tolerant multiprocess system, it is often necessary for the individual processes to maintain clocks that are synchronized with one another Lamport [15, 16] and Wensley [34]. Since physical clocks do not keep perfect time, but can drift with respect to one another, the clocks must be resynchronized periodically. Such a fault-tolerant system needs a clock-synchronization algorithm that works despite faulty behavior by some processes and clocks. This paper describes three such algorithms.

It is easy to construct fault-tolerant synchronization algorithms if one restricts the type of failures that are permitted. However, it is difficult to find algorithms that can handle arbitrary failures - in particular, failures that can result in "two-faced" clocks. As an example, consider a network of three processes. We would like an algorithm in which a fault in one of the processes or in its clock does not prevent the other two processes from synchronizing their clocks. However, suppose that:

- Process 1's clock reads 1:00
- Process 2's clock reads 2:00
- Process 3's clock is faulty in such a way that when read by Process 1 it gives the value 0:00 and when read by Process 2 it gives the value 3:00.

Processes 1 and 2 are in symmetric positions; each sees one clock reading an hour earlier and one clock reading an hour later than its own clock. There is no reason why Processes 1 and 2 should change their clocks in a way that would bring their values closer together.

The algorithms described in this paper work in the presence of any kind of fault, including such malicious, two-faced clocks. The first one is called an *interactive convergence* algorithm. In a network of at least $3m + 1$ processes, it will handle up to m faults. Its name is derived from the fact that the algorithm causes correctly working clocks to converge, but the closeness with which they can be synchronized depends upon how far apart they are allowed to drift before being resynchronized.

The final two algorithms are called *interactive consistency* algorithms, so named because the nonfaulty processes obtain mutually consistent views of all the clocks. The closeness with which clocks can be synchronized depends only upon the accuracy with which processes can read each other's clocks and how far they can drift during the synchronization procedure. They are derived from two basic interactive consistency algorithms presented in Lamport [17]. The first one requires at least $3m + 1$ processes to handle up to m faults. The second algorithm assumes a special method of reading clocks, requiring the use of unforgeable digital signatures, to handle up to m faults with as few as $2m + 1$ processes. We conjecture that, like the original interactive consistency problem Pease [24], $3m + 1$

[1]From Lamport[17]

processes are required to allow clock synchronization in the presence of m faults if digital signatures are not used. Although we have not been able to prove it, this conjecture is strongly suggested by the similarity between clock synchronization and interactive consistency. Moreover, the number $3m + 1$ arises in the interactive convergence algorithm for seemingly different reasons than in the interactive consistency algorithm, suggesting some general principle about the need for $(3m + 1)$-fold replication to handle m faults.

2. The Problem

Clocks

When discussing clocks, there are two kinds of time that are involved:

real time: an assumed Newtonian time frame that is not directly observable.

clock time: the time that is observed on some clock.

We adopt the convention of using lower-case letters to denote quantities that represent real time and upper-case letters to denote quantities that represent clock time. Thus, we will let the "second" denote the unit of real time and the "SECOND" denote the unit of clock time. Within this convention, we use Roman letters to denote large values and Greek letters to denote small values. In most applications, "large" times will be on the order of milliseconds or more and "small" times on the order of microseconds.

It is customary to define a clock to be a mapping C from real time to clock time, where $C(t) = T$ means that at real time t the clock reads T. We find it more convenient to define a clock to be the inverse of this function, so a clock is a mapping c from clock time to real time, with $c(T)$ denoting the real time at which the clock c has the value T.

Two clocks c and c' are synchronized to within δ at a clock time T if $|c(T) - c'(T)| < \delta$, so they reach the value T within δ seconds of one another. This is the kind of synchronization that is needed for a system such as SIFT [34], in which clocks are used to generate actions. If two processes' clocks are synchronized to within δ at time T, then actions generated by the two processes at that time occur within δ seconds of one another.

If the clocks are used to measure when events occur, rather than to generate events, then one is concerned with the difference between the inverse clocks – the mappings from real time to clock time. It is easy to show that for clocks c and c' that run at approximately the correct rate, if $|c(T) - c'(T)| < \delta$ for all times T in some interval, then $|C(t) - C'(t)| \lesssim \delta$ for all t in an appropriate interval, where C and C' are the inverse clocks. Hence, our synchronization algorithms can be used in this situation too.

It is most convenient to consider the clock c to be a function on a continuous interval. The discreteness of a real clock is modeled as an error in reading the clock.

Definition 1: *A clock c is a* good clock *during the interval $[T_1, T_2]$ if it is a monotonic, differentiable function on $[T_1, T_2]$, and for all T in $[T_1, T_2]$:*

$$\left|\frac{dc}{dT}(T) - 1\right| < \rho/2 \quad .$$

We consider a network of n processes, where each process p contains a clock c_p. We assume that these clocks are initially synchronized to within δ_0 of one another at the "starting time" $T^{(0)}$, so we have:

A0. For all processes p and q: $\left|c_p(T^{(0)}) - c_q(T^{(0)})\right| \ < \ \delta_0$.

We will not consider the problem of how this initial synchronization is achieved.

A nonfaulty process's clock is assumed to be good. More precisely, we assume:

A1. If process p is nonfaulty during the real time interval $[t_1, t_2]$, then c_p is a good clock during the interval $[T_1, T_2]$, where $c_p(T_i) = t_i$, $i = 1, 2$.

Synchronization

In order to maintain synchrony despite the difference in clock rates (two processes' clocks can drift apart at the rate of up to ρ seconds per SECOND), the processes must periodically increment their clocks. To increment a clock c by A SECONDS - i.e., to add A to the value read from the clock - formally means defining a new clock c' by

$$c'(T) = c(T - A) \quad .$$

For simplicity, we assume that the clocks are resynchronized every R SECONDS. Let $T^{(i)} = T^{(0)} + iR$, and let $R^{(i)}$ be the interval $[T^{(i)}, T^{(i+1)}]$. Resynchronizing the clocks every R SECONDS means having each process p use a clock $c_p^{(i)}$ on the time interval $R^{(i)}$, where

$$c_p^{(i)}(T) = c_p(T + C_p^{(i)}) \tag{1}$$

for some constant $C_p^{(i)}$. We assume that $C_p^{(0)} = 0$, so $c_p^{(0)} = c_p$.

For convenience, we introduce the following terminology.

Definition 2: *A process p is said to be nonfaulty up to time $T^{(i+1)}$ if it is nonfaulty during the real-time interval* $[c_p^{(0)}(T^{(0)}), c_p^{(i)}(T^{(i+1)})]$.

Note that this interval runs from the time process p is started until the time its clock reaches the end of the i^{th} synchronization interval $R^{(i)}$.

We require that a synchronization algorithm satisfy the following condition for any $i \geq 0$:

Clock Synchronization Condition: There exist constants m, δ, and Σ such that for all p and q, if all but at most m processes are nonfaulty up to time $T^{(i+1)}$ then:

S1. If Processes p and q are nonfaulty up to time $T^{(i+1)}$, then for all T in $R^{(i)}$:

$$\left|c_p^{(i)}(T) - c_q^{(i)}(T)\right| \ < \ \delta \quad .$$

S2. If Process p is nonfaulty up to time $T^{(i+1)}$, then:

$$\left|c_p^{(i+1)} - c_p^{(i)}\right| \ < \ \Sigma \quad .$$

Our problem is to find an algorithm for choosing the values $C_p^{(i+1)}$ such that if the Clock Synchronization Condition holds for i, then it will hold for $i+1$.

Condition S1 is elementary: it states that the clocks are synchronized to within δ. Condition S2 states that for each resynchronization, a nonfaulty process always increments its clock by less than Σ. This is important for two reasons:

- The resynchronization procedure introduces an error of at most Σ/R into the average running rate of the clocks. Hence, a small value of Σ means that the processes' clocks maintain a good approximation to absolute real time.
- The resynchronization can cause a process to move its clock ahead by some amount A. This means that A SECONDS of clock time have been lost, and none of the events that the process should have generated during that lost interval can be generated at the proper time. To prevent this from causing errors, each synchronization interval must begin (or end) with an interval of length Σ during which nothing is scheduled to happen, so the process is idle for Σ of every R SECONDS.

If the discontinuity introduced by incrementing the clock is undesirable, then the correction could be spread over the entire interval $R^{(i)}$ by changing the clock's running rate. Our results are easily applied to this case by analyzing the difference between an adjusted clock and the discretely incremented one that we consider.

We place no restriction on the clock of a process that has failed. Thus, we are not considering the problem of restarting a failed process and bringing it into synchrony with the other processes. This is a nontrivial problem, whose solution depends upon the details of how a process reads other processes' clocks, and is beyond the scope of this paper.

Reading Clocks

Any clock synchronization algorithm requires that processes read other processes' clocks. We assume that all the reading of clocks and transmitting of information in the computation of $C_p^{(i+1)}$ is done in the final S seconds of the interval $R^{(i)}$–i.e., during the interval $S^{(i)} \equiv [T^{(i+1)} - S, T^{(i+1)}]$. For our first two algorithms, we require that during the interval $S^{(i)}$, each process p reads the clock of every other process q with an error of less than ϵ. More precisely, we make the following assumption.

A2. If the Clock Synchronization Condition holds for i, and process p is nonfaulty up to time $T^{(i+1)}$, then, for each other process q, p obtains a value Δ_{qp} during the interval $S^{(i)}$. If q is also nonfaulty up to time $T^{(i+1)}$, then

$$\left|c_p^{(i)}(T_0 + \Delta_{qp}) - c_q^{(i)}(T_0)\right| < \epsilon \tag{2}$$

for some time T_0 in $S^{(i)}$.

For $p = q$, we take $\Delta_{pq} = 0$, so (2) holds in this case, too.

The actual method of reading the clocks by which A2 is satisfied is of no concern for our first two algorithms. It might be possible for p to read q's clock directly, or it might require some cooperative action by both processes. In the latter case, determining Δ_{qp} may require the synchrony of the two processes' clocks, which is why we assume in A2 that the Clock

Synchronization Condition holds for i. For our third algorithm, we will make a different assumption from which it is possible to derive a method of reading clocks that satisfies A2.

Approximations

For real clocks, ρ can be reduced to the order of 10^{-6} or less. We will simplify our calculations by making approximations based upon the assumption that $n\rho \ll 1$, where n is the number of processes. This means that we will neglect quantities of order $n\rho\epsilon$, and $n\rho^2$ in our calculations. The reader will be able to check the validity of these calculations by showing that for an approximate inequality of the form $x \lesssim y$, the neglected terms are at most of order $n\rho y$. Note that the inequality $x \lesssim y$ can be interpreted as $x < y'$ for some $y' \approx y$. We also assume $R \gg \epsilon$, which is the only case of practical interest.

3. The Interactive Convergence Algorithm

The Clock Synchronization Condition for i implies that if p and q are nonfaulty, then the clocks $c_p^{(i)}$ and $c_q^{(i)}$ will differ by less than δ. Process p can therefore assume that process q is faulty, and ignore the value of q's clock, if that clock differs from its own by more than δ. In the interactive convergence algorithm, Process p does that by setting its clock to the average of all the clocks that are not too different from its own.

The algorithm is based upon the following simple result.

Lemma 1: *If S1 holds for i and processes p and q are nonfaulty up to time $T^{(i+1)}$, then*

$$|\Delta_{qp}| \lesssim \delta + \epsilon \quad .$$

Proof: Let T_0 and Δ_{qp} be as in A2. Writing

$$c_p^{(i)}(T_0) - c_p^{(i)}(T_0 + \Delta_{qp}) = c_p^{(i)}(T_0) - c_q^{(i)}(T_0) + c_q^{(i)}(T_0) - c_p^{(i)}(T_0 + \Delta_{qp}) \, ,$$

it follows easily from S1 and A2 that

$$\left|c_p^{(i)}(T_0) - c_p^{(i)}(T_0 + \Delta_{qp})\right| < \delta + \epsilon \quad .$$

The desired result then follows from A1 and the assumption that $\rho \ll 1$. ∎

This lemma leads us to the following algorithm, which defines how process p computes the correction to its clock for the $(i+1)^{st}$ synchronization period using the values Δ_{qp} obtained during the i^{th} period. We assume that there are n processes, numbered from 1 through n.

Algorithm CNV(δ): *For all p:*

```
$C_p^{(i+1)} = C_p^{(i)} + \Delta_p$
    where $\Delta_p$  $\equiv$  $(1/n) \sum_{r=1}^{n} \overline{\Delta}_{rp}$
          $\overline{\Delta}_{rp}$  $\equiv$  if $r \neq p$ and $|\Delta_{rp}| < \Delta$ then $\Delta_{rp}$
```

else 0 .

$$\Delta \approx \quad \delta + \epsilon$$

Theorem I: *If*

- $3m < n$
- $\delta \gtrsim \max(n'(2\epsilon + \rho(R + 2S')), \delta_0 + \rho R)$,

  where $n' \equiv n/(n-3m)$

  $S' \equiv (n-m)S/n$
- $\delta \ll \min(R, \epsilon/\rho)$

then Algorithm CNV(δ) *satisfies the Clock Synchronization Condition with* $\Sigma = \Delta$.

Proof: Condition S2 is easy, since Δ_p is the average of n terms, each less than Δ. We prove Condition S1 by induction on i. For $i = 0$, A1 implies that two nonfaulty clocks that are synchronized to within δ_0 up to time $T^{(0)}$ will remain synchronized to within $\delta_0 + \rho R$ at time $T^{(1)} = T^{(0)} + R$. Condition S1 then follows immediately from A0 and the hypothesis.

We therefore assume that S1 holds for i and prove it for $i+1$. We begin with the following lemmas.

Lemma 2: *If S1 holds for i and process p is nonfaulty up to time* $T^{(i+2)}$, *then for any* Π *such that* $|\Pi| < R$ *and any* T *in* $S^{(i)}$:

$$\left|c_p^{(i)}(T + \Pi) - [c_p^{(i)}(T) + \Pi]\right| < (\rho/2)\Pi$$

Hence, if $\rho\Pi$ *is negligible, then*

$$c_p^{(i)}(T + \Pi) \approx c_p^{(i)}(T) + \Pi \quad .$$

Proof: This follows easily from A1. ∎

Lemma 3: *If S1 holds for i, and p and q are nonfaulty up to time* $T^{(i+2)}$, *then for any* T *in* $S^{(i)}$ *and any* Π *such that* $|\Pi| < R$ *and* $\rho\Pi$ *is negligible:*

$$\left|c_p^{(i)}(T + \Pi + \Delta_{qp}) - c_q^{(i)}(T + \Pi)\right| \lesssim \epsilon + \rho S$$

Proof: Letting T_0 be as in A2, we have

$$\begin{aligned}
&\left|c_p^{(i)}(T + \Pi + \Delta_{qp}) - c_q^{(i)}(T + \Pi)\right| \\
&\quad = \left|c_p^{(i)}(T_0 + \Delta_{qp} + T - T_0 + \Pi) - c_q^{(i)}(T_0 + T - T_0 + \Pi)\right| \\
&\quad \leq \left|c_p^{(i)}(T_0 + \Delta_{qp}) - c_q^{(i)}(T_0)\right| + \rho\,|T - T_0 + \Pi|
\end{aligned}$$

[by two applications of Lemma 2]

$\lesssim \epsilon + \rho\,|T - T_0|$ [by A2 and the hypothesis that $\rho\Pi$ is negligible]

The result now follows from the hypothesis that T is in $S^{(i)}$. ▮

Lemma 4: *If S1 holds for i, and processes p, q and r are nonfaulty up to time $T^{(i+2)}$, then for any T in $S^{(i)}$:*

$$\left|c_p^{(i)}(T) + \overline{\Delta}_{rp} - [c_q^{(i)}(T) + \overline{\Delta}_{rq}]\right| \lesssim 2(\epsilon + \rho S) \quad .$$

Proof: It follows from Lemma 1 that $|\Delta_{rp}|$ and $|\Delta_{qp}|$ are both less than Δ, so $\overline{\Delta}_{rp} = \Delta_{rp}$, $\overline{\Delta}_{rq} = \Delta_{rq}$, and $\rho\Delta_{rp}$ and $\rho\Delta_{rq}$ are both negligible. We therefore have:

$$\begin{aligned}
&\left|c_p^{(i)}(T) + \overline{\Delta}_{rp} - [c_q^{(i)}(T) + \overline{\Delta}_{rq}]\right| \\
&= \left|c_p^{(i)}(T) + \Delta_{rp} - [c_q^{(i)}(T) + \Delta_{rq}]\right| \\
&\approx \left|c_p^{(i)}(T + \Delta_{rp}) - c_q^{(i)}(T + \Delta_{rq})\right| && \text{[by Lemma 2]} \\
&\leq \left|c_p^{(i)}(T + \Delta_{rp}) - c_r^{(i)}(T)\right| + \left|c_r^{(i)}(T) - c_q^{(i)}(T + \Delta_{rq})\right| \\
&\lesssim 2(\epsilon + \rho S) && \text{[by Lemma 3]}
\end{aligned}$$

proving the result. ▮

Lemma 5: *If S1 holds for i, and Processes p and q are nonfaulty up to time $T^{(i+2)}$, then for any r and any T in $S^{(i)}$:*

$$\left|c_p^{(i)}(T) + \overline{\Delta}_{rp} - [c_q^{(i)}(T) + \overline{\Delta}_{rq}]\right| < \delta + 2\Delta \quad .$$

Proof: By the assumption that S1 holds for i, we have:

$$\left|c_p^{(i)}(T) - c_q^{(i)}(T)\right| < \delta \quad .$$

Since $\left|\overline{\Delta}_{rp}\right|$ and $\left|\overline{\Delta}_{rq}\right|$ are by definition no larger than Δ, the result follows immediately. ▮

We now complete the proof of the theorem. Assume that processes p and q are both nonfaulty until time $T^{(i+2)}$. For notational convenience, let T denote $T^{(i+1)}$. For any T' in $R^{(i+1)}$ we have:

$$\begin{aligned}
&\left|c_p^{(i+1)}(T') - c_q^{(i+1)}(T')\right| < \left|c_p^{(i+1)}(T) - c_q^{(i+1)}(T)\right| + \rho R && \text{[by A1]} \\
&= \left|c_p^{(i)}(T + \Delta_p) - c_q^{(i)}(T + \Delta_q)\right| + \rho R && \text{[from the algorithm]} \\
&\approx \left|c_p^{i}(T) + \Delta_p - [c_q^{(i)}(T) + \Delta_q]\right| + \rho R \\
& && \text{[by Lemma 2, since } |\Delta_p|, |\Delta_q| < \Delta] \\
&= \left|(1/n)\textstyle\sum_{r=1}^{n}(c_p^{(i)}(T) + \overline{\Delta}_{rp} - [c_q^{(i)}(T) + \overline{\Delta}_{rq}])\right| + \rho R \\
& && \text{[by definition of } \Delta_p \text{ and } \Delta_q]
\end{aligned}$$

$$\leq (1/n)\Sigma_{r=1}^{n}\left|c_p^{(i)}(T)+\overline{\Delta}_{rp}-[c_q^{(i)}(T)+\overline{\Delta}_{rq}]\right| + \rho R$$
$$\lesssim (1/n)[\,2(n-m)(\epsilon+\rho S) + m(\delta+2\Delta)\,] + \rho R \quad ,$$

where the last inequality is obtained by applying Lemma 4 to the $n-m$ nonfaulty processes r and Lemma 5 to the remaining m processes. Since $\Delta \approx \delta + \epsilon$, a little algebraic manipulation shows that if

$$\delta \gtrsim n'(2\epsilon + \rho(R+2S')) \quad ,$$

then:

$$\frac{1}{n}[\,2(n-m)(\epsilon+\rho S)+m(\delta+2\Delta)\,] + \rho R \lesssim \delta \quad .$$

Combining this with the above string of inequalities, we see that for any T' in $R^{(i+1)}$:

$$\left|c_p^{(i+1)}(T')-c_q^{(i+1)}(T')\right| \lesssim \delta \quad ,$$

so S1 holds for $i+1$. This completes the proof of Theorem I. ▮

For any clock synchronization algorithm, we will have $\delta \geq \delta_1 + \rho R$, where δ_1 is the closeness with which the clocks can be resynchronized and ρR is how far they can drift apart during an R SECOND interval. In the interactive convergence algorithm CNV(δ), there is a term $(n'-1)\rho R$ in δ_1, so how close the clocks can be resynchronized depends upon how far apart they are allowed to drift.

4. Interactive Consistency Algorithms

The obvious approach to fault-tolerant clock synchronization is to let each process read all the processes' clocks, and set its own clock to some function of the values that it read, such as the median. If a majority of the clocks are "good," then the median will be a good clock, so all the processes will synchronize to the same good clock. However, as the example of the "two-faced clock" shows, this assumes that any two nonfaulty processes get approximately the same value when reading any clock. In fact, it requires that the following two conditions hold for every process r:

1. Any two nonfaulty processes read approximately the same value for r's clock – even if r is faulty.
2. If r is nonfaulty, then every nonfaulty process reads approximately the correct value of its clock.

If we replace "reading process r's clock" by "obtaining General r's order," then these conditions are very similar to the approximate Byzantine Generals problem discussed in Lamport [17]. In fact, the two interactive consistency synchronization algorithms we describe are derived from the two algorithms for a completely connected network given in Lamport [17].

When p reads r's clock, what it actually finds is a constant $\overline{\Delta}_{rp}$ such that

$$c_r^{(i)}(T) \approx c_p^{(i)}(T+\overline{\Delta}_{rp}) \quad .$$

(The values $\overline{\Delta}_{rp}$ are not the same ones defined in the interactive convergence algorithm.) We also allow the possibility that if r is faulty, then p may not be able to read r's clock. This is denoted by letting $\overline{\Delta}_{rp}$ have the special value NULL. Recalling the definition of Δ_{qp} given by A2, and letting a NULL clock be approximately equal only to another NULL clock, we see that the above two informal conditions can be stated precisely as follows:

CC. For some constant $\Omega \ll R$ and all i: if the Clock Synchronization Condition holds for i, then for any processes p and q that are nonfaulty up to time $T^{(i+2)}$:

1. For all $r \neq p, q$: either
   (a) $\left|\overline{\Delta}_{rp} - [\Delta_{qp} + \overline{\Delta}_{rq}]\right| < \Omega$, or
   (b) $\overline{\Delta}_{rp} = \overline{\Delta}_{rq} = \text{NULL}$.
2. $\overline{\Delta}_{qp} \neq$ NULL , and $\left|\overline{\Delta}_{qp} - \Delta_{qp}\right| < \Omega$.

For convenience, we let $\overline{\Delta}_{pp} \equiv 0$ for all p. Condition CC2 is then equivalent to CC1(a) for $r = q$.

Before stating our next result, we introduce some notation. We let $\mathbf{N}$ denote the set $\{1, \ldots, n\}$. A *multiset* is a set in which the same element can appear more than once. We use ordinary set notation for describing multisets, so the multiset $\{1, 1, 2\}$ contains three elements, two of which are equal. The multiset $\{a_i : i \in \mathbf{N}\}$ contains n elements, not all of which need be distinct. If $\mathbf{M}$ is a multiset, then "median $\mathbf{M}$" denotes the median of $\mathbf{M}$, defined by

$$\text{median}\,\mathbf{M} \equiv a_{\lfloor n/2 \rfloor} \quad ,$$

where $\mathbf{M} = \{a_1, \ldots, a_n\}$ with $a_1 \leq a_2 \leq \ldots \leq a_n$.

Our two interactive consistency algorithms are based upon the following result.

Theorem II: *If $m \leq \lfloor n/2 \rfloor$, $\delta_0 < \Omega + \epsilon + \rho S$, and CC holds for all i, then letting*

$$c_p^{(i+1)}(T) \equiv c_p^{(i)}(T + \Delta_p)$$

where

$$\Delta_p \equiv \text{median}\{\overline{\Delta}_{rp} : r \in \mathbf{N} \textit{ and } \overline{\Delta}_{rp} \neq \textit{NULL}\} \quad ,$$

satisfies the Clock Synchronization Condition for all i, with

$$\begin{aligned} \delta &\approx \Omega + \epsilon + \rho(R + S) \\ \Sigma &\approx 2(\Omega + \epsilon) + \rho(R + S) \end{aligned}$$

The proof of Theorem II requires the following two results about medians.

Lemma 6: *If $|a_r - b_r| < \pi$ for all $r \in \mathbf{N}$, then:*

$$|\text{median}\{a_r : r \in \mathbf{N}\} - \text{median}\{b_r : r \in \mathbf{N}\}| < \pi \quad .$$

Proof: We prove the stronger result that for any k: the k^{th} highest values of the multisets $\{a_r\}$ and $\{b_r\}$ lie within π of one another. Let the permutations α and β be chosen such that:

$$a_{\alpha(1)} \leq a_{\alpha(2)} \leq \ldots \leq a_{\alpha(n)}$$
$$b_{\beta(1)} \leq b_{\beta(2)} \leq \ldots \leq b_{\beta(n)}$$

We prove that for all k: $\left|a_{\alpha(k)} - b_{\beta(k)}\right| < \pi$.

We consider the two cases $k = 1$ and $k > 1$. For $k = 1$ we have:

$$\begin{aligned} a_{\alpha(1)} &\leq a_{\beta(1)} && \text{[by definition of } \alpha\text{]} \\ &< b_{\beta(1)} + \pi && \text{[by hypothesis]} \end{aligned}$$

Symmetrical reasoning shows that

$$b_{\beta(1)} < a_{\alpha(1)} + \pi$$

and combining the two inequalities gives

$$\left|a_{\alpha(1)} - b_{\beta(1)}\right| < \pi$$

proving the result for $k = 1$.

For $k > 1$, consider the two multisets $A \equiv \{\alpha(1), \ldots, \alpha(k-1)\}$ and $B \equiv \{\beta(1), \ldots, \beta(k-1)\}$. If these two multisets are equal, then applying the result for $k = 1$ to the two multisets $\{a_{\alpha(r)} : r \geq k\}$ and $\{b_{\beta(r)} : r \geq k\}$ yields the desired result. We therefore assume that $A \neq B$.

Let $\alpha(k')$ be an element of A that is not in B, so $\alpha(k') = \beta(k'')$ for $k'' \geq k$. We have:

$$\begin{aligned} b_{\beta(k)} &\leq b_{\beta(k'')} && \text{[since } k \leq k''\text{]} \\ &= b_{\alpha(k')} \\ &< a_{\alpha(k')} + \pi && \text{[by hypothesis]} \\ &\leq a_{\alpha(k)} + \pi && \text{[since } k' < k\text{]} \end{aligned}$$

Symmetrical reasoning shows that

$$a_{\alpha(k)} < b_{\beta(k)} + \pi$$

and combining these two inequalities gives the desired result. ▮

Lemma 7: *If* $|a_r - a| < \pi$ *for a majority of values* r *in* $\mathbf{N}$, *then* $|\text{median}\{a_r : r \in \mathbf{N}\} - a| < \pi$.

Proof: It is easy to see that if A is any submultiset containing a majority of the elements of $\{a_r\}$, then:

$$\min(A) \leq \text{median}\{a_r : r \in \mathbf{N}\} \leq \max(A) \quad .$$

Letting A be the multiset $\{a_r : |a_r - a| < \pi\}$, this implies that

$$a - \pi \leq \text{median}\{a_r\} \leq a + \pi,$$

which proves the lemma. ▮

Proof of Theorem II: The proof is by induction on i. For $i = 0$, the result is trivial. (Note that S2 is vacuous for $i = 0$.) Assume that the theorem is true for i. By CC2 and Lemma 1, we see that

$$\left|\overline{\Delta}_{qp}\right| < \Omega + \delta + \epsilon$$

for all nonfaulty p and q, so S2 follows easily from Lemma 7.

Since Δ_p is the median of the $\overline{\Delta}_{rp}$, and a majority of the processes r are nonfaulty, the above inequality shows that we can neglect terms of order $\rho\Delta_p$, and likewise terms of order $\rho\Delta_q$. Lemma 1 implies that we can neglect terms of order $\rho\Delta_{qp}$. Letting $T = T^{(i+1)}$, we then have:

$$\begin{aligned}
&\left|c_p^{(i+1)}(T) - c_q^{(i+1)}(T)\right| \\
&\quad = \left|c_p^{(i)}(T+\Delta_p) - c_q^{(i)}(T+\Delta_q)\right| && \text{[by hypothesis]} \\
&\quad = \left|c_p^{(i)}(T+\Delta_{qp}+\Delta_p-\Delta_{qp}) - c_q^{(i)}(T+\Delta_q)\right| \\
&\quad \approx \left|c_p^{(i)}(T+\Delta_{qp}) + \Delta_p - \Delta_{qp} - [c_q^{(i)}(T)+\Delta_q]\right| && \text{[by Lemma 2]} \\
&\quad \le |\Delta_p - \Delta_{qp} - \Delta_q| + \epsilon + \rho S && \text{[by Lemma 3]} \\
&\quad = \left|\text{median}\{\overline{\Delta}_{rp} - \Delta_{qp} : r \in \mathbf{N} \text{ and } \overline{\Delta}_{rp} \neq \text{NULL}\}\right. \\
&\qquad\quad \left. - \text{median}\{\overline{\Delta}_{rq} : r \in \mathbf{N} \text{ and } \overline{\Delta}_{rq} \neq \text{NULL}\}\right| + \epsilon + \rho S \\
&\quad \lesssim \Omega + \epsilon + \rho S && \text{[by CC and Lemma 6]}
\end{aligned}$$

Condition S1 follows easily from this inequality and A1. ▮

The Algorithm COM

Achieving Condition CC requires that the processes not only read each other's clocks, but also send values to one another. If x_p is a value that process p sends to the other processes, then we let x_{pq} denote the value that q receives from p. The manner in which the value is transmitted is irrelevant - the value might be sent as a message from p to q, or p might leave it in some register where q can read it. We assume that if p and q are both nonfaulty, then $x_{pq} = x_p$. If p or q is faulty, then x_{pq} may have any value.

Our first interactive consistency algorithm is based upon the following algorithm by which process q obtains a value $\text{COM}(m, \mathbf{P}, x_p, p)_q$ from process p. It is essentially the same as Algorithm OM(m) of Lamport [17], except that whenever a process r receives a value from process s, it adds Δ_{sr} to that value before using it or relaying it to other processes. This is done because a value held by process r represents the difference between r's clock and p's clock. The algorithm for obtaining the value $\text{COM}(m, \mathbf{P}, x_p, p)_q$ is defined inductively as follows, where we write $\mathbf{P} - p$ to denote $\mathbf{P} - \{p\}$, for any set $\mathbf{P}$.

Algorithm COM: *For any integer* $m \geq 0$*, any subset* $\mathbf{P}$ *of* $\mathbf{N}$*, any value* x_p *and any*

$p, q \in \mathbf{P}$:

$$\begin{aligned} \mathrm{COM}(0, \mathbf{P}, x_p, p)_q &\equiv x_{pq} + \Delta_{pq} \\ \mathrm{COM}(m, \mathbf{P}, x_p, p)_q &\equiv \mathrm{median}\{\mathrm{COM}(m-1, \mathbf{P}-p, x_{pr} + \Delta_{pr}, r)_q : r \in \mathbf{P}-p\} \end{aligned}$$

To prove the required properties of Algorithm COM, we need the following result.

Lemma 8: *If S1 holds for i, and processes p, q and r are nonfaulty up to time $T^{(i+2)}$, then:*

$$|\Delta_{pr} + \Delta_{rq} - \Delta_{pq}| \lesssim 3\epsilon + 2\rho S \quad .$$

Proof: Let T be the time, obtained from A2, such that

$$\left|c_p^{(i)}(T) - c_q^{(i)}(T + \Delta_{pq})\right| < \epsilon \quad .$$

We then have:

$$\begin{aligned} &|\Delta_{pr} + \Delta_{rq} - \Delta_{pq}| \\ &\quad \approx \left|c_q^{(i)}(T + \Delta_{pr} + \Delta_{rq}) - c_q^{(i)}(T + \Delta_{pq})\right| && \text{[by Lemma 2]} \\ &\quad \le \left|c_q^{(i)}(T + \Delta_{pr} + \Delta_{rq}) - c_r^{(i)}(T + \Delta_{pr})\right| \\ &\qquad + \left|c_r^{(i)}(T + \Delta_{pr}) - c_p^{(i)}(T)\right| + \left|c_p^{(i)}(T) - c_q^{(i)}(T + \Delta_{pq})\right| \\ &\quad \lesssim 3\epsilon + 2\rho S && \text{[by A2 and Lemma 3]} \end{aligned}$$

which is the required result. ∎

The following result is the analogue of Lemma 1 of Lamport [18].

Lemma 9: *For all m and k, if $n > 2k+m$, S1 holds for i, and all but at most k processes in* **P** *are nonfaulty up to time $T^{(i+2)}$, then for any of those nonfaulty processes p and q, and any x_p:*

$$|\mathrm{COM}(m, \mathbf{P}, x_p, p)_q - [x_p + \Delta_{pq}]| \lesssim m(3\epsilon + 2\rho S) \quad .$$

Proof: The proof is by induction on m. The result is trivial for $m = 0$. Assume it for $m - 1$. For any process r in $\mathbf{P} - p$ that is nonfaulty up to time $T^{(i+2)}$, we have:

$$\begin{aligned} &|\mathrm{COM}(m-1, \mathbf{P}-p, x_{pr} + \Delta_{pr}, r)_q - [x_p + \Delta_{pq}]| \\ &\quad = |\mathrm{COM}(m-1, \mathbf{P}-p, x_{pr} + \Delta_{pr}, r)_q - [x_{pr} + \Delta_{pr} + \Delta_{rq}] \\ &\qquad + [\Delta_{pr} + \Delta_{rq} - \Delta_{pq}]| && \text{[since } p \text{ and } r \text{ nonfaulty implies } x_{pr} = x_p] \\ &\quad \lesssim (m-1)(3\epsilon + 2\rho S) + (3\epsilon + 2\rho S) \end{aligned}$$

where the last inequality comes from Lemma 8 and the induction hypothesis, which can be applied because $\mathbf{P} - p$ has $n - 1$ elements and $n - 1 > 2k + (m - 1)$. Therefore, for every nonfaulty process r, we have

$$|\mathrm{COM}(m-1, \mathbf{P}-p, x_{pr} + \Delta_{pr}, r)_q - [x_p + \Delta_{pq}]| \lesssim m(3\epsilon + 2\rho S) \quad .$$

The lemma now follows from Lemma 7, since $n > 2k + m \geq 2k + 1$. ∎

Our next lemma is the analogue of Theorem 1 of Lamport [17].

Lemma 10: *If S1 holds for i, and* **P** *is a set containing more than 3m processes, all but at most m of which are nonfaulty up to time $T^{(i+2)}$, then for any of the nonfaulty processes p and q:*

1. *For all r in* **P**:
$|\mathrm{COM}(m, \mathbf{P}, x_r, r)_p - [\Delta_{qp} + \mathrm{COM}(m, \mathbf{P}, x_r, r)_q]| \lesssim (2m+1)(3\epsilon + 2\rho S)$
2. $|\mathrm{COM}(m, \mathbf{P}, x_q, q)_p - x_q - \Delta_{qp}| \lesssim m(3\epsilon + 2\rho S)$.

Proof: Part 2 follows immediately from Lemma 9 by letting $k = m$. Part 1 is proved by induction. For $m = 0$, it follows easily from the definition of COM and Lemma 8. Let $m > 0$ and assume it holds for $m - 1$. We consider two cases: (i) r faulty and (ii) r nonfaulty.

If r is faulty, then there are at most $m - 1$ faulty processes in $\mathbf{P} - r$, and we can apply the induction hypothesis to obtain:

$$|\mathrm{COM}(m-1, \mathbf{P} - r, x_{rs}, s)_p - \Delta_{qp} - \mathrm{COM}(m-1, \mathbf{P} - r, x_{rs}, s)_q| \lesssim (2(m-1)+1)(3\epsilon + 2\rho S)$$

for any s in $\mathbf{P} - r$. The result now follows easily from Lemma 6.

If r is nonfaulty, then we can apply the inequality from part 2 to obtain:

$$\begin{aligned} |\mathrm{COM}(m, \mathbf{P}, x_r, r)_p - [x_r + \Delta_{rp}]| &\lesssim m(3\epsilon + 2\rho S) \\ |\mathrm{COM}(m, \mathbf{P}, x_r, r)_q - [x_r + \Delta_{rq}]| &\lesssim m(3\epsilon + 2\rho S) \end{aligned} \tag{3}$$

We then have:

$$\begin{aligned} &|\mathrm{COM}(m, \mathbf{P}, x_r, r)_p - \Delta_{qp} - \mathrm{COM}(m, \mathbf{P}, x_r, r)_q| \\ &\quad = |\mathrm{COM}(m, \mathbf{P}, x_r, r)_p - [x_r + \Delta_{rp}] - (\mathrm{COM}(m, \mathbf{P}, x_r, r)_q - [x_r + \Delta_{rq}]) \\ &\qquad + \Delta_{rp} - \Delta_{rq} - \Delta_{qp}| \\ &\quad \lesssim 2m(3\epsilon + 2\rho S) + (3\epsilon + 2\rho S) \end{aligned}$$

where the last inequality follows from the triangle inequality, (3) and Lemma 8. This finishes the proof of part 1 for m. ∎

Taking $x_r = 0$, Lemma 10 yields the following result.

Theorem III: *If all but at most m processes are nonfaulty up to time $T^{(i+2)}$, and $n > 3m$, then Condition CC is satisfied by*

$$\overline{\Delta}_{qp} = \mathrm{COM}(m, \mathbf{N}, 0, q)_p \quad ,$$

with $\Omega \approx (2m+1)(3\epsilon + 2\rho S)$.

Combining this with Theorem II yields our first interactive consistency clock synchronization algorithm, with

$$\begin{aligned} \delta &\approx (6m+4)\epsilon + (4m+3)\rho S + \rho R \\ \Sigma &\approx (12m+8)\epsilon + (8m+5)\rho S + \rho R \end{aligned}$$

This algorithm requires that n be greater than $3m$–i.e., that more than two-thirds of the processes be nonfaulty. It can be shown that given an algorithm to achieve Condition CC, one can construct a solution to the approximate Byzantine Generals Problem described in Lamport [17]. This means that any interactive consistency algorithm that does not use digital signatures cannot work unless more than two-thirds of the processes are nonfaulty.

The Algorithm CSM

Our second interactive consistency algorithm is derived from algorithm SM of Lamport [17]. Each process p sends a message to every other process stating that its clock has reached a certain value T_p, and these messages are relayed among the other processes. To use such messages for synchronizing clocks, the message transmission delay must be known. We therefore make the following assumption, which replaces A2.

A2′. If an event in a process q occurring at (real) time t_0 causes q to send a message to a process p, then that message arrives at a time t_1 such that:

(a) if p and q are nonfaulty, then:

$$|t_1 - t_0 - \gamma| < \epsilon \quad .$$

(b) $t_1 - t_0 > \gamma - \epsilon$

for some constant γ such that $n\rho\gamma$ is negligibly small.

Assumption A2′(a) states that the time taken by a nonfaulty process to generate a message and transmit it to another nonfaulty process equals $\gamma \pm \epsilon$. Part (b) states that a faulty process cannot generate and transmit a message in any less time, but it can take longer. In practice, the value of γ may depend upon p and q and on the type of event generating the message. To avoid having to cope with all these different values, we assume a single γ for all messages.

As with Algorithm SM of Lamport [17], we need to send unforgeable signed messages, so we must assume a method for generating digital signatures. A digital signature mechanism consists of a function S_p for each process p, satisfying the following condition:

A3. For any process p and any data item D:

(a) No faulty process other than p can generate $S_p[D]$.

(b) For any X, any process can determine if X equals $S_p[D]$.

We refer the reader to Lamport [17] for a discussion of how digital signatures are constructed. Note that the first assumption is stronger than the one made in Lamport [17], since it does not permit one faulty process to forge the signatures of another faulty process.

Assumption A3(a) means that a faulty process cannot generate any arbitrary value, so it restricts the class of faults that may occur. Hence, we can hope to find a clock synchronization algorithm to handle m faults with fewer than $3m + 1$ processes, and, indeed, our second interactive consistency algorithm requires that only a majority of the processes be nonfaulty.

We define the message $M(T, p_0 \dots p_s)$, for any sequence $p_0, \dots, p_s$ of processes – including the null sequence λ – as follows:

$$\begin{array}{lcl} M(T,\lambda) & \equiv & (T, p_0, S_{p_0}(T)) \\ M(T, p_0 \ldots p_s) & \equiv & (M(T, p_0 \ldots p_{s-1}), p_s, S_{p_s}[M(T, p_0 \ldots p_{s-1})]) \end{array}$$

By A3(a), the value $M(T, p_0 \ldots p_s)$ can only be generated as the result of process p_0 sending the message $M(T, p_0)$ to process p_1, which sends the message $M(T, p_0 p_1)$ to process $p_2 \ldots$ which sends the message $M(T, p_0 \ldots p_{s-1})$ to process p_s, which generates $M(T, p_0 \ldots p_s)$. Moreover, A3(b) implies that any process can determine whether a given data item X equals $M(T, p_0 \ldots p_s)$ for some T and $p_0 \ldots p_s$.

In Algorithm CSM, for some time T_p in $S^{(i)}$, process p sends the message $M(T_p, p)$ to all other processes when its clock reaches T_p. Immediately upon receiving this message, each other process p_1 sends the message $M(T_p, pp_1)$ to all processes other than itself and p, and so forth. Any process q will therefore receive messages $M(T_p, pp_1 \ldots p_s)$ for many different sequences $p_1 \ldots p_s$. Each such message tells q that p's clock read T_p approximately $(s+1)\gamma$ SECONDS ago. If p and all the p_i are nonfaulty, then this message is correct. If one or more of the p_i are faulty, then they can either fail to relay the message, so q never receives it, or they can delay it. However, they cannot alter the value of T_p or cause the message to arrive too early. Hence, process q believes the message indicating the earliest time at which p's clock reached T_p.

There is a practical problem in implementing this approach. In order to perform the appropriate message relaying, a process must be prepared to receive the incoming message. This may require that the process not do anything else while waiting, so it should know when the message will arrive and be able to ignore the message if it does not arrive when it should. Since the uncertainty in message transmission time is ϵ, and the difference between p's clock and q's clock is δ, q can expect to receive the message $M(T_p, p)$ within about $\epsilon + \delta$ seconds of when its clock reads $T_p + \gamma$. If q only relays this message if it arrives when it should, then another process r can expect to receive the message $M(T_p, pq)$ within about $2(\epsilon + \delta)$ of when its clock reads $T_p + 2\gamma$. Continuing, this leads us to the following definition:

Definition 3: *The message $M(T, p_0 \ldots p_s)$ is said to arrive* on time *at Process q if either*

1. *$s \geq 0$ and the message arrives at (real) time $c_q^{(i)}(T')$, or*
2. *$s = -1$ (so $p_0 \ldots p_s$ is the null sequence) and $T' = T$*

and $|T' - T - (s+1)\gamma| \leq (s+1)(\delta + \epsilon)$.

The δ in this definition is the same one as in the Clock Synchronization Condition. Its value will be given later.

The following algorithm describes how each process q determines the value $\overline{\Delta}_{pq}$ for every $p \neq q$.

Algorithm CSM(m): *For each process p, and for some clock time T_p in $S^{(i)}$:*

1. *When its clock $c_p^{(i)}$ reaches T_p, process p sends the message $M(T_p, p)$ to every other process.*
2. *For each process $q \neq p$:*

   A. *Process q initializes $\overline{\Delta}_{pq}$ to ∞.*

B. *If the message $M(T_p, pp_1 \dots p_s)$ arrives on time at q, at time $c_q^{(i)}(T)$, and $T - T_p - (s+1)\gamma < \overline{\Delta}_{pq}$, then:*

(a) *Process q sets $\overline{\Delta}_{pq}$ equal to $T - T_p - (s+1)\gamma$.*

(b) *If $s < m$, then immediately upon receiving this message, q sends the message $M(T_p, pp_0 \dots p_s q)$ to every other process q' not contained among the processes $pp_0 \dots p_s$.*

C. *At time $T_p + (m+1)(\gamma + \delta + \epsilon)$, if $\overline{\Delta}_{pq} = \infty$, then p sets $\overline{\Delta}_{pq}$ equal to* NULL.

We have assumed A2′ instead of A2, so the values Δ_{qp} are not yet defined. In order to apply Theorem II, we must define the Δ_{qp} and prove A2.

Lemma 11: *Assumption A2 is satisfied, except with the strict inequality replaced by approximate inequality, if Δ_{qp} is defined to equal $T - T_q - \gamma$, where T is the value such that p receives the message $M(T_q, q)$ at time $c_p^{(i)}(T)$, and to have any value if p receives no such message.*

Proof: If p and q are nonfaulty, then the message $M(T_q, q)$ is sent by q and received by p. Taking $T_0 = T_q$, we find

$$\begin{aligned} &\left|c_p^{(i)}(T_0 + \Delta_{qp}) - c_q^{(i)}(T_0)\right| \\ &\quad = \left|c_p^{(i)}(T - \gamma) - c_q^{(i)}(T_q)\right| \\ &\quad \approx \left|c_p^{(i)}(T) - \gamma - c_q^{(i)}(T_q)\right| && \text{[by Lemma 2]} \\ &\quad < \epsilon && \text{[by A2′(a)]} \end{aligned}$$

which proves the lemma. ∎

Lemma 11 allows us to use the earlier results that assumed A2. (Since these results all involve approximate inequalities, they are not invalidated when the exact inequality in A2 is replaced by an approximate inequality.) We now prove the main result for Algorithm CSM.

Theorem IV: *If all but at most m processes are nonfaulty up to time $T^{(i+2)}$, then the values $\overline{\Delta}_{pq}$ found by algorithm* CSM(m) *satisfy condition CC with $\Omega \approx (m+5)\epsilon + 2\rho S$.*

The proof uses the following lemmas.

Lemma 12: *Let S1 hold for i, and let p and q be nonfaulty up to time $T^{(i+2)}$. If the message $M(T, p_0 \dots p_s)$ arrives on time at p, and $s < m$, then the message $M(T, p_0 \dots p_s p)$ arrives on time at q.*

Proof: Let $c_p^{(i)}(T')$ be the time at which $M(T, p_0 \dots p_s)$ arrives at p, letting $T' = T$ if $p_0 \dots p_s$ is the null sequence, and let $c_q^{(i)}(T'')$ be the time at which $M(T, p_0 \dots p_s p)$ arrives at q. Then

$$\begin{aligned} &\left|c_q^{(i)}(T'') - c_p^{(i)}(T' + \gamma)\right| \\ &\quad \approx \left|c_q^{(i)}(T'') - c_p^{(i)}(T') - \gamma\right| \end{aligned}$$

$$\leq \left|c_q^{(i)}(T'') - c_p^{(i)}(T') - \gamma\right| + \delta \qquad \text{[by S1]}$$
$$\lesssim \epsilon + \delta \qquad \text{[by A2]}$$

It follows from A1, and the assumption that $\rho\gamma$ is negligible, that:

$$|T'' - T' - \gamma| \lesssim \delta + \epsilon \quad . \tag{4}$$

We then have:

$$\begin{aligned} &|T'' - T - (s+2)\gamma)| \\ &\quad \leq |T'' - T' - \gamma| + |T' - T - (s+1)\gamma| \\ &\quad \lesssim \delta + \epsilon + (s+1)(\delta + \epsilon) \end{aligned}$$

[by (4) and the on-time arrival of $M(T, p_0 \ldots p_s)$]

which implies that $M(T, p_0 \ldots p_s p)$ arrives on time at q. ∎

Lemma 13: *If S1 holds for i, all but at most m processes are nonfaulty up to $T^{(i+2)}$, and p and q are among the nonfaulty ones, then for any process r: if $\overline{\Delta}_{rp} \neq$ NULL then $\overline{\Delta}_{rq} \neq$ NULL and*

$$\overline{\Delta}_{rq} + \Delta_{qp} \lesssim \overline{\Delta}_{rp} + (m+3)\epsilon + \rho S \quad .$$

(For $r = q$ or p, $\overline{\Delta}_{rr}$ is defined to be 0.)

Proof: Let $r_0 = r$, and let T be the time such that

$$\overline{\Delta}_{rp} = T - T_r - (s+1)\gamma \tag{5}$$

and the message $M(T_r, r_0 \ldots r_s)$ arrived at p (on time) at time $c_p^{(i)}(T)$. (If $r = p$, so $s = -1$, then $T = T_r$.) We consider two cases:

1. $q = r_j$
2. q not in the sequence $r_0 \ldots r_s$.

In case 1, it follows from A3(a) that the message $M(T_r, r_0 \ldots r_{j-1})$ must have arrived on time at q at some time $c_q^{(i)}(T')$, so $\overline{\Delta}_{rq} \neq$ NULL, and

$$\overline{\Delta}_{rq} \leq T' - T_r - j\gamma \quad . \tag{6}$$

A simple induction argument using A2′(b) shows that

$$c_p^{(i)}(T) - c_q^{(i)}(T') - (s-j+1)\gamma > -(s-j+1)\epsilon \quad . \tag{7}$$

We then have:

$$\begin{aligned} &c_p^{(i)}(T) - c_p^{(i)}(T' + (s-j+1)\gamma) \\ &\quad \gtrsim c_p^{(i)}(T) - c_q^{(i)}(T' + (s-j+1)\gamma - \Delta_{qp}) - \epsilon - \rho S && \text{[by Lemma 3]} \\ &\quad \approx c_p^{(i)}(T) - c_q^{(i)}(T') - (s-j+1)\gamma + \Delta_{qp} - \epsilon - \rho S && \text{[by Lemma 2]} \\ &\quad > -(s-j+2)\epsilon - \rho S + \Delta_{qp} && \text{[by (7)]} \end{aligned}$$

By A1, this yields:

$$T-(T'+(s-j+1)\gamma) \gtrsim -(s-j+2)\epsilon-\rho S+\Delta_{qp} \quad ,$$

so

$$T'-j\gamma+\Delta_{qp} \lesssim T-(s+1)\gamma+(s-j+2)\epsilon+\rho S \quad .$$

Subtracting T_r from both sides of this inequality, we see that (5) and (6) imply:

$$\overline{\Delta}_{rq}+\Delta_{qp} \lesssim \overline{\Delta}_{rp}+(s-j+2)\epsilon+\rho S \quad ,$$

which yields the desired result, since $s \leq m$.

For case 2, q not equal to any of the r_j, we consider two subcases: $s < m$ and $s = m$. If $s < m$, then p sends q the message $M(T_r, r_0 \ldots r_s p)$, which, by Lemma 12, arrives on time at q. Hence, $\overline{\Delta}_{rq} \neq$ NULL and

$$\overline{\Delta}_{rq} \leq T'-T_r-(s+2)\gamma \tag{8}$$

where $c_q^{(i)}(T')$ is the time at which the message arrives. We then have:

$$\begin{aligned} & c_p^{(i)}(T'+\Delta_{qp}-\gamma)-c_p^{(i)}(T) \\ & \approx c_p^{(i)}(T'+\Delta_{qp})-c_p^{(i)}(T)-\gamma && \text{[by A1]} \\ & \lesssim c_q^{(i)}(T')-c_p^{(i)}(T)-\gamma+\epsilon+\rho S && \text{[by Lemma 3]} \\ & < 2\epsilon+\rho S && \text{[by A2}'\text{(a)]} \end{aligned}$$

This implies that

$$T'+\Delta_{qp}-\gamma-T \lesssim 2\epsilon+\rho S$$

which can be rewritten as

$$T'-(s+2)\gamma+\Delta_{qp} \lesssim T-(s+1)\gamma+2\epsilon+\rho S \quad .$$

Subtracting T_r from both sides of this inequality shows that the desired result follows immediately from (5) and (8).

Finally, we consider the case $s = m$, where q is not one of the r_j. Since there are at most m faulty processes, there is at least one process r_j that is nonfaulty up to time $T^{(i+2)}$. Let $c_q^{(i)}(T')$ be the time at which r_j received the message $M(T_r, r_0 \ldots r_{j-1})$–or at which it sent the message $M(T_r, r)$, if $j = 0$. The same argument as before shows that (7) again holds. By Lemma 12, the message $M(T_r, r_0 \ldots r_j)$ arrives on time at q, so $\overline{\Delta}_{rq} \neq$ NULL, and

$$\overline{\Delta}_{rq} \leq T''-T_r-(j+1)\gamma \quad , \tag{9}$$

where $c_q^{(i)}(T'')$ is the time at which the message arrives. By A2′(a) we have

$$c_q^{(i)}(T'')-c_q^{(i)}(T')-\gamma < \epsilon$$

and combining this with (7) yields:

$$c_p^{(i)}(T) - c_q^{(i)}(T'') - (s-j)\gamma \ > \ -(s-j+2)\epsilon \quad .$$

Using Lemma 3, we can deduce from this that

$$c_p^{(i)}(T) - c_p^{(i)}(T'' + \Delta_{qp}) - (s-j)\gamma \ \gtrsim \ -(s-j+3)\epsilon - \rho S \quad .$$

Since $\rho(s-j)\gamma$ is negligible, this implies by A1 that

$$T - T'' - \Delta_{qp} - (s-j)\gamma \ \gtrsim \ -(s-j+3)\epsilon - \rho S$$

which can be rewritten as

$$T'' - (j+1)\gamma + \Delta_{qp} \ \lesssim \ T - (s+1)\gamma + (s-j+3)\epsilon + \rho S \quad .$$

Subtracting T_r from both sides, we obtain the desired result from (9) and (5). ▮

Proof of Theorem IV: Let p, q be as in Condition CC. It follows easily from Lemma 12 that $\overline{\Delta}_{qp} \neq$ NULL. Condition CC2 then follows from CC1(a) for $r = q$. It therefore suffices to prove CC1 for all r. This requires proving that if $\overline{\Delta}_{rp} \neq$ NULL, then $\overline{\Delta}_{rq} \neq$ NULL and

$$\begin{aligned} \overline{\Delta}_{rp} - \Delta_{qp} - \overline{\Delta}_{rq} \ &\lesssim (m+5)\epsilon + 2\rho S \\ \overline{\Delta}_{rq} + \Delta_{qp} - \overline{\Delta}_{rp} \ &\lesssim (m+5)\epsilon + 2\rho S \end{aligned}$$

The fact that $\overline{\Delta}_{rq} \neq$ NULL and the second inequality follow from Lemma 13. Reversing p and q in Lemma 13, we obtain,

$$\overline{\Delta}_{rp} + \Delta_{pq} - \overline{\Delta}_{rq} \ \lesssim \ (m+3)\epsilon + \rho S \quad .$$

To prove the theorem, we therefore need only show that

$$|\Delta_{pq} + \Delta_{qp}| \ \lesssim \ 2\epsilon + \rho S \quad .$$

We write

$$\begin{aligned} &\left|c_p^{(i)}(T_0 + \Delta_{qp}) - c_p^{(i)}(T_0 - \Delta_{pq})\right| \\ &\quad \leq \left|c_p^{(i)}(T_0 + \Delta_{qp}) - c_q^{(i)}(T_0)\right| + \left|c_q^{(i)}(T_0) - c_p^{(i)}(T_0 - \Delta_{pq})\right| \\ &\quad \lesssim \epsilon + \epsilon + \rho S \qquad\qquad \text{[by A2 and Lemma 3]} \end{aligned}$$

where T_0 is as in A2, and the result follows from A1. ▮

Combining Theorem IV with Theorem II gives an interactive consistency algorithm with

$$\begin{aligned} \delta \ &\approx \ (m+6)\epsilon + 3\rho S + \rho R \\ \Sigma \ &\approx \ (2m+12)\epsilon + 5\rho S + \rho R \end{aligned}$$

This is the value of δ that should be used in the definition of on-time arrival.

5. Conclusion

We have described three clock-synchronization algorithms. The interactive convergence algorithm CNV is the simplest, requiring only that every process read every other process's clock. The interactive consistency algorithms require a great deal of message passing – Algorithm COM generates approximately n^{m+1} messages, while Algorithm CSM generates at most that many, but on the average generates approximately $(n/2)^{m+1}$ messages. Algorithms CNV and COM require that more than two-thirds of the processes be nonfaulty, while Algorithm CSM requires only that a majority be nonfaulty. However, Algorithm CSM assumes a digital signature mechanism and a known message transmission delay.

To compare the closeness of synchronization achieved by these algorithms, we assume that $\rho S \ll \epsilon$. This is a reasonable assumption, since, for most practical applications, ϵ will be on the order of microseconds, S, at most a few milliseconds, and $\rho \lesssim 10^{-6}$. To simplify comparisons with the interactive convergence algorithm, in which δ depends upon n, we assume that $n = 3m + 1$. This will be the case if the only reason for having multiple processes is to achieve fault-tolerance through redundancy. We then get the following values of δ:

Algorithm CNV: $(6m + 2)\epsilon + (3m + 1)\rho R$

Algorithm COM: $(6m + 4)\epsilon + \rho R$

Algorithm SCM: $(m + 6)\epsilon + \rho R$

This shows that Algorithm SCM achieves the closest synchronization, especially for $m > 1$. If R is small enough – i.e., if the clocks are resynchronized often enough – then Algorithm CNV can achieve slightly better synchronization than Algorithm COM. However, one usually wants to resynchronize only as often as is necessary to achieve a desired value of δ. If this value of δ is much larger than $6m\epsilon$, then it is necessary to synchronize $3m + 1$ times more often with Algorithm CNV than with the interactive consistency algorithms. For example, if $m = 2$, $\epsilon = 2$ microseconds, $\rho = 10^{-6}$, and $\delta = 50$ microseconds, then we obtain the following resynchronization intervals R:

Algorithm CNV:	3.1 Seconds
Algorithm COM:	18 Seconds
Algorithm SCM:	34 Seconds.

We suspect that, in most applications, Algorithm CNV will provide sufficiently short resynchronization times. Its more serious drawback will be that it requires two-thirds of the processes to be nonfaulty in order to guarantee synchronization. When this is not satisfactory – for example, when using three-fold redundancy to protect against a single fault – Algorithm SCM must be used.

We have assumed a system in which each process can communicate with all the others, and have considered only process failure, not communication failure. Our interactive consistency algorithms can be generalized to incompletely connected networks of processes. In the same way that Algorithm COM was derived from Algorithm OM(m) of Lamport [17], Algorithm OM(m, p) of Lamport [17] and the algorithm of Dolev [3] can be used to obtain clock synchronization algorithms for incompletely connected networks. It can be shown, that for a network of diameter d, Algorithm CSM(m) satisfies Theorem IV, except with the maximum number of faulty processes reduced to $m - d + 1$.

In algorithms not using digital signatures, the failure of a communication line joining two processes must be considered a failure of one of the two processes. Indeed, a two-faced clock is perhaps more likely to be caused by communication failure than by failure of the clock itself. For Algorithm CSM, assuming that a faulty communication line cannot "forge" properly signed messages, a faulty communication line is equivalent to a missing one. Hence, Algorithm CSM($m+d-1$) can handle up to m process faults plus any number of communication line failures, so long as the remaining network of nonfaulty processes and communication lines has diameter at most d.

Appendix C. Summary of the SPECIAL Specification Language

1. Introduction

This Appendix presents a brief tutorial on the SPECIAL specification language. A more detailed description may be found in Weinstock [30]. Isner [14] gives a very readable tutorial to data abstraction and specification, using a SPECIAL-like notation, and Robinson, et al., [26] presents a substantial example of an operating system specification in SPECIAL.

2. General Characteristics

SPECIAL permits a computer program or system to be described in terms of operations on externally visible state variables, called "items" or "objects". The operations are defined as state transitions, which are described as functions over well-defined sets of values of the objects. The specifications may be hierarchical, so a given set of states and state transitions may represent a large number of states and state transitions that are invisible at that level. A specification may be looked upon as the instruction set of an abstract machine, and a specification hierarchy may be looked upon as a set of interface descriptions for a hierarchy of abstract machines, in which lower level machines implement the state transitions defined at higher levels.

A specification of the kind produced by SPECIAL serves to hide design details that are not essential to an understanding of the system's intended behavior. SPECIAL is thus not a programming language (e.g., successive statements are not "executed" and have no sequentiality) and should not be used as such.

The heart of a SPECIAL specification consists of collections ("modules") of functions of various kinds (called V-, O-, and OV-functions). These functions are defined in terms of effects on objects, and the effects are stated as mathematical expressions.

SPECIAL is a "typed" language in that a type (a well-defined set of values) is associated with each item when declared, thus permitting subsequent appearances of the items in a specification to be checked for consistency with their declared type. The type INTEGER* (a primitive type of SPECIAL) has, as values, all of the integers--positive and negative (including zero). The type BOOLEAN (also a primitive type of SPECIAL) has, as values, TRUE and FALSE. Although not needed for this example, there are additional primitive types. New types, e.g., sets, vectors, structures (records), subtypes, may also be constructed out of existing types.

*All reserved words are in capital letters.

One or more types noted as *designator types* can be associated with a module. The values of these types, called *designators*, serve as names for abstract objects of the module. The interface description of a module lists all of its designator types. For example, the "stacks" module interface description, illustrated in Figure C-1, declares the designator type "stack_name" (an abbreviation for name-of-stack).

Following the designator types, the interface description lists the module's parameters. A parameter of a module is a symbolic constant which, upon initialization of the module, acquires a value that is not subsequently changed by any operation invocation. The parameter mechanism enables a module specification to have some generality. Often a module can appear in different machines in the hierarchy, with a different value for the parameters. Another reason for leaving the values of parameters unbound at specification time is that they are often dependent on the values of lower-level parameters, in a manner that is not decided until later stages. The "stacks" module has the single integer-value parameter "max_stack_size", whose value is the maximum number of elements that can be in a stack. The reader should observe that we have made the decision for this example that all stacks of the module are of the same fixed size.

The interface description for a module is a list of its operations. Depending on whether its invocation returns a value and/or causes a state change, an operation is declared to be one of the following three kinds:

- V-function (VFUN) -- returns a value, but causes no state change.
- O-function (OFUN) -- causes a state change, but does not return a value.
- OV-function (OVFUN) -- returns a value and causes a state change.

The "stacks" module has two operations:

- "push" -- causes an integer v to be placed on top of stack s. *
- "pop" -- causes the integer value v on the top of the stack s to be removed and returned.

The reader should note that the decision to provide integer stacks is manifested by declaring the second argument of "push" and the returned value of "pop" to be of type INTEGER. The remaining sections describe the detailed features of the language.

*Hereafter we will refer to "stack s" as a shorthand for "the stack that corresponds to the designator s".

```
MODULE stacks  $( maintains a fixed number of stacks of integers,
            each of the same fixed maximum size)

   TYPES

stack_name: DESIGNATOR $( names for stacks) ;

   PARAMETERS

INTEGER max_stack_size $( maximum size for a given stack) ;


   FUNCTIONS

VFUN ptr(stack_name s) -> INTEGER i;  $( stack pointer, or
                    number of elements, of stack s)
   HIDDEN;
   INITIALLY
     i = 0;

VFUN stack_val(stack_name s; INTEGER i) -> INTEGER v;
  $( v is the ith value of stack s)
   HIDDEN;
   INITIALLY
     v = ?;

OFUN push(stack_name s; INTEGER v);
  $( puts the value v on top of stack s)
   EXCEPTIONS
     stack_overflow : ptr(s) = max_stack_size;
   EFFECTS
     'stack_val(s, 'ptr(s)) = v;
     'ptr(s) = ptr(s) + 1;

OVFUN pop(stack_name s) -> INTEGER v;
  $( pops the stack s and returns the old top)
   EXCEPTIONS
     stack_underflow : ptr(s) = 0;
   EFFECTS
     'stack_val(s, ptr(s)) = ?;
     'ptr(s) = ptr(s) - 1;
     v = stack_val(s, ptr(s));

END_MODULE
```

FIGURE C-1 SPECIFICATION OF THE STACKS MODULE

3. Expressions in SPECIAL

SPECIAL operations are defined in terms of *expressions*. Each expression is of a particular type, characterizing the type of the values returned by the expression. Expressions are constructed using constants, variables declared in the specification, built-in functions and connectives of the language, functions (O, OV, and V) of the module being specified, and additional functions declared to produce a more readable specification. The following are examples of types of expressions supported by SPECIAL.

Arithmetic Expressions

The value returned by an arithmetic expression is of type INTEGER or REAL. An arithmetic expression is a single constant, a variable or a user-defined function of type INTEGER or REAL, or is built out of existing arithmetic expressions using the operations "+", "*", "-", "/".

Boolean Expressions

The value returned by a boolean expression is of type BOOLEAN. The constants TRUE and FALSE are boolean expressions, as are variables and functions declared to be of type BOOLEAN. The operations AND, OR, "~" (NOT) and "=>" (IMPLIES) are used to construct new boolean expressions from existing boolean expressions.

Relational Expressions

Using the infix relational operators (namely "=", "<", "<=", ">", ">=", "~="), boolean expressions are constructed from existing expressions. For "=" (or "~="), the resulting expression is of the form A = B (or A ~= B) where A and B are required to have the same type. For the other operators, each of the two component expressions is required to be of type INTEGER or REAL.

Conditional Expressions

A conditional expression is of the form IF P THEN Q ELSE R, where P is of type boolean, and Q and R are of the same arbitrary type. The type of the resulting expression is the type of Q (or R).

Quantified Expressions

To express properties relating to a large number of values, SPECIAL provides quantified expressions in the first-order predicate calculus. The universal quantified statement is written as

FORALL x | P(x): Q(x)

or

FORALL x: P(x)=>Q(x).

The meaning is "For all values of x such that P(x) is true, Q(x) is also true." Clearly, P(x) and Q(x) are of type BOOLEAN, as is the type of resulting expression. The variable x can be of any type, usually declared prior to its introduction in the specification.

The existentially quantified statement is written as

EXISTS x | P(x): Q(x),

which has the meaning "There exists a value x such that, if P(x) is true, then Q(x) is also true."

4. Role of "?" in SPECIAL

SPECIAL provides the particular value UNDEFINED (abbreviated as "?") to stand for "no value". It is used in a specification where the designer wishes to associate the absence of a meaningful value with a data structure. (UNDEFINED should not be confused with "don't care," which stands for some value.) UNDEFINED is only used in a specification, not in an implementation; no operation can return "?" as a value. For purposes of establishing type-matching rules, however, "?" is assumed to be a value of every type.

5. Specification of "stacks"

Now we are ready to discuss the SPECIAL specification of the module "stacks" (Figure C-1). This specification consists of three *paragraphs*: TYPES, PARAMETERS, and FUNCTIONS. More complex modules would require additional paragraphs, omitted here for simplicity.

TYPES paragraph

Here the types referred to in the specification are declared. It is required that all designator types (e.g., "stacks" for this module) be declared, but the declaration of other types can be deferred until the first appearance of an item of that type. Note that comments -- $(This is a comment) -- can appear anywhere in a specification.

PARAMETERS paragraph

All of the parameters are listed as they appear in the interface description of the module.

FUNCTIONS paragraph

Most of the functionally interesting information in a module specification is embodied in the FUNCTIONS paragraph. Each of the operations of the module ("push" and "pop" for the module "stacks") is listed and individually specified. In addition, other functions, typically V-functions corresponding to data structures, are introduced to assist in the specification of the operations. It is emphasized that, except for the primitive machine, the data structures serve only for purposes of specification.

We separately consider V-functions and O- and OV-functions.

6. Specification of V-functions

For purpose of specification, a V-function returns a value and never causes a state change. A V-function is classified as [primitive or derived] and [visible or hidden]. Thus a V-function has one of four flavors, identified by the combination of reserved words that appear in its specification.

The *primitive* V-functions -- "ptr" and "stack_val" for the "stacks" module -- correspond to the module's data structures. Their specification requires the association of an initial value with each possible argument value. That is, all primitive functions are defined to be "total," although many argument values correspond to physically meaningless conditions. For such conditions, the value of the function is usually "?". The expression following INITIALLY specifies the initial value. The primitive v-function "stack_val" returns the INTEGER v corresponding to the i-th location in stacks. We have decided that the initial value v of "stack-val" for any stacks is to be "?" for all i. The expression

v = ?

which is understood to mean

FORALL s; i: stack_val(s, i)=?

captures this decision. Note that, in general, the expression need not determine a unique initial value for a primitive V-function.

The other primitive V-function, "ptr" returns the value i of the stack pointer for stack s. The initial value of "ptr" is 0 for all stacks, reflecting the decision that all stacks are to be initially empty.

A *hidden* V-function cannot be called from outside the module; i.e., it is not an operation. The reserved word HIDDEN in the V-function specification declares the function to be hidden. Clearly, "stack_val" should be hidden since only the top element of the stack is to be accessible. However, some designs for a stack allow the pointer to be accessible.

The *visible* V-functions are operations that return a value, but do not cause a state change. They are identified by the absence of the word HIDDEN in the specification. As is the case for all operations, the specification can indicate a list of exception conditions. Since the "stacks" module has no visible V-functions, we defer discussion of exception conditions to the next section.

The value of a *derived* V-function is specified in terms of the values of the primitive V-functions. In the specification of a derived V-function, an expression that defines the returned value appears following the reserved word DERIVATION.

Because a V-function can serve multiple roles (say as an operation and a data structure), the length of a SPECIAL specification can be reduced greatly, compared with an alternative specification technique in which operations and data structures are separately specified.

7. Specification of O- and OV-functions

All O- and OV-functions are state-changing operations. An operation can return one of n exceptions ex1, ex2, ..., exn (we use the descriptive term "raise" in referring to exceptional returns), or can return "normally". No state change occurs when an operation invocation raises an exception. A value-returning operation (V- or OV-function) will return an actual value upon the NORMAL return; an O-function merely returns. Exception returns are a way of associating particular events with classes of states and values of the operation's arguments. In the specification of an operation, the specification of each exception condition consists of a name (typically a mnemonic for the condition) followed by a boolean expression that characterizes the condition. The list of exception conditions follows the reserved word EXCEPTIONS.

The behavior of an operation that has n exception conditions is determined as follows: if the expression corresponding to ex1 evaluates to true, then the first exception is raised; if the expression corresponding to ex1 evaluates to false and the expression corresponding to ex2 evaluates to true, then the second exception is raised; ...; finally, if the expressions corresponding to ex1, ..., exn evaluate to false, the operation returns normally.

For the O-function "push," there is the single exception condition, specified as:

stack_overflow: ptr(s) = max_stack_size

The expression evaluates to true when the number of elements in the stack is equal to the maximum size of a stack.

Following the reserved word EFFECTS, the state changes that can occur as associated with O- and OV-functions, together with the value corresponding to the NORMAL return of an OV-function, are specified. The specification consists of a collection of boolean expressions, each called an *effect* (in which the order of presentation is irrelevant). Semantically, the collection of effects should be read as a single expression that is the conjunction of the expressions corresponding to each of the effects. An effect can reference the following: arguments to the operation, values of primitive V-functions before the invocation ("old" values) of the operation, and values that primitive V-functions will obtain after the invocation ("new" values). In the specification, a single quote, "'" , preceding a primitive V-function indicates the value of the V-function after the invocation. The collection of effects defines the new value of each primitive V-function in terms of old values and argument values in the following way: the feasible new values for the primitive V-functions are those for which each of the effects evaluates to TRUE. Thus the specifications need not be *deterministic*, i.e., they need not define a unique new value for each primitive V-function argument list. However, the specifications for our simple example are deterministic.

When the new value of a primitive V-function for some argument is not constrained by the specification, it is assumed that the new value is identical to the old value.

For "push," the effects are:

```
'stack_val(s, 'ptr(s)) = v;
'ptr(s) = ptr(s) + 1;
```

They constrain the new value of "ptr(s)" to be the old value incremented by one, and the new value of the pointer for s to be the value v pushed onto the stack. Note that, since the effects do not constrain the values of stack_val(s,i) for i ~= 'ptr(s), such values remain unchanged.

We will not burden the reader with a discussion of the effects for "pop," except for a few remarks. First, note that the returned value v is specified to be the INTEGER on the top of the stack in the old state. Second, the location at the top of the stack is the old state changed to be "?". It should be clear that this latter state change is apparent only in the specification. The implementation need not be concerned with this apparent storing of "?".

Appendix D. SIFT Specifications in SPECIAL

This appendix presents the detailed SPECIAL specifications of the SIFT executive systems. A general guide to the specifications is presented in Section B.

```
MODULE SIFT

$( The TYPES)

  TYPES
        $( currently the activity kinds are vote, dummy_vote, and execute )

    activity_kinds : INTEGER;

        $( the task kinds are the names of the various tasks.  The special
           ones are global_exec, reconfig, and error_report )

    task_kinds : INTEGER;

        $( This is an array of processor numbers used by the 3-way voter,
           etc )

    proc_array : ARRAY of INTEGER;

 $( The schedule table type definition )

        $( The array of schedules is a configuration schedule for each
           possible processor )

    sched_array : ARRAY of processor_array;

    activity_record : RECORD (activity_kinds activity;
                              task_kinds taskname;
                              INTEGER elem);

        $( This is a sequence of activities for a given subframe. )

    subframe_array : ARRAY of activity_record;

        $( This is a sequence of subframe actions for each configuration. )

    config_array : ARRAY of subframe_array;

        $( This is a set of configurations for each processor )

    processor_array : ARRAY of config_array;

 $( The datafile type definition )

        $( A datafile contains an array of task data spaces for each
           process which exists. )

    datafile_array : ARRAY of taskname_array;

        $( An input/output for a process is an array of data )
        elem_array : ARRAY of integer;

            $( Each task has an array of data of its own )

        taskname_array : ARRAY of elem_array;

    $( The configuration table type definition )

            $( For each configuration, indicate whether a processor executes
               a task )

        poll_array : ARRAY of poll_proc_array;

            $( Each task either is executed or not {true or false} )

        poll_task_array : ARRAY of BOOLEAN;

            $( For each processor, indicate whether the various tasks are
               executed or not )

        poll_proc_array : ARRAY of poll_task_array;

    $( Miscellaneous type definitions )

            $( The number of errors seen from each processor. )

        error_array : ARRAY of INTEGER;

            $( For each task, the element array which is input to that task )

        input_array : ARRAY of elem_array;

            $( A set of integer, for reasoning about properties of things )

        set_of_int : SETOF INTEGER;

            $( An array of integer, one for each task )

        task_array : ARRAY of INTEGER;

            $( An array of boolean )

        bool_array : ARRAY of BOOLEAN;


    $(  The PARAMETERS )


      PARAMETERS INTEGER frame_size, max_processors, my_processor,
                         max_activities, max_elems, max_tasks, bottom_val,
                         err_threshold, vote, dummy_vote, execute, reconfig,
                         global_exec, error_report, null_task;

            $( The sched table is a sequence of schedules for each configuration.
               It is of the form:
                    sched_table [proc_num][configuration][subframe][activity_num]
              and gives a record of activities to do.  Given a processor
              number, and a configuration number, and a subframe number,
        then there are a sequence of activities to do, each one
        described by its ACTIVITY field. The activities are currently
        VOTE, DUMMY_VOTE, and EXECUTE. For votes, the taskname field is
        the task to vote on and the element number is the element to
        vote on.  For dummy_votes, the entire element sequence of the
        taskname is set to bottom.  For executes, the taskname is invoked. )

     sched_array sched_table;

     $( The poll tells whether a processor ran a task in a given
        configuration.  It is referenced as:
             poll [configuration][processor][taskname] )

     poll_array poll;

     $( The inputs tell which tasks have produced input for a
        particular task.  It is indexed by the task to run and
        the number of the task which is input {from 1 to n}. )

     taskname_array inputs;

     $( This indicates which tasks are interactive consistency tasks. )

     bool_array i_c;

     $( This returns the number of elements output by each task )

     task_array result_size;

     $( The error report tasks are a group of tasks {one for each
        processor} which do the error reporting.  They are indicated
        here. )

     proc_array error_report_tasks;

     $( The error_i_c_tasks are the interactive consistency tasks
        which broadcast around the error reports.  There are three
        of these tasks with the specified task numbers. )

     proc_array error_i_c_tasks;


$( The DEFINITIONS )

  DEFINITIONS

    BOOLEAN is_in_majority (INTEGER c, t, e, val) IS

      $( given a configuration c, a taskname t, and an element number e,
         return true if val is in the majority of the outputs of all of
         the processors which produced output according to POLL.  If there
         is no majority, then val must be the default value. )

     IF EXISTS INTEGER maj_val :
         CARDINALITY ({ INTEGER q | q >= 1 AND q <= max_processors
                                  AND poll[c][real_to_virt()[q]][t]
                                  AND maj_val = datafile()[q][t][e]}) * 2
           > CARDINALITY ({ INTEGER p | p >= 1 AND p <= max_processors
                                          AND poll[c][p][t]}) END_EXISTS
    THEN FORALL INTEGER maj :
            CARDINALITY ({ INTEGER q | q >= 1 AND q <= max_processors
                                AND poll[c][real_to_virt()[q]][t]
                                AND maj = datafile()[q][t][e]}) * 2
             > CARDINALITY ({ INTEGER p | p >= 1 AND p <= max_processors
                                          AND poll[c][p][t]})
          => val = maj END_FORALL
    ELSE val = bottom_val END_IF;


$( The PARAMETER INVARIANTS )


  PARAMETER_INVARIANTS

        $( Constraints on the simple parameters.  Various constants have
           appropriate values.)

   frame_size > 0;  max_processors > 0;
   my_processor >= 1 AND my_processor <= max_processors;
   max_activities > 0; max_elems > 0; max_tasks > 0;
   vote > 0 AND dummy_vote > 0 AND execute > 0;
   vote ~= dummy_vote AND vote ~= execute AND dummy_vote ~= execute;
   FORALL INTEGER i : FORALL INTEGER j :
       reconfig ~= global_exec AND
       reconfig ~= null_task AND
       null_task ~= global_exec AND
       error_report_tasks[i] ~= reconfig AND
       error_report_tasks[i] ~= global_exec AND
       error_report_tasks[i] ~= null_task AND
       error_report_tasks[i] ~= error_i_c_tasks[j] AND
       error_i_c_tasks[i] ~= reconfig AND
       error_i_c_tasks[i] ~= null_task AND
       error_i_c_tasks[i] ~= global_exec END_FORALL END_FORALL;

        $( Constraints on the schedule table and associated data structures )

   FORALL INTEGER p : FORALL INTEGER c : FORALL INTEGER sf :
       FORALL INTEGER j1 : FORALL INTEGER j2 : FORALL INTEGER i :

                $( any execute which needs the results of a vote or
                   dummy_vote must follow the vote )

             sched_table[p][c][sf][j1].activity = execute
         AND (   sched_table[p][c][sf][j2].activity = vote
              OR sched_table[p][c][sf][j2].activity = dummy_vote)
         AND inputs[sched_table[p][c][sf][j1].taskname][i]
                 = sched_table[p][c][sf][j2].taskname
        => j1 > j2
     END_FORALL END_FORALL END_FORALL END_FORALL END_FORALL END_FORALL;

   FORALL INTEGER p : FORALL INTEGER c : FORALL INTEGER sf :
       FORALL INTEGER j1 : FORALL INTEGER j2 :

                $( there does not exist a vote and a dummy_vote on the
                   same task during a subframe. )
        NOT (     sched_table[p][c][sf][j1].activity = vote
              AND sched_table[p][c][sf][j2].activity = dummy_vote
              AND sched_table[p][c][sf][j1].taskname =
                    sched_table[p][c][sf][j2].taskname)
     END_FORALL END_FORALL END_FORALL END_FORALL END_FORALL;

FORALL INTEGER p : FORALL INTEGER c : FORALL INTEGER sf :
    FORALL INTEGER j1 : FORALL INTEGER j2 :

            $( there does not exist a vote on the results of an execute
                    in the same subframe )

         NOT (     sched_table[p][c][sf][j1].activity = execute
               AND sched_table[p][c][sf][j2].activity = vote
               AND sched_table[p][c][sf][j1].taskname =
                     sched_table[p][c][sf][j2].taskname)
       END_FORALL END_FORALL END_FORALL END_FORALL END_FORALL;

FORALL INTEGER p : FORALL INTEGER c : FORALL INTEGER sf :
    FORALL INTEGER j1 : FORALL INTEGER j2 :

            $( Any reconfiguration done must be at the end of a subframe )

            sched_table[p][c][sf][j1].activity = execute
        AND sched_table[p][c][sf][j1].taskname = reconfig
        AND (   sched_table[p][c][sf][j2].activity = execute
             OR sched_table[p][c][sf][j2].activity = vote)
      => j1 > j2 END_FORALL END_FORALL END_FORALL END_FORALL END_FORALL;

FORALL INTEGER p : FORALL INTEGER c : FORALL INTEGER sf :

            $( no vote and error_report allowed in the same subframe. )

            EXISTS INTEGER j :
                sched_table[p][c][sf][j].activity = vote END_EXISTS
         => NOT EXISTS INTEGER i :
                       sched_table[p][c][sf][i].activity = execute
                   AND sched_table[p][c][sf][i].taskname
                             = error_report_tasks[p] END_EXISTS
  END_FORALL END_FORALL END_FORALL;

FORALL INTEGER p : FORALL INTEGER c : FORALL INTEGER sf :
    FORALL INTEGER i : FORALL INTEGER j :

            $( There do not exist two votes on the same element of
               the same task in the same subframe )

               sched_table[p][c][sf][i].activity = vote
           AND sched_table[p][c][sf][j].activity = vote
           AND i ~= j
        =>     sched_table[p][c][sf][i].taskname
                   ~= sched_table[p][c][sf][j].taskname
           OR sched_table[p][c][sf][i].elem
                   ~= sched_table[p][c][sf][j].elem
  END_FORALL END_FORALL END_FORALL END_FORALL END_FORALL;

FORALL INTEGER p : FORALL INTEGER c : FORALL INTEGER sf :
    FORALL INTEGER i : FORALL INTEGER j :
```

```
               $( There do not exist two dummy_votes on the same element of
                  the same task in the same subframe )

                  sched_table[p][c][sf][i].activity = dummy_vote
              AND sched_table[p][c][sf][j].activity = dummy_vote
              AND i ~= j
           => sched_table[p][c][sf][i].taskname
                 ~= sched_table[p][c][sf][j].taskname
    END_FORALL END_FORALL END_FORALL END_FORALL END_FORALL;

   FORALL INTEGER p : FORALL INTEGER c : FORALL INTEGER sf :
       FORALL INTEGER i : FORALL INTEGER j :

               $( There do not exist two executes on
                  the same task in the same subframe )

                  sched_table[p][c][sf][i].activity = execute
              AND sched_table[p][c][sf][j].activity = execute
              AND i ~= j
           =>     sched_table[p][c][sf][i].taskname
              ~= sched_table[p][c][sf][j].taskname
    END_FORALL END_FORALL END_FORALL END_FORALL END_FORALL;

   FORALL INTEGER p : FORALL INTEGER c : FORALL INTEGER sf :
       FORALL INTEGER j :

               $( all activities are either vote, dummy_vote, or execute. )

          sched_table[p][c][sf][j].activity = vote
       OR sched_table[p][c][sf][j].activity = dummy_vote
       OR sched_table[p][c][sf][j].activity = execute
       OR sched_table[p][c][sf][j].activity = 0
    END_FORALL END_FORALL END_FORALL END_FORALL;

   FORALL INTEGER p : FORALL INTEGER c : FORALL INTEGER sf :
       FORALL INTEGER j : FORALL INTEGER i :

               $( zero fill in sched table )

               sched_table[p][c][sf][j].activity = 0 AND i > j
            => sched_table[p][c][sf][i].activity = 0
       END_FORALL END_FORALL END_FORALL END_FORALL END_FORALL;

   FORALL INTEGER c : FORALL INTEGER ti :

               $( The number of processors running a particular task
                      is 1 for interactive consistency tasks or 3
                      otherwise )

       CARDINALITY({ INTEGER p | p >= 1 AND p <= max_processors
                                           AND poll[c][p][ti]})
           = IF i_c[ti] THEN 1 ELSE 3 END_IF
    END_FORALL END_FORALL;

               $( The inputs to the global executive are itself and the
                  error interactive consistency tasks. )
    inputs[global_exec][1] = global_exec
AND FORALL INTEGER i : (i >= 1 AND i <= max_processors
                      => inputs[global_exec][i+1] = error_i_c_tasks[i])
                  AND (i < 1 OR i > max_processors
                      => inputs[global_exec][i] = null_task)
     END_FORALL;

    $( these are the constraints on the output of various tasks )

FORALL input_array inp : FORALL INTEGER i :
    task_results (error_i_c_tasks[i], inp) = inp[1]
END_FORALL END_FORALL;

        $( returns 1 {not-working} if it was previously not working
            or if a majority of those working consider it bad. )

FORALL input_array inp : FORALL INTEGER p :
   IF   inp[1][p] = 1   $( = input[global_exec][p] )
     OR CARDINALITY({INTEGER q1 | inp[1][q1] = 0
                             AND p ~= q1 AND inp[q1+1][p] = 1}) * 2
                                     $( = input[error_i_c_tasks[q1]] )
           > CARDINALITY ({INTEGER q2 | inp[1][q2] = 0 AND p ~= q2})
   THEN task_results (global_exec, inp)[p] = 1
   ELSE task_results (global_exec, inp)[p] = 0 END_IF
 END_FORALL END_FORALL;

  FUNCTIONS

$( The State Functions )

        $( The subframe count.  Used to index into various tables )

 VFUN subframe() -> INTEGER s;

        $( The current configuration {ie, the number of processors
           currently assumed to be working}. )

 VFUN config() -> INTEGER c;

        $( This is the input values for a task.  It is referenced as:
                input [taskname][element] )

 VFUN input() -> input_array value;

        $( The datafile is the broadcast area.  It is referenced as:
                datafile [processor][taskname][element] )

 VFUN datafile() -> datafile_array value;

        $( The errors accumulated for each processor, indexed by processor )

 VFUN errors() -> error_array v;

        $( Given a real processor number, this returns the processor number
           used in the various tables for this configuration. )

 VFUN real_to_virt() -> proc_array v;
```

```
        $( Given a processor number in the tables, this maps to the current
           real processor which is associated with it )

     VFUN virt_to_real() -> proc_array v;

  $( The Operations )

     OFUN dispatcher ();

        $( The dispatcher is invoked each subframe.  It bumps the subframe
           number and does each of the activities for that subframe. )

      ASSERTIONS

        subframe() < frame_size and subframe() >= 0;
        FORALL INTEGER c : FORALL INTEGER ti :
            CARDINALITY({ INTEGER p | p >= 1 AND p <= max_processors
                                      AND poll[c][real_to_virt()[p]][ti]})
                = IF i_c[ti] THEN 1 ELSE 3 END_IF
         END_FORALL END_FORALL;

      EFFECTS

        $( changes to subframe )

        'subframe() = (subframe + 1) MOD frame_size;

        $( poll set still has 1 or 3 )

        FORALL INTEGER c : FORALL INTEGER ti :
            CARDINALITY({ INTEGER p | p >= 1 AND p <= max_processors
                                      AND poll[c][real_to_virt()[p]][ti]})
                = IF i_c[ti] THEN 1 ELSE 3 END_IF
         END_FORALL END_FORALL;

        $( Changes to INPUT:
           For all tasks ti and for all data elements of the task,
           if this element of this task was voted on then the input now
           has the majority value, else if it was dummy voted it has
           bottom as value, else it hasn't changed. )

               $( Frame axiom: if no vote or dummy vote, nothing changed )

       FORALL INTEGER ti : FORALL INTEGER ei :
           NOT EXISTS INTEGER j : j >= 1 AND j <= max_activities
                AND ti = sched_table[real_to_virt()[my_processor]][config()]
                                    [subframe()][j].taskname
                AND (    (     sched_table[real_to_virt()[my_processor]]
                                          [config()][subframe()][j].activity
                                  = vote
                           AND ei = sched_table[real_to_virt()[my_processor]]
                                          [config()][subframe()][j].elem)
                      OR sched_table[real_to_virt()[my_processor]][config()]
                                  [subframe()][j].activity
                           = dummy_vote) END_EXISTS
         => 'input()[ti][ei] = input()[ti][ei]
       END_FORALL END_FORALL;
```

```
        $( Vote activity )

FORALL INTEGER ti : FORALL INTEGER ei :
        EXISTS INTEGER j : j >= 1 AND j <= max_activities
         AND sched_table[real_to_virt()[my_processor]][config()]
                        [subframe()][j].activity = vote
         AND ti = sched_table[real_to_virt()[my_processor]][config()]
                             [subframe()][j].taskname
         AND ei = sched_table[real_to_virt()[my_processor]][config()]
                             [subframe()][j].elem END_EXISTS
         => is_in_majority (config(), ti, ei, 'input()[ti][ei])
END_FORALL END_FORALL;

        $( Dummy vote activity )

FORALL INTEGER ti : FORALL INTEGER ei :
        EXISTS INTEGER j : j >= 1 AND j <= max_activities
                AND sched_table[real_to_virt()[my_processor]]
                    [config()][subframe()][j].activity
                               = dummy_vote
                AND ti = sched_table[real_to_virt()[my_processor]]
                                   [config()][subframe()][j].taskname
          END_EXISTS AND ei >= 1 AND ei <= result_size[ti]
         => 'input()[ti][ei] = bottom_val
END_FORALL END_FORALL;

$( Changes to ERRORS:
   For all processors p, for every non interactive consistency
   vote that p was involved in for which p was not in the majority,
   the error count for p goes up by one. )

FORALL INTEGER p :
    'errors()[p] = errors()[p]
        + CARDINALITY ({INTEGER j | j >= 1 AND j <= max_activities
        AND sched_table[real_to_virt()[my_processor]][config()]
                       [subframe()][j].activity = vote
        AND NOT i_c[sched_table[real_to_virt()[my_processor]]
               [config()][subframe()][j].taskname]
        AND poll[config()][real_to_virt()[p]]
                [sched_table[real_to_virt()[my_processor]][config()]
                            [subframe()][j].taskname]
        AND NOT is_in_majority
                (config(),
                 sched_table[real_to_virt()[my_processor]][config()]
                            [subframe()][j].taskname,
                 sched_table[real_to_virt()[my_processor]][config()]
                            [subframe()][j].elem,
                 datafile()[p]
                       [sched_table[real_to_virt()[my_processor]
                                   [config()][subframe()]
                                   [j].taskname]
                       [sched_table[real_to_virt()[my_processor]]
                                   [config()][subframe()][j].elem])})
END_FORALL;

FORALL INTEGER j : FORALL INTEGER p :
    j >= 1 AND j <= max_activities
    AND sched_table[real_to_virt()[my_processor]][config()]
```

```
                    [subframe()][j].activity = execute
         AND sched_table[real_to_virt()[my_processor]]
                    [config()][subframe()][j].taskname
                  = error_report_tasks[my_processor]
               => 'errors()[p] = 0 END_FORALL END_FORALL;

$( Changes to DATAFILE:
   For each task that is executed, the datafile contains the
   output for that task. )

             $( Frame axiom: if no execute is done on a task, then
                its datafile area stays the same. )

 FORALL INTEGER p : FORALL INTEGER ti : FORALL INTEGER ei :
     NOT (     p = my_processor
          AND EXISTS INTEGER j : j >= 1 AND j <= max_activities AND
              sched_table[real_to_virt()[my_processor]][config()]
                         [subframe()][j].activity = execute
             AND sched_table[real_to_virt()[my_processor]][config()]
                            [subframe()][j].taskname = ti
                   END_EXISTS)
            => 'datafile()[p][ti][ei] = datafile()[p][ti][ei]
    END_FORALL END_FORALL END_FORALL;

             $( Execute activity )

 FORALL INTEGER ti : FORALL INTEGER ei : FORALL INPUT_ARRAY inp :
          EXISTS INTEGER j : j >= 1 AND j <= max_activities AND
              sched_table[real_to_virt()[my_processor]][config()]
                         [subframe()][j].activity = execute
             AND sched_table[real_to_virt()[my_processor]][config()]
                            [subframe()][j].taskname = ti
             AND ti ~= reconfig
             AND ti ~= error_report_tasks[my_processor]
           END_EXISTS
      AND FORALL INTEGER taski : FORALL INTEGER j : FORALL INTEGER elemi:
                  j >= 1 AND j <= result_size[taski]
              AND inputs[ti][j] = taski AND taski ~= null_task
             => inp[j][elemi] = 'input()[taski][elemi]
           END_FORALL END_FORALL END_FORALL
      => 'datafile()[my_processor][ti][ei] = task_results (ti, inp)[ei]
   END_FORALL END_FORALL END_FORALL;

 FORALL INTEGER ei : FORALL INTEGER j :
            j >= 1 AND j <= max_activities
        AND sched_table[real_to_virt()[my_processor]][config()]
                       [subframe()][j].activity = execute
        AND sched_table[real_to_virt()[my_processor]]
                       [config()][subframe()][j].taskname
                  = error_report_tasks[my_processor]
     => 'datafile()[my_processor][error_report_tasks[my_processor]][ei]
              = IF errors()[ei] > err_threshold THEN 1 ELSE 0 END_IF
   END_FORALL END_FORALL;

$( Changes to CONFIG, REAL_TO_VIRT, and VIRT_TO_REAL:
   These are only changed by reconfig.  Config is set to the number of
   processors which are currently working, as reported by the global_exec
   task.  Real_to_virt is set so that the nth processor is mapped to
the mth working processor.  Virt_to_real is set so that the nth
working processor is mapped to the mth processor. )

           $( Frame axiom: if there is no reconfiguration, then the
              reconfiguration data stays the same. )

    NOT EXISTS INTEGER j : j >= 1 AND j <= max_activities
          AND sched_table[real_to_virt()[my_processor]][config()]
                         [subframe()][j].activity = execute
          AND sched_table[real_to_virt()[my_processor]][config()]
                         [subframe()][j].taskname = reconfig
         END_EXISTS
       => 'config() = config() AND 'real_to_virt() = real_to_virt() AND
          'virt_to_real() = virt_to_real();

           $( reconfiguration activity )

    EXISTS INTEGER j : j >= 1 AND j <= max_activities
          AND sched_table[real_to_virt()[my_processor]][config()]
                         [subframe()][j].activity = execute
          AND sched_table[real_to_virt()[my_processor]][config()]
                         [subframe()][j].taskname = reconfig
         END_EXISTS
       => FORALL INTEGER x :
              'config() = CARDINALITY ({INTEGER p1 |
                                       input()[global_exec][p1] = 0})
          AND 'real_to_virt()[x] = CARDINALITY ({INTEGER p2 |
                        p2 <= x AND input()[global_exec][p2] = 0})
          AND 'virt_to_real()[x]
                 = x + CARDINALITY ({INTEGER p3 |
                            p3 <= x AND input()[global_exec][p3] = 1})
         END_FORALL;


OFUN vote_activity (INTEGER c, t, e);
  ASSERTIONS
    FORALL INTEGER c : FORALL INTEGER ti :
        CARDINALITY({ INTEGER p | p >= 1 AND p <= max_processors
                                   AND poll[c][real_to_virt()[p]][ti]})
           = IF i_c[ti] THEN 1 ELSE 3 END_IF
     END_FORALL END_FORALL;

  EFFECTS
    is_in_majority (c, t, e, 'input()[t][e]);
    FORALL INTEGER ti : FORALL INTEGER ei :
        ti ~= t OR ei ~= e => 'input()[ti][ei] = input()[ti][ei]
      END_FORALL END_FORALL;
    FORALL INTEGER q :
      IF poll[c][real_to_virt()[q]][t] AND NOT i_c[t] THEN
        IF is_in_majority (c, t, e, datafile()[q][t][e])
          THEN 'errors()[q] = errors()[q]
          ELSE 'errors()[q] = errors()[q] + 1 END_IF
      ELSE 'errors()[q] = errors()[q] END_IF END_FORALL;

OVFUN vote3 (INTEGER t, e; proc_array p) -> INTEGER result;
  ASSERTIONS
        p[1] ~= p[2] AND p[1] ~= p[3] AND p[2] ~= p[3]
    AND p[1] >= 1 AND p[1] <= max_processors
```

```
      AND p[2] >= 1 AND p[2] <= max_processors
      AND p[3] >= 1 AND p[3] <= max_processors;
    EFFECTS
      IF EXISTS INTEGER maj_val :
            CARDINALITY ({INTEGER q1 | q1 >= 1 AND q1 <= max_processors
                                       AND datafile()[q1][t][e] = maj_val
                                       AND (q1 = p[1] OR q1 = p[2] OR
                                            q1 = p[3])}) > 1 END_EXISTS
      THEN FORALL INTEGER maj :
               CARDINALITY ({INTEGER q1 | q1 >= 1 AND q1 <= max_processors
                                       AND datafile()[q1][t][e] = maj
                                       AND (q1 = p[1] OR q1 = p[2] OR
                                            q1 = p[3])}) > 1
            =>     result = maj
               AND FORALL INTEGER j :
                      IF j ~= p[1] AND j ~= p[2] AND j ~= p[3]
                      THEN 'errors()[j] = errors()[j]
                      ELSE IF datafile()[j][t][e] = maj
                           THEN 'errors()[j] = errors()[j]
                           ELSE 'errors()[j] = errors()[j] + 1 END_IF END_IF
                 END_FORALL
            END_FORALL
      ELSE     result = bottom_val
           AND FORALL INTEGER j :
                  IF j = p[1] OR j = p[2] OR j = p[3]
                  THEN 'errors()[j] = errors()[j] + 1
                  ELSE 'errors()[j] = errors()[j] END_IF
            END_FORALL END_IF;

  OFUN dummy_vote_activity (INTEGER c, t);
    EFFECTS
      FORALL INTEGER ti : FORALL INTEGER ei :
        'input()[ti][ei] =
              IF ti = t AND ei >= 1 AND ei <= result_size[t]
              THEN bottom_val ELSE input()[ti][ei] END_IF
      END_FORALL END_FORALL;

  OFUN gexectask ();
    EFFECTS
      FORALL INTEGER t : FORALL INTEGER e : FORALL INTEGER p :
             IF p = my_processor AND t = global_exec THEN
                IF    inp[global_exec][p] = 1
                   OR CARDINALITY ({INTEGER q1 | inp[global_exec][q1] = 0
                        AND p ~= q1 AND inp[error_i_c_tasks[q1]][p] = 1}) *
                     2 > CARDINALITY ({INTEGER q2 | inp[global_exec][q2] = 0
                                                     AND p ~= q2})
                THEN 'datafile()[p][t][e] = 1
                ELSE 'datafile()[p][t][e] = 0 END_IF
             ELSE 'datafile()[p][t][e] = datafile()[p][t][e] END_IF
      END_FORALL END_FORALL END_FORALL;

  OFUN errtask ();
    EFFECTS
      FORALL INTEGER p : FORALL INTEGER t : FORALL INTEGER e :
         'datafile()[p][t][e] =
            IF t = error_report AND p = my_processor THEN
              IF errors()[e] > err_threshold THEN 1 ELSE 0 END_IF
            ELSE datafile()[p][t][e] END_IF END_FORALL END_FORALL
```

```
          AND (p >= 1 AND p <= max_processors => 'errors()[p] = 0) END_FORALL;

    OFUN recftask ();
      EFFECTS
        'config() = CARDINALITY ({INTEGER p1 | input()[global_exec][p1] = 0});
        FORALL INTEGER x2 :
           'real_to_virt()[x2] = CARDINALITY ({INTEGER p2 |
                                     p2 <= x2 AND input()[global_exec][p2] = 0})
         END_FORALL;
        FORALL INTEGER x3 :
           'virt_to_real()[x3] = x3 + CARDINALITY ({INTEGER p3 |
                                     p3 <= x3 AND input()[global_exec][p3] = 1})
         END_FORALL;

    OVFUN do_err_ic (INTEGER ti) -> BOOLEAN task_done;
      EFFECTS
        IF EXISTS INTEGER i : error_i_c_tasks[i] = ti END_EXISTS
        THEN FORALL INTEGER p : FORALL INTEGER taski : FORALL INTEGER ei :
                  'datafile()[p][taski][ei] =
                      IF p = my_processor AND ti = taski
                        THEN input()[inputs[ti][1]][ei]
                        ELSE datafile()[p][taski][ei] END_IF
                END_FORALL END_FORALL END_FORALL
             AND task_done = TRUE
        ELSE     task_done = FALSE
             and 'datafile() = datafile() END_IF;

    OFUN general_task (INTEGER ti);
      EFFECTS
        FORALL INTEGER p : FORALL INTEGER taski : FORALL INTEGER ei :
            IF p = my_processor AND taski = ti THEN
                FORALL input_array inp :
                   FORALL INTEGER input_task : FORALL INTEGER j :
                        FORALL INTEGER elemi :
                               j >= 1 AND j <= result_size[input_task]
                          AND inputs[ti][j] = input_task
                          AND input_task ~= null_task
                       => inp[j][elemi] = 'input()[input_task][elemi]
                     END_FORALL END_FORALL END_FORALL
                 => 'datafile()[p][ti][ei] = task_results (ti, inp)[ei]
                END_FORALL
            ELSE 'datafile()[p][taski][ei] = datafile()[p][taski][ei] END_IF
           END_FORALL END_FORALL END_FORALL;
        'input() = input();

END_MODULE
```

Appendix E. The 1553A Aircraft I/O Controller

1. Introduction

This appendix describes the design of the module that controls aircraft input-output, according to MIL STD 1553A for each processor.

2. General Characteristics

The 1553A controller is a 32-bit word-sized, microcoded processor. It has address computation capability, microcoded test routines, ability to program branch special-purpose registers, and ability to operate on a prioritized interrupt or polling basis. The controller shares memory with one BDX-930 processor through the data file of the processor. The interface to the data file provides for 16-bit parallel words. The interface to the 1553A bus is serial by bit.

The 1553A controller is composed of an analog section and a digital section. The analog section is a waveform and impedance converter. It converts the pulsed digital data received from the digital section to a 1 megabit serial 1553A bus-compatible signal for transmission over the 1553A bus. In turn, it converts the received 1553A bus signals to pulsed digital data that can be processed by the digital section. The digital section responds to commands from the BDX-930 processor to transmit, receive, or idle. In addition, it encodes and decodes bus data as required by mode logic (Figure E-1).

3. Detailed Design

The 1553A bus employs three modes of information transfer: (1) bus-controller to remote-terminal (RT) transfer, (2) RT to controller transfer, and (3) RT to RT transfer (Figure E-2).

The sequence of events for controller-to-terminal transfer is as follows:

- -Reset encoder/decoder
- -Check bus activity
- -Encode contents of command register
- -Access data file and load into data register
- -Encode contents of data register
- -Increment address counter
- -Decrement data word counter
- -Verify that all data words have been transmitted
- -Verify that remote terminal responds with a valid status word within a specified time frame or flag an error.

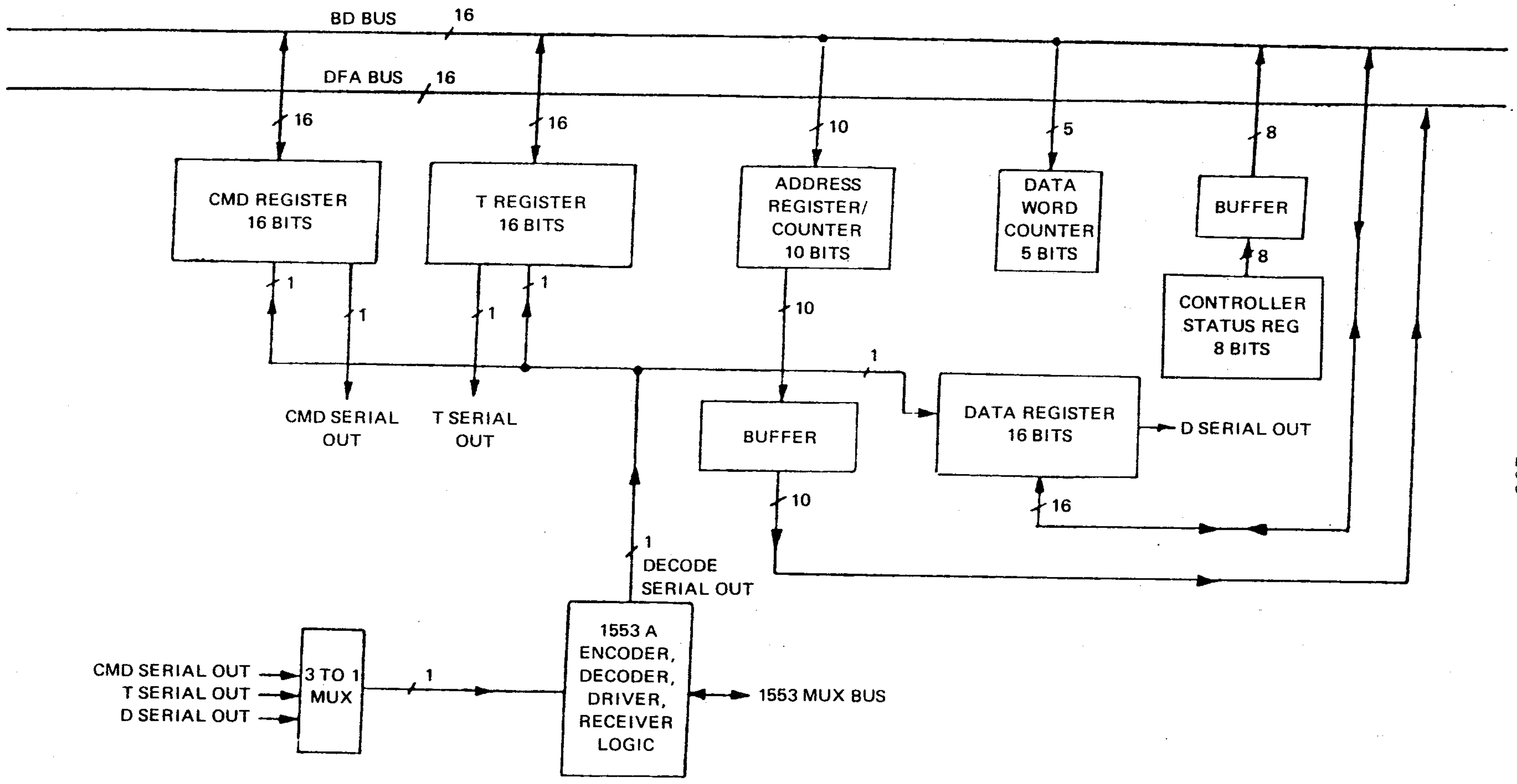


FIGURE E-1 1553 LINK

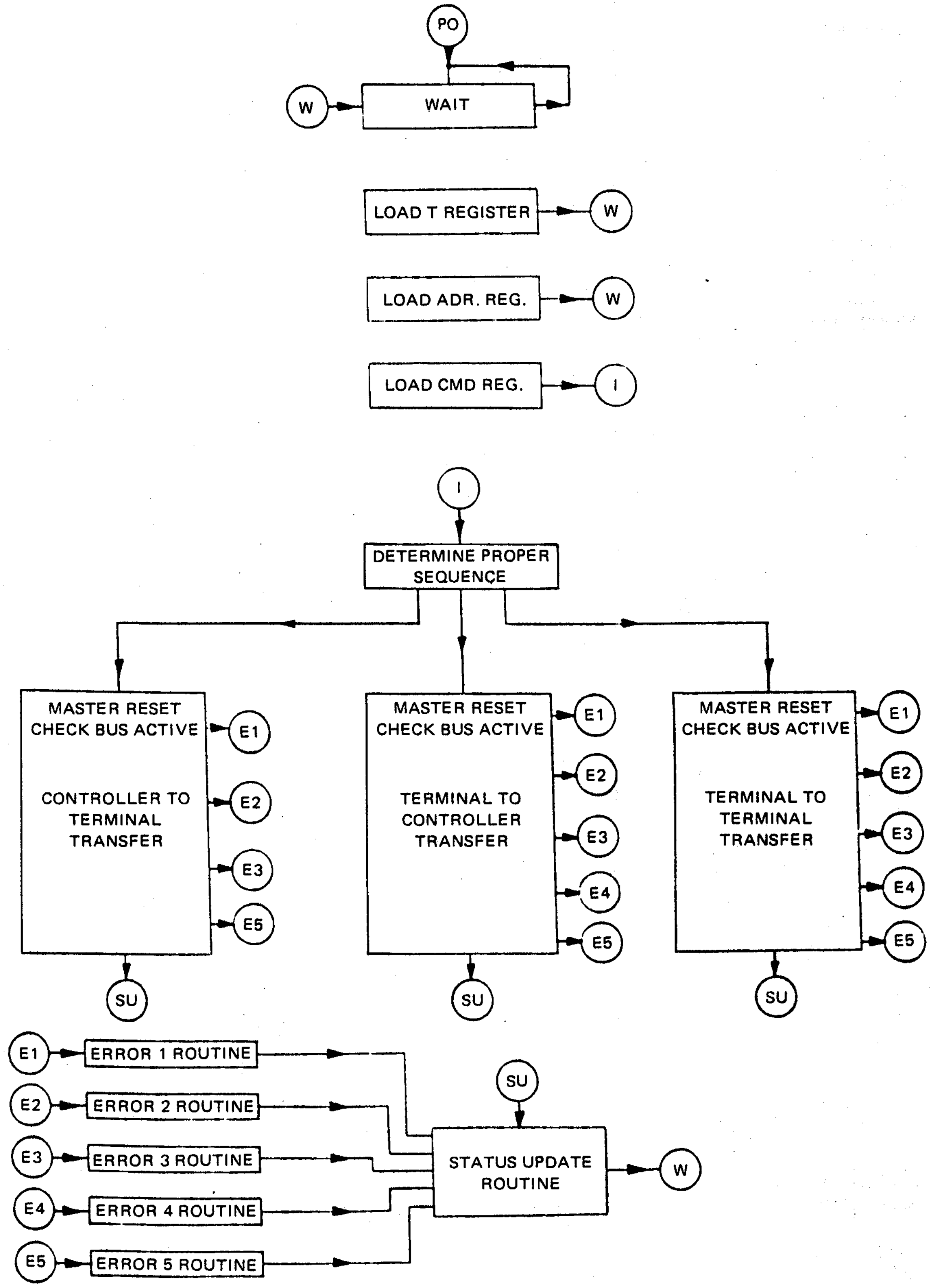

PO
W
WAIT
LOAD T REGISTER
W
LOAD ADR. REG.
W
LOAD CMD REG.
I
I
DETERMINE PROPER SEQUENCE
MASTER RESET CHECK BUS ACTIVE
CONTROLLER TO TERMINAL TRANSFER
E1
E2
E3
E5
SU
MASTER RESET CHECK BUS ACTIVE
TERMINAL TO CONTROLLER TRANSFER
E1
E2
E3
E4
E5
SU
MASTER RESET CHECK BUS ACTIVE
TERMINAL TO TERMINAL TRANSFER
E1
E2
E3
E4
E5
SU
E1
ERROR 1 ROUTINE
E2
ERROR 2 ROUTINE
E3
ERROR 3 ROUTINE
E4
ERROR 4 ROUTINE
E5
ERROR 5 ROUTINE
SU
STATUS UPDATE ROUTINE
W

FIGURE E-2 1553A FLOW CHART

Terminal-to-controller transfers are made as follows:

-Reset encoder/decoder
-Check bus activity
-Encode contents of command register
-Verify that a valid status word is received from RT within a specified time frame (or flag an error), and load the decoded word into the T register
-Verify that valid data word(s) immediately follows status word or flag an error, and load the decoded word into the data register
-Load contents of data register into data file
-Increment address counter
-Decrement word counter
-Verify that all data words have been transmitted.

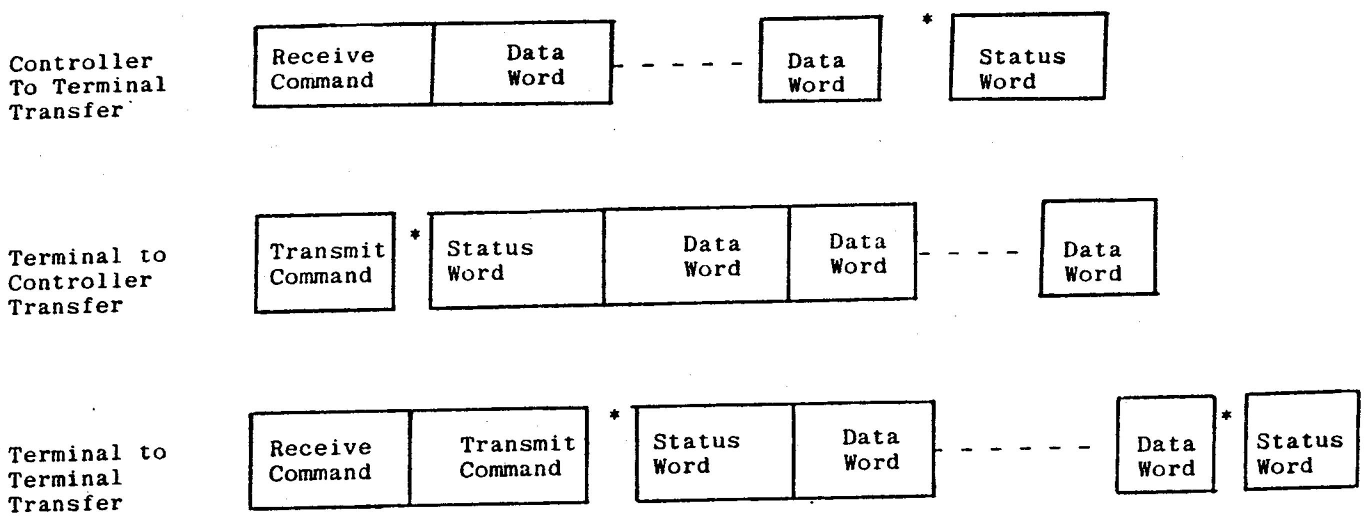
The sequence of events for terminal-to-terminal transfer is:

-Reset encoder/decoder
-Check bus activity
-Encode contents of command register
-Encode contents of T register
-Verify that a valid status word is received from RT within a specified time frame or flag an error, and load the decoded word into the T register
-Verify that valid data word(s) immediately follows status word or flag an error, and load the decoded word into the data register
-Increment address counter
-Decrement word counter
-Verify that all data words have been transmitted
-Verify that RT responds with a valid status word within a specified time frame or flag an error, and load the decoded word into the T register.

The message formats for the information transfers discussed in the preceding paragraphs are shown in Figure E-3.

Errors that are detected during information transfers are indicated in the eight-bit-controller status register (Figure E-1). The contents of this register indicate the following:

-A logic '1' in bit 0 indicates that the avionics multiplex bus is active. A logic '0' means the bus is inactive.
-A logic '1' in bit 1 indicates that the remote terminal has responded too quickly with its status response (error 1). A logic '0' indicates no failure.
-A logic '1' in bit 2 indicates that the remote terminal has not responded with its status word within the allotted time (error 2). A logic '0' indicates no failure.
-A logic '1' in bit 3 indicates that the information being received into the controller from the RT is of the wrong type, i.e., data when it should be status or vice versa (error 3). A logic '0' indicates that it is of the correct



*Gap

FIGURE E-3 MESSAGE FORMATS

type.
-A logic '1' in bit 4 indicates that the interword gap was too long (error 4). A logic '0' represents a no-failure indication.
-A logic '1' in bit 5 indicates that the word received from the remote terminal was invalid, with parity incorrect, timing off, or sync incorrect (error 5). A logic '0' represents no failure.
-Bit 6 is not presently being used.
-A logic '1' in bit 7 indicates that the 1553 controller has completed the requested transfer (controller to RT, RT to controller, or RT to RT). A logic '0' indicates transfer not complete.

The command, status, and data words referred to in the preceding paragraphs have the word formats shown in Figure E-4. The word size is 16 bits plus the synch waveform and the parity bit.

The command word comprises a synch waveform, RT address, transmit/receive bit, subaddress/mode, data word count, and a parity bit (Figure E-4). The five bits following the synch specify the RT address. A maximum of 32 RTs can be attached to each bus. The transmit/receive bit indicates the action required of the RT. A logic zero indicates that the RT is to receive; a logic one indicates that it is to transmit. The subaddress/mode field can be used for either a RT subaddress or mode control. It is not presently used. The data word count field specifies the quantity of data words to be sent out or received by the RT. A maximum of 32 data words may be transmitted or received in any one message block. The last bit in the command word is used for parity over the sixteen preceding bits. Odd parity is used.

The data word comprises a sync waveform, data bits, and a parity bit (Figure E-4). The five bits following the sync specify the address of the terminal that is transmitting the status word. A logic one in the message-error field indicates that the preceding message failed to pass the RT's validity tests. The error condition includes parity errors. A logic zero indicates the absence of a message error. The nine bits following the message-error bit can be used to indicate the RT's status. Presently, they have not been defined. A logic one setting of the terminal flag bit indicates the need for the bus controller to examine the built-in test data available from the terminal. Utilization of the parity bit is the same as for the command word.

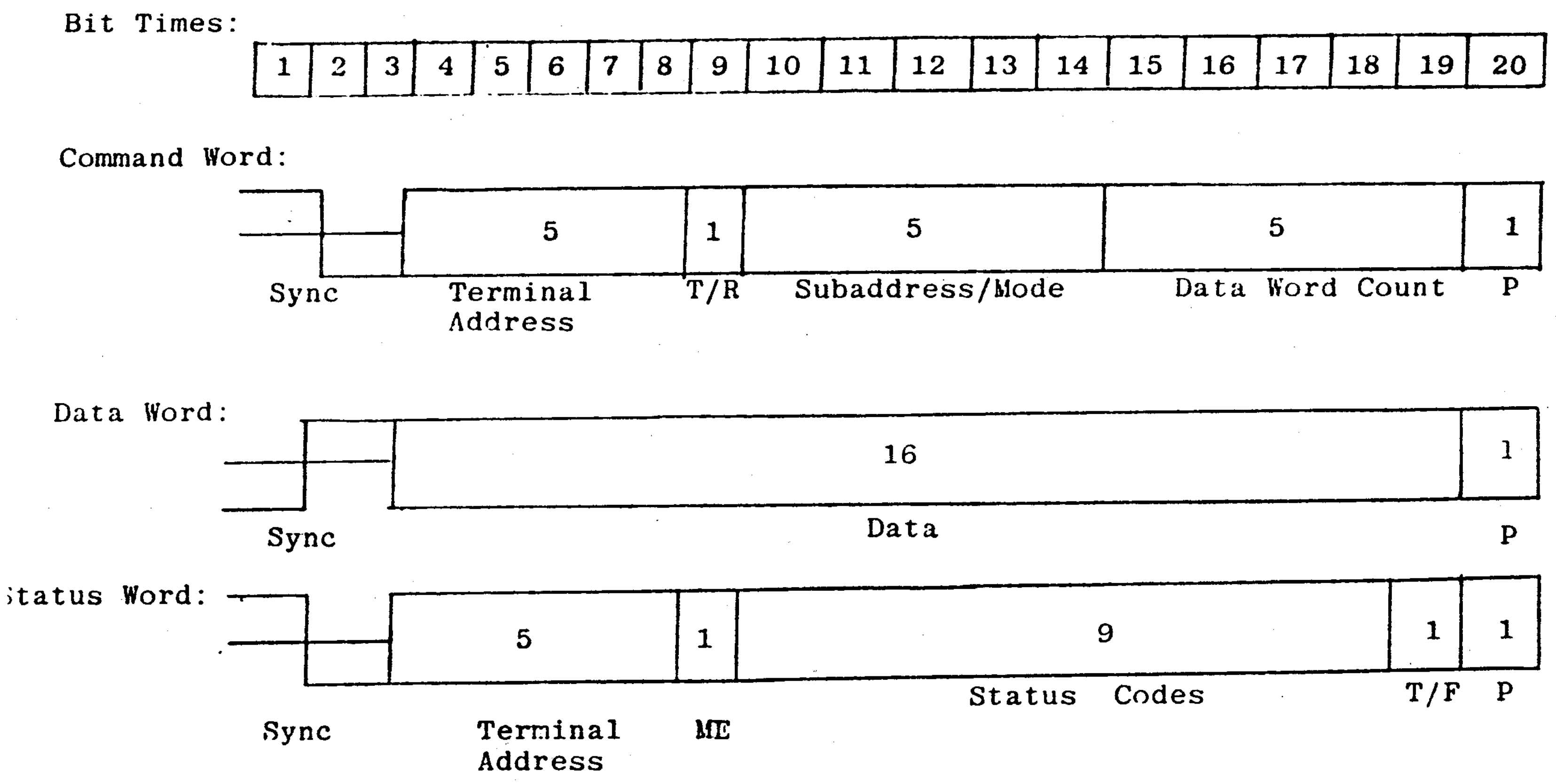


FIGURE E-4 WORD FORMATS

Appendix F. Task Programming

1. Introduction

SIFT programs must be specially prepared to provide for task scheduling and voting. Presently this preparation must be done manually. This Appendix gives instructions for this preparation. There are three major steps: task dependency analysis, programming, and table creation.

2. Task Dependency Analysis

Analysis of dependency relations among tasks is conducted in the following steps:

1. Divide the application into tasks. The tasks should be iterative in nature, and each one should be capable of completion in a short period of time (well under 3 milliseconds).

2. Determine what data will have to be saved from one iteration to the next. This includes state information that the task will need the next time it executes, as well as information that needs to be passed from one task to another.

3. Having made a list of all data that need to be saved, determine the dependency of tasks. For example, if Task A broadcasts a value that Task B needs to execute properly, then it is clear that all processors must have completed Task A before any processor may run Task B. Assume we have Tasks A, B, and C. Task A broadcasts X, Task B needs X and Y to compute Z, which it broadcasts, and Task C needs X to compute Y, which it broadcasts. Then Task A must run first, Task C must run second, and Task B must run third.

3. Application Program Scheduling

A schedule for application programs is constructed as follows:

1. Assume that the complete set of data to be saved or transmitted is X, Y, and Z. Assign small integers to each, preferably sequentially. No value should be lower than 4 (values less than this are used by the operating system) and no value should be higher than 119. Using SIFTDEC.REQ as an example (the constants xreset through qz) assume X is assigned 4, Y is assigned 5, and Z is assigned 6. Call these numbers the variable idents.

2. Insert broadcasts for each piece of data needing it into the code for the computing tasks. If the constants have been properly assigned, this should be

almost transparent to the user. To broadcast the value 32 for X, insert the procedure call

```
STOBROADCAST(X,32)
```

3. The value of any variable that has been broadcast, may be obtained by a task needing it by referring to POSTVOTE[<variable>]. For instance, POSTVOTE[X] would contain the value of X if it has been broadcast. As the name implies this is the value after a vote.

4. Table Creation

Schedule tables are created as follows.

1. The module SIFTTA.BDX (implemented in assembly language) contains the precomputed tables for SIFT. The first entry should be a set of EQUates for the symbolic variables that associate their idents with their names. For instance:

```
X     EQU    4
Y     EQU    5
Z     EQU    6
```

2. The next entry is a set of EQUates to assign task names to task numbers. Numbers 0 through 5 are reserved for the operating system, and should be defined as they are in SIFTTA.BDX. Continuing with our example:

```
A     EQU    6
B     EQU    7
C     EQU    8
```

3. The TASK macro has been defined to allow declaration of a state space for each of the tasks. It puts two parameters: the task name (A, B, C) and an offset into the buffer information table (to be described shortly). For example:

```
T6     TASK     A,AOFF
T7     TASK     B,BOFF
T8     TASK     C,COFF
```

4. The buffer information table contains information, used during reconfiguration, that associates tasks and computed values. It is a list of buffer numbers (the variable idents already declared) separated by 0 entries, one such list for each task that computes values. The symbol NLOFF may be used in declaring the task table entry for a task that does not compute any values. The EVENT macro is used in building this and other tables. It takes as its single parameter, the variable ident or task number assigned.

Following is a complete buffer information table (including the operating system part) for our example:

```
STLOC   EQU     *
EROFF   EQU     *-STLOC
        EVENT   ERRER
NLOFF   EQU     *-STLOC
        EVENT   0
*       gexec
GEOFF   EQU     *-STLOC
        EVENT   GEREC
        EVENT   GEMEM
        EVENT   0
AOFF    EQU     *-STLOC
        EVENT   X
        EVENT   0
BOFF    EQU     *-STLOC
        EVENT   Z
        EVENT   0
COFF    EQU     *-STLOC
        EVENT   Y
        EVENT   0
```

5. The next entry is the schedule table. It determines exactly which tasks run during which subframe on which processors. There may be many events scheduled per subframe, or there may be no events scheduled per subframe. In addition to the task schedule there is a schedule that determines when data are to be voted upon. The key problem in building these schedules is to make sure that all dependencies are met. If Task A must be run before X can be voted, it is imperative that the vote schedule so indicate. In addition to EVENT, three other macros exist to aid in creation of these schedules: SFEND is used to indicate the end of a subframe in a schedule: SCHED is used to initialize a particular schedule; it takes four parameters, (a) the number, N, of active processors this schedule uses, (b) which of the N processors this one is, (c) the symbol that refers to the start of this schedule, and (d) the symbol that refers to the end of this schedule (this caters to limitations of the BDX assembler; the special processor number 99 is reserved to indicate the VOTE schedule), and (3) SEND is used to terminate a schedule, even prior to using up all of the allowed subframes. For our example the following would be a legal schedule for the first processor in a one-processor system. Since there are so few tasks, for a larger system all tasks would run on all processors, and each of the schedules would be identical (for a more complex example, see the schedule in SIFTTA.BDX).

```
S11   SCHED  1,1,S11,E11
      EVENT  A       Subframe 0
```

```
         SFEND
         EVENT  C      Subframe 1
         SFEND
         EVENT  B      Subframe 2
         SFEND
         EVENT  ERT    Subframe 3  The error task
         SFEND
         EVENT  GET    Subframe 4  The global executive
         SFEND
         EVENT  RET    Subframe 5  The reconfiguration task
         SFEND
         EVENT  CLT    Subframe 6  The clock task
         SFEND
    E11  SEND

    S199 SCHED  1,99,S199,E199
         SFEND         Subframe 0  (No voting)
         EVENT  X      Subframe 1  (X is available after A runs)
         SFEND
         EVENT  Z      Subframe 2  (Z is available after B runs)
         SFEND
         EVENT  Y      Subframe 3  (Y is available after C runs)
         SFEND
         EVENT  ERRER  Subframe 4  (ERRER is available after
                                              Error task)
         SFEND
         EVENT  GEREC  Subframe 5  (Available after Global Exec)
         EVENT  GEMEM  Subframe 5  (Available after Global Exec)
         SFEND
         EVENT  -1 If there is a "copy" schedule it follows
                             this
    E199 SEND
```

6. The final table needed is the Buffer Table. This is where SIFT holds information about active buffers, etc. (A buffer is essentially a broadcast variable). There should be one entry in the buffer table for each of the variables for which idents have been assigned. The entries should appear in the order in which the idents were assigned. The macro BUF has been defined to take one parameter, the ident assigned to a variable. For our example:

```
    NUL    BUF    0       0      UN-NAMED
           BUF    ERRER   1
           BUF    GEREC   2
           BUF    GEMEM   3
```

```
BUF     X     4
BUF     Y     5
BUF     Z     6
```

These tables must appear in a known location in memory, in order for SIFT to find them. In SIFTDEC.REQ the constants TLOC, BLOC, SLOC, and ILOC define where the compiled version of the operating system expects to find the task, buffer, schedule, and buffer information tables. Constants of similar names are defined in SIFTTA.BDX, and should reference the same locations. See SIFTTA.BDX. for an illustrative case that handles the current guidance and control tasks.

References

[1] Boyer, R. and Moore J.
A Computational Logic.
Academic Press, 1979.

[2] Daly, W.M., Hopkins, A.L., Jr., and McKenna, J.F.
A Fault-Tolerant Digital Clocking System.
The State of the Art: From Device Testing to Reconfigurable Systems, Digest of Papers, Fault Tolerant Computing Symposium (FTC/3) :17-22, Jun, 1973.

[3] Dolev, D.
The Byzantine Generals Strike Again.
Journal of Algorithms 3(1):14-30, 1982.

[4] Forman, P. and Moses, K.
SIFT: Multiprocess Architecture for Software Implemented Fault Tolerance Flight Control and Avionics Computers.
In *3rd Digital Avionics Systems Conference*, pages 325-329. IEEE, 1979.

[5] Goldberg, J.
Logical Design Techniques for Error Control.
In *IEEE Proceedings of the Western Electronic Show and Convention (WESCON) '66.* IEEE, 1966.

[6] Goldberg, J.
SIFT: A Provable Fault-Tolerant Computer for Aircraft Flight Control.
In S.H. Lavington (editor), *Information Processing 80*, , pages 151-156. International Federation for Information Processing (IFIP), 1980.
8th World Computer Congress.

[7] Goldberg, J.
The SIFT Computer and Its Development.
In *Proceedings of the 4th Digital Avionics Systems Conference.* IEEE, Nov, 1981.

[8] Goldberg, J.
The SIFT Approach to Fault Tolerant Computing.
In *Real-Time Signal Processing.* Society of Photo-Optical Instrumentation Engineers (SPIE), 1981.
SPIE 25th Annual International Technical Symposium and Exhibition, V.298.

[9] Goldberg, J., Levitt, K.N., and Wensley, J.H.
An Organization for a Highly Survivable Memory.
IEEE Transactions on Computers C-23(7):693-705, Jul, 1975.

[10] Goldberg, J., Levitt, K.N., and Short, R.A.
Techniques for the Realization of Ultra-Reliable Spaceborne Computers.
Final Report-Phase 1 SRI Project 5580, SRI International, Menlo Park, CA, Sep, 1966.
Prepared for National Aeronautics And Space Administration-Langley Research Center, Contract NAS 12-33.

[11] Hopkins, A.L., Jr.
A Fault-Tolerant Information Processing Concept for Space Vehicles.
IEEE Transactions on Computers C-20(11):1394-1403, Nov, 1971.

[12] Hopkins, A.L., Jr., Smith, T.B., III, and Lala, J.H.
FTMP--A Highly Reliable Fault-Tolerant Multiprocessor for Aircraft.
In *Proceedings of the IEEE,* pages 1221-1239. IEEE, Oct, 1978.

[13] Igarashi, S., London, R., Luckham, D.
Automatic Program Verification I: A Logical Basis and its Implementation.
Acta Informatica 4:145-182, 1975.

[14] Isner, J.F.
A Fortran Programming Methodology Based on Data Abstraction.
Communications of the ACM 25(10), Oct, 1982.

[15] Lamport, L.
The Implementation of Reliable Distributed Multiprocess Systems.
Computer Networks 2:95-114, 1978.

[16] Lamport, L.
Using Time Instead of Timeout for Fault-tolerant Distributed Systems.
To appear in *ACM Tran. on Prog. Lang. and Sys.*.

[17] Lamport, L. and Melliar-Smith, P.M.
Synchronizing Clocks in the Presence of Faults.
CSL Technical Report 141, SRI International, 1982.
Submitted to *The Journal of the ACM,* revised from 1981 edition.

[18] Lamport, L., Shostak, R., and Pease, M.
The Byzantine Generals Problem.
ACM Tran. on Prog. Lang. and Sys. 4(3):382-401, Jul, 1982.

[19] Levitt, K.N., Neumann, P.G., and Robinson, L.
The SRI Hierarchical Development Methodology (HDM) and its Application to the Development of Secure Software.
Technical Report, National Bureau of Standards Special Publication 500-67, 1980.

[20] Melliar-Smith, P.M. and Schwartz, R.L.
Hierarchical Specifications of the SIFT Fault Tolerant Flight Control System.
AGARD conference Proceedings: Tactical Airborne Distributed Computing and Networks (303):22.1-22-15, 1981.

[21] Melliar-Smith, P.M. and Swartz, R.L.
Formal Specification and Mechanical Verification of SIFT: A Fault-Tolerant Flight Control System.
IEEE Transactions on Computers C-31(7):616-630, Jul, 1982.

[22] Melliar-Smith, P.M., et al.
Investigation, Development, and Evaluation of Performance Proving for Fault-Tolerant Computer.
Final Report, Contract No NAS1-15528, SRI International, Menlo Park, CA, Jul, 1982.

[23] Murray, N.D., Hopkins, A.L., and Wensley, J.H.
Highly Reliable Multiprocessors.
In Kurzhals, P.R. (editor), *Integrity in Electronic Flight Control Systems*, pages 17.1-17.16. Advisory Group for Aerospace Research and Development, AGARD, 1977.
AGARDograph No. 224.

[24] Pease, M., Shostak, R., and Lamport, L.
Reaching Agreements in the Presence of Faults.
Journal of the Association for Computing Machinery 27(2):228-234, Apr, 1980.

[25] Ratner, R.S., et al.
Design of a Fault Tolerant Airborne Digital Computer.
Technical Report National Aeronautics And Space Administration CR-132253, SRI International, 1973.
Vol. II, Computational Requirements and Technology, Final Report.

[26] Robinson, L., Levitt, K.N., Neumann, P.G., and Saxena, A.R.
A Formal Methodology for the Design of Operating System Software.
In *Software Specification and Design.* Volume I: *Current Trends in Programming Methodology*, . Prentice Hall, Englewood Cliffs, N.J., 1977.

[27] Robinson, L., Silverberg, B., and Levitt, K.N.
The HDM Handbook.
Technical Report, SRI International, 1979.
Volumes I, II, and III.

[28] Shostak, R.
Deciding Combinations of Theories.
In *6th Conference on Automated Deduction.* International Federation for Information Processing (IFIP), Jun, 1982.

[29] Shostak, R., Schwartz, R. and Melliar-Smith, P.M.
STP: A Mechanized Logic for Specification and Verification.
In *6th Conference on Automated Deduction.* International Federation for Information Processing (IFIP), Jun, 1982.

[30] Weinstock, C.B.
SIFT: System Design and Implementation.
Fault Tolerant Computing Symposium-10, IEEE Computer Society International Conference in Kyoto, Japan, October 1-3, 1980.

[31] Wensley, J.H.
SIFT Software Implemented Fault Tolerance.
In *Conference Proceedings Fall Joint Computer Conference, 1972* , pages 243-253. American Federation of Information Processing Societies (AFIPS) Press, 1972.
Volume 41.

[32] Wensley, J.H., et al.
Design of a Fault-Tolerant Airborne Digital Computer.
Technical Report National Aeronautics And Space Administration CR-132252, SRI International, 1973.
Vol. I, Architecture, Final Report.

[33] Wensley, J.H., et al.
The Design, Analysis, and Verification of the SIFT Fault-Tolerant System.
In *Proceedings 2nd International Conference on Software Engineering*, pages 458-469. IEEE Computer Society, 1976.

[34] Wensley, J.H., et al.
SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control.
Proceedings of the IEEE 66(10):1240-1255, Oct, 1978.
Also in: Siewiorek, D.P. and Swarz, R.S., The Theory and Practice of Reliable System Design, Digital Press, 1982.

[35] Wensley, J.H., et al.
Design Study of Software-Implemented Fault-Tolerance (SIFT) Computer.
Technical Report National Aeronautics And Space Administration Contractor Report 3011, Contract No. NAS1-13792, SRI International, Jun, 1982.

1. Report No. NASA CR-172146	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle Development and Analysis of the Software Implemented Fault-Tolerance (SIFT) Computer		5. Report Date February 1984
		6. Performing Organization Code
7. Author(s) Jack Goldberg, William H. Kautz, P. Michael Melliar-Smith, Milton W. Green, Karl N. Levitt, Richard L. Schwartz, and Charles B. Weinstock		8. Performing Organization Report No.
9. Performing Organization Name and Address SRI International 333 Ravenswood Avenue Menlo Park, CA 94025		10. Work Unit No.
		11. Contract or Grant No. NAS1-15428
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Washington, DC 20546		13. Type of Report and Period Covered Contractor Report
		14. Sponsoring Agency Code

15. Supplementary Notes
Langley technical monitor: Charles W. Meissner, Jr.
Final Report

16. Abstract
SIFT (Software Implemented Fault Tolerance) is an experimental, fault-tolerant computer system designed to meet the extreme reliability requirements (e.g., 10^{-9} probability of failure in a ten-hour flight) for safety-critical functions in advanced aircraft. Errors are masked by performing a majority voting operation over the results of identical computations, and faulty processors are removed from service by reassigning computations to the nonfaulty processors. This scheme has been implemented in a special architecture using a set of standard Bendix BDX930 processors, augmented by a special asynchronous-broadcast communication interface that provides direct, processor to processor communication among all processors. Fault isolation is accomplished in hardware; all other fault-tolerance functions, together with scheduling and synchronization are implemented exclusively by executive system software. The system reliability is predicted by a Markov model. Mathematical consistency of the system software with respect to the reliability model has been partially verified, using recently developed tools for machine-aided proof of program correctness. General results have been obtained for achieving consistent replication of data among multiple processors in the presence of faults. A general procedure for testing general-purpose fault-tolerant computers for the purpose of estimating failure and recovery rates was developed and partially applied. The system is operational and has been delivered to the AIRLAB facility at NASA Langley Research Center.

17. Key Words (Suggested by Author(s)) Reliable flight control, fault tolerant computing, software-implemented fault tolerance, reliability modeling, design verification, proof of correctness, data consistency, clock synchronization, fault-tolerant power supply		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 214	22. Price A10