NASA Contractor Report **182004**

Design of an Integrated Airframe/ Propulsion Control System Architecture

G. C. Cohen
Boeing Advanced Systems
Seattle, Washington

C. W. Lee
Boeing Advanced Systems
Seattle, Washington

M. J. Strickland
Boeing Advanced Systems
Seattle, Washington

NASA Contract NAS1-18099
March 1990

[redacted]

Date for general release March 31, 1992

National Aeronautics and
Space Administration
Langley Research Center
Hampton, Virginia 23665

PREFACE

This report describes the design of an embedded architecture for an integrated flight/propulsion control system. The design procedure is based on a prevalidation methodology. This work has been supported under NASA contract NAS1-10899, Integrated Airframe/Propulsion Control System Architecture (IAPSA II).

The NASA technical monitor for this work is Daniel L. Palumbo of the NASA Langley Research Center, Hampton, Virginia.

The work was accomplished by the Flight Controls Technology organization at Boeing Advanced Systems in Seattle, Washington. Personnel responsible for the work performed include:

D. Gangsaas	Responsible manager
T. M. Richardson	Program manager
G. C. Cohen	Principal investigator
C. W. Lee	System design and reliability analysis
M. J. Strickland	Performance analysis

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF FIGURES (Continued)

LIST OF FIGURES (Continued)

LIST OF FIGURES (Continued)

LIST OF TABLES

1.0 SUMMARY

During the detailed design effort for the IAPSA II contract, a candidate architecture design based on AIPS fault-tolerant system building blocks was evaluated for its ability to meet the demanding performance and reliability requirements of a flight-critical system. This effort was conducted in accordance with the IAPSA II prevalidation methodology. This methodology was defined and an advanced fighter configuration was selected during an earlier phase of this contract. A mission analysis of the high-performance, multirole, twin-engine fighter was conducted to define a set of flight-critical requirements for this study during the earlier effort.

The preliminary evaluations showed that the candidate needed some refinements to meet the system requirements. It is significant that several weaknesses in the candidate architecture became apparent that were not evident in the initial rough performance and reliability calculations. This effort shows that it is both possible and preferable to perform detailed evaluation of concepts based on specifications before committing a project to a hardware and software design.

The refined configuration was evaluated for reliability using improved Markov modeling techniques. Although this was much better than earlier evaluation techniques, improvements are needed in the handling of very large systems with a high degree of interdependency.

A set of objectives and experiments was defined for testing critical performance characteristics using a small-scale system. The small-scale system consists of existing proof-of-concept AIPS building-block hardware and software components. It embodies key features of the IAPSA II design and will be used to explore issues identified as a result of the performance and reliability modeling effort. Experimental data will be obtained that verify performance estimates obtained during the preliminary simulation effort and measure timing characteristics critical to successful operation.

2.0 INTRODUCTION

The purpose of this report is to document the results of the detailed design of the IAPSA II system. This effort was carried out using the prevalidation methodology developed during an earlier phase of this contract. This report discusses the preliminary simulation experiments and the reliability evaluation of the candidate architecture and shows how they affect the resulting design. Additionally, a small-scale system that captures the fundamental characteristics of the IAPSA II design and a plan for using it in AIRLAB experimentation are defined.

The IAPSA II analysis and design effort is the continuation of a research and technology program to investigate the benefits of integrated system architectures and to demonstrate by experimentation in the NASA Langley Avionics Integration Research Laboratory (AIRLAB) the properties of promising architectures. Work under previous contracts achieved the following: (1) defined major characteristics of an Integrated Airframe Propulsion Control System Architecture, (2) proposed several candidate system configurations, and (3) selected one as a basis for a preliminary system design.

The overall objectives of the IAPSA II program are (1) analysis and detailed design of an integrated control system architecture that satisfies stringent performance and reliability requirements, (2) an analytical and experimental approach for evaluating the architecture, and (3) installation and limited experimentation on a small scale system test specimen in AIRLAB.

The first phase of this contract, documented in reference 1, defined an advanced fighter configuration for analysis, a prevalidation methodology, and a candidate architecture based on the use of fault-tolerant system building blocks. The advanced fighter is a twin-engine design with a high degree of coupling between the propulsion system and the airframe. A mission analysis was conducted on mission scenarios for this fighter to

PRECEDING PAGE BLANK NOT FILMED

PAGE 2 INTENTIONALLY BLANK

derive the control system requirements. These requirements formed the basis for the design of a control system architecture.

The methods used to design and validate the control system architecture are as important to the IAPSA II contract as the architecture itself. The prevalidation methodology defined in reference 1 emphasizes the early evaluation of key performance and reliability characteristics of system concepts using models of system behavior. This early evaluation ensures that the system design is capable of meeting critical requirements. System concept changes needed to meet these requirements can then be made early when they have the greatest performance benefit and the least impact on schedule and cost. Key performance and reliability assumptions identified by the modeling effort will be tied to activities to validate the implemented system.

A candidate system architecture defined by our subcontractor, Charles Stark Draper Laboratory (CSDL), was evaluated to exercise the methodology. The basic outline of the candidate architecture is provided in chapter five of reference 1. The more detailed definition of the candidate system necessary to support the modeling effort is presented in sections 3 and 4. The reliability evaluation effort was accomplished in four parts: (1) system operating details and key reliability assumptions were defined to support system modeling; (2) based on the key reliability measures (safety, mission success, etc.), a failure analysis was conducted to define how the system fails; (3) the ASSIST program was used to create a corresponding failure model; and (4) the SURE model was executed and its results used to indicate the candidate's strengths and weaknesses. The reliability effort is covered in section 3.

The performance characteristics of the candidate architecture were evaluated in normal and failure situations as required by the prevalidation methodology. The evaluation effort consisted of four major parts: (1) the key application sequencing and control options in the candidate system were defined; (2) critical performance issues and simulation experiments were defined for the candidate configuration; (3) a model of the critical system workload and its use of the configuration elements was built using the DENET tool; and (4) the DENET experiments were executed and the results analyzed. This performance evaluation effort is described in section 4 of this report.

The candidate system evaluation described in sections 3 and 4 showed that it was not capable of meeting the system requirements. The predicted safety and mission reliability values exceeded the system constraints. Additionally, the predicted timing needs of the major control functions executing on the concept system did not leave adequate growth capability. The flight control group application workload strained the system capacity in both computing and I/O activity. As a result the IAPSA II system concept was refined to improve its performance and reliability. The three approaches taken to refine the candidate architecture to better match the system needs are described in section 5. The results of a detailed reliability evaluation of the refined configuration and a preliminary performance analysis are also presented in this section.

Section 6 outlines the objectives and presents experiment definitions for the small-scale system testing effort. The small-scale system embodies key features of the IAPSA II design that will be evaluated in a limited experimentation effort. The limited effort will explore a set of critical aspects of the IAPSA II candidate architecture that were identified as a result of the performance and reliability modeling effort. The small-scale system consists of existing proof-of-concept AIPS building-block hardware and software components. Two kinds of experimental data will be obtained. First, certain performance estimates obtained during the preliminary simulation effort will be verified. Performance simulation results will be directly compared with applicable measurements made on the small-scale system. Second, certain timing characteristics critical to successful operation in normal and faulted situations will be measured experimentally.

One limitation of this small-scale system, of course, is that system level interactions (e.g., communication between the flight control group and the engine groups) cannot be tested. Another limitation is that the experiments must make use of existing system testability features. Because existing building block elements are used, no new "hooks" or test points can be used to enhance visbility during experimentation.

3.0 CANDIDATE ARCHITECTURE EVALUATION

The prevalidation methodology defined in reference A emphasizes the early evaluation of key performance and reliability characteristics of system concepts using models of the system behavior. Early evaluation ensures that the system design is capable of meeting critical requirements. System concept changes needed to meet these requirements can then be made early when they have the greatest performance benefit and the least effect on schedule and cost. In the IAPSA II detailed design effort a candidate system architecture defined by our subcontractor, Charles Stark Draper Laboratory (CSDL), was evaluated to exercise the methodology. The basic outline of the candidate architecture was provided in chapter five of reference A. A more detailed definition of the candidate system necessary to support the modeling effort is developed in sections 3 and 4. Section 3 deals with reliability issues and section 4 with performance issues.

To support system modeling, three key aspects of the system must be defined: (1) function partitioning, (2) physical and functional interconnection, and (3) failure protection. Function partitioning defines how the system functions (or processes) are allocated to system configuration elements. Interconnection defines how the system elements are interrelated. Finally, failure protection describes how the critical system functions are preserved when system elements fail. The following sections will provide elaboration of these characteristics for the candidate architecture.

3.1 Development of Major Control Functions

The IAPSA II system performs control functions that are critical to the flight safety and mission effectiveness of an advanced fighter. Table 3.1-1 presents the major control functions developed for the vehicle in reference 1. These functions impose demanding performance and reliability requirements on the system. Furthermore, the designed system must have the capacity to handle the demanding workload of these functions in normal situations and in failure situations where a specified level of performance

Table 3.1-1. IAPSA II Major Control Functions

System functions	Capabilities	Needed for:
Manual control	Basic flight path control	SFL
Flutter control	High speed ingress with stores	FMC
Trajectory following	Track optimized flight paths	FMC
Wing camber control	Optimized wing performance for mission segment	FMC
Trim control		FMC
Inlet control	Full supersonic capability	FMC
Engine control		SFL
Nozzle control	Thrust vectoring/reversing	FMC

Notes:
SFL: Safe flight and landing
FMC: Full mission capability

must be provided after faults. The specific reliability goals are that (1) the system contribution to loss of aircraft probability will be less than 10-7 for a 3-hour flight, and (2) the system contribution to loss of mission probability will be less than 10-4. The major control functions are developed in more detail in the following sections.

Table 3.1-1 specifies the high-level capability provided by each major control function. Two of the major functions, manual control and engine control, are needed to allow continued flight to a safe landing. The rest of the functions are needed to provide full mission capability. No attempt was made during the reliability study to distinguish lesser or intermediate levels of mission capability. The effects of system element failures and combinations of failures were categorized in terms of the three system failure conditions: safe flight and landing (SFL), fully mission capable (FMC), and unsafe.

The major control functions have specific sensing and actuation requirements. These requirements are shown for each of the major control functions in figures 3.1-1 through 3.1-8. The figures also list the required cyclic execution rate derived in the function development effort documented in reference 1. The major control functions were decomposed into subfunctions based on this information. The resulting subfunction definition and data transfer details are presented in the following two subsections.

3.1.1 Derivation of Subfunctions

The IAPSA II functional design was developed in more detail by decomposing the major control functions into subfunctions. The design at the subfunction level identifies the sensor and actuator redundancy management processes. The detailed development was based on several ground rules. A major design ground rule is the sharing of sensors and computing processes among the major functions. If more than one function requires a particular computed parameter, it is computed only once.

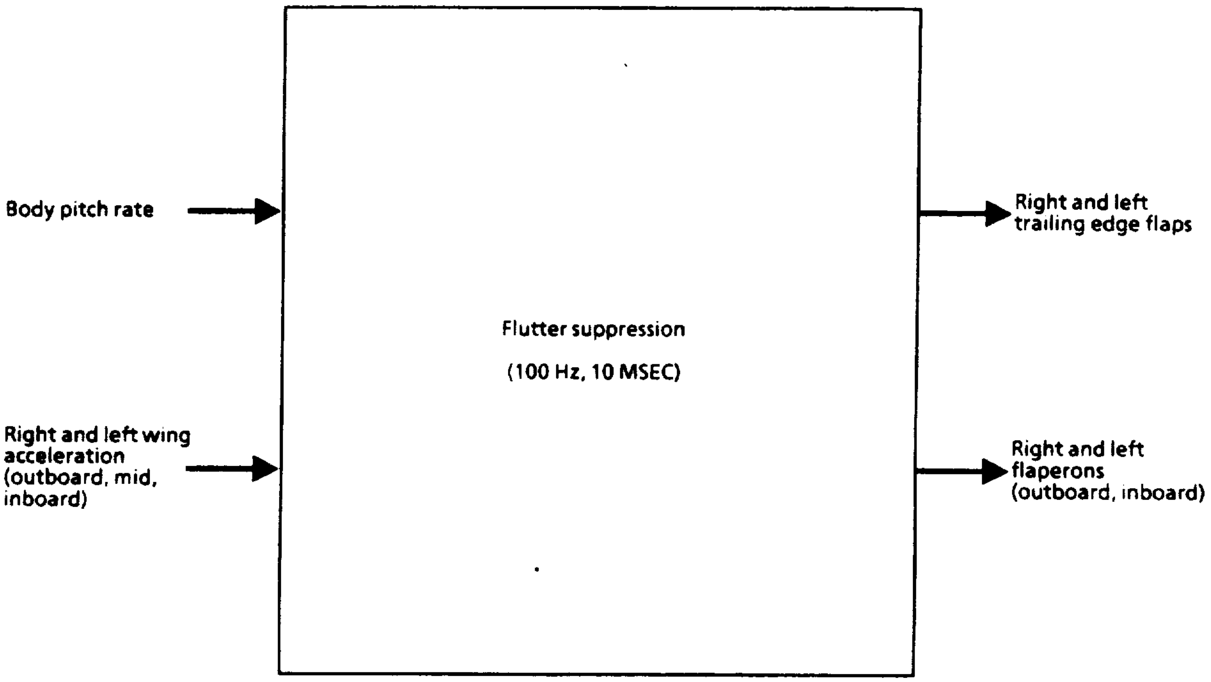

Body pitch rate
Right and left wing acceleration (outboard, mid, inboard)
Flutter suppression
(100 Hz, 10 MSEC)
Right and left trailing edge flaps
Right and left flaperons (outboard, inboard)

Figure 3.1-1. Flutter Control Function

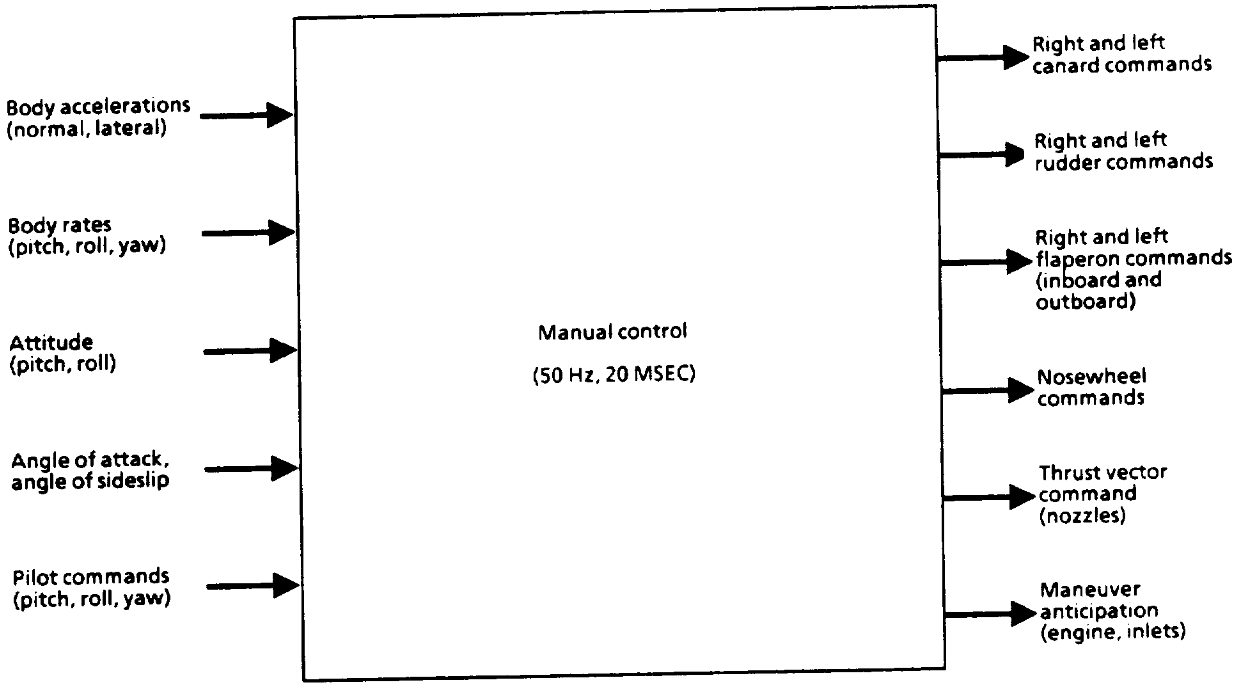


Figure 3.1-2. Manual Control Function

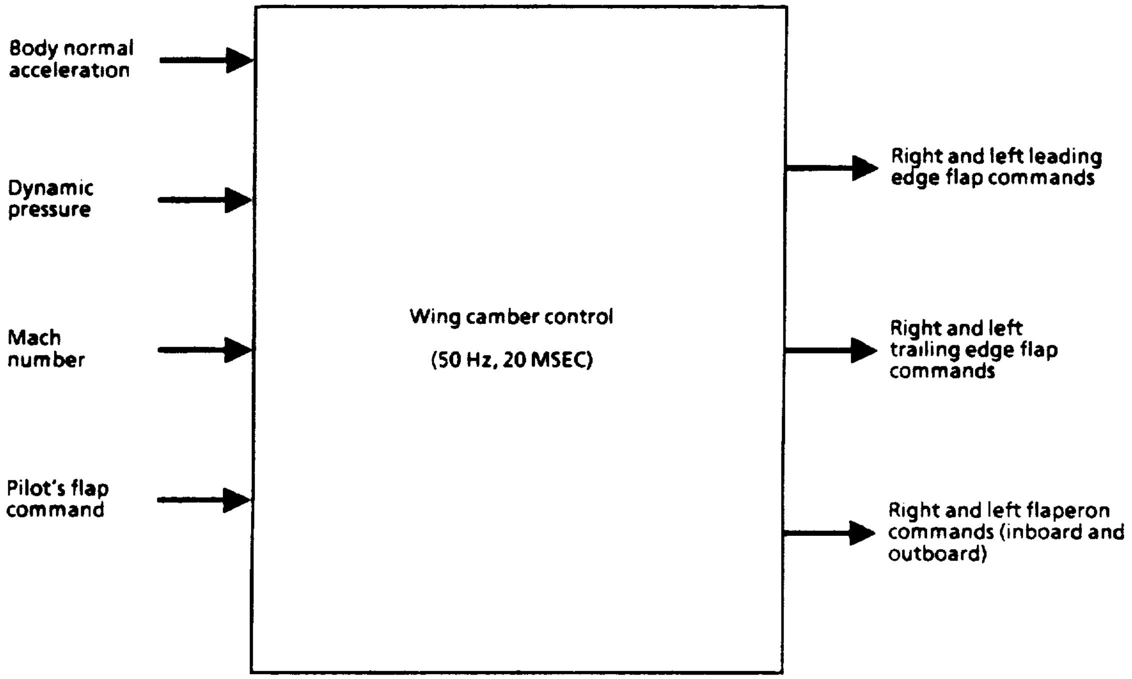

Body normal acceleration
Dynamic pressure
Mach number
Pilot's flap command
Wing camber control
(50 Hz, 20 MSEC)
Right and left leading edge flap commands
Right and left trailing edge flap commands
Right and left flaperon commands (inboard and outboard)

Figure 3.1-3. Wing Camber Control Function

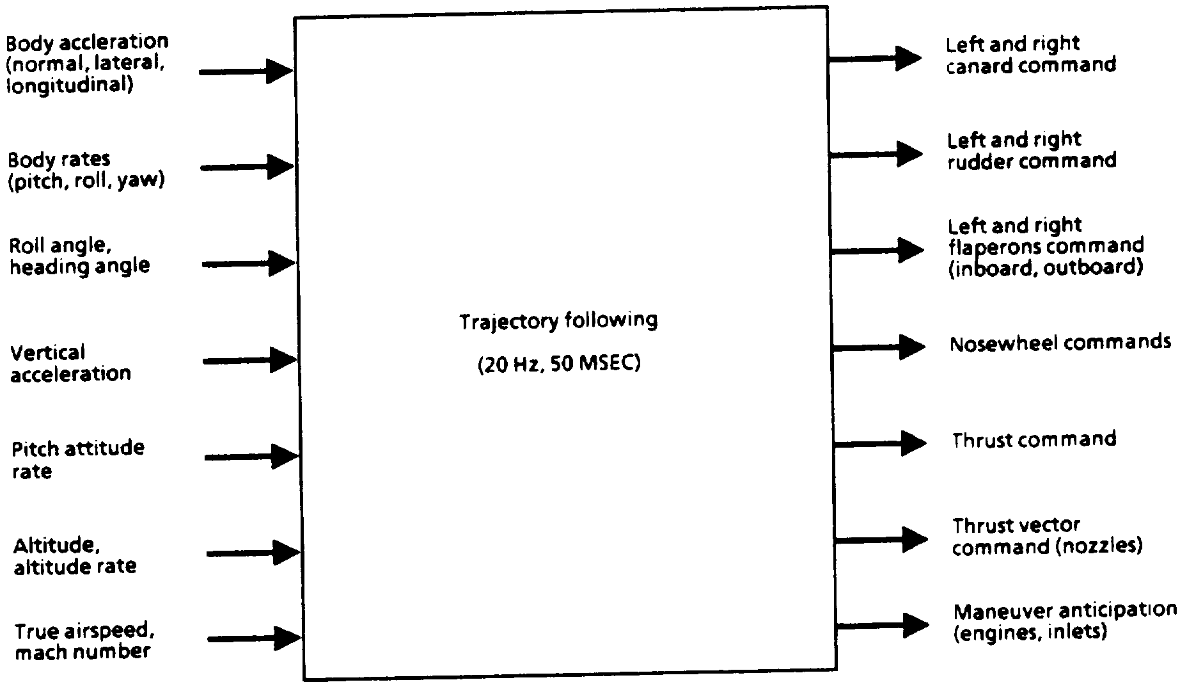

Body accleration (normal, lateral, longitudinal)
Body rates (pitch, roll, yaw)
Roll angle, heading angle
Vertical acceleration
Pitch attitude rate
Altitude, altitude rate
True airspeed, mach number
Trajectory following
(20 Hz, 50 MSEC)
Left and right canard command
Left and right rudder command
Left and right flaperons command (inboard, outboard)
Nosewheel commands
Thrust command
Thrust vector command (nozzles)
Maneuver anticipation (engines, inlets)

Figure 3.1-4. Trajectory Following Function

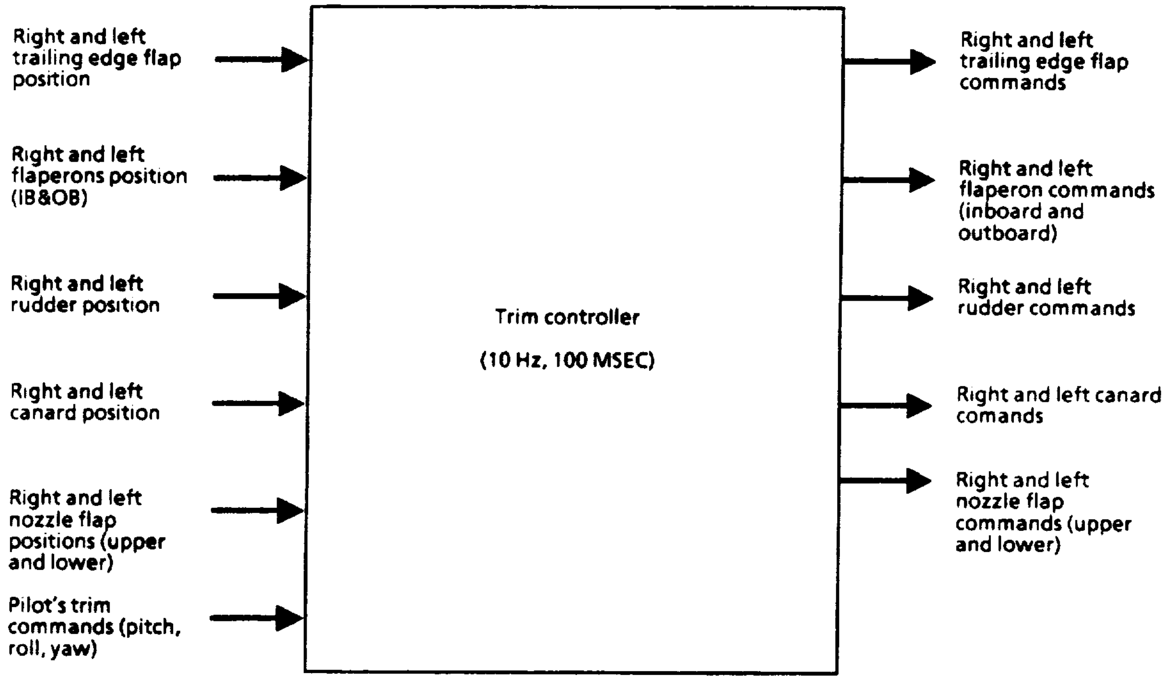

Right and left trailing edge flap position
Right and left flaperons position (IB&OB)
Right and left rudder position
Right and left canard position
Right and left nozzle flap positions (upper and lower)
Pilot's trim commands (pitch, roll, yaw)
Trim controller
(10 Hz, 100 MSEC)
Right and left trailing edge flap commands
Right and left flaperon commands (inboard and outboard)
Right and left rudder commands
Right and left canard comands
Right and left nozzle flap commands (upper and lower)

Figure 3.1-5. Trim Control Function

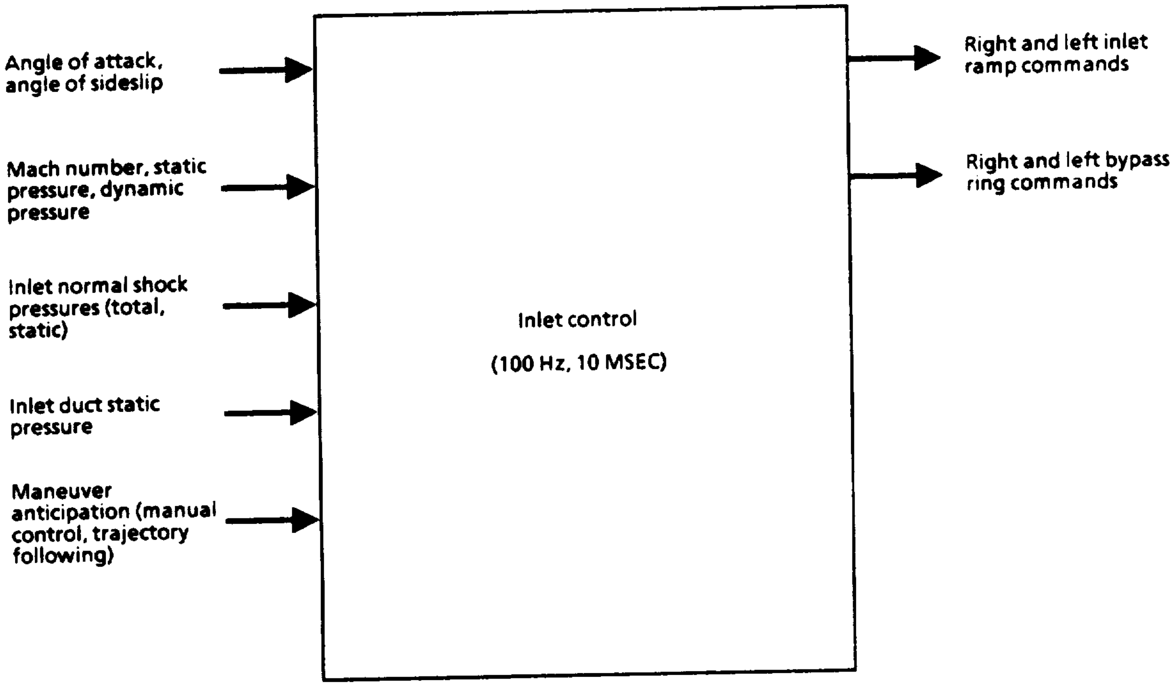

Angle of attack, angle of sideslip
Mach number, static pressure, dynamic pressure
Inlet normal shock pressures (total, static)
Inlet duct static pressure
Maneuver anticipation (manual control, trajectory following)
Inlet control
(100 Hz, 10 MSEC)
Right and left inlet ramp commands
Right and left bypass ring commands

Figure 3.1-6. Inlet Control Function

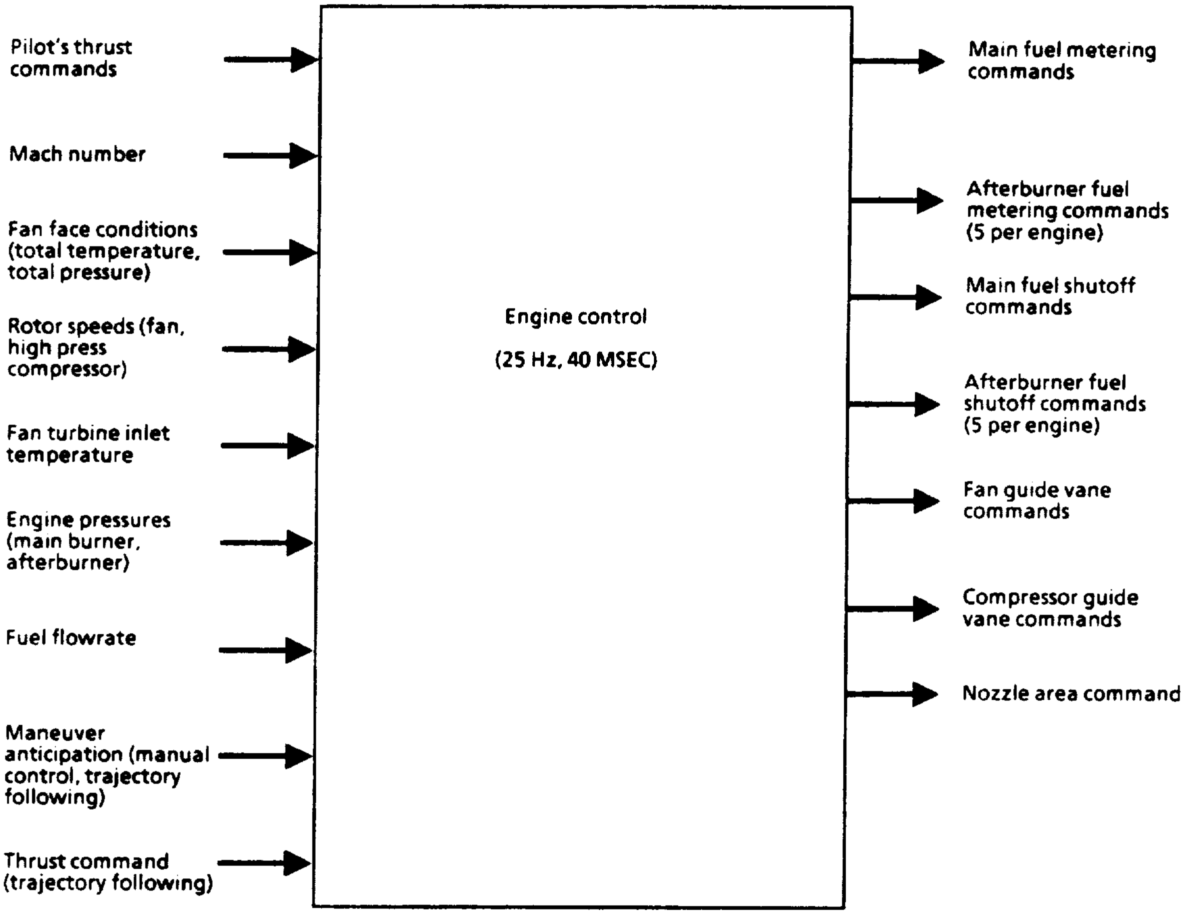

Pilot's thrust commands
Mach number
Fan face conditions (total temperature, total pressure)
Rotor speeds (fan, high press compressor)
Fan turbine inlet temperature
Engine pressures (main burner, afterburner)
Fuel flowrate
Maneuver anticipation (manual control, trajectory following)
Thrust command (trajectory following)
Engine control
(25 Hz, 40 MSEC)
Main fuel metering commands
Afterburner fuel metering commands (5 per engine)
Main fuel shutoff commands
Afterburner fuel shutoff commands (5 per engine)
Fan guide vane commands
Compressor guide vane commands
Nozzle area command

Figure 3.1-7. Engine Control Function

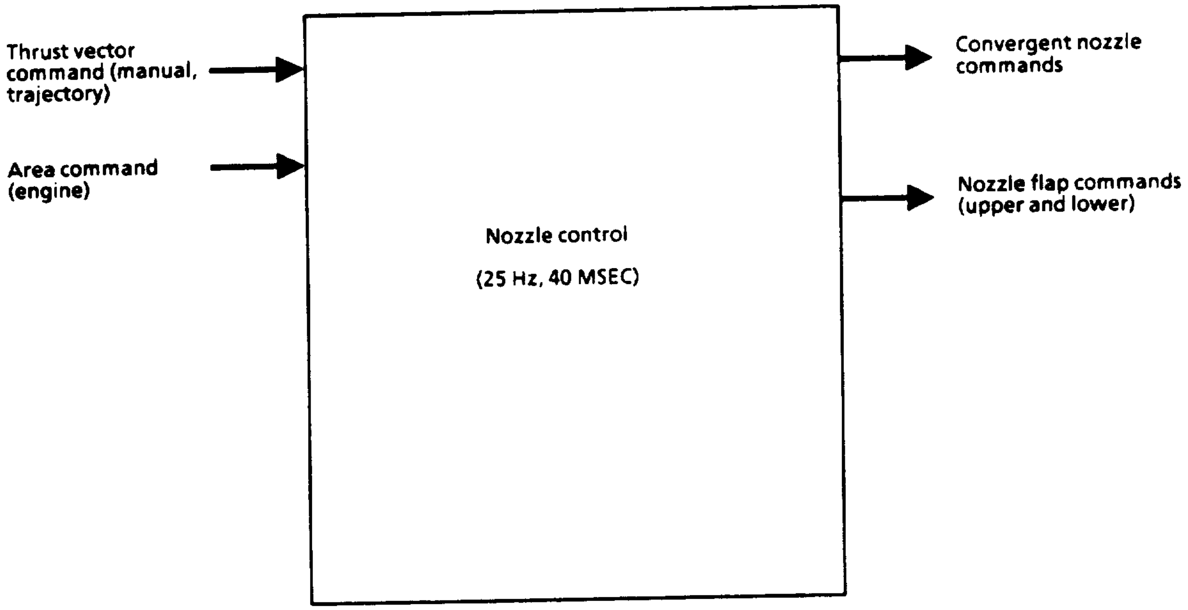

Thrust vector command (manual, trajectory)
Area command (engine)
Nozzle control
(25 Hz, 40 MSEC)
Convergent nozzle commands
Nozzle flap commands (upper and lower)

Figure 3.1-8. Nozzle Control Function

The shared elements must satisfy the requirements of all dependent functions. This means that a shared sensor must have adequate performance and adequate failure protection for its most critical user. For shared computing processes, the most demanding function defines the failure protection requirement and also sets the cyclic execution rate. For this study, the rate used for all derived subfunctions and sensor and actuator signaling is based on the fastest control function. Detailed control law performance analysis might show that some of these rates can be relaxed; however, because control law design and analysis is beyond the scope of the contract, this possibility was not investigated.

The resulting central computing subfunctions are presented in figures 3.1.1-1 through 3.1.1-15.

The first group of figures (3.1.1-1 through 3.1.1-8) are organized by major control function. The sensor management computing subfunctions shown perform the redundancy management for sensor failures. The process provides parameter estimates to all using functions. The actuator management functions shown on the figures have two purposes: first, they combine actuator commands from all control law functions that share the control surface or device; second, they perform any necessary central actuator redundancy management. Failure protection details are covered in section 3.3.

The next two figures (3.1.1-9 and 3.1.1-10) detail the inertial data and air data sensor processing subfunctions. Both processes have subfunctions that execute at different rates to support different users. Air data computing is divided into two rates. The fast rate provides the sensor redundancy management and the computing needed by the inlet control function. The slow rate provides the redundancy management functions for the rest of the sensors and calculated variables needed by the trajectory-following function. Inertial data has a fast subfunction that does some redundancy management and estimation necessary for the flutter law. The rest of the inertial data processing can be done at a slower rate. Note that these figures only show the interfaces for the integrated control functions. In a complete system, air data and inertial data are also needed by the pilot display functions of the pilot-vehicle interface.

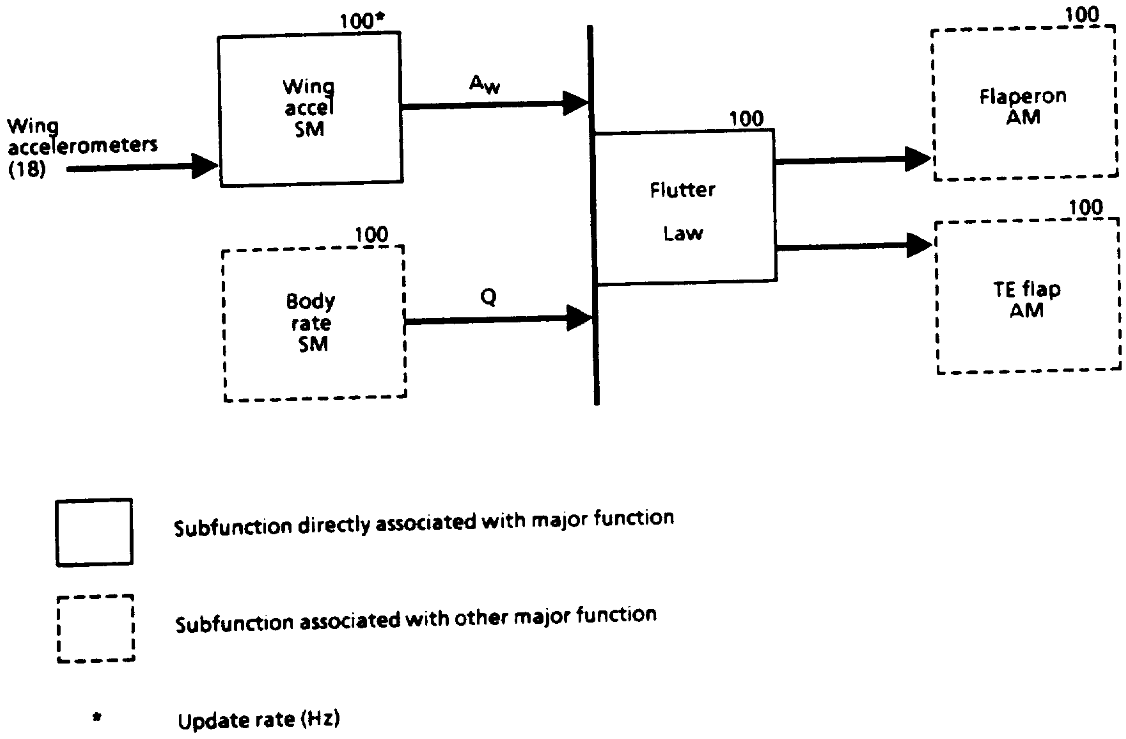


Figure 3.1.1-1. Flutter Subfunctions

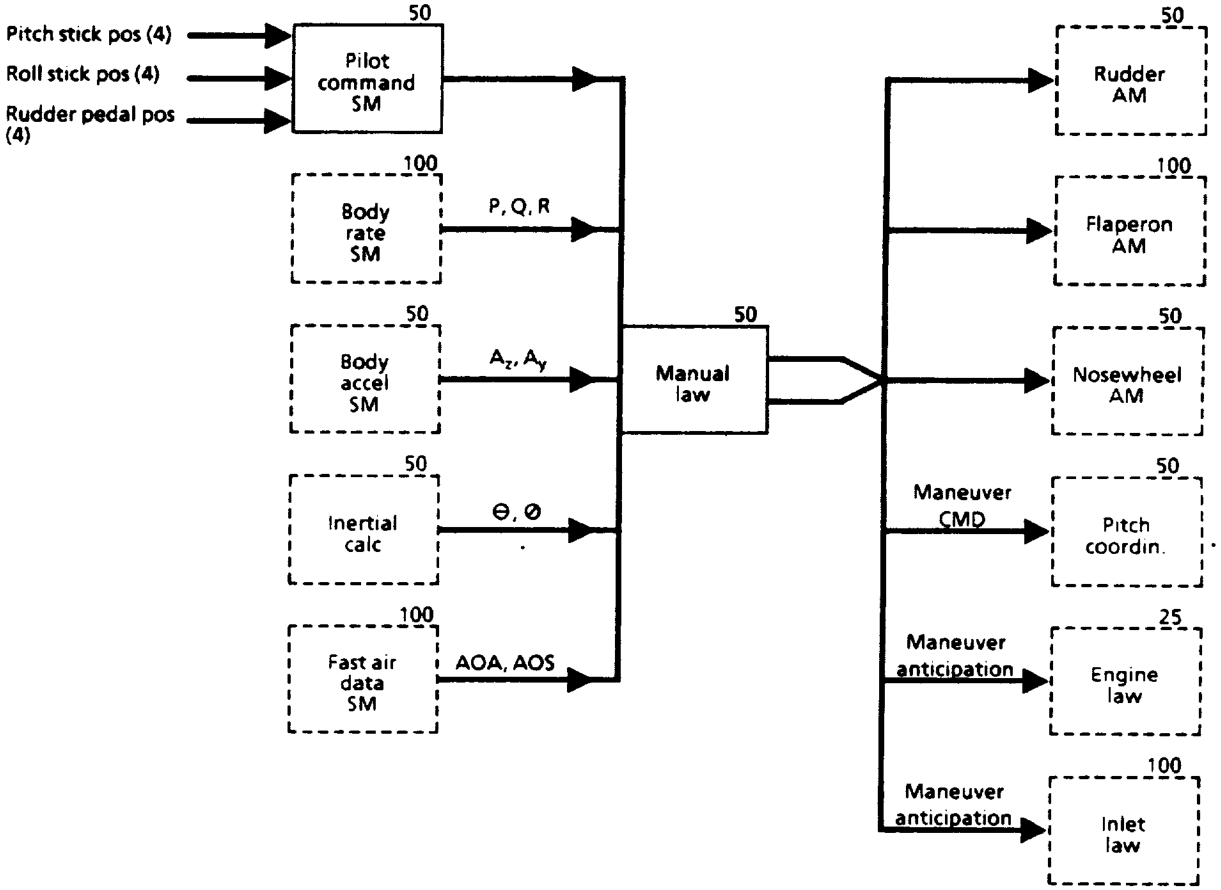


Figure 3.1.1-2. Manual Control Subfunctions

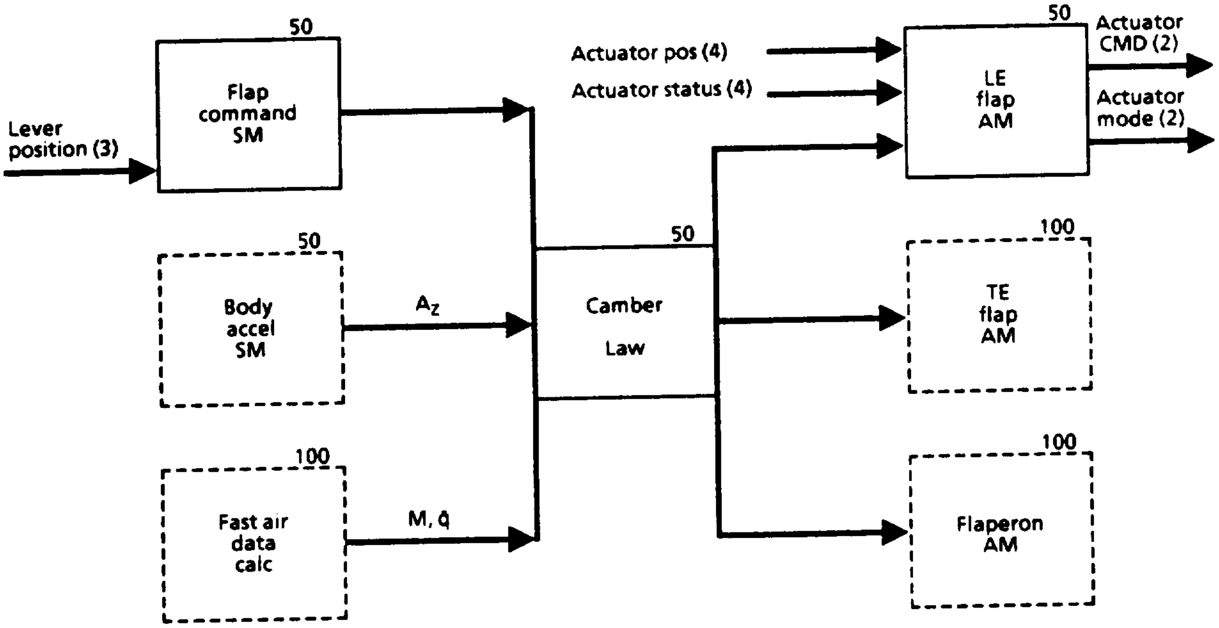


Figure 3.1.1-3. Camber Subfunctions

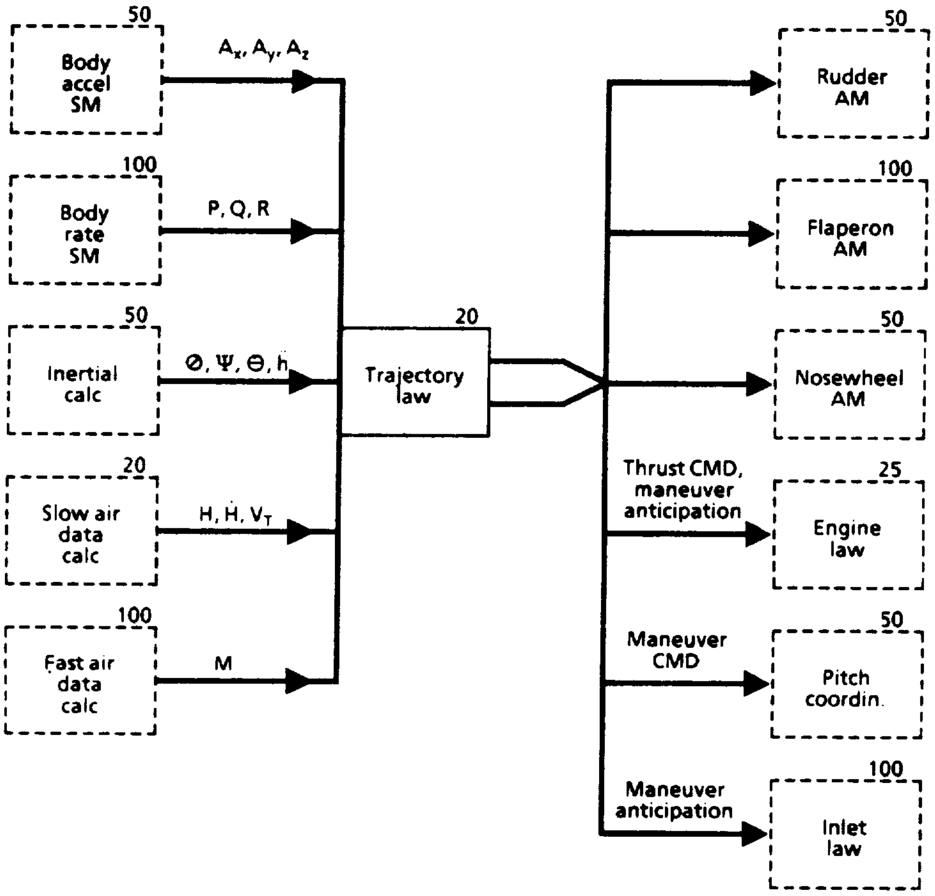


Figure 3.1.1-4. Trajectory Subfunctions

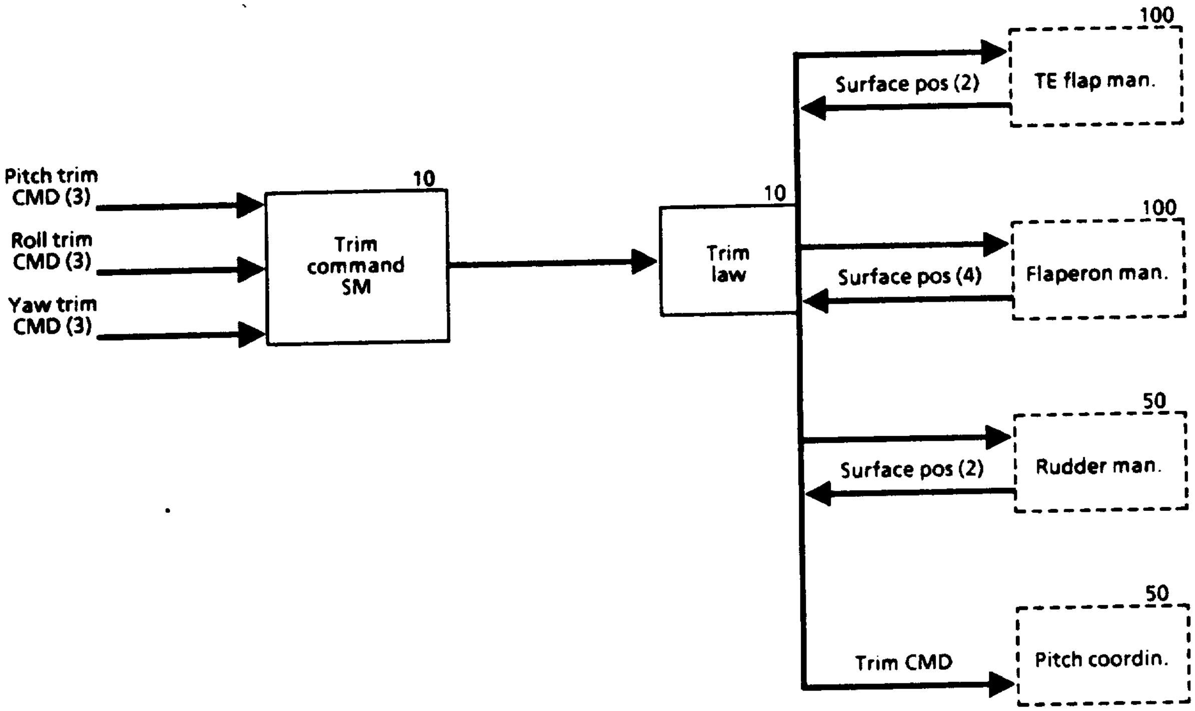


Figure 3.1.1-5. Trim Subfunctions

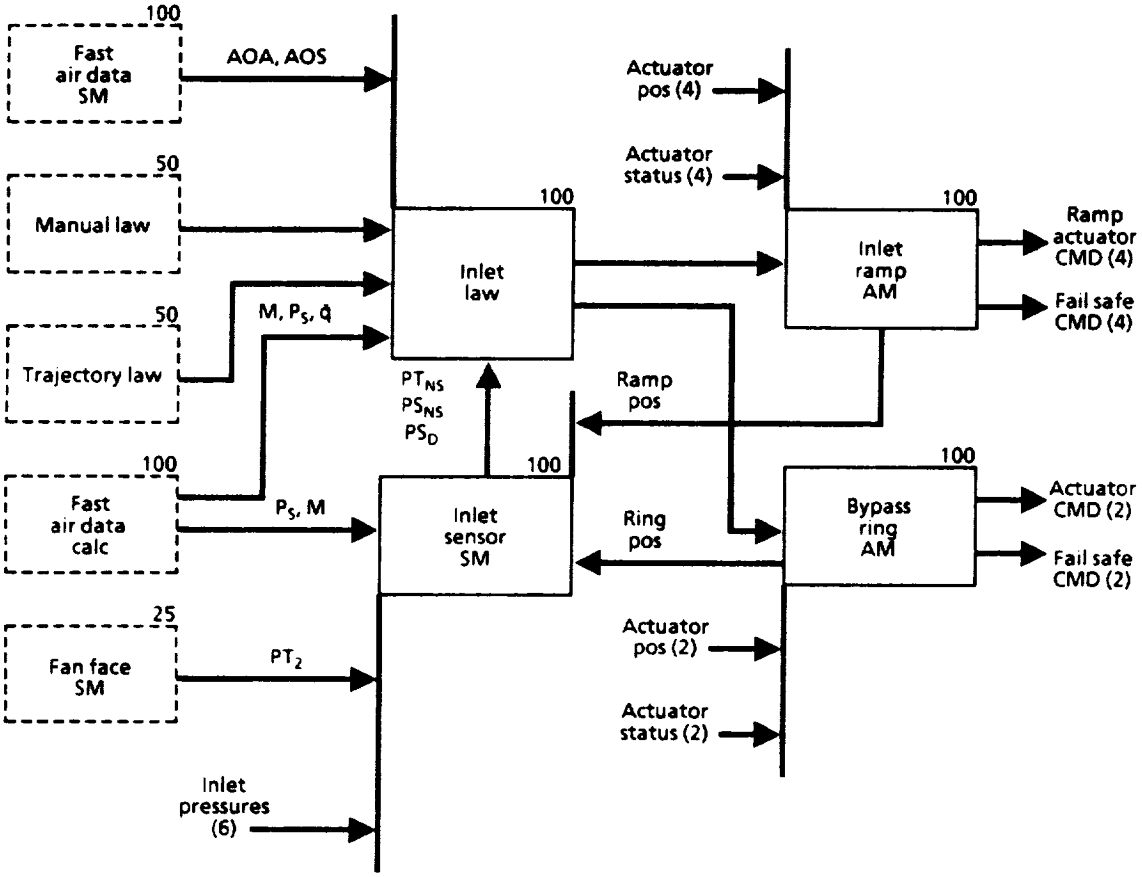


Figure 3.1.1-6. Inlet Subfunctions

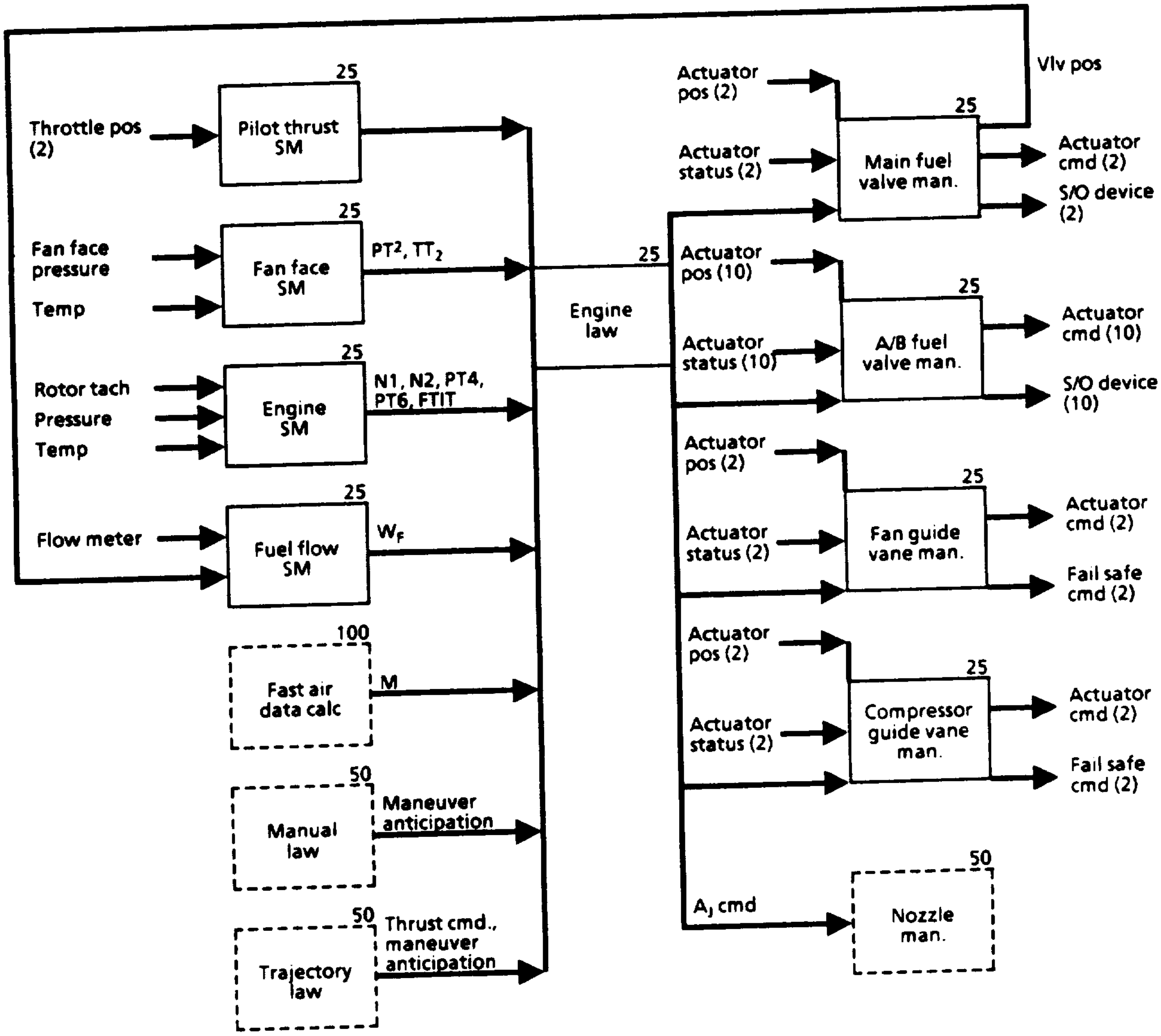


Figure 3.1.1-7. Engine Subfunctions

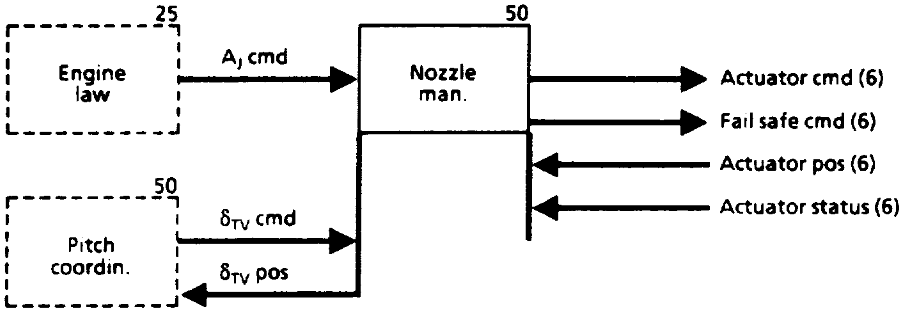


Figure 3.1.1-8. Nozzle Subfunctions

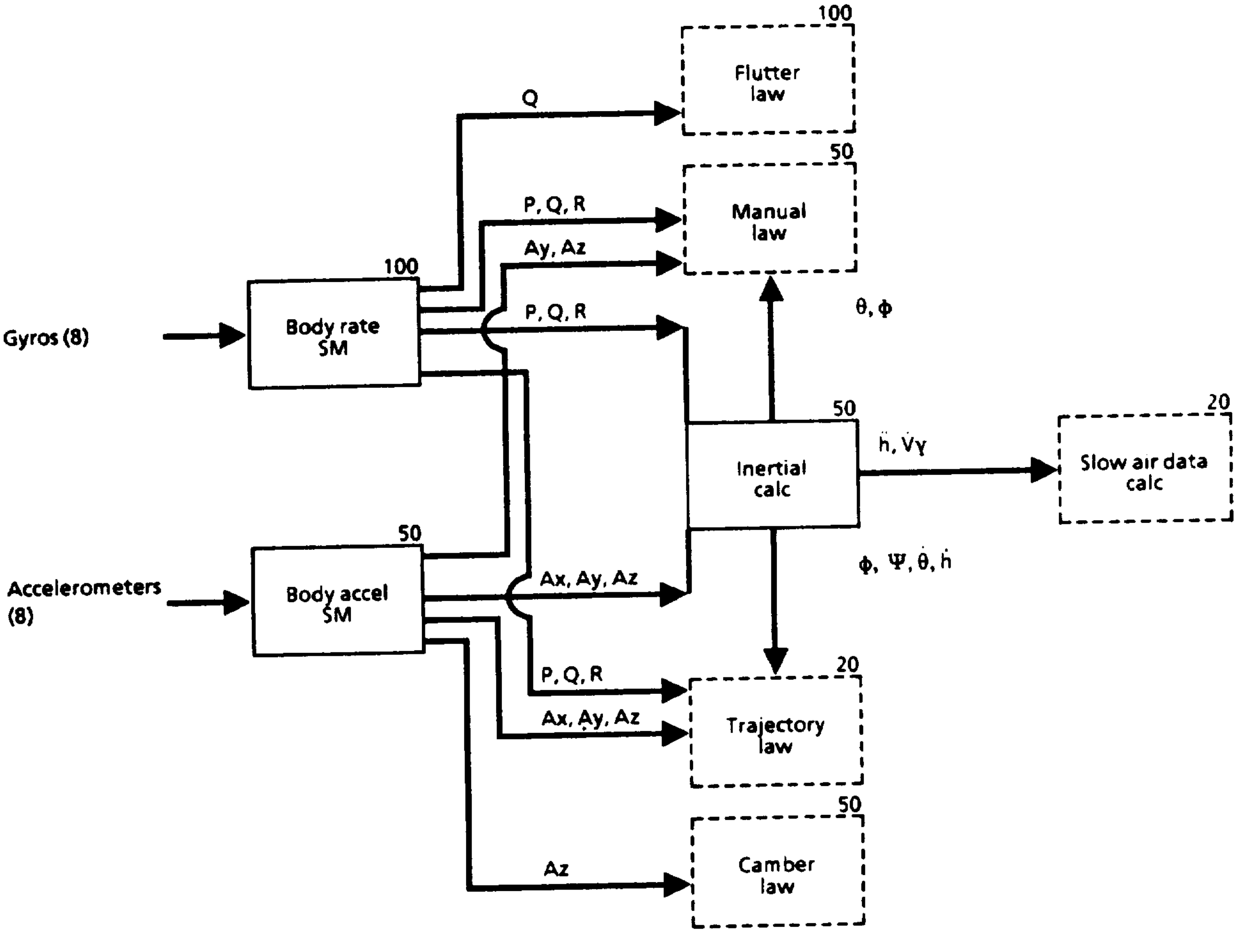


Figure 3.1.1-9. Inertial Subfunctions

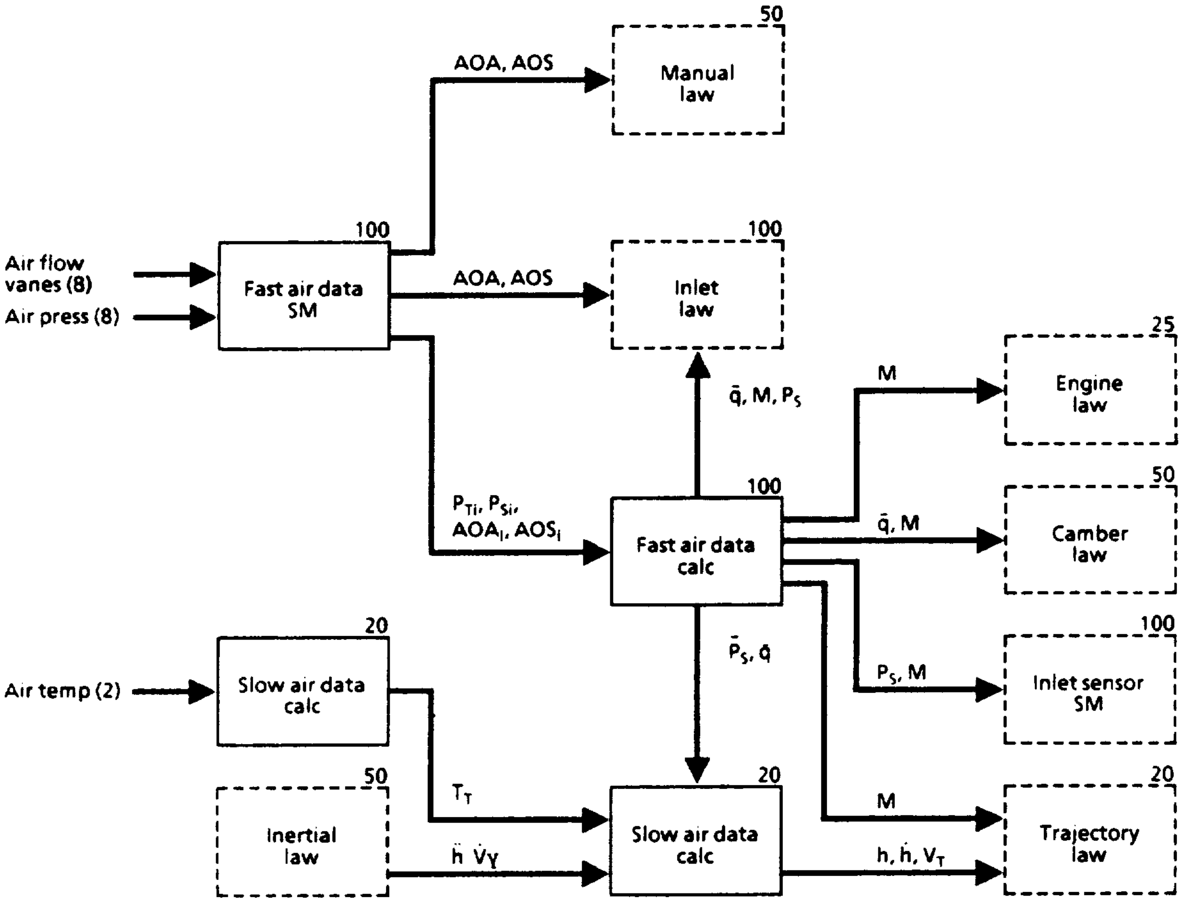


Figure 3.1.1-10. Air Data Subfunctions

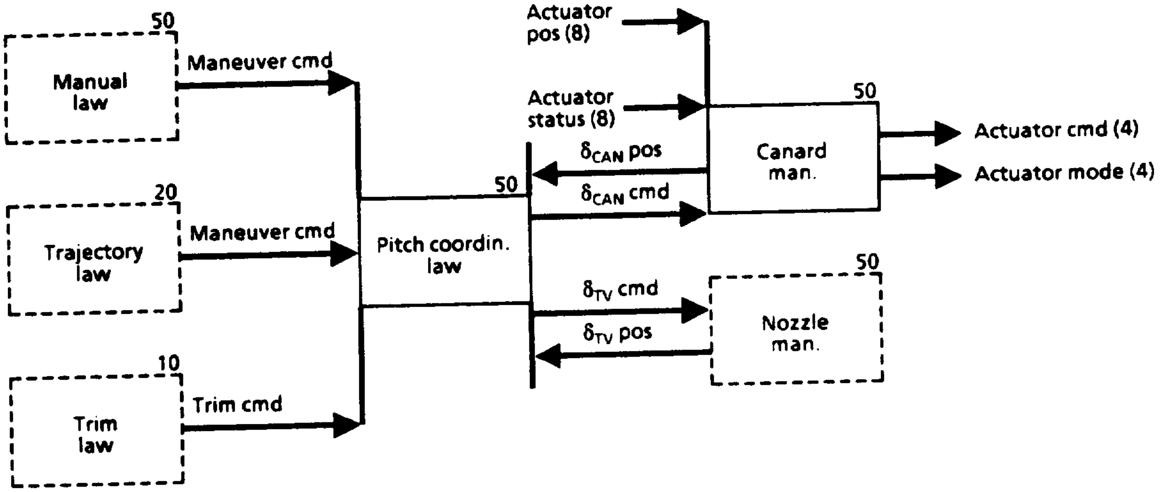


Figure 3.1.1-11. *Pitch Coordination Subfunctions*

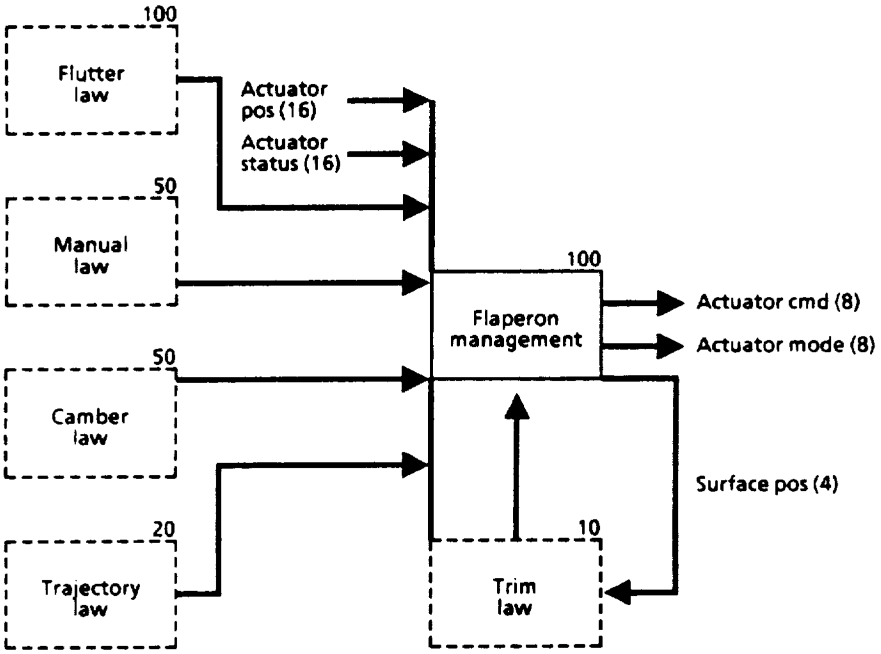


Figure 3.1.1-12. Flaperon Management

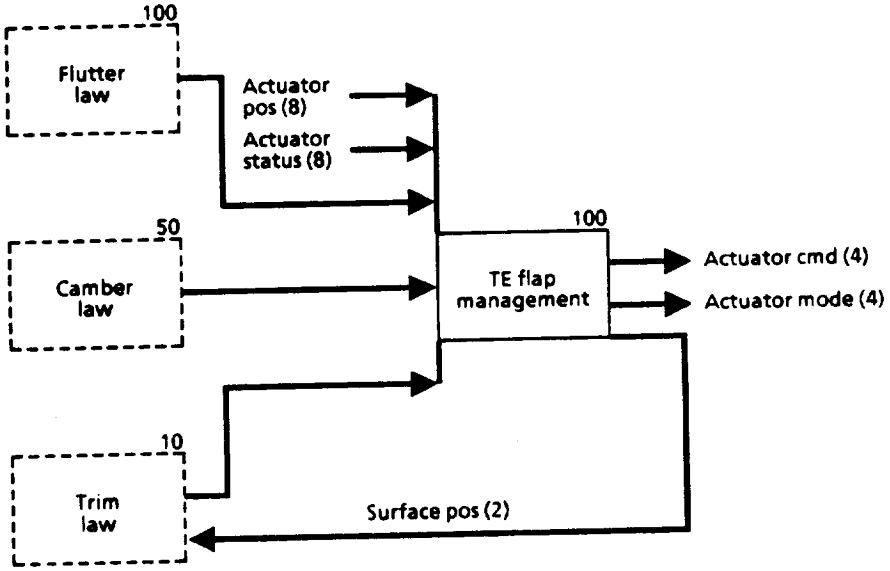


Figure 3.1.1-13. TE Flap Management

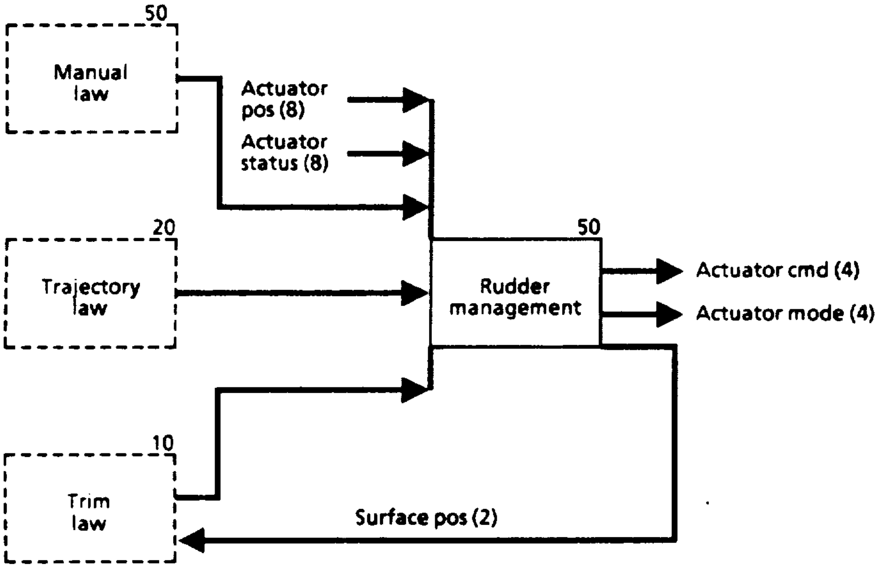


Figure 3.1.1-14. Rudder Management

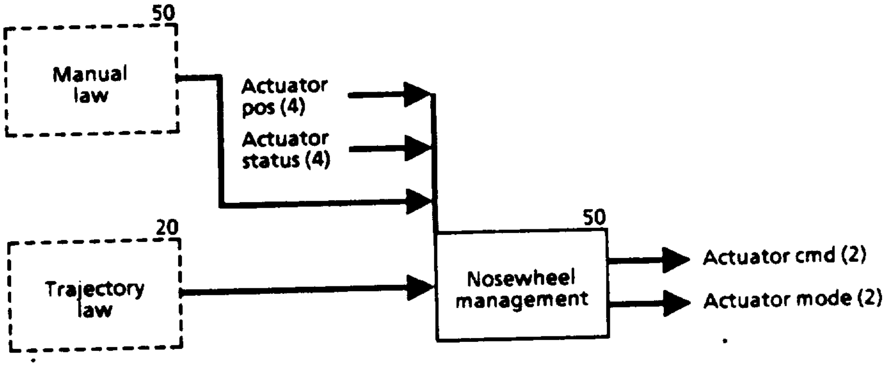


Figure 3.1.1-15. Nosewheel Management

A pitch coordination subfunction handles the blending of control effectors for longitudinal axis control. This allocation is primarily one of convenience. It provides a coordinated interface for the pitch control surfaces and the vectored nozzle. Another integrated control design alternative might signal the actuator devices directly, without the need for an explicit coordination process.

The subfunction figures conclude with the actuator management subfunctions. These figures show which major functions interface to each set of actuators.

The candidate architecture definition from reference 1 allocated the IAPSA II computing functions to the flight control computing site and to an engine control computing site for each engine. The resulting staightforward allocation of computing subfunctions and associated update rates are shown in tables 3.1.2-1 and 3.1.2-2. The pitch coordination subfunction responsible for pitch control blending is performed at the flight control site. The engine control site performs the nozzle actuator management subfunction. Note that the execution rates for the slower functions at the flight control site have been adjusted upward. By maintaining an integer multiple relationship between all rates, potential problems associated with tasks drifting in and out of phase with each other are avoided. The figures show the computing allocation for the functions of the candidate architecture to be discussed in this section and section 4 as well as the refined configuration discussed in section 5.

In addition to division by computing sites, the subfunctions are also organized into units by rate group. That is, all subfunctions executing at one site at the same cyclic execution rate are treated together. This has performance implications in terms of decreased overhead processing necessary for sequencing and control of the application computing, which are discussed in section 4. The results of this grouping are special data transfer requirements incorporated into the system design. Note that these are examples of requirements that are due to the way the system is implemented.

Table 3.1.2-1. Computing Allocation – Flight Control

100 Hz	50 Hz	25 Hz	12.5 Hz
Wing accelerometer SM	Pilot command SM	Trajectory law[1]	Trim command SM
Flutter law	Manual law	Slow air data SM[1]	Trim law
Body rate SM	Camber law	Slow air data calculation[1]	
Fast air data SM[1]	LE flap AM		
Fast air data calculation[1]	Body accelerometer SM		
Flaperon AM	Inertial calculation		
TE flap AM	Pitch coordination		
	Canard AM		
	Rudder AM		
	Nosewheel AM		
	Flap command SM		

SM – Sensor management
AM – Actuator management
1 – Reference configuration

Table 3.1.2-2. Computing Allocation – Engine Control

100 Hz	50 Hz	25 Hz
Inlet SM	Nozzle AM	Pilot thrust SM
Inlet law		Fan face SM
Inlet ramp AM		Engine SM
Inlet ring AM		Fuel flow SM
Fast air data SM[2]		Engine law
Fast air data[2] calculation		Main fuel AM
		Afterburner fuel AM
		Fan guide vane AM
		Compressor guide vane AM
		Trajectory law[2]
		Slow air data SM[2]
		Slow air data calculation[2]

Notes:
SM – Sensor management
AM – Actuator management
2 – Refined configuration

3.1.2 Derivation of Data Transfer

Any data transfer between functions in different computing sites must take place on the intercomputer (IC) network. Intercomputer data transfer takes place between independent computers. A computer with data must first gain access control of the network and then transmit its message. The receiving computer processes the message based on its priority and the current processing workload. A key assumption is that the corresponding time delay is acceptable to the control law performance. As discussed in section 4, IC network operation was not modeled in this study.

Data transfer among subfunctions in different rate groups at the same site will generate intertask activity. The integrity of such transfers will undoubtedly be protected by some operating system kernel or executive system service. This protection will require some overhead processing and corresponding time delay. The operation of these features was not modeled in this study.

The activity in these two categories resulting from the candidate system computing allocation is presented in table 3.1.2-3. The data in this table are listed under the major subfunction that is the data source.

3.2 Reference Configuration Layout

The physical configuration of the IAPSA II candidate architecture is shown in figure 3.2-1. The components are arranged in three major groups: (1) a flight control group, (2) a right engine control group, and (3) a left engine control group. The candidate makes extensive use of advanced information processing system (AIPS) fault-tolerant system building blocks. These are the fault- and damage-tolerant IC networks, input/output (I/O) networks, and FTP. These elements are described in detail in reference 1, appendix A.

The flight control group consists of flight control sensors and actuators connected to a quadruple-channel FTP via two I/O networks. The FTP

Table 3.1.2-3. Data Transfer Activity by Source

Major subfunction (rate)	IC activity	Intertask activity
Manual law (50)	Maneuver anticipation	Flaperon commands
Camber law (50)		TE flap commands Flapperon commands
Trajectory law (25)	Thrust commands Maneuver anticipation Rudder commands[2] Flaperon commands[2] Nosewheel command[2] Maneuver command[2]	Rudder commands[1] Flaperon commands[1] Nosewheel command[1] Maneuver command[1]
Trim law (12.5)		TE flap commands Flaperon commands Rudder commands Pitch trim command
Engine (25)	Throttle position[1]	Nozzle area commands
Nozzle (50)	Thrust vectoring position	
Inertial (100/50)	Body rates[2] Body accelerations[2] Roll angle[2] Heading[2] Pitch attitude rate[2] Vertical acceleration[2] Flight path acceleration[2]	Body rates Body accelerations Roll angle[1] Heading[1] Pitch attitude rate[1] Vertical acceleration[1] Flight path acceleration[1]
Air data (100/25)	Angle of attack[2] Angle of sideslip[2] Mach Dynamic pressure Static pressure[1]	Angle of attack[1] Angle of sideslip[1] Mach Dynamic pressure Static pressure
Pitch coordination (50)	Thrust vector command	
Flaperon AM (100)		Flaperon position
TE flap AM (100)		TE flap position
Rudder AM (50)		Rudder position

[1] Reference configuration

[2] Refined configuration

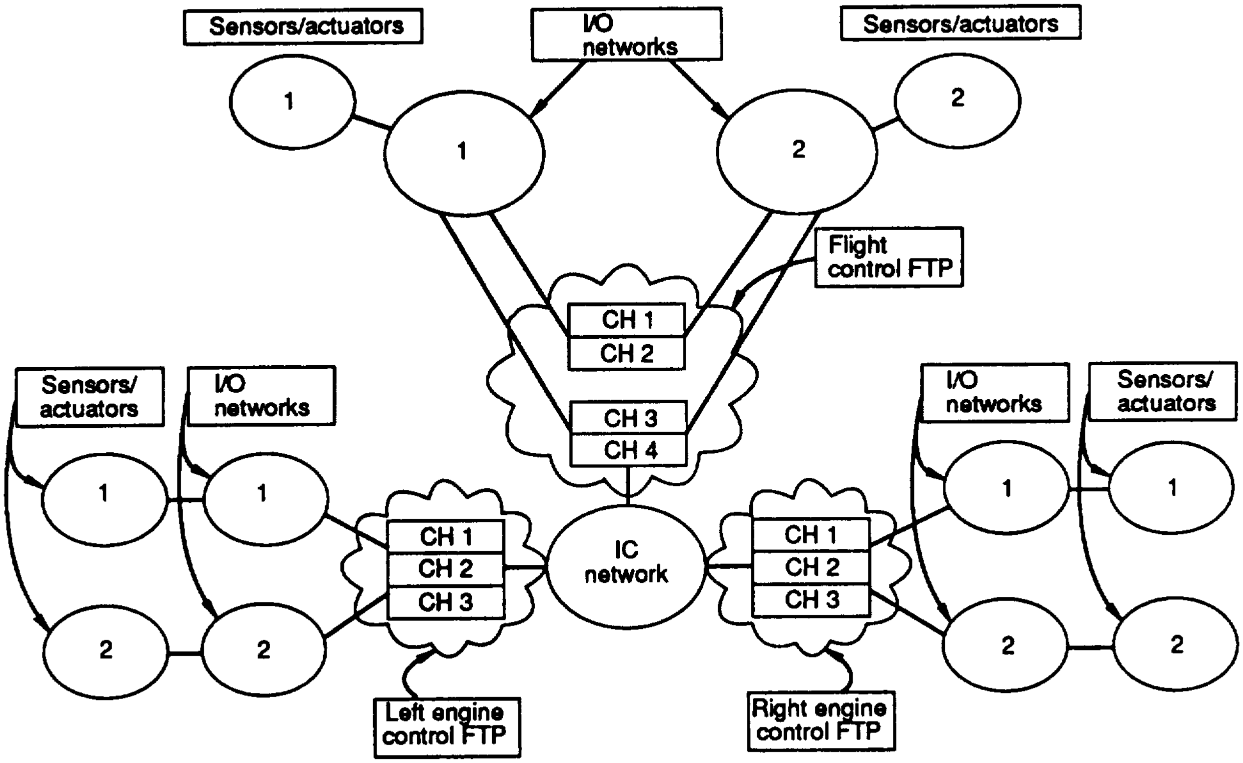


Figure 3.2-1. Reference Configuration Overview

channels are physically dispersed for damage tolerance. Each of the two engine groups controls the operation of one of the two propulsion systems. An engine group consists of the sensors and actuators of one propulsion system connected to a triple-channel FTP through two I/O networks. The three groups are interconnected by an IC network joining the three computing sites. The following two subsections present details about the flight and engine groups from an I/O network point of view.

3.2.1 Flight Control I/O Network

One of the two flight control I/O networks is shown in figure 3.2.1-1. Half of the flight control sensors and actuators are connected to network 1; half are connected to network 2. The sensors and actuators interface the network through device interface units (DIU). The DIU provides signal conditioning and conversion for the devices and handles the network communication protocol. Each DIU connects to a single network node. The lack of cross-connection between sensor and actuators, nodes, and DIUs alleviates any fault containment concerns. This form of "brickwalled" system organization introduces an element of dependency into the system, which will be evaluated in the reliability analysis.

The I/O network consists of a mesh of 18 nodes connected by full duplex point-to-point links. The nodes can be commanded to turn links on or off. In operation, enough links are turned on to create a path from the FTP to all nodes and DIUs in the network. The remaining links act as spares. Each network node in the candidate architecture is connected to three other network nodes. This means that two-thirds of the links are active and the remaining one-third are spare during operation. With three connections, at least three faults are necessary to destroy communications with a good node and its attached devices.

The flight control I/O networks are connected to the FTP with three root links, each connected to a different channel. During operation, only one of the root links is active, and the others act as spares. With three root links, at least three faults must occur to isolate an entire network of sensors and actuators.

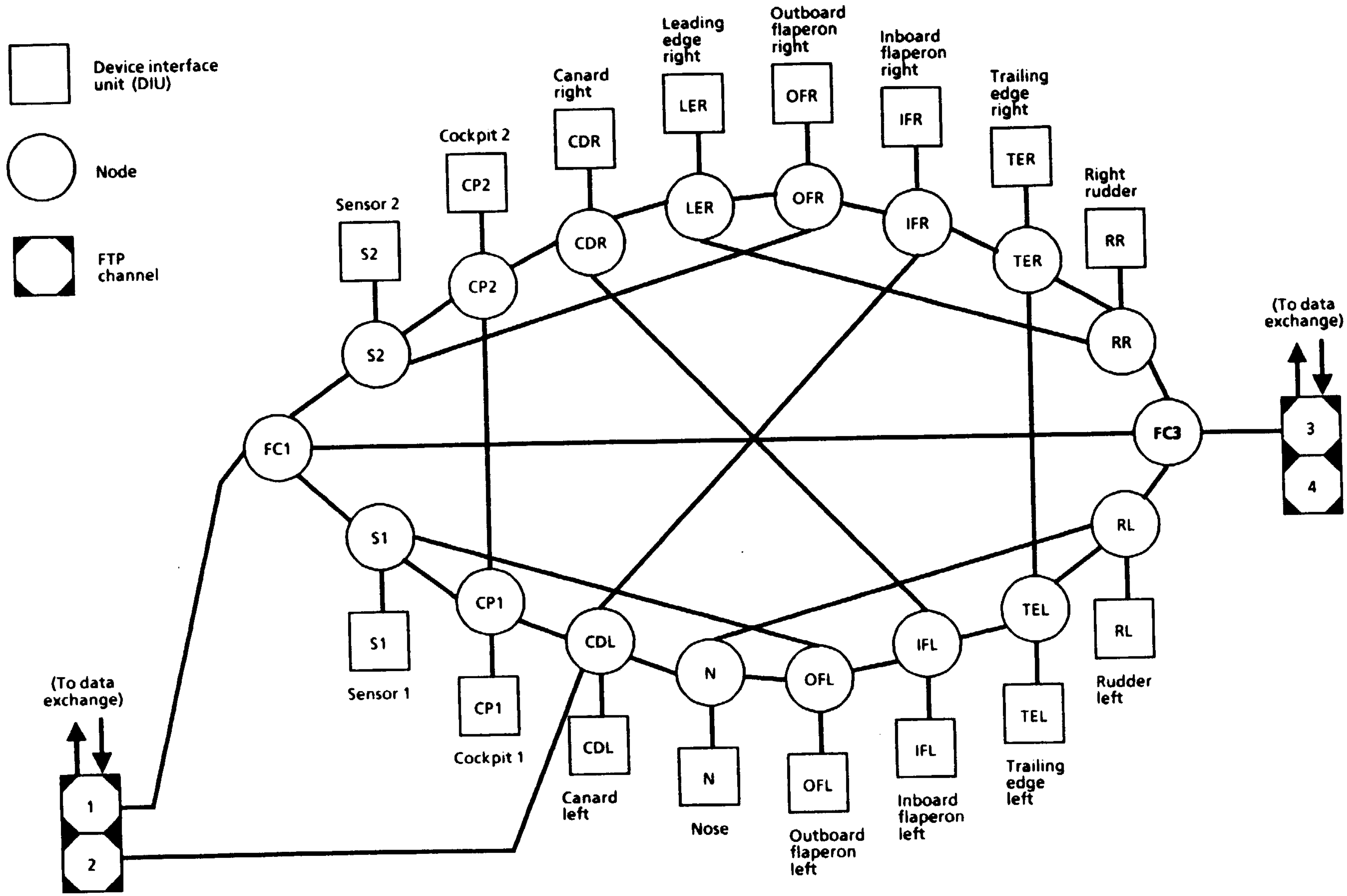


Figure 3.2.1-1. Flight Control I/O Network 1 Layout

The redundant flight control sensors and actuators are spread evenly across the two networks and the redundant DIUs. The specific assignment of these elements is shown in table 3.2.1-1. The safety-critical flight control sensors are primarily quadruple redundant. The skewed body motion sensors are an exception because a different redundancy level is required to achieve the same degree of failure protection. A total of eight were needed in the IAPSA I system due to interconnection dependencies. Eight were used in the candidate architecture because a similar situation exists for IAPSA II. The mission-critical flight control sensors are triple redundant. The flight control surface actuators have a dual redundant control channel arrangement. Each actuator channel is connected to a different network and is capable of full operation. If communications to one channel of the actuator are interrupted for any reason, control can be maintained through communications on the unaffected network. Failure protection details for these elements will be provided in section 3.3.

3.2.2 Engine Control I/O Network

The two I/O networks for one propulsion system are shown in figure 3.2.2-1. Like the flight control arrangement, the propulsion control sensors and actuators are connected half to one network and half to the other network. The devices are also connected in a "brickwall" manner, with one device connected to only one DIU, which in turn is connected to only one network node and therefore one network.

The engine control networks each contain four nodes, and because the network is a system building block entity, its operation is identical to the flight control networks. Like the flight control network, each network node is connected to three network links. Unlike the flight control network, each network is connected to the FTP through two root links, which means that an entire set of sensors and actuators can be isolated after two failures.

The candidate system requires a lower level of failure protection for each propulsion system because safe flight and landing is possible after loss of

Table 3.2.1-1. ***Sensor/Actuator Connection – Flight Control Networks***

Devices	Redundancy	DIU/node ID	NW
Body accelerometers	2 2 2 2	S1 S2 S3 S4	1 1 2 2
Body gyros	2 2 2 2	S1 S2 S3 S4	1 1 2 2
Angle of attack	1 1 1 1	S1 S2 S3 S4	1 1 2 2
Angle of sideslip	1 1 1 1	S1 S2 S3 S4	1 1 2 2
Static pressure	1 1 1 1	S1 S2 S3 S4	1 1 2 2
Total pressure	1 1 1 1	S1 S2 S3 S4	1 1 2 2
Total temperature	1 1	S1 S3	1 2
Pitch stick	1 1 1 1	CP1 CP2 CP3 CP4	1 1 2 2
Roll stick	1 1 1 1	CP1 CP2 CP3 CP4	1 1 2 2
Rudder pedal	1 1 1 1	CP1 CP2 CP3 CP4	1 1 2 2
Left throttle	1 1	CP2 CP3	1 2
Right throttle	1 1	CP1 CP4	1 2
Flap lever	1 1 1	CP1 CP2 CP3	1 1 2
Pitch trim	1 1 1	CP2 CP3 CP4	1 2 2
Roll trim	1 1 1	CP1 CP3 CP4	1 2 2
Yaw trim	1 1 1	CP1 CP2 CP4	1 1 2

Table 3.2.1-1. Sensor/Actuator Connection – Flight Control Networks (Continued)

Devices	Redundancy	DIU/node ID	NW
Left canard actuation	1 1	CDL1 CDL2	1 2
Right canard actuation	1 1	CDR1 CDR2	1 2
Nosewheel actuation	1 1	N1 N2	1 2
Leading edge actuation	1 1	LER LEL	1 2
L outboard flaperon actuation	1 1	OFL1 OFL2	1 2
L inboard flaperon actuation	1 1	IFL1 IFL2	1 2
L TE flap actuation	1 1	TEL1 TEL2	1 2
L rudder actuation	1 1	RL1 RL2	1 2
R rudder actuation	1 1	RR1 RR2	1 2
R TE flap actuation	1 1	TER1 TER2	1 2
R inboard flaperon actuation	1 1	IFR1 IFR2	1 2
R outboard flaperon actuation	1 1	OFR1 OFR2	1 2
L outboard wing accelerometers	1 1 1	OFL2 OFL1 IFL2	2 1 2
L mid-wing accelerometers	1 1 1	IFL2 IFL1 TEL2	2 1 2
L inboard wing accelerometers	1 1 1	IFL1 TEL2 TEL1	1 2 1
R inboard wing accelerometers	1 1 1	TER2 TER1 1FR2	2 1 2
R mid-wing accelerometers	1 1 1	TER1 1FR2 1FR1	1 2 1
R outboard wing accelerometers	1 1 1	1FR1 OFR2 OFR1	1 2 1

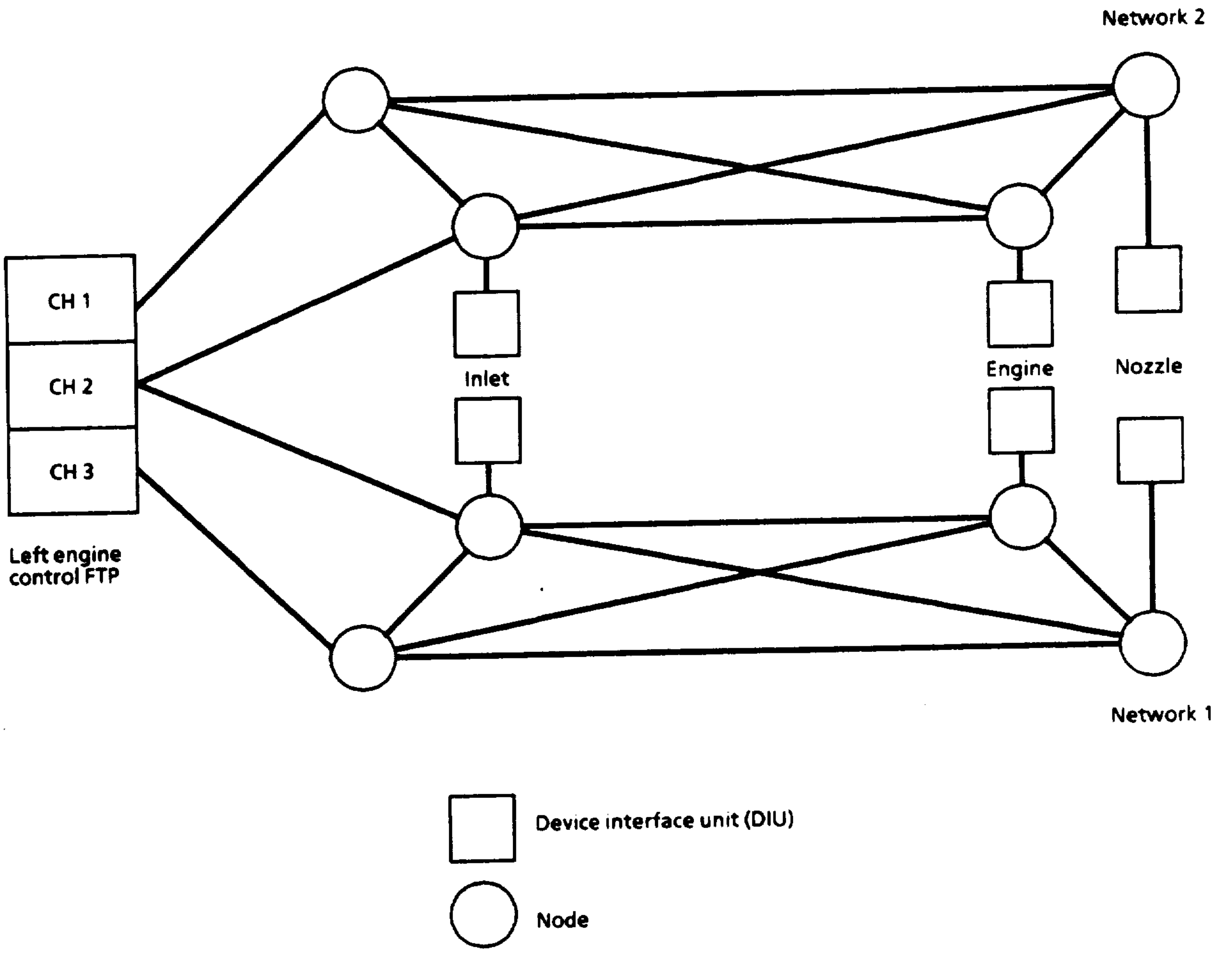


Figure 3.2.2-1. Left Engine I/O Network Layout

one of the two engines. The specific assignment of the redundant sensors and actuators for one propulsion system is presented in table 3.2.2-1. Notice that most propulsion system sensors are dual redundant. An exception is the engine core sensors that are covered by an analytic redundancy management scheme. The propulsion system actuators are dual channel. Each actuator channel is connected to a different I/O network like the flight control actuators. More detail on the failure protection for these elements is presented in section 3.3.

3.3 Reference Configuration Failure Protection

Failure protection is the central issue in the design of flight-critical systems. To provide the necessary levels of safety and mission reliability, the system must be able to tolerate faults without affecting the operation of its critical functions. This capability is provided by redundancy management processes that are responsible for the detection and identification of system element faults and any necessary reconfiguration of functions to maintain safety or mission capability. This section will discuss the failure protection assumptions for the candidate system. Elements will be discussed by functional category. The primary function of each candidate configuration element will be used to group it in one of four major categories: computing, data transfer, sensing, or actuation. Before presenting these assumptions, function migration will be discussed.

A key failure protection issue for the candidate architecture is function migration. This is a high-level scheme that can be used on the group level to provide failure protection for the computing functions. In an AIPS system, function migration is a nonroutine change of computing assignments for the different system computers. An external stimulus such as a fault, a change in mission phase, or crew direction is necessary for function migration. An early design decision was to not implement this capability for failure protection in the candidate architecture. The capability was judged to be relatively immature for the time frame of the IAPSA II application.

Table 3.2.2-1. Sensor/Actuator Connection – Engine Control Networks

Devices	Redundancy	DIU/node ID
Upper ramp actuation	1 1	INL1 INL2
Inner ramp actuation	1 1	INL1 INL2
Bypass ring actuation	1 1	INL1 INL2
Duct static pressure	1 1	INL1 INL2
Normal shock total pressure	1 1	INL1 INL2
Normal shock static pressure	1 1	INL1 INL2
Convergent nozzle actuation	1 1	NOZ1 NOZ2
Upper nozzle flap actuation	1 1	NOZ1 NOZ2
Lower nozzle flap actuation	1 1	NOZ1 NOZ2
Fan face pressure	1 1	ENG1 ENG2
Fan face temperature	1 1	ENG1 ENG2
Fan speed	1 1	ENG1 ENG2
Compressor speed	1	ENG1
Fuel flowmeter	1 1	ENG1 ENG2
Burner pressure	1	ENG2
Fan turbine inlet temperature	1 1	ENG1 ENG2
Afterburner pressure	1 1	ENG1 ENG2
Fan guide vane actuation	1 1	ENG1 ENG2
Compressor guide vane actuation	1 1	ENG1 ENG2
Fuel metering valve actuation	1 1	ENG1 ENG2
Afterburner fuel metering valve actuation (each of 5)	1 1	ENG1 ENG2
Afterburner light off detector	1 1	ENG1 ENG2
Main fuel S/O device	1 1	ENG1 ENG2
Afterburner zone fuel S/O device (1 of 5)	1 1	ENG1 ENG2

Function migration imposes very challenging requirements on a system performing flight-critical functions with critical timing needs. The function must be reliably terminated at one computing site and started at another with minimum delay. Performance transients must be minor. Additionally, communications responsibility for the associated sensors and actuators must be transferred to the new site with minimum delay. For the new site to fulfill this responsibility, there must first be a physical path from the new site to the appropriate networks. Second, all the software definitions necessary to pick up the periodic communications with the devices must be resident at the new site. Finally, the new site must be able to perform redundancy management on the networks. The critical software elements are AIPS system service functions: system manager, I/O services, and I/O redundancy management.

The decision to include function migration would necessitate changes to the candidate architecture. Spare links would be needed between the computing sites and the networks of the different groups to support the alternative communications capability. The I/O networks might be operated differently, for example, as regional networks instead of local networks, if function migration was a possible reconfiguration action. However, because of the uncertain availability of this technique, the candidate system was evaluated with no function migration capability.

3.3.1 Application Computing

The first functional category for the failure protection discussion is application computing. The application computing functions are centered on control laws that provide integrated flight and propulsion control capabilities. These control laws run on the FTP general-purpose computers. A key feature of the AIPS system is that application software functions can be written as if they execute on a perfectly reliable single channel computer. The AIPS system hardware and software elements provide protection from computing element failure.

The FTP operating concept has all redundant channels executing exactly the same software in instruction synchronism. All of the computed outputs are voted to ensure bit-for-bit agreement. An unsuccessful vote points out a faulty channel. All inputs go through a byzantine fault-tolerant data exchange process to ensure that each good channel is operating with exactly the same data. Special fault-tolerant clock (FTC) hardware keeps each channel in synchronization, while special data exchange (DX) hardware allows for fast, reliable exchange of interchannel data.

A key element of the FTP concept is the AIPS system software FDIR process. FDIR has the overall responsibility for FTP redundancy management. The portion of FDIR that executes most frequently is called Fast-FDIR. This process checks every cycle for indications of output disagreement and ensures that all channels are in instruction synchronism. When necessary, processor interlock hardware is used to disable faulty channel outputs. FDIR programs running in background perform self-tests on the channel hardware. A watchdog timer monitors the periodicity of the channel cyclic execution. More detailed information about the FTP failure protection is provided in reference 1.

Two good FTP channels are needed for operation when a guaranteed shutdown is required for a subsequent channel fault. Two are necessary so that comparison between channels can immediately detect the fault. The FTP building block has the ability to be safely reconfigured into a special simplex operating mode if the failures can be rapidly identified by self-test. This ability allows for continued operation for those which do not cause a loss of synchronization. However, the capability to degrade to simplex operation was not evaluated in the candidate architecture. Based on this decision, the quadruple flight control computer provides fail-operational/fail-off failure protection capability for the safety-critical functions. Similarly, the engine computer provides fail-operational/fail-off capability for each propulsion system.

3.3.2 Application I/O Activity

The next functional category is data transfer. Sensor and actuator data transfer takes place on the I/O networks. As previously mentioned, not all of the links in the mesh network are active during operation. During system startup before flight, two out of three mesh network links are activated to form a virtual communications bus. This virtual bus provides a path between the FTP and all good nodes and DIUs. Responsibility for maintaining a communication path to all good devices during operation rests with the I/O redundancy management process, which is a software building block element of the AIPS system services software.

Network data transfer involves many of the FTP hardware and software elements. These are shown in figure 3.3.2-1. The application process requests I/O activity from one of the AIPS system service software processes, which primarily resides on the IOP coprocessor. The network activity called for by the I/O request consists of a sequence or chain of transactions. Each transaction involves a command message addressed to a specific DIU followed (usually) by a response message from that DIU. Each chain is setup for execution, initiated, and post processed by the services software on the IOP. Once initiated, the chain runs without IOP involvement. The network interface (NI) hardware sends each command message and receives each response message until the chain sequence is complete.

The NI hardware, which transmits and receives all chains, detects and logs network protocol errors. Missing response messages are caught by the NI timer that sets a timeout interval for each transaction. The I/O services process implements a chain timeout to detect faults that result in an incomplete chain. The data transfer status information and all data returned by the DIUs are distributed to all FTP channels by the I/O services through the data exchange. The status information is analyzed by I/O redundancy management and, when necessary, network repair action is taken to restore communications.

Most network repair actions command nodes to enable or disable network links using special command messages over the I/O network. The repair strategy fundamentally consists of turning links on and off to isolate faulty network elements and to provide an alternate data path to the affected DIU(s). An example of this is shown in figure 3.3.2-2, where a spare link is used to bypass a failed active link. The specific repair action must be based on fault indications available to I/O redundancy

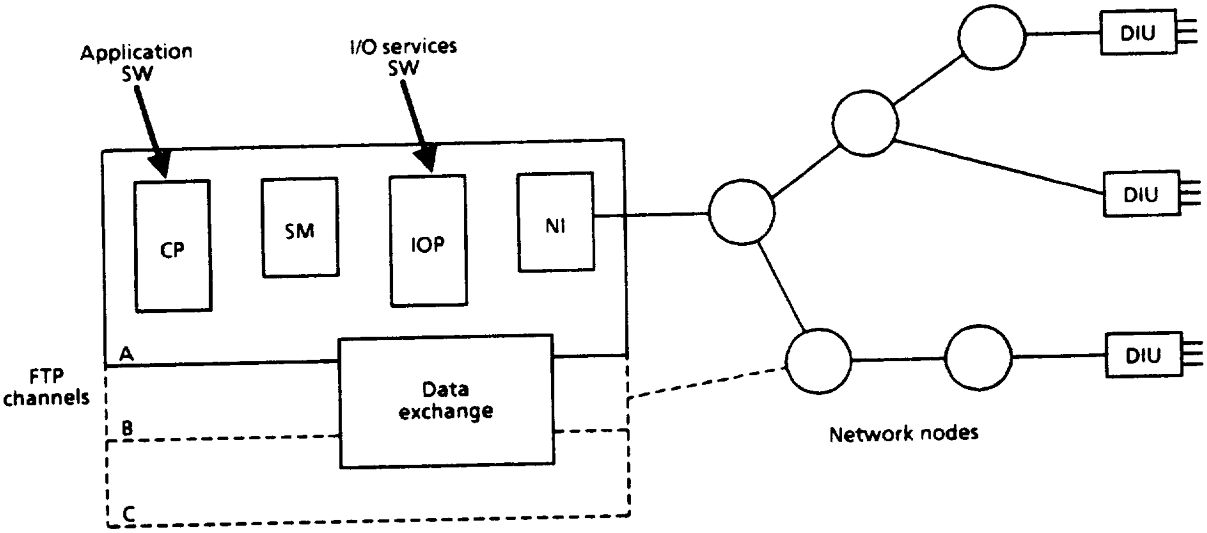
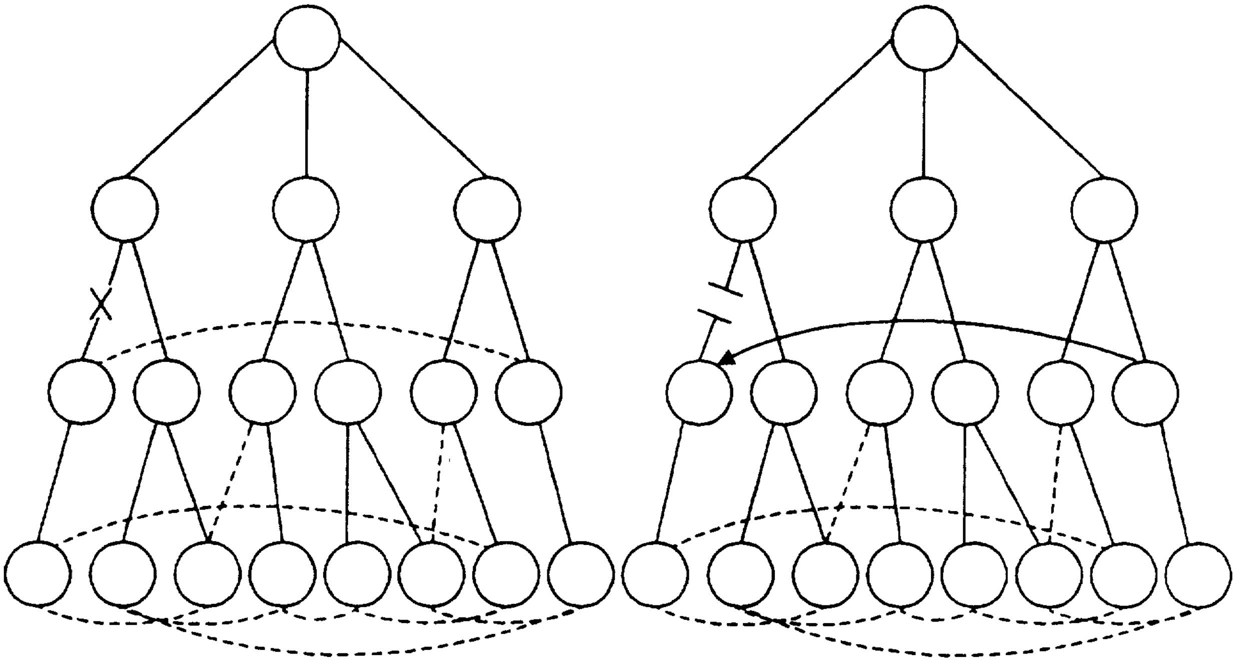


Figure 3.3.2-1. I/O Network Operation



- Spare link used to bypass failure

Figure 3.3.2-2. Link Failure Reconfiguration

management at the FTP. Because many faults have similar indications, special chains are usually necessary before the repair activity to help diagnose the fault. In some cases, the fault may not be identified except by a process of elimination during the repair sequence. Certain candidate architecture faults, such as DIUs or nodes, will permanently disable the directly connected sensors and actuators because no alternative path is possible. The specific I/O network repair strategy assumed for the candidate architecture is discussed further in section 4.

3.3.3 Flight Control Sensing

Most of the safety critical sensors listed in table 3.2.1-1 in section 3.2 were quadruple redundant to provide full operation after two like sensor failures. Mission critical sensors are triple redundant to provide fail-operational/fail-off failure protection for the mission-critical control functions. Voting processes executing in the FTP compare redundant sensor readings to detect and identify sensor failures. These voting processes have demanding false alarm, missed alarm, and failure transient performance requirements. Sophisticated dynamic processes are used to meet these needs for full-time critical operation. Distinguishing failures from normal channel mismatches is a very challenging engineering task. As a result, a significant failure recovery time is needed to react to a sensor failure. Because a comparison process is used, only failure detection can be accomplished when two sensors remain operational.

The skewed axis sensor readings are processed to provide estimates of the three axis rates and accelerations. A sophisticated process compares the readings for consistency in order to detect and identify sensor failures. In this situation, four sensors are needed for the process to provide failure detection capability. Five are needed to identify the failed sensor.

All of the sensor redundancy management processes can use external information to aid fault identification. When data is unavailable to a comparison process because of a known communication fault, operation can be continued with a single remaining sensor (or three skewed sensors).

However, in this situation, the voting process is unable to detect a subsequent sensor fault.

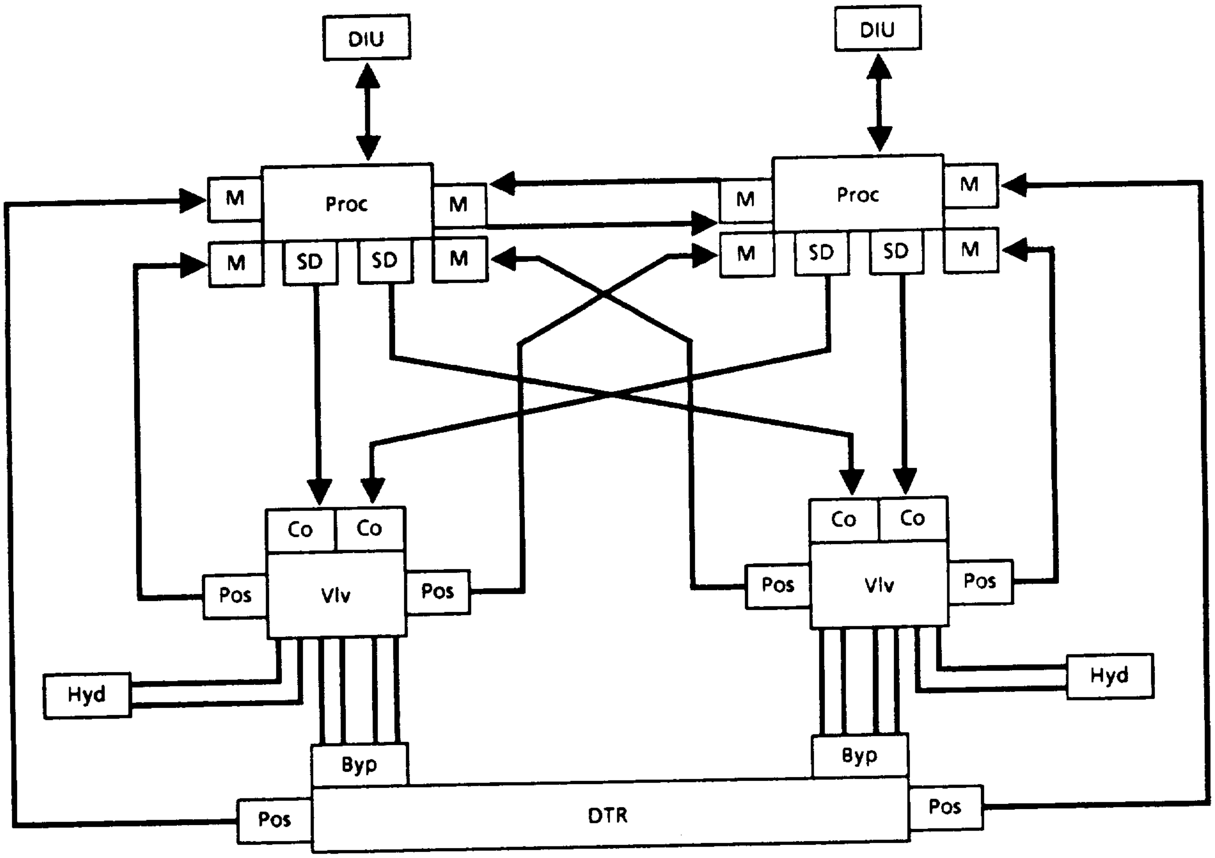
3.3.4 Flight Control Actuation

Eight primary surfaces provide basic flight path control. At least two of the surfaces contribute most of the control moment for each axis. Pitch axis control is provided by two canards. Two flaperons on each wing control roll axis motion. Similarly, two rudders control motion around the yaw axis. Secondary surfaces and devices include leading edge flaps, trailing edge flaps, and nosewheel steering.

Each surface or device is moved by a dual actuator. The actuator is based on a dual coil-dual monitored valve approach. Figure 3.3.4-1 shows the configuration of the standard actuator. Most of the actuator components are dual with the exception of the servodrive and valve monitoring elements. Two processors, each of which has the capability to drive both control valves, close the actuator position loop. Position errors cause control valve movement that routes hydraulic flow to one side of a dual tandem power ram.

Local redundancy management is used to react to most failures. Special monitor hardware detects most failures of the actuator position and valve position sensors. When failures are detected, the other actuator processor can take control. The actuator processor computes a model of the control valve dynamics to detect valve failure. Valve failure will lead to bypass of that side of the dual tandem ram and continued operation using the other valve. A self-test process and watchdog timer hardware detect failures of the actuator processor hardware. Detected failures result in control of the surface by the other processor.

Some actuator failures may be missed due to a lack of coverage by the monitor processes. As a last line of defense, the actuator management process in the FTP generates a "safe" command for the surface when it sees indications of an unresponsive surface. Undetected failures may lead to force fight situations, seen as a stuck or slowly drifting surface. They



M	Monitor	Hyd	Hydraulic system
Proc	Processor	Vlv	Control valve
SD	Servo drive	Byp	Bypass device
Co	Coil	DTR	Dual tandem ram
Pos	Position sensor		

Figure 3.3.4-1. Surface Actuation – Reference Configuration

can also lead to a surface moving rapidly hardover if, for example, the other actuator channel has been bypassed due to a previous failure. Because it can deactivate an entire surface, the central process must satisfy stringent false alarm requirements. It must correctly handle anomalies occurring during normal operation or caused by failures in other system elements.

The response to an uncovered failure is more time critical in the case of an actuator that is controlled by a single channel due to a previously detected fault. In this situation, the surface may be moving rapidly hardover until the central process commands passive operation. This condition sets the time response requirements for the central actuator management process.

3.3.5 Propulsion Control Sensing

As previously described, most of the propulsion sensors are dual redundant. For the candidate system, model-based redundancy management processes were assumed to allow fail-operational/fail-off failure protection capability. The specific approaches used for the different sensors are described in this section.

It should be noted that a more detailed examination and definition of the propulsion control concept was carried out during the refined configuration evaluation. The concept and failure analysis described in this section were reevaluated during that study. Shown here are the results of the initial analysis. The results of more detailed evaluation are presented in terms of changes to this baseline in section 5.

An inlet flow model identifies failures among the inlet pressure sensors, fan face sensors, and inlet device position sensors. The model executes on the engine control FTP using air data sensor information and engine fan speed. It identifies the failed sensor of a pair and detects a second like-sensor failure by looking for an inconsistency among the measurements and the device positions. Loss of any of the required measurements will deactivate this redundancy management process.

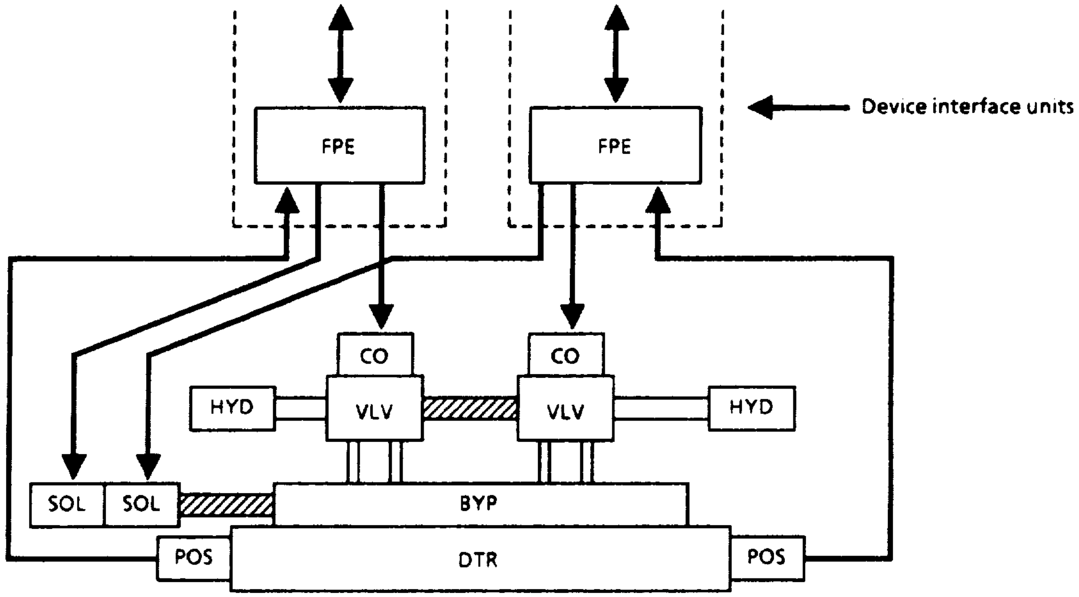
Throttle command sensor management uses the throttle setting of the other engine to help identify sensor failures. The logic assumes that the two engine commands will be nearly identical. When the logic indicates that the final sensor of a pair has failed, the engine reverts to a fixed-thrust operation. In this situation, or at any time the engine does not follow commands, the pilot can shut the engine down when conditions permit.

Redundancy management for the engine core sensors employs a sophisticated algorithm described in detail in reference 2. The core sensors include: fan speed, compressor speed, burner pressure, fan turbine inlet temperature, and afterburner pressure. The analytic redundancy method detects and identifies failures among the five sensor types to provide fail-operational/fail-off capability. When measurements from a specific sensor type are lost, the algorithm provides an estimated sensor value to be used by the control law. In addition to the core sensor measurements, the algorithm needs fuel flow, nozzle area, and fan and compressor guide vane position measurements to perform its function. Some of the core sensors are dual redundant to maintain failure detection capability after communication element failures. Because the sensors are divided between two node/DIUs, one failure can eliminate half of them.

The afterburner lightoff detectors, used in the staged control of the afterburner, use a simple consistency scheme to identify detector failures. The scheme is based on the commanded afterburner operating mode. When the detectors disagree, the one consistent with the commanded operation will be used while the other is declared failed. Similarly, a remaining detector indicating uncommanded lightoff will be declared failed. If the remaining detector fails to indicate lightoff when operation is commanded, afterburner sequencing will be stopped.

3.3.6 Propulsion Control Actuation

All propulsion devices employ the same general actuation control concept shown in figure 3.3.6-1. A propulsion actuator is basically a dual-channel device incorporating fail-passive electronics. Each channel drives its control valve based on the error between the position command from the DIU



Devices (per actuator)

FPE	Fail passive electronics	2
CO	Coil	2
VLV*	Control valve	2
BYP	Bypass device	1
SOL	Engage solenoid	2
POS	Position sensor	2
DTR*	Dual tandem ram	1
HYD	Hydraulic system	2

*Active failure mode

Figure 3.3.6-1. Propulsion Actuation

and the position feedback sensor. The two control valves are mechanically linked together. Each valve controls hydraulic flow to one side of the dual tandem power ram. Either channel can engage the actuator through a dual solenoid-operated bypass valve.

Generally, propulsion actuation element failures are detected using self-test methods. Failures detected in the electronic elements cause one channel to fail passive, which means it stops driving its control valve and takes away its engage power from the bypass solenoid. When both sides fail passive, disengagement causes the device to move to a preferred fixed position. The propulsion system operates at a degraded performance level when one of its devices is in the fixed position. This will be further discussed in section 3.4.

Failures that are not detected by the self-test methods result in disagreement between the feedback sensors or force fight at the control valve. This will be manifested by a stuck or slowly drifting propulsion device. The actuator management process in the FTP detects this situation and commands disengagement to force the device to move to its preferred fixed position. Certain mechanical failures in the actuator will also be detected by the central redundancy management. Mechanical jam of the control valve will cause the device to move toward a hardover position until it is disengaged; however, mechanical jam of the power ram, which is an unlikely failure mode, will jam the device permanently.

A fuel flow model identifies metering valve position sensor failures and fuel flowmeter failures. The model can identify the first failure of a pair of sensors and detect the second failure to provide fail-op /fail-off capability. Its operation is based on consistency between the valve position and the measured flow.

The fuel handling portion of the system includes special fuel shutoff devices where needed for additional safety. The fuel to each engine passes through a dual device fuel shutoff valve. Either shutoff drive can terminate fuel flow to the engine. This capability is used by the fuel metering actuator management process as a last resort to protect against hazardous overspeed or overtemperature situations. Additionally, fuel flow to the afterburner incorporates dual solenoid zone flow shutoff valves. Either drive can open the valve to allow zone flow that is modulated by an afterburner metering valve. Flow to a specific zone is cut off when neither solenoid drive is powered. Central RM commands fuel shutoff if continued operation of a specific engine is hazardous, for example, if the thrust vectoring and reversing devices won't move to a safe position.

In the next section, this detailed description of the candidate system will be used to evaluate its key reliability characteristics.

3.4 Reliability Evaluation of Candidate

The candidate system performs functions that greatly enhance the mission effectiveness of an advanced fighter. Two key measures were established for this system to address the reliability specifications presented in section 3.1. The first, safe flight and landing (SFL), is a measure of the safety implications of the system design. Safe flight and landing capability means that the aircraft can fly to a recovery airfield and land safely. Aircraft operation may require the use of emergency procedures and diversion to an emergency base. This reliability measure is based on a 3-hour period, which is representative of a long-deployment mission.

The second measure, full mission capability (FMC), indicates the ability of the aircraft to complete its mission. Full mission capability means that the aircraft can continue to fly any of its possible missions after the failure of a system element. The applicable redundancy management process must automatically perform any necessary reconfiguration so that continued operation requires no special procedures and no significant performance degradation. A 1-hour time period consistent with a combat mission is used for numerical evaluation.

A key study assumption, implicit in most reliability evaluations, is that all significant elements are fully operational at the start of the mission. This implies that preflight system tests, necessary to guarantee correct operation of all critical system elements, are implemented. Creation of these tests is another challenging task for flight-critical systems. Most successful approaches for ensuring integrity before flight rely heavily on the redundancy management processes designed for use during flight. Preflight test definition has not been addressed in this study.

The reliability evaluation process was accomplished in three phases. The first phase is a functional failure analysis, undertaken to define how the system fails. Next, an abstract model of the resulting failure behavior is formulated and converted for input to a reliability tool. Finally, the system loss probabilities are numerically computed and evaluated to understand the particular system concept's strengths and weaknesses. The failure analysis phase is described next.

The candidate system operation must be analyzed to understand how the system fails. A combination of top-down and bottom-up techniques was used in the failure analysis. Top-down techniques were used to classify and group the failure conditions at different levels of the organization hierarchy. Bottom-up techniques, based on failure mode effects analysis, were used to identify the effects of an element failure on the next higher level of system organization. As mentioned earlier, the two system-level failure conditions of interest are loss of SFL capability and loss of FMC capability.

The flight control portion of the system performs the functions organized into the functional blocks illustrated in figure 3.4-1. Similarly, the propulsion control functional blocks for one of the two systems are presented in figure 3.4-2. The top-down technique determined the significant operational states of these functional blocks by relating system performance after failures within the blocks to the two system failure conditions of interest. The goal is to identify those functional block states which by themselves or in combination with the states of other blocks leads to a loss of system capability.

The functional blocks were used to organize the system elements for the failure analysis. Specifically, each element was assigned to a functional block based on its primary function. Each element's failure effect is

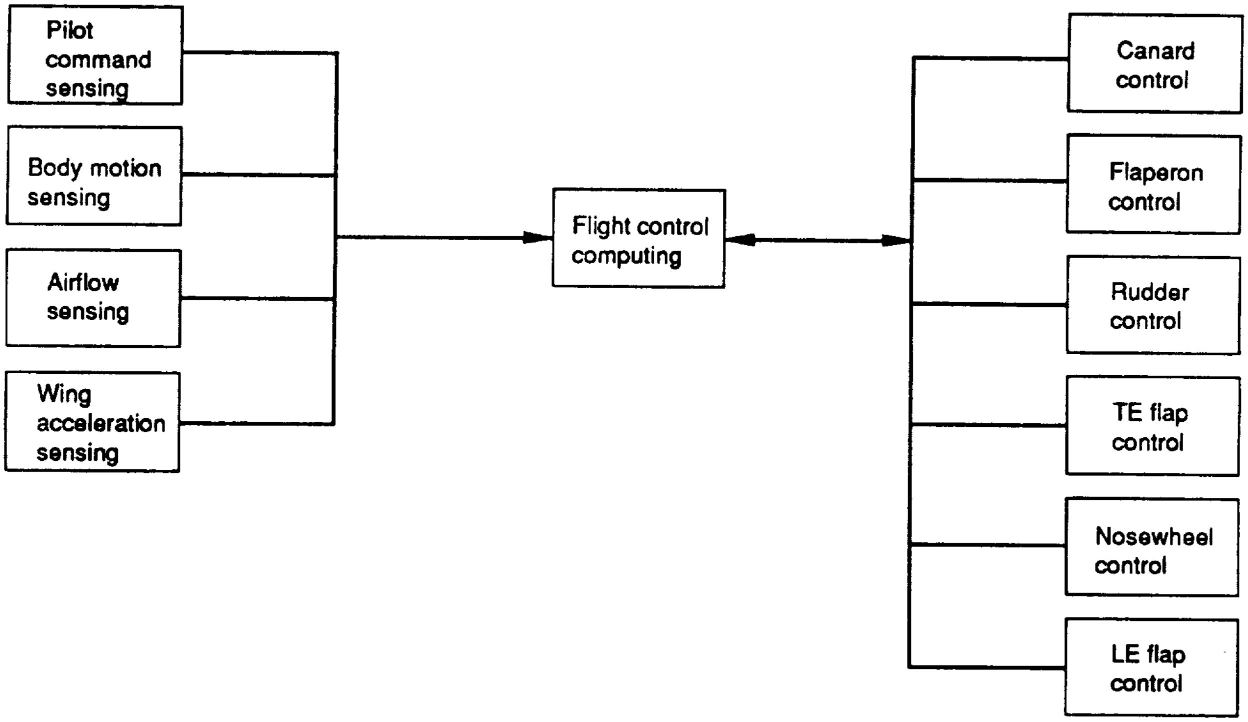


Figure 3.4-1. Flight Control Functions

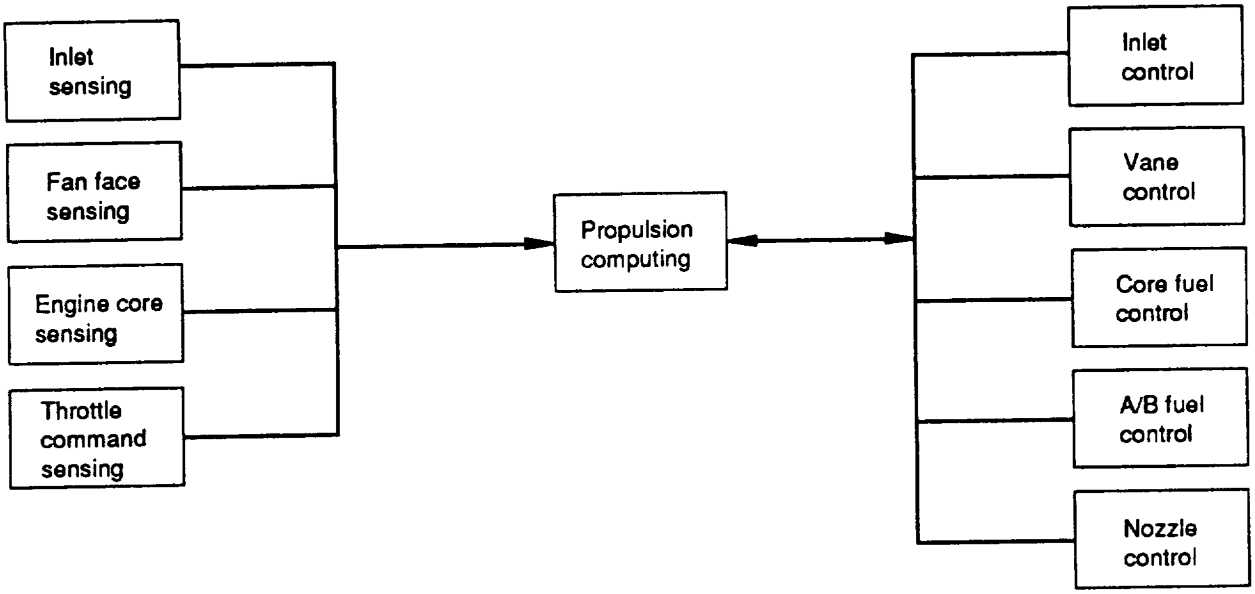


Figure 3.4-2. Propulsion Control Functions

determined by the resulting operational state of the functional block. The process then can relate an element failure and the resulting redundancy management action to an effect on the system operational capability through the state of the functional block.

The failure effect ultimately depends on how the element is used by the major control functions and the element failure mode. The basic failure mode assumed in the analysis is loss of element function. The failure analysis also considers possible active failure modes to see if any can have a more serious effect on the system. Any significant failure modes were included in the reliability models. The failure analysis must also consider the performance of the redundancy management process that handles the affected element. The current operating configuration is usually important to the performance of the redundancy management. These processes can perform imperfectly, failing to detect certain faults and incorrectly diagnosing others.

3.4.1 Flight Control Group Failure Analysis

The failure analysis for elements in the flight control sensing functional blocks was based on some standard assumptions and groundrules. The voting processes used for sensor redundancy management were assumed to operate perfectly. This means that no false alarms, missed alarms, or incorrect identifications occur as long as good sensors outnumber bad sensors. When only two sensors remain (four for skewed sensors), it is assumed that the process can detect that a failure has occurred but cannot identify which of the remaining sensors is bad.

For safety-critical sensing elements, it is assumed that when the redundancy level supports failure detection only, a subsequent sensor failure causes loss of safety. However, in the same situation, if the subsequent failure is loss of data due to a known communication failure, the assumption is that safe operation continues using the remaining sensor. Of course, loss of good data from the final sensor causes loss of the aircraft.

The assumed operating rules for most of the mission-critical sensing elements are slightly different. When data are lost or unreliable, the affected function can be terminated with no affect on aircraft safety. Therefore, when there are only enough sensors to support detection, it is assumed that the function is deactivated when the sensor fails. In the same situation, if data from one sensor becomes unavailable due to a known communication failure, function deactivation is also assumed. Full mission capability is lost when deactivation occurs, but the policy prevents hazardous operation resulting from mission-critical sensing failures.

One aspect of redundancy management operation considered deals with the time required to identify a failed sensor and reconfigure the algorithm accordingly. A possible hazard exists if, during a sensor failure recovery period, another sensor from the redundant set fails. This situation will be referred to as nearly coincident sensor failure. In the case of a quadruple set of sensors, it means that two good sensor values will be processed with two bad values. The resulting inability to "outvote" the bad data is assumed to lead to the use of bad data by the system. For a safety-critical function, this means loss of SFL capability. The results in section 3.4.4 show that the likelihood of this situation is very small and therefore significant only for the safety-critical sensors.

The results of the failure analysis by functional block are presented in table 3.4.1-1 for the flight control group of elements.

Flight Control Sensing

The pilot command sensing functional block includes the pitch, roll, and yaw command sensors as well as the flap lever and trim command switches. Only the flight path command sensors are safety critical, the rest are mission critical. Nearly coincident like sensor failures are a hazard for the flight path command sensors. The standard ground rules apply to the elements in this block.

Table 3.4.1-1. Function Failure Analysis - Flight Control

Function	Total loss effect	Active failure mode considerations
Pilot command		
Pitch, roll, yaw, sensing	Unsafe	
Trim command sensing	SFL	
Flap lever	SFL	
Body motion sensing		
Rate sensing	Unsafe	
Acceleration sensing	Unsafe	
Air flow sensing		
Angle of attack	Unsafe	
Angle of sideslip	Unsafe	
Static pressure	SFL	
Total pressure	SFL	
Total temperature	-	
Wing acceleration sensing	SFL	Total loss of capability during critical phase of flight is catastrophic
Flight control computing	Unsafe	
Canard control	Unsafe	Loss of single surface - SFL (if all primary surfaces operational) Surface stuck/jammed - Unsafe
Flaperon control	Unsafe	Loss of single surface or two symmetrical surfaces - SFL (if other primary surfaces operational) Surfaced stuck/jammed - Unsafe
Rudder control	Unsafe	Loss of single surface - SFL (if all primary surfaces operational) Surface stuck/jammed - Unsafe
TE flap control	SFL	Loss of single surface - SFL
Nosewheel control	SFL	
LE flap control	SFL	

The body rate sensors and the body acceleration sensors are included in the body motion sensing functional block. It is assumed that the redundancy management can identify failures perfectly as long as five or more sensors are being used. Detection is possible only when four sensors are operating; however, operation can continue with three sensors if data is lost due to a known communication failure. Because dual simultaneous failures can theoretically be identified with seven or more sensors, nearly coincident sensor failures are not a concern with the assumed complement of eight instruments.

The airflow sensing functional block includes the angle of attack, angle of sideslip, static, and total pressure sensors. The angle of attack and angle of sideslip sensors are safety critical while the static and total pressure sensors are mission critical. The safety-critical sensors are vulnerable to nearly coincident sensor failure situations.

The wing acceleration sensing block includes the wing accelerometers at six wing sites used by the flutter control major function. The fault reaction for flutter control devices is slightly different from that assumed for other mission-critical elements. Flutter control allows high-speed, low-altitude ingress with external stores. The capability is needed for only part of some mission scenarios. However, if the aircraft is operating in the critical part of its flight envelope, total loss of this function will cause loss of aircraft. Flutter control is different from the other mission critical functions in that it cannot be immediately deactivated. The aircraft must fly out of the critical region before the function can be turned off.

A major analysis assumption is that the flutter control law design is constrained to drastically lower the failure protection requirements for sensing and actuation. The specific design objective is to provide a minimum safe level of performance when sensing at a single site or actuation of a single surface is lost. The resulting degraded performance with a single passive surface or a loss of sensing at one location must be adequate to allow safe flight out of the critical flutter envelope.

With this constraint, flutter sensing needs can be met with triple redundant sensors at each site, providing fail-operational/fail-off capability. The operating assumption is that the aircraft will slow down out of critical envelope before flutter control is deactivated. Full operation continues after the first sensor failure, when only two good sensors remain at a site. If one of the two remaining sensors fail, data from that site is not used while the aircraft slows down. After slowdown, the flutter control function is deactivated. Aircraft slowdown and function deactivation is also assumed if sensor data becomes unavailable due to a known communication failure. This fault reaction will prevent flutter control operation with a single sensor, thus precluding exposure to a subsequent fault with catastrophic effects.

The control law performance tradeoffs associated with this design constraint were not evaluated because control law design and analysis is not a part of this study. If it is impractical to design a control law to this constraint, there will be an additional safety hazard with the candidate architecture if flutter control operation is continued after sensor failure (or surface actuator channel failure). This additional hazard was not quantified during this study.

These assumptions on system behavior after sensor failures mean that except for nearly coincident faults, three similar sensing failures must occur before SFL capability is affected. Similarly, FMC capability is not lost until two like failures have occurred.

Flight Control Computing

The major flight control functions all have computing subfunctions performed in the FTP. The flight control computing functional block includes all of these subfunctions. Manual control, needed for SFL capability, is the most critical flight control function dependent on the FTP. Because FTP throughput performance is not dependent on channel redundancy level, all functions will continue execution after channel faults until safe operation is impossible. Therefore, the flight control computing functional block remains fully operational after most FTP channel failures. Exact agreement of channel outputs can only detect faults when two FTP channels remain. Thus, the final FTP channel failure occurs, causing all functions to fail when there are only two channels operating.

There were a couple of worst case FTP channel failure mode possibilities. Worst case computation failure modes would result from events or faults that could affect correct computation in more than one channel; however, because avoidance of these situations is a key FTP design goal, they were not considered further in this analysis. In another failure scenario, an FTP channel could fail, generating valid but incorrect messages to the actuator interfaces on one of the two networks. In this situation, an actuator force fight will occur due to the disagreement between the good commands on one network and the bad commands on the other. It was assumed that the system could tolerate this situation for one or two application cycles. Furthermore, it was assumed that all such failures would be detected and full operation restored within this time period. Thus, no active computation failure modes were modeled.

Flight Control Actuation

The common ground rules and assumptions used for flight control actuation will be presented before the specifics of each functional block. The flight control devices include the primary surfaces used in basic flight path control, canards, flaperons, and rudders. Flight control secondary devices include the nosewheel and the leading and trailing edge flaps. The primary control surfaces are used by the safety critical manual control function. A key failure analysis assumption is that continued safe flight and landing is possible if a single primary surface fails passively. For roll axis control, it is assumed that symmetrical pairs of flaperons can be lost. In these situations, the performance reduction caused by the loss of a single surface takes away FMC capability. Another key assumption is that failures that leave any primary control surface stuck or hardover causes loss of safety.

Most actuator control element failures are handled by the local redundancy management processes. One assumption is that control valve and hydraulic power failures are perfectly identified by the local process resulting in an operational surface using the redundant devices. The remaining actuation elements have uncovered failure modes that cause the central actuator management process to command passive operation of the device. A worse surface failure mode occurs when a failure leaves a device in a jammed or stuck position. One cause is a rare mechanical jam failure mode of the hydraulic power ram. Other combinations of failures can also lead to this situation. The failure effect on system capability of a jammed or stuck actuator is device specific.

Generally, passive device operation eliminates FMC capability, but safe flight and landing is still possible. For example, loss of high lift or nosewheel steering may require diversion to an alternative base, but landing is possible using emergency procedures. Similarly, a passive flaperon will degrade the aircraft roll response below FMC standards, but will allow a safe landing. A summary of these assumptions is that covered actuation element failures will result in full operational capability, while uncovered failures will lead to central safing action and a corresponding loss of FMC capability.

The special fault reaction considerations for flutter control have been previously described. For actuation, the assumption is that the control law is designed with the capability to fly out of the critical flutter envelope with a single passive flutter control surface. Fault reaction will take place if a flaperon or trailing edge flap fails passive for any reason during flutter operation. The flutter control function will be deactivated after the aircraft slows down out of the critical flight envelope.

The canard control, flaperon control, and rudder control functional blocks include all eight primary control surfaces. The standard operating assumptions apply to these surfaces. Passive operation of a single surface results in loss of FMC capability. SFL capability is lost if any two primary surfaces are passive (except for symmetrical flaperons). Jam of any primary control surface is assumed to prevent safe flight and landing. The flaperon control blocks are subject to the special fault reaction considerations when flutter control is active.

The nosewheel control and leading edge flap functional blocks include a dual-channel actuation device for each function. Passive operation of either device results in a loss of FMC performance. The leading edge flaps are assumed to remain in their last commanded position during passive operation. It is assumed that a safe emergency landing can be accomplished with a jam of the nosewheel.

The trailing edge flap functional blocks include the two trailing edge surfaces and dual-channel actuator devices. A passive trailing edge flap results in a loss of FMC capability. Additionally, if the failure occurs during flutter control operation, the special flutter fault reaction operation is required. Jam of either trailing edge flap results in a loss of FMC capability.

3.4.2 Propulsion Group and Common Device Failure Analysis

Before discussing the propulsion control elements, the capabilities provided by the propulsion system will be evaluated. The most critical capability is control of thrust adequate to support safe flight and landing. For the candidate twin-engine aircraft, a certain level of single-engine performance is assumed to be necessary. The remaining propulsion system capabilities primarily support the advanced fighter missions; for example, the variable inlet is necessary for supersonic missions, and the vectoring and reversing nozzles support short takeoff and landing, enhanced supersonic maneuvering, and other mission capabilities.

The capability of each propulsion system after failures can be divided into three major performance categories: full, normal, and low capability. Full capability means that all functions are fully operational; this includes full supersonic inlet control; full afterburner thrust control and full thrust vector, and thrust reverse capability. The normal capability category allows some degradation from the full performance level. As a minimum requirement, the engine must be capable of providing the full unaugmented thrust range. The nozzle and the inlet can be operating in a fixed position mode. In the low capability category, the system cannot meet the normal category minimum requirements. The engine has either suffered a serious malfunction and cannot be operated, or it can only run at a fixed thrust level.

The performance levels of both systems must be considered together in order to determine vehicle capability. To fly all of the mission scenarios envisioned for the advanced fighter, both engines must have full performance capability. Full mission capability is lost when either system suffers a failure reducing its capability below full. Based on the initial discussion, safe flight and landing capability is provided if one of the systems has at least normal capability. Safe flight and landing capability is lost when a failure occurs that results in both systems having less than normal performance capability. An argument could be made for requiring afterburner capability for single-engine operation; however, this more restrictive ground rule was not used in this analysis.

The propulsion control system moves several devices to provide its capabilities. If sensing or actuation failures disable the control function, some of these devices are assumed to have safe positions that allow continued operation with degraded performance. These assumptions are indicated in table 3.4.2-1 along with the consequences of this fixed operation. These consequences are used in the detailed failure analysis.

Table 3.4.2-2 presents the results of the failure analysis of the propulsion control elements for a single engine. The total loss effect is expressed in terms of the single propulsion system capability. The next column summarizes any special failure considerations for that functional block. The rest of this section will discuss these results in more detail.

Propulsion Sensing

The inlet sensing functional block includes the three sensor types needed to support variable inlet operation. The mission-critical inlet operation occurs only during transonic and supersonic flight. The inlet sensor redundancy management is based on the inlet flow model. The inlet model is assumed to allow perfect identification of the first inlet pressure sensor

Table 3.4.2-1. Fixed Propulsion Device Operation

Device	Fixed position	Consequences
Upper ramp	Subsonic	Limited supersonic capability
Inner ramp	Subsonic	Limited supersonic capability
Bypass ring	Subsonic	Limited supersonic capability
Fan guide vanes	Run	Limited thrust Range/reduced accel/decel
Compressor guide vanes	Run	Limited thrust Range/reduced accel/decel
Main fuel metering	Off	Engine shut down
Afterburner fuel metering	Off	No afterburner capability
Convergent nozzle	Minimum area	No afterburner capability/ reduced accel/decel
Thrust vectoring nozzle	Centerline thrust	No vectored thrust capability

Table 3.4.2-2. Function Failure Analysis - Propulsion Control Loss Effect

Function	Propulsion System	(Vehicle)	Special considerations
Inlet sensing			
Duct static pressure	Normal	(SFL)	
Normal shock static pressure	Normal	(SFL)	
Normal shock static pressure	Normal	(SFL)	
Fan face sensing			
Fan face pressure	Low	(SFL)	
Fan face temperature	Low	(SFL)	
Engine core sensing			Full operation with 4 of 5 sensor types. Run/off if only 3 sensor types available in engine core
Fan speed	Full	(FMC)	
Compressor speed	Full	(FMC)	
Burner pressure	Full	(FMC)	
Fan turbine inlet temperature	Full	(FMC)	
Afterburner pressure	Full	(FMC)	
Throttle command sensing	Low	(SFL)	Pilot shut down of engine if failure detection process doesn't detect last sensor failure
Propulsion computing	Low	(SFL)	
Inlet control			Active failure mode may cause reduced thrust operation at subsonic speed
Upper ramp	Normal	(SFL)	
Inner ramp	Normal	(SFL)	
Bypass ring	Normal	(SFL)	
Vane control			Active failure mode may cause flameout or compressor stall
Fan guide vane	Low	(SFL)	
Compressor guide vane	Low	(SFL)	
Main fuel control			
Metering valve	Low	(SFL)	
Flowmeter	Low	(SFL)	
Fuel shutoff	-		Passive failure unsafe in conjunction with active metering valve failure Active failure mode may cause overspeed/overtemp or flameout
A/B fuel control			
Zone metering valve	Normal	(SFL)	
Zone fuel shutoff	Normal	(SFL)	
Light off detector	Normal	(SFL)	
Nozzle control			
Lower flap	Normal	(SFL)	Active failure mode requires engine shutdown
Upper flap	Normal	(SFL)	Active failure mode requires engine shutdown
Convergent nozzle	Normal	(SFL)	Active failure mode may cause overspeed or compressor stall

failure and perfect detection of the second like sensor failure. This allows the inlet to remain fully operational after the first failure. Without the inlet flow model, failures cause fixed subsonic inlet operation and a corresponding loss of FMC capability.

The fan face pressure and temperature sensors are covered by the fan face sensing functional block. These sensors are used throughout the full engine operating range. Detection and identification of failures is also accomplished with the inlet model. The redundancy management is assumed to operate perfectly while the inlet model is operational. When sensor data is not available due to a dual like sensor failure or loss of inlet model capability, the engine reverts to fixed-thrust level operation. This reduces capability to the low performance level.

The engine core sensing includes the fan and compressor speed sensors, the burner and afterburner pressure sensors, and the fan turbine inlet temperature sensor. The engine core sensors are critical to normal performance. Failures are detected and identified using analytic redundancy techniques. The process handles certain sensor failure modes very easily, while other failure modes present more difficulty. A fail-operations/fail-off level of failure protection is assumed, which means that the engine is fully operational with the loss of one sensor type. When two sensor types are lost, the assumed fault reaction causes fixed-thrust level operation, which is low performance capability.

The first throttle command sensor failure of a pair is assumed to be perfectly identified with the redundancy management scheme. When the final sensor of a pair fails, the engine reverts to low performance capability. If the sensor failure is detected by the process, fixed-thrust operation results. If the other throttle logic fails to detect the second failure, the pilot can shut the engine down for not following commands. In either case, the result is a loss of normal performance capability. With these operating assumptions, it isn't necessary to distinguish between covered or uncovered second failures in the reliability model.

Propulsion Computing

Propulsion computing operates like flight control computing in failure situations. All computing functions are fully operational until the failure of one of a remaining pair of FTP channels. The assumption is that redundancy management automatically commands fixed-thrust operation when computing is lost. There may also be a procedural requirement to shut the engine down when conditions permit. In either case, performance is reduced below the normal capability level.

Propulsion Actuation

Some standard assumptions and ground rules were used in the failure analysis for the propulsion actuation functional blocks. A common propulsion actuator was used for all devices. The key failure behavior assumptions are as follows. All but a fraction of the propulsion actuation element failures are detected by the self-test processes. These covered failures result in the disengagement of one actuator channel, but the device still has full operational capability through the remaining channel. If the remaining channel then suffers a covered failure, disengagement causes the device to move to the preferred fixed position.

Failures not detected by the self-test processes cause the central actuator management process to command the propulsion device to the preferred fixed position. These failures include undetected feedback position and actuator drive failures, as well as a mechanically jammed control valve. These are single failure situations resulting in fixed-device operation. Finally, the consequences of the rare mechanical jam of the main actuator ram are device specific, which will be described in each functional block discussion.

The inlet control functional group moves three inlet devices. If any inlet device fails to the preferred subsonic position, both engines are assumed to operate with limited supersonic performance. Full mission capability is lost because this eliminates the ability to perform some of the aircraft missions. The jam failure mode could leave an inlet device in the supersonic position. This is assumed to lead to reduced thrust capability at slower speeds and therefore a loss of normal performance capability.

The vane control functional group includes both the fan guide vanes and the compressor guide vanes. Loss of ability to move either set degrades the engine performance while allowing continued operation. A conservative assumption is that this degraded operation does not provide the normal level of performance for the system. A jam failure of the device is assumed to have the same result.

The main fuel metering valve, the fuel pumps and the fuel flow meters are included in the main fuel control functional group. The fuel metering valve moves to the shutoff position when continued control is impossible. The resulting performance is below the normal level, resulting in at least a loss of full mission capability. The main fuel flow model is assumed perfect in detecting and identifying flowmeter and actuator position sensor failures. Continued full operation is therefore possible after the first failure of either sensor type. The model also is assumed to detect second like failures. The main fuel metering valve reverts to the fixed shutoff position when both flowmeters or both position sensors are lost. Because of the flow model, there are no uncovered metering valve position sensor failures.

The unlikely jam failure of the main fuel metering valve causes a safety hazard. In these situations, the central actuator management commands a shutdown of the engine through the fuel cutoff valve. The fuel cutoff valve is dual so that there are two ways to accomplish shutdown. A significant active failure mode for this device would be uncommanded fuel cutoff. This failure mode would of course also lead to loss of normal performance for the system. The fuel pumps are sized to handle the afterburner fuel flow. Full operation is therefore assumed possible if one pump fails. Failure of both pumps leads to engine fuel shutdown and loss of normal performance capability.

The afterburner fuel control functional group includes the A/B zone flow metering valves, the zone flow fuel cutoff valves, the light off detectors, and the ignitors. The afterburner is operated during limited periods of the flight scenarios. The preferred fixed position for the metering valves shuts off the fuel to that zone. Loss of metering capability for any zone is assumed to disable all A/B capability. An assumption is that the central actuator management process detects a jammed metering valve and reacts by commanding disengagement of the zone flow cutoff valves. The fault reaction also disables afterburner operation for the remainder of the flight. The zone flow cutoff valves can be activated two ways to allow zone flow. The active failure mode of concern would be uncommanded opening of the valves. This failure is hazardous only in conjunction with a stuck-open A/B fuel metering valve. Because this situation is a combination of two rare failure modes, it was not included in the reliability models.

Perfect failure detection was assumed for the light off detectors. Full operation is possible as long as one detector remains functional. If both detectors fail, afterburner control sequencing is impossible, causing loss of afterburner capability. Similarly, one ignitor will allow full afterburner operation. If both fail, afterburner capability is lost. In all situations involving loss of afterburner capability, the system performance is reduced to the normal level.

The nozzle control functional block includes the thrust vectoring/reversing flaps and the convergent nozzle. Fixed operation for the convergent nozzle is assumed to lead to normal capability for the system with no afterburner thrust capability. The assumption for a jammed nozzle area device is loss of normal performance capability.

Fixed operation for the vectoring/reversing flaps results in centerline thrust capability only. This would reduce the engine capability to the normal performance level. Because jammed vectoring/reversing flaps could have a severe effect on attitude and speed control, the assumed fault reaction is an engine shutdown with a loss of normal performance capability.

Communication Devices

All of the major control functions in the three groups of the candidate system depend on data transfer provided by network operation. Communication device failures primarily affect sensing and actuation functional blocks. That is, their primary function of sensing the environment or moving actuators for the control function is interrupted by the communication failures. The following subsection presents details on the consequences of communication device failures.

Table 3.4.2-3 summarizes a high-level failure effect study for the elements composing an I/O network. Two generic failure modes are considered, the active mode generally having a more serious affect on network operation. Communications device failures can affect sensors and actuators on one DIU, a subset of the DIUs, or all of the DIUs on a network. The effect depends on the device failure mode and the location of the failure in the active network. All sensors and actuators affected by the failure are unusable until network repair action restores communication. Communication failures can have permanent as well as temporary effects on system operation. When elements like nodes or DIU links fail, it causes a permanent loss of the sensors and actuators dependent on them for connection to the system.

Network redundancy management is challenging because it is difficult to identify specific failures with the information available after an unsuccessful chain of transactions. Many of the failures have similar effects when observed from the FTP. Repair attempts based on the most likely failure or the most easily repaired failure are made to localize the problem. Failure modes not included in the table exist that present especially difficult problems to the network redundancy management. Particularly bad are those that look like other failures and cause the resulting repair action to come to a halt at an inopportune time during the repair sequence. Current techniques to handle such failure modes employ special time-consuming tests at each step in the repair to ensure success. These tests drastically increase the repair time and must be custom designed to catch each and every unique failure mode.

Table 3.4.2-3. Communication Device Failure Summary

Device type	Fault type	Fault effect	Repair action
Network node	Passive	Loss of comm to all downstream devices	Rebuild network around failed node
	Active	Nw unusable Node does not obey reconfiguration command	Rebuild network around failed node
Network link	Passive	Loss of comm to /from all downstream devices	Rebuild path around failed link
	Active	NW unusable or loss of comm to all downstream devices	Rebuild path around failed link
Root link	Passive	NW unusable	Switch to alternate root link
	Active	NW unusable	Switch to alternate root link/reconfigure old root node to disable old root link
Network interface	Passive	NW unusable	Switch to alternate root link
	Active	NW unusable	Switch to alternate root link/disable old root link at old root node
DIU Link	Passive	Loss of comm to DIU and all serviced devices	
	Active	NW unusable	Disable DIU link at servicing node
DIU	Passive	Loss of comm to DIU and all serviced devices	
	Active	NW unusable Actuator Command values corrupted Sensor values corrupted	Disable DIU link at servicing node

While the network is being repaired, system operation must continue using the remaining accessible sensors and actuators. This includes those devices on the remaining good network and any devices that can still be reached on the network with a failure.

There are several ways network operation during network failure recovery can cause system failure. The three most significant have been termed (1) temporary exhaustion, (2) nearly coincident network-sensor/actuator recovery, and (3) nearly coincident dual network recovery. Temporary exhaustion failures occur when previous device failures leave the system without enough devices to safely fly during a subsequent network repair period. This is shown in figure 3.4.2-1 for the candidate architecture situation with quadruple sensors. In the scenario presented by the figure, both sensors on one network have previously failed to make the system vulnerable. A subsequent failure on the other network will disrupt communications with the remaining devices, putting the system out of control until the repair is completed. Normal operation is possible after communications are restored, but if the repair period is too long, it will be too late to save the system. Hence, the name temporary exhaustion.

A nearly coincident network-sensor recovery situation is shown in figure 3.4.2-2. Normally, the bad data from a faulty sensor is prevented from disturbing the system by the voting redundancy management process. Because of normal sensor mismatches, the process needs time to reliably exclude the faulty sensor data from further consideration. In this coincident network recovery situation, the voting process is temporarily without enough good sensors to outvote the faulty sensor. Bad data is assumed to propagate to the control function, causing loss of safety.

Surface actuation for the primary control surfaces is also vulnerable to nearly coincident network recovery. As a consequence of the dual network configuration, an actuation channel will disengage the actuator a few cycles after losing command updates. Disengagement is necessary to allow the channels on the other network to provide control during network outages or communication device failures. The nearly coincident situation occurs when an actuator channel has an undetected failure that is causing a force

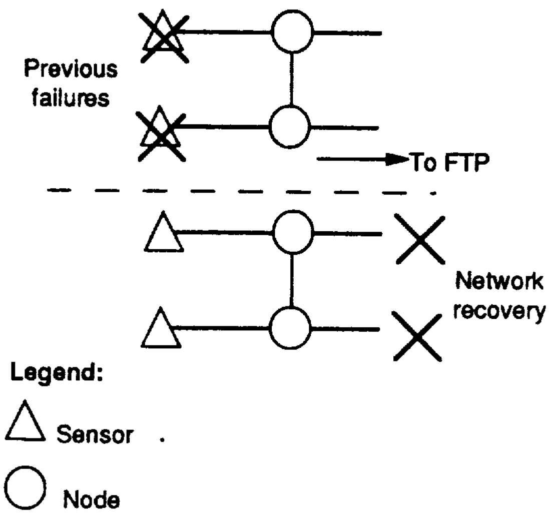


Figure 3.4.2-1. Temporary Exhaustion

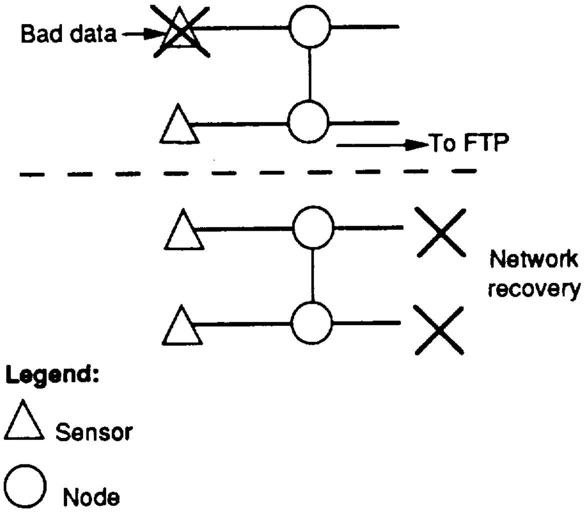


Figure 3.4.2-2. Nearly Coincident Network—Sensor Recovery

fight with the good channel at the same time a network fault occurs. In this situation, the central actuator management fault reaction commands the surface to passive operation. If a network fault interrupts communications, the safe command cannot reach the good actuator channel. The channel with an undetected failure may therefore drive the surface hardover when the good channel disengages due to lost communications.

The final situation is nearly coincident dual network recovery. This is a straightforward case in which both networks undergo repair at the same time. Because there are only two networks, all affected redundant sensing and actuation is lost during the mutual repair period. All three of these network operation situations are assumed to lead to loss of safety because of the effect on safety-critical sensing and actuation.

Some of the communication elements have active failure modes that directly affect sensing and actuation performance. One example from the table is a postulated active DIU failure mode that corrupts the actuator command while satisfying the DIU-actuator protocol. The problem will be detected by the central actuator management process and will result in central safing action. This significant active failure mode was included in the reliability models to assess the possible hazard.

Another significant communication failure mode would be a bad NI that continues to send valid messages containing outdated actuator commands. This would cause channels on the bad network to oppose channels on the good network thereby freezing the surfaces. However, it was assumed that this situation would be detected and terminated within a few application execution cycles, and so it is not included in the reliability models.

Central Power Distribution

Two central systems are used extensively by elements of the candidate architecture: the hydraulic power distribution and the electric power distribution systems. In this study, the details of the secondary power configuration, which includes the engine, accessory drive, and emergency power unit, were considered part of basic airframe system. Only the distribution elements were considered part of the IAPSA II system to be analyzed.

Hydraulic power is supplied to the actuators of both the flight control group and the propulsion control group by two independent aircraft hydraulic systems. All system actuators have two control valves operating a dual tandem power ram. Loss of one hydraulic system is assumed to result in a passive loss of one of the two hydraulic channels. All actuators can continue full operation with the remaining channel. If both hydraulic channels fail, all control devices fail passive, resulting in loss of the aircraft.

There are many two-failure situations leading to a passive loss of an actuation device. The loss of a hydraulic system makes all devices vulnerable to a failure in the other actuator channel. The actuator redundancy management processes must operate correctly during hydraulic system failures or anomalous performance situations that might occur during peak demand periods or operation after loss of one system. For this evaluation, it was assumed that actuator redundancy management performs perfectly and that hydraulic failures lead to passive channel failures.

The electrical power distribution for the architecture was based on an approach for fault-tolerant electric power presented in reference C. Critical power is provided to the IAPSA elements through four electrical load management centers. The candidate architecture assumption is that each system element is connected to a single power source. It was assumed that the connection was organized so that loss of a single power source could not reduce the redundancy of any sensing, computing, or actuation device by more than one level. With this assumption, safety is not affected by a sequence of electrical source failures until the loss of three eliminates the computing function and body motion sensing.

There are many three-failure loss of safety situations and two-failure loss of mission situations involving electric power sources. Because the communication devices and the dependent devices use the same power source in a single connection system, redundancy management in the FTP sees the effect of a power loss as a known loss of communications with one-fourth of the system devices. A detailed electrical connection plan was not developed for the candidate architecture analysis. More detailed modeling of the electric power distribution was performed for the refined architecture described in section 5.

3.4.3 Reliability Modeling

The functional block organization of the system elements used for failure analysis was also used for reliability modeling. Each of the reliability models covers a section of the system containing key sensing or actuating or computing elements. Data transfer elements are included in the section models where their failure has a permanent effect. Table 3.4.3-1 shows the elements included in each of the flight control section models. The same information for the propulsion control elements on a single propulsion system is presented in table 3.4.3-2.

The reliability models are used to estimate the likelihood of the failure situations identified during the failure analysis. Each section model includes the local effect of hydraulic system, electrical power system, and network failures. The element failure rates and other related information used in the evaluation are shown in tables 3.4.3-1 and -2.

These tables also show the failure rates assumed for the system components. Other parameters assumed for the reliability models are also shown in the tables. Included are the coverage values used for the critical self-test processes described in section 3.4.2. For example, the self-test hardware is assumed to detect 99 percent of the surface actuator position feedback sensor failures as a baseline. Therefore 1 percent of the failures lead to undetected failures and a resulting surface shutdown. There are also entries for components with significant active failure modes. In these cases, the table shows the fraction of total device failures assumed to be "active" failures.

To simplify the reliability evaluation, some conservative assumptions were made about the temporary effects of network element failures. The goal was

Table 3.4.3-1. Section Models - Flight Control

Name	Devices	Number	Failure rate (X10^{-6}/hr)	Comments
Cockpit	Pitchstick sensors	4	10	
	Rollstick sensors	4	10	
	Rudder pedal sensors	4	10	
	Cockpit nodes	4	15	
	Cockpit DIUs	4	15	
Sensors	Body rate gyros	8	50	
	Body accelerometers	8	30	
	Sensor nodes	4	15	
	Sensor DIUs	4	15	
Air	Total pressure sensors	4	20	
	Static pressure sensors	4	20	
	Angle of attack sensors	4	33	
	Angle of sideslip sensors	4	33	
	Sensor nodes	4	15	
	Sensor DIUs	4	15	
Fcom	FTP channels	4	200	
	Root link/root node	6	20	
	NW interface	6	20	
Pit (2 surfaces)	Actuator processor	4	50	COV = .95
	Actuator position sensor	4	10	COV = .99
	Valve drive group	8	-	
	Control valve	4	15	AFF = .333x10^{-4}
	Canard DIUs	4	37.5	
	Canard nodes	4	37.5	
	Electrical load Management center	4	20	
	Hydraulic power supply	2	20	

Notes:
AFF – active failure fraction
COV – coverage fraction

Table 3.4.3-2. Section Models - Propulsion

Name	Devices	Number	Failure rate $(X10^{-6}/hr)$	Comments
Inlet	PS_D sensor	2	15	
	PT_{NS} sensor	2	15	
	PS_{NS} sensor	2	15	
	Inlet nodes	2	37.5	
	Inlet DIUs	2	37.5	AFF = .05
	Upper ramp actuator	1		
	Inner ramp actuator	1		
	Bypass ring actuator	1		
Face	PT2 sensor	2	15	
	TT2 sensor	2	85	
	Fan guide vane actuator	1		
	Compressor guide vane actuator	1		
	Engine nodes	2	37.5	
	Engine DIUs	2	37.5	AFF = .05
Engine	N1 sensors	2	50	
	N2 sensor	1	50	
	PT4 sensor	1	40	
	FTIT sensors	2	100	
	PT6 sensors	2	40	
	Engine nodes	2	37.5	
	Engine DIUs	2	37.5	AFF = .05
Fuel	Main fuel metering actuator	1		
	Fuel flow sensor	2	40	
	Main fuel shutoff	2	11	AFF = .01
	Main fuel pump	2	100	
	Engine nodes	2	37.5	
	Engine DIUs	2	37.5	AFF = .05
Ecom	FTP channels	3	200	
	Root link/root node	4	20	
	NW interface	4	20	
After	Zone fuel metering actuator	5		
	Zone fuel shutoff solenoid	10	11	AFF = .01
	A/B lightoff detector	2	5	
	Engine nodes	2	37.5	
	Engine DIUs	2	37.5	AFF = .05
	A/B ignitors	2	80	
Nozzle	Upper flap actuator	1		
	Lower flap actuator	1		
	Convergent actuator	1		
	Nozzle nodes	2	37.5	
	Nozzle DIUs	2	37.5	AFF = .05
Prop	Fail passive electronics	2	15	COV = .99
	Actuator position sensor	2	10	COV = .99
	Control valve	2	15	AFF[1] = .0333
				AFF[2] = $.333 \times 10^{-4}$
	Engage solenoid	2	11	AFF = .01

Notes:

[1] Control valve jam

[2] Power ram jam

AFF – active failure fraction

COV – coverage fraction

to use the evaluation results to indicate potential problems with the network operation. All network element failures, regardless of failure mode, were assumed to cause loss of all devices on the entire network during the repair period. To scope the hazard, it was assumed that all repair periods are 1 second long.

A special failure analysis concern was the hazard associated with active DIU failure modes. To assess the potential problem, a fixed fraction of all DIU faults were assumed to be active failures.

Two models were created for some of the sections, one version to predict the probability of loss of flight safety and one to predict the probability of loss of full mission capability. There were two reasons for this. First, the evaluation tool only provides the probability of reaching the model absorbing states that correspond to either a loss of safety or a loss of mission capability. Second, it turned out to be more convenient (especially during the refined configuration evaluation effort) to build separate models so that failure condition-peculiar model reduction techniques could be used.

The overall probability of system failure was estimated by combining the results from the individually solved section models. If the sections are completely independent, the system failure results from each section are added together to provide a good first order estimate for total unreliability. Higher order correction terms are missing, but the answer is adequate for highly reliable systems. On the other hand, combining section results must be done very carefully for the candidate system because the sections are not totally independent. Some of the common elements affect the state of more than one section. Therefore, certain failure sequences appear in more than one section model. Also, some section failure states need to be considered together to determine system success or failure. For example, two sections may have failure conditions with a level of performance that is not safety critical when considered one at a time. However, the coexistence of the degraded states in both sections may result in system failure. It should be noted that these kinds of problems can be minimized by careful grouping of the elements into sections.

The section models were developed in an iterative manner in the candidate evaluation effort. Usually, the first version of the section model covered just a few of the failures and failure modes. A relatively complete section model was built containing all possible failure states, regardless of likelihood. The model was then evaluated and simplified to eliminate unimportant characteristics. Next, a more detailed version was created by adding more failure situations to the model and repeating the evaluation and simplification cycle. Early in the evaluation, certain failure sequences were seen to dominate the unreliability of the candidate configuration. Because configuration changes were necessary, the analysis was not carried to completion. Once it was clear that the dominant failure sequences had been determined, the modeling effort was terminated. Some specific points about the resulting section models are discussed in the remainder of this section.

Flight Control Models

The flight control section models are shown in table 3.4.3-1. These models were built to evaluate the loss of safety failure conditions, with the exception of the PIT model. During model development and evaluation, it became clear that the mission failure likelihood would be dominated by single-failure situations. Because none of these were identified in the failure analysis for most of the flight control functional blocks, a mission failure version of the model was not built.

The COCKPIT model was based on the safety-critical components, nodes, and associated DIUs of the pilot command sensing functional block. The model captures sensor exhaustion failures, dependency on communication elements, and nearly coincident sensor failures. Also included were the temporary exhaustion and nearly coincident network recovery failure situations.

The body motion sensing functional block was evaluated with the SENSOR model. All of the failure situations covered in COCKPIT are included, except for nearly coincident like sensor failures. As described previously, nearly coincident sensor failures are not a threat with eight devices until two sensors have failed. However, the model includes the nearly coincident network-sensor failure situation because it is still a problem.

The AIR model was based on the airflow sensing functional block. This model treated the quadruple-redundant static and total pressure sensors as though they were safety critical. This was a conservative approach because the defined manual control function does not require them. This inclusion had no significant effect on the model results. The model includes all of the failure situations covered in the COCKPIT model.

Two FCOM models were built based on the flight control computing functional block. One of the models evaluates the loss of safety failure condition. Another model was defined to evaluate the likelihood of permanent loss of either network. In such a situation, the subsequent failure of any safety-critical sensor is catastrophic. The evaluation confirmed the assumption that the permanent failure of a single network in the candidate system was not a part of any of the significant system failure sequences.

The PIT model treated the failure behavior of a pair of control surfaces based on the canard control functional block. As mentioned previously, two versions of PIT were built, one to the safety criteria and one to the mission criteria. The safety model included the following failure situations: (1) a single surface jammed or hardover, (2) a pair of passive critical surfaces, and (3) a dual hydraulic failure. Additionally, the temporary exhaustion and nearly coincident actuator-network recovery failure situations were covered. The mission model only included the dominant failure sequences, all of which cause a single passive surface.

The results of these two PIT models were extended to estimate failure contributions due to the four primary surface pairs. A weakness of the candidate architecture results is that the dual passive surface failure situations did not account for passive surface combinations on different axes. For example, the likelihood of system failure due to a passive canard and passive flaperon was not calculated. It should also be noted that the contribution to mission failure from the secondary flight control devices was not evaluated.

Propulsion Control Models

The section models for the propulsion system are shown in table 3.4.3-2. These models were built to evaluate the likelihood of a single system losing either full or normal performance capability. In accordance with the general failure analysis ground rules presented earlier, loss of full performance on either engine causes loss of mission. Loss of vehicle safety occurs when both engines have less than normal performance. Temporary exhaustion situations are included in many of the propulsion control models to quantify their likelihood. It was not clear whether propulsion control functions could be designed to tolerate temporary exhaustion situations. In any case, the numerical results showed that such failures were not significant. The following discussion provides additional information about the specific models.

The INLET model includes elements from both the inlet sensing and inlet control functional blocks. Loss of full performance capability is the model failure condition because that is the result of most inlet element failures. The model includes sensor and actuator channel exhaustion failures and dependency on communication elements. Inlet device control valve jams are incorporated as a significant active failure mode. Inlet device jams, leading to a loss of normal capability, are modeled and visible in the section results.

The inlet flow model was assumed to be fully operational for redundancy management. Total loss of one of the necessary sensors results in either loss of normal performance capability or fixed-inlet operation. In either case this means that the flow model is available when needed. The only situation that might violate the assumption is the nearly coincident failure of two sensors covered by the inlet flow model. Because this unlikely situation cannot contribute significantly to system failure, it was not treated in the model.

The fan face pressure and temperature sensors from the fan face sensing block and the fan and compressor guide vanes from the vane control functional block are included in the FACE model. This model was built to evaluate the likelihood of element failures causing loss of normal performance capability. The same failure situations included in INLET are covered in FACE.

The ENGINE model includes just the engine sensors from the engine core sensing functional block. The model failure condition was loss of data from two or more types of core sensors. This is a loss of normal performance failure condition. ENGINE also included the nearly coincident sensor-network recovery situation and temporary exhaustion failure situation. In addition, ENGINE modeled the likelihood of nearly coincident failures among the five sensor types.

Most of the elements in the main fuel control functional block are contained in the FUEL model. The exception is the dual fuel pump that was modeled separately. The model failure condition was loss of normal performance capability. One special situation for this model is that there are no undetected position sensor failures in the main fuel metering actuator. Also, active fuel shutoff valve failures that cause uncommanded engine shutdown are modeled.

FCOM models the behavior of a triplex FTP that performs the propulsion system computing for a single system. Two versions of FCOM were built. The failure condition for one version was loss of normal performance capability. The second version included loss of a single network as a special failure condition. This version verified that failures associated with single network operation are not significant contributors to system failure.

Most of the elements from the afterburner fuel control functional block are handled in the AFTER model. The exceptions are the ignitors and light off detectors, which are modeled separately. AFTER models device failures that cause the system to lose full performance capability. This model includes exhaustion failures, dependency on communication elements, device jam, control valve jam, and temporary exhaustion.

NOZZLE includes the devices from the nozzle control functional block. Most nozzle element failures reduce system performance to the normal level. The model includes nozzle device jam failures that lead to a worse condition, loss of normal performance. NOZZLE contains all the failure situations included in the AFTER model.

3.4.4 Reliability Results

The most important benefit of early system evaluation is that it provides indications of the system's strengths and weaknesses. The reliability results available to the system design team must be detailed enough to be used for making improvements. The system failure probabilities should be categorized by specific failure situation to assist the reliability evaluation. This information should provide insight into the operation of the fault-tolerant system, which can then be used to guide necessary system design changes.

A fault-tolerant system can fail in several ways. Redundancy exhaustion is the most obvious failure mechanism. When enough of the redundant devices fail, the function can no longer operate. For highly reliable systems, imperfect performance of the redundancy management processes becomes important. In this study, nearly coincident like sensor failures were identified as a cause of early system failures. Because this failure mechanism is related to the duration of the failure recovery process, reliability modeling is necessary to specify a requirement for the system. This study also defined several failure mechanisms involving operation of the I/O network. The detailed results show whether or not the specific building block configuration is satisfactory for the IAPSA II system.

The design also incorporated elements that were subject to postulated rare active failure modes or failure modes that could not be detected by simple redundancy management processes. One evaluation question is whether the simpler, lower performance processes can support the system requirements. If not, more complicated hardware and software is needed. Therefore, the detailed results must quantify the contribution of these special failures to system unreliability. A complication is that the input failure mode data is generally not available for devices early in the design process. Good quality data of this kind is usually the result of detailed failure analysis of inservice equipment failures.

Likewise, failure data to support the evaluation of the undetected failure mechanism is difficult to obtain. To determine good coverage values for the processes used in the candidate system definition, a full redundancy management performance analysis must be performed using device failure characteristic data. Again, information at this level of detail is usually not available for early concept development work. The approach taken for this study is to use assumed values to indicate the magnitude of the threat associated with the respective failure mechanism. These results must then be treated as having a much higher than normal level of uncertainty.

Design concept results containing sequences with a lot of uncertainty must be evaluated carefully. Different designers will interpret the same results very differently. There are several options available for design concepts whose reliability is dominated by failure sequences with a high degree of uncertainty. One option is early prototype development of the devices and redundancy management processes in question. Evaluation of the prototypes would provide accurate data for assessment and improvement of their design and performance. Alternatively, compatible design concepts could be developed in parallel to be available as a backup if the original concept turns out unsatisfactory after undergoing detailed development. The program benefit of the rough evaluation of these more complicated failure mechanisms is the early warning provided for specific areas of possible development risk.

The model results for the loss of safe flight and landing capability are shown in table 3.4.4-1. The safety data are based on mission time of 3 hours. The results are divided into functional blocks, which show how the loss of specific functions contribute to the loss of safety. These results come from the flight control models for COCKPIT, SENSOR, AIR, FCOM, and PIT (safety condition).

Table 3.4.4-1. Safety Reliability

Functional block	Probability x 10^{-7}
FC sensing	
Pilot	.0023
Body motion	5.08
Airflow	.0078
FC computing	.012
FC surface control	
Pitch	.19
Roll	.38
Yaw	.19
Hydraulic power	.036
Dual propulsion control	.0076
Total	5.9

The contribution to loss of safety from the propulsion system was based on situations where failures cause one system to have less than normal performance capability. The FACE, ENGINE, and FUEL models were evaluated for a 3 hour period to establish this probability. Additionally, the small contribution due to jam failures in the INLET and NOZZLE models was included in the single system value. This value was then used to estimate the likelihood of the unsafe situation where both systems have less than normal performance.

Before combining the results from the separate models, it was necessary to eliminate the double counting of communication element failure sequences. For example, the ENGINE and FUEL models both contaian the engine nodes/DIUs. Identical dual failure sequences are contained in both models. The probability sums were adjusted where necessary so that common sequences were only counted once.

A few failure sequences dominated the safety reliability for the candidate architecture, preventing it from meeting the system requirement. The predominant sequence was loss of body motion sensing in a temporary exhaustion failure situation. This two-failure sequence occurs when a node or DIU fails, leaving the system vulnerable to subsequent repair on another network. When the subsequent network element failure causes the other network to shutdown for repair, only two good sensors are accessible instead of the three needed to estimate the three axis states. A key assumption in temporary exhaustion failure modes is that the network repair exceeds the time the system can tolerate loss of control. Another assumption affecting the likelihood of this failure sequence is that all network element failures lead to a long repair period.

The other dominant loss of safety sequences are associated with surface control. The first situation is a jammed or stuck single surface. The two most common failure sequences are (1) a mechanical jam of the power ram and (2) a detected failure-undetected failure sequence in the two actuator channels for one surface. The second dominant surface control situation is a pair of critical surfaces failing passive. The most common cause of this situation is an undetected actuator channel failure on one surface leading to central safe action, followed by an undetected failure on a channel of the second surface. It should be noted that the contribution to loss of safety from passive pairs is understated in table 3.4.4-1 because the probability was calculated for pairs on the same control axis. Passive surface pairs on different axes were not included.

The element failure modes that take part most often in surface control failure sequences are undetected actuator channel failures and mechanical jam failures. Undetected failures include actuator processor or position sensor faults not covered by the local redundancy management as well as active DIU faults. There is a large uncertainty associated with the probability of these failures. Mechanical jam of the power ram, for example, is usually considered to fall in the extremely improbable category. The other channel failures were characterized by a coverage and active failure fraction in the reliability model. As mentioned earlier, these values are not well known early in the system life cycle; however, for the nominal values used in this analysis, the surface control failure sequences were significant to aircraft safety.

The full mission capability reliability for the system is shown in table 3.4.4-2. All of the propulsion control models were used in the mission evaluation. The models were evaluated using a mission time of 1 hour. Any failure reducing a single propulsion system below full performance capability causes a loss of full mission capability. Active DIU failures were not incorporated directly in the propulsion control models. The effect was calculated separately assuming that inlet and nozzle active DIU failures caused loss of FMC capability due to fixed operation, while active engine DIU failures cause loss of normal performance capability for the engine.

With the exception of the surface control devices, elements from the flight control group were not included in the mission capability evaluation. The flight control models described earlier covered safety critical elements that lose mission capability at the same time they lose safety. These elements therefore don't affect mission reliability until the third failure. Early evaluation results showed that only one and two failure

Table 3.4.4-2. Mission Capability Reliability

Functional block	Probability x 10^{-4}
FC surface control	
Pitch	.18
Roll	.36
Yaw	.18
Propulsion system (per engine)	
Fixed inlet	.046
Fixed guide vanes	.031
Engine core sensing	00066
Core fuel metering	.016
Afterburner metering	.076
Fixed nozzle	.045
Engine computation	.0015
Propulsion DIU active fault*	.11
Aircraft total	1.4

*Not in models

sequences contributed significantly to loss of mission capability for the candidate architecture. Of all of the flight control models described earlier, only PIT had a loss of mission capability at the one-failure level. Therefore only the mission capability version of the PIT model, which evaluates the likelihood of a single passive surface, was used for the loss of mission capability evaluation.

The mission success likelihood was unsatisfactory based largely on failures requiring central actuator management action to safe actuation devices. These are all cases in which full mission capability is lost at first failure. The specific failures consisted of active DIU failures, undetected actuator controller faults, and propulsion actuator control valve jams. Like the flight control actuation safety situation, all of these failures are modeled with parameter values that have a large range of uncertainty.

Several aspects of the candidate architecture were not completely modeled; however, the results were carried far enough to show the need to change the design concept to better meet the reliability requirements. The key safety concern is that certain two-failure sequences cause loss of capability. Similarly, certain single failures can cause loss of mission capability. The specific situations and the resulting design refinements are discussed in section 5.

4.0 CANDIDATE PERFORMANCE EVALUATION

This section details the performance analysis of the IAPSA II candidate system architecture. The performance evaluation steps shown in figure 4.0-1, are described in reference 5. An overview of the key methodology steps is presented in the following paragraphs.

Application Performance Requirements

Performance requirements deal with allowable update rates and time parameters associated with the defined control functions. These requirements were defined in section 3. After the control functions have been mapped onto an architecture configuration, the configuration must be evaluated to guarantee that the control function performance requirements are met.

This effort defines the digital implementation timing characteristics needed to satisfy the mission and safety requirements. These timing requirements include control law update rates; allowable end-to-end time delay between sensor reads and resulting actuator commands; and limits on the frame-to-frame variability of control cycle actions (jitter).

Architecture Description

The performance analysis is based on an architecture description that defines how the timing of the key application functions is to be controlled. These key sequencing and control details must be defined whether the architecture is based on a custom design or on a building-block approach, such as that used for IAPSA II.

Synthesize Candidate Architecture

This step maps the control functions onto the candidate configuration and details the operation of the sequencing and control functions. Simple

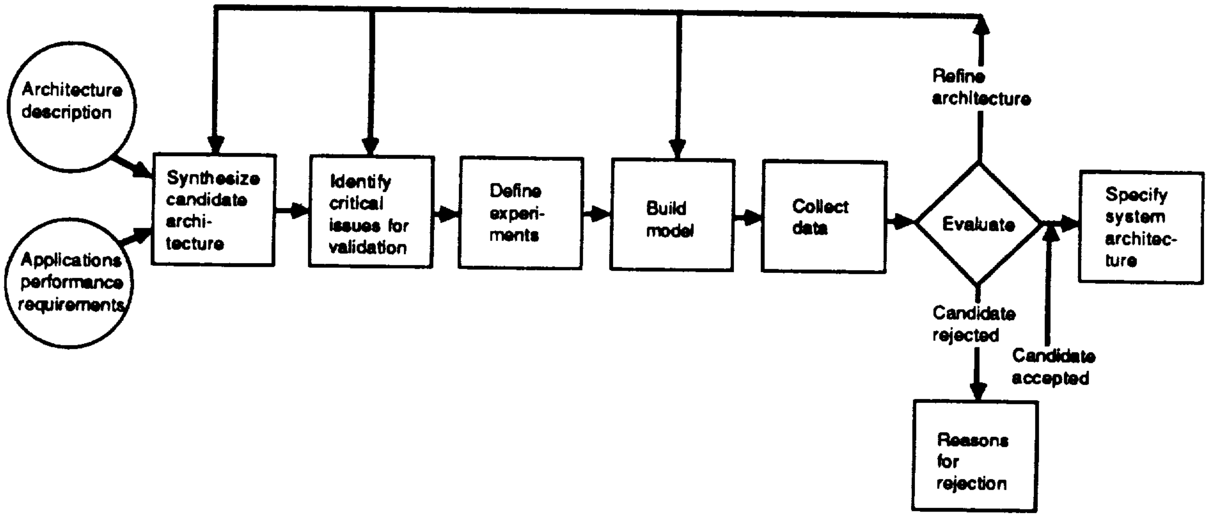


Figure 4.0-1. Performance Evaluation Methodology

timing charts are used to organize the application computing, application I/O, and system processes. This step may eliminate many configurations using these simple timing charts. During this step the underlying system or executive functions may need definition (specification).

Identify Critical Validation Issues

In addition to the normal performance requirements, the performance model should be able to be used for early evaluation of any critical validation issues related to timing. In general, these include situations in which stringent timing needs must be met for safe operation of the system. Identification and evaluation of these issues early in the system design phase will greatly enhance the development and validation effort.

Define Experiments

Experiments are defined for evaluation of the candidate under normal performance and failed conditions. In addition, the critical validation issues are investigated. The experiment definition includes the key input and output variables, the technique to control the experiments, and also the number of runs needed to assess the statistical variations in the data.

Build Model

A simulation model is developed to investigate the critical system behavior. The simulation is normally composed of high-level, low-fidelity functional models of the key application processes, together with models of the associated sequencing and control functions. The decision about what functional behavior should be included or how much detail needs to be modeled is based on the data needed to support the defined experiments. The simulation can be updated as the detailed design is developed. This expanded simulation model can then be used in future development phases to verify the correct implementation of the hardware and software. Finally, the detailed simulation model can be used as part of the supporting deliverable data for validation.

Collect Data and Evaluate

The simulation model is used to perform the experiments defined to study normal performance and the critical validation issues. The data results are evaluated in terms of the application performance requirements. Based on the data evaluation, four possible decisions can be made, as shown in figure 4.0-1: (1) reject the candidate architecture, (2) accept the candidate architecture, (3) refine the architecture, and (4) identify new critical issues.

The candidate architecture is acceptable when all application performance requirements are satisfied. Refinement actions modify the implementation concept, which will require changes in the simulation model. As a result of the insight gained during the evaluation, new critical issues may become apparent. This will require changes in the set of evaluation experiments. Furthermore, as more detail of the configuration is added, additional requirements will be generated.

The following sections describe in detail how the performance evaluation process was applied in the IAPSA II work.

4.1 Synthesize Candidate

4.1.1 Application Performance Requirements

The purpose of the IAPSA II study is the design and validation of a control system architecture for a twin-engine fighter with significant coupling between the airframe and engines. This section describes the application performance requirements adopted for the IAPSA II reference configuration.

The control law design effort defines the necessary timing requirements for each function. The control function is implemented with a repetitive timing cycle that reads the sensors, updates the control law variables, and commands the actuators. The design effort defines the necessary update rate needed for satisfactory performance of the control function. The fundamental performance requirement is then to perform all the computing and I/O activity defined by the design effort in the available update period.

The system specification also requires that the application activity have 100 percent growth capability. In general, growth capability is difficult to measure because of the system complexity. In this study, a simple measure of growth capability was used indicating how much the application activity can increase before reaching system throughput limits.

Control law performance is affected by the end-to-end time delay between the reading of a sensor and the start of the resulting actuator movement. This time delay interval is illustrated in figure 4.1.1-1 for a specific sensor-actuator pair. The effect of time delay on control law performance ranges from imperceptible to rough handling characteristics to loss of control in extreme cases. Specific performance-related time delay limits were not available for the IAPSA II control functions; a time delay value of one control cycle period or frame was assumed to be the criterion for satisfactory performance.

The control law design is usually based on a sampled data approach that implicitly assumes uniform sampling periods (regularity in the control cycle repetition rate). The important control cycle actions with respect to lack of regularity or jitter, are the reading of sensors and commanding of actuators (fig. 4.1.1-1). As with time delay, specific performance requirements were not available for control cycle jitter. Small variations with respect to the update period are acceptable, while large variations are unsatisfactory.

4.1.2 Reference Configuration Analysis

This analysis develops the high-level system performance data that will be used to evaluate the candidate architecture. The focus is on how hardware and software elements of the candidate architecture perform the application functions. In this analysis, the performance of each major group (flight control and engine control) is treated separately. Ultimately a separate simulation model is created for each group.

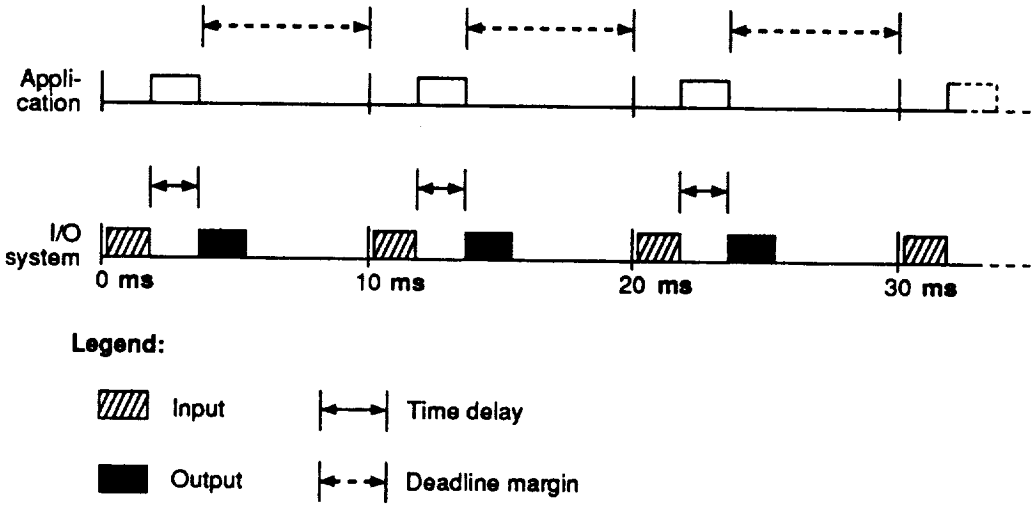


Figure 4.1.1-1. Example Application - Update Rate 100 Hz

The performance model was developed in three distinct, sequential phases. The three phases were (1) the initial timing, (2) initial AIPS timing, and (3) revised AIPS timing. In these phases the application timing data are built up and organized manually using simple timing charts. Situations involving variable timing needs or contention for shared resources are deferred to the simulation model development for promising configurations.

4.1.3 Initial Timing

The organization of the application activity follows the computing subfunction allocation defined in section 3. All functions within the same update rate are lumped into a single rate group. The computing and I/O activities for all the functions in the same rate group are combined. Therefore, if more than one function needs to communicate with the same DIU, the DIU is accessed only once, obtaining all data needed by the functions in that rate group. This consolidation of message traffic reduces the I/O demands of the application.

The initial timing assumes that the control cycle for each application rate group starts with the input I/O activity needed by the rate group, followed by the compute activity, and finally by all output I/O activity. This particular organization of the I/O activity is referred to as separated I/O. The input I/O activity for each application rate group supplies all data from the sensors, actuator positions, etc., requested by all functions during the current control cycle update. The compute activity performs any sensor and actuator redundancy management and updates the control law variables. The output activity consist of sending position and redundancy management commands to all the necessary actuators via the DIUs. This organization of I/O activity and computing activity is shown in figure 4.1.1-1 for a single application rate group.

The initial timing development phase considers a single I/O network that must be shared by all rate groups. The input I/O activity starts each control cycle. I/O activity is nonpreemptable; once started it runs to completion. A sequencing and control function is assumed that operates each rate group in order of priority, with the fastest rate first.

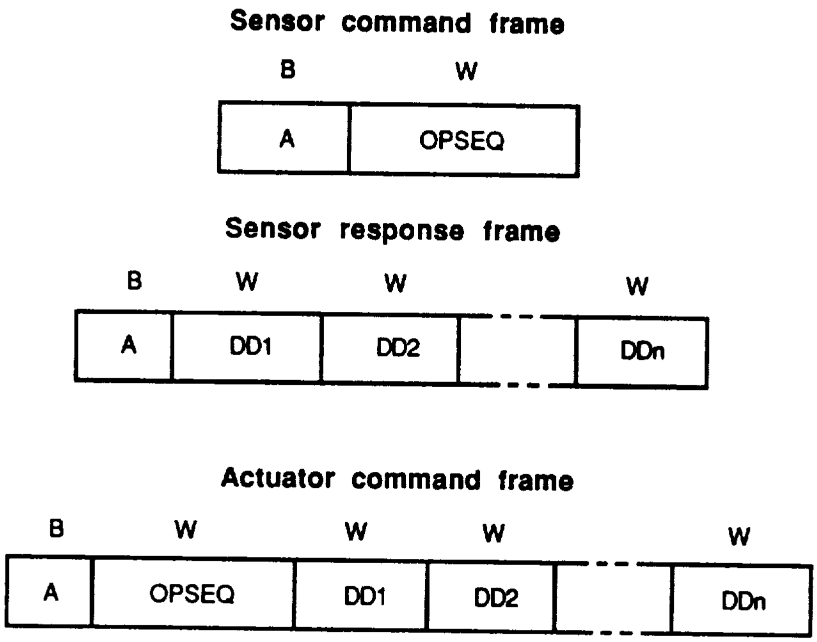
In contrast to the single shared I/O network, the computing for each application rate group is assumed to have its own dedicated processor. Once the input I/O activity for a rate group is completed, the computing activity begins. At the conclusion of the computing activity the control function starts the output I/O activity. When the output activity for that rate group is completed, the control function starts the next rate group scheduled for that minor frame. Note that this assumes that a rate group does not relinquish the network for lower priority activity until the completion of its output I/O activity. The final control assumption is that all rate groups are scheduled for initial execution at the very first minor frame.

The remaining initial model assumptions deal with the I/O activity duration. The two major elements are DIU processing and the duration of the command and response messages. The first element to be discussed is DIU processing. The system operates in a command-response mode. The application rate group sends an I/O message containing a unique operation code to each DIU in sequential order. If appropriate, the DIU returns a response message. The command message and response message (optional) pair that is the basis for communication with devices on a single DIU is termed a transaction. The unique operation code sent in the command message causes a sequence of simple DIU operations that reads sensors or actuator status registers or writes to actuator command registers. Additionally, the DIU formats any resulting sensor data into a response message and transmits it as needed. The DIU requires some overhead processing time to decode and verify the message and prepare the response message. The times assumed for all operations in the timing estimate are illustrated in figure 4.1.3-1.

The second major I/O activity element is the duration of the command and response messages sent on the I/O network. The transmission rate on the I/O network is 2 megabits per second. A primitive format assumed for these messages is illustrated in figure 4.1.3-2. The input I/O activity for a rate group consists of a sequence of transactions with all its DIUs. Each transaction consists of a sensor command frame followed by DIU processing as described above and finally a sensor response frame. The output I/O

Operation	Execution time/ operation (μ s)
Sensor read	15
Actuator command	5
Actuator safe command	5
Actuator status read	5
Actuator position read	15
DIU overhead	10

Figure 4.1.3-1. DIU Operation Times



Legend:

A – DIU address
OPSEQ – DIU operation sequence
DD – Device data
B – Byte
W – Word

Figure 4.1.3-2. Primitive DIU Command Response Assumptions

activity consists of a sequence of output-only transactions for all the appropriate DIUs. These output-only transactions contain only an actuator command frame. The equations describing the duration of input and output I/O activity are detailed in figure 4.1.3-3.

Flight Control Major Group

The sensor and actuator data transfer requirements for the flight control group are illustrated in figure 4.1.3-4 (DIU names are shown in fig. 3.2.1-1). This figure lists the specific number of words required by each DIU within each rate group for the application functions allocated to the flight control group. Evaluation of I/O activity duration data yields the results summarized in figure 4.1.3-5. The computing duration is based on execution of the allocated control subfunctions on a 3 mips processor. The mean computing workload for each control function (less the allowance for growth) was taken from section 4 of reference A. A timeline for the application activity during a major frame is illustrated in figure 4.1.3-6.

Figure 4.1.3-6 shows an overlap in the application computing between the 100 Hz rate and the 25 Hz rate. This indicates that the rate groups will contend for the computing processor in this organization. The major characteristics of time delay for this organization can also be seen in the figure. The values are clearly a fraction of a single minor frame, thus well below the update rate period criterion.

Engine Control Group

The sensor and actuator data transfer requirements for each engine control group are illustrated in figure 4.1.3-7. The resulting input and output I/O activity duration along with the computing duration are shown in figure 4.1.3-8. A timeline for the application activity by rate during a major frame is illustrated in figure 4.1.3-9.

It is clear that the engine control group is very lightly loaded by the application when compared to the flight control group. No timing problems are apparent at this stage of the performance development.

(Eqn 1) input = time to transmit sensor command frames + time to execute sensor command frames + time to transmit sensor response frames

(Eqn 2) output = time to transmit actuator command frames + time to execute actuator command frames

Time to transmit sensor command frames:

= # input command frames X time to transmit word X words/input command frame

Time to execute sensor command frames:

= # input command frames X DIU overhead + total # of sensor reads X sensor read time + total # of actuator status reads X actuator status read time + total # actuator position reads X actuator position read time

Time to transmit sensor response frames:

= total # words in all sensor response frames X time to transmit word

Time to transmit actuator command frames:

= ((# actuator command frames X overhead words/actuator command frame) + total # of actuator commands + total # of actuator safe commands) X time to transmit word

Time to execute actuator command frames:

= # actuator command frames X DIU + (total # of actuator commands X actuator command time + total # of actuator safe commands X actuator safe command time

Figure 4.1.3-3. Input-Output Activity Execution

Rate	DIU operations					Sensor command frame (words)		Sensor response frame (words)		Actuator command frame (words)	
100 Hz	SR	AC	ASC	AS	AP	Overhead	Data	Overhead	Data	Overhead	Data
S1	6					1 1/2		1/2	6		
S2	6					1 1/2		1/2	6		
OFL	1	1	1	1	1	1 1/2		1/2	3	1 1/2	2
OFR	1	1	1	1	1	1 1/2		1/2	3	1 1/2	2
IFL	2	1	1	1	1	1 1/2		1/2	4	1 1/2	2
IFR	2	1	1	1	1	1 1/2		1/2	4	1 1/2	2
TEL	1	1	1	1	1	1 1/2		1/2	3	1 1/2	2
TER	2	1	1	1	1	1 1/2		1/2	4	1 1/2	2
Totals	21	6	6	6	6	12		4	33	9	12
50 Hz											
S1	2					1 1/2		1/2	2		
S2	2					1 1/2		1/2	2		
CP1	3					1 1/2		1/2	3		
CP2	3					1 1/2		1/2	3		
CDL		1	1	1	1	1 1/2		1/2	2	1 1/2	2
CDR		1	1	1	1	1 1/2		1/2	2	1 1/2	2
RL		1	1	1	1	1 1/2		1/2	2	1 1/2	2
RR		1	1	1	1	1 1/2		1/2	2	1 1/2	2
N		1	1	1	1	1 1/2		1/2	2	1 1/2	2
LER		3	3	3	3	1 1/2		1/2	6	1 1/2	6
Totals	10	8	8	8	8	15		5	26	9	16
25 Hz											
S1	1					1 1/2		1/2	1		
CP1	1					1 1/2		1/2	1		
Totals	2					3		1	2		

Legend:

SR	– Sensor read
AC	– Actuator command
ASC	– Actuator safe command
AS	– Actuator status
AP	– Actuator position

Figure 4.1.3-4. Flight Control Computer Estimated Application Timing

Rate	Inputs (µ s)	Compute (µ s)	Outputs (µ s)
100	907	1723	288
50	778	3016	340
25	98	7100	–

Resulting utilization:

Application computing	50.06%
Application I/O	14.83%

Figure 4.1.3-5. Flight Control Computer Estimated Application Timing - Rate Values

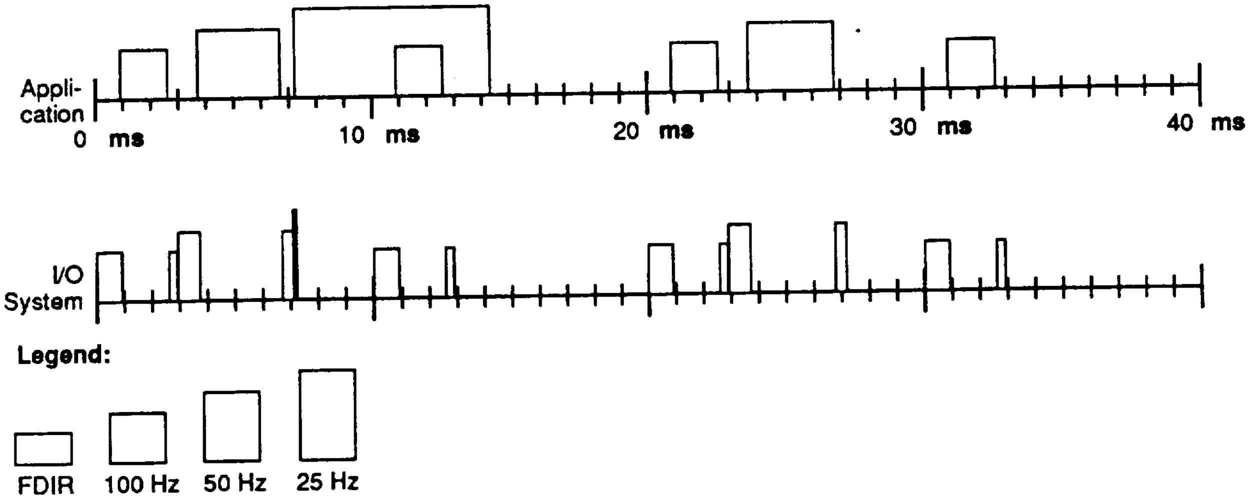


Figure 4.1.3-6. Flight Control Computer Estimated Application Timing

Rate	DIU operations					Sensor command frame (words)		Sensor response frame (words)		Actuator command frame (words)	
	SR	AC	ASC	AS	AP	Overhead	Data	Overhead	Data	Overhead	Data
100 Hz	3	3	3	3	3	1 1/2		1/2	9	1 1/2	6
50 Hz		3	3	3	3	1 1/2		1/2	6	1 1/2	6
25 Hz	7	8	8	8	8	1 1/2		1/2	23	1 1/2	16

Legend:

SR – Sensor read
AC – Actuator command
ASC – Actuator safe command
AS – Actuator status
AP – Actuator position

Figure 4.1.3-7. Engine Control Computer Estimate Application Timing

Rate	Input (µs)	Compute (µs)	Output (µs)
100	213	158	100
50	120	94	100
25	475	1293	230

Resulting utilization:

Application computing	5.28%
Application I/O	6.17%

Figure 4.1.3-8. Engine Control Computer Estimated Application Timing - Rate Values

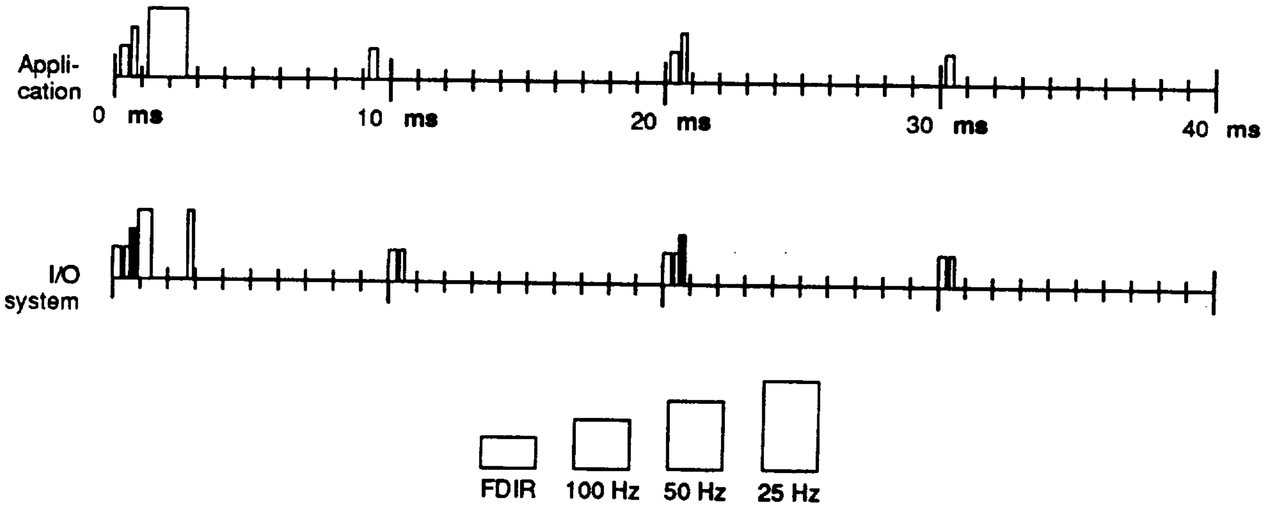


Figure 4.1.3-9. Engine Control Timing - Estimated Application Timing

4.1.4 Initial AIPS Timing

The next phase in the performance model development adds more detail to the modeling of the I/O activity. The elements of the AIPS-based candidate configuration are shown in figure 4.1.4-1. One aspect of AIPS operation is the exact voting of output data and the source-congruent voted exchange of input data across FTP channels. The model of I/O activity duration presented in the previous section is expanded to include this voting process, in which the IOP uses the data exchange.

The IOP executes some preprocessing and postprocessing functions for the I/O activity. The IOP prepares each command frame for execution by loading the input/output sequencer (IOS). Similarly, the IOP completes processing of sensor response frames (input data) by unloading the IOS. Before sending actuator command frames, the IOP votes all of the associated data while loading the IOS. Similarly, after all sensor response frames are received the data are distributed to all FTP channels via the data exchange. Since the sensor command frames for the input activity do not change from cycle to cycle, it is assumed that the IOP does not vote the associated data. Once loading is complete, the IOS sends command frames to the DIU and receives any response frames, executing without IOP involvement until all the transactions in the chain are completed. The IOS is assumed to require 10 microseconds for overhead processing between consecutive network transactions. This is called transaction turnaround time.

In this phase, each rate group is now assumed to have its own CP and IOP. The rate group computing is explicitly allocated to the CP. The assumed sequencing and control function must now arbitrate between the rate groups for the use of the single network. The extra system loading due to the output voting and input distribution causes some additional delay in the completion of lower priority output I/O activity due to more urgent high priority input I/O activity.

The speed of the data exchange used to calculate the duration of IOP involvement in the I/O activity was 6 microseconds per word for loading the IOS and 8 microseconds per word for unloading the IOS. There were two

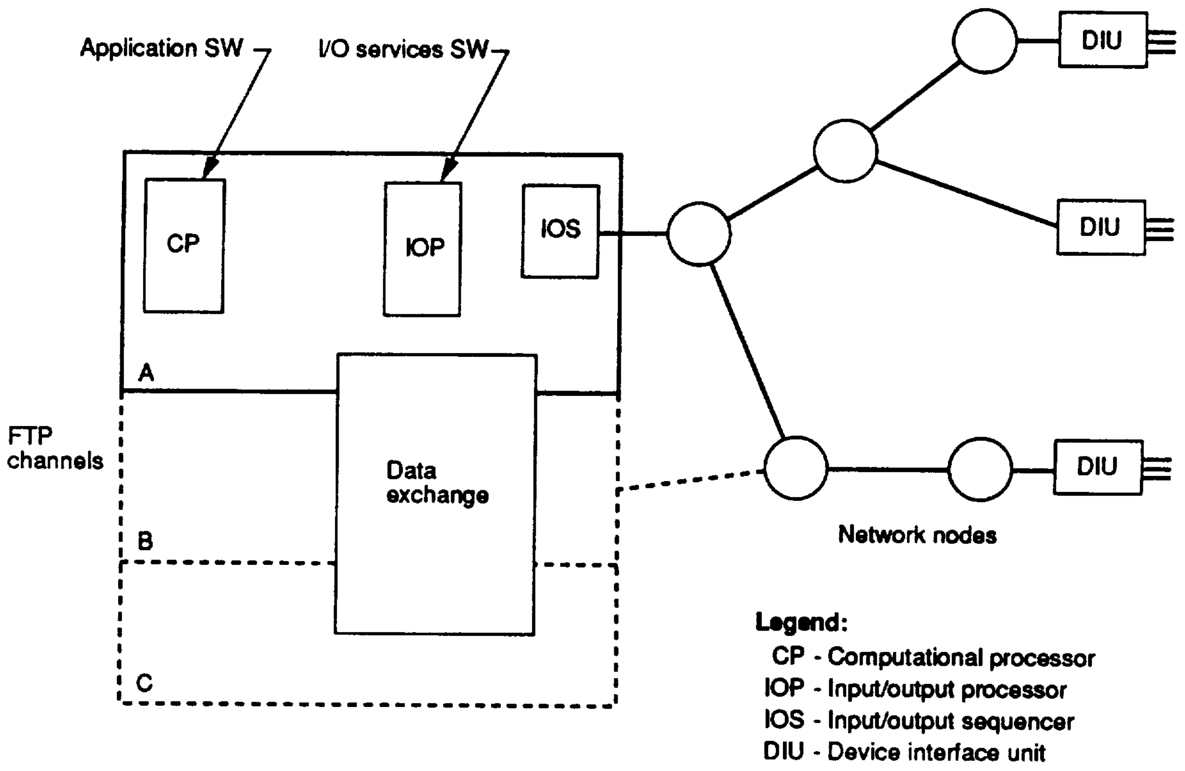

Application SW
I/O services SW
CP
IOP
IOS
A
FTP channels
Data exchange
B
C
DIU
DIU
DIU
Network nodes
Legend:
CP - Computational processor
IOP - Input/output processor
IOS - Input/output sequencer
DIU - Device interface unit

Figure 4.1.4-1. Application I/O Operation

further changes to the modeling of the I/O activity in this initial AIPS timing phase. First, the actual AIPS network protocol is used for data transfer over the network (fig. 4.1.4-2). This figure also indicates the amount of data that must pass through the data exchange for each actuator command frame and sensor response frame. Second, the DIU fixed overhead time was increased to 20 microseconds. The overall I/O activity time equations for the initial AIPS timing estimate are illustrated as equation 3 and equation 4 in figure 4.1.4-3.

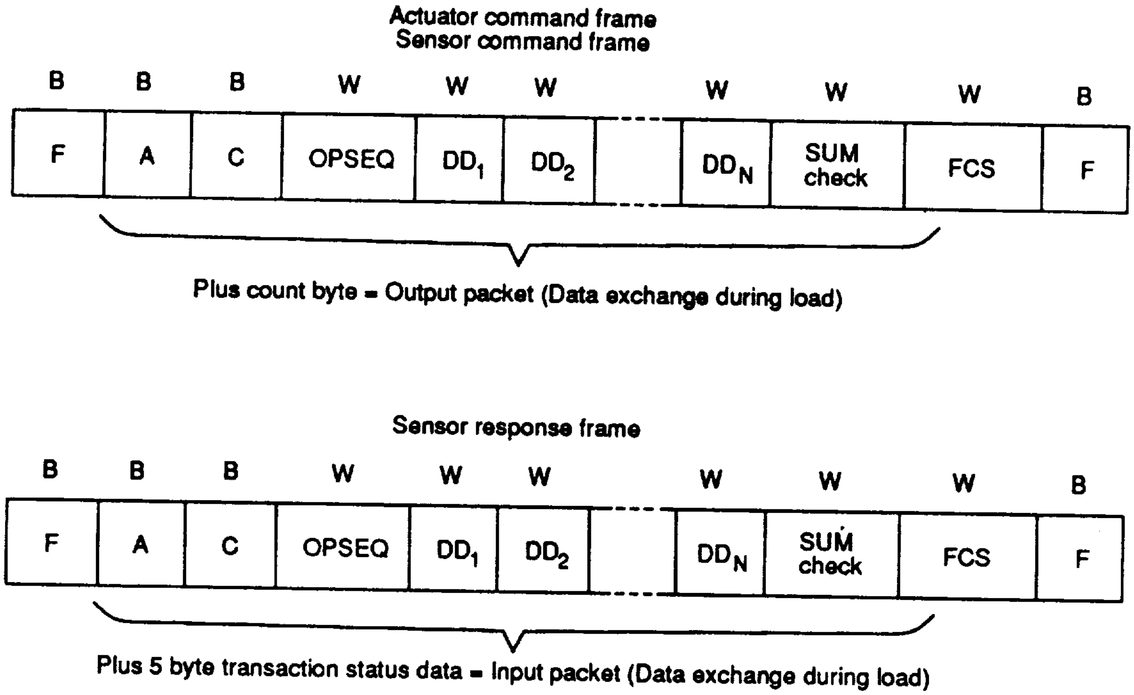
Flight Control Group

The phasing of activity in the flight control group was altered slightly by changing the initial scheduling of the 25 Hz rate group. Rather than start all three rate groups in the very first minor frame, the activity for the 25 Hz is offset by starting it in the second minor frame. The sensor and actuator data requirements for the flight control group are updated to reflect the data exchange usage and the more extensive high level data link control (HDLC) frame formats, as illustrated in figure 4.1.4-4. The resulting timing data are summarized in figure 4.1.4-5. A timeline of this information is illustrated in figure 4.1.4-6.

Two problems are observed in figure 4.1.4-6. First, there is an overlap in the second and fourth minor frame between the 100 Hz input activity and the 50 Hz output activity in the IOP. Second, there is overlap between the 100 Hz application processing and 25 Hz application processing in the CP. This indicates that contention for the processor will occur with this initial AIPS organization. Time delay is still well within the one update period criteria.

Engine Control Group

The sensor and actuator data requirements for each engine control group are updated and shown in figure 4.1.4-7. A summary of the resulting I/O activity duration data is presented in figure 4.1.4-8. A new timeline is illustrated in figure 4.1.4-9. There are no apparent timing problems.



Actuator command frame
Sensor command frame

B	B	B	W	W	W		W	W	W	B
F	A	C	OPSEQ	DD_1	DD_2		DD_N	SUM check	FCS	F

Plus count byte = Output packet (Data exchange during load)

Sensor response frame

B	B	B	W	W	W		W	W	W	B
F	A	C	OPSEQ	DD_1	DD_2		DD_N	SUM check	FCS	F

Plus 5 byte transaction status data = Input packet (Data exchange during load)

Legend:

F - HDLC flag
A - HDLC address
C - HDLC control (used to check address)
W - 16 bit work
OPSEQ - Commanded operation sequence
DD - Device data
FCS - HDLC frame check sequence
B - 8-bit byte

Figure 4.1.4-2. HDLC Protocol - DIU Command/Response Frame

(Eqn 3) input = time to transmit sensor command frames + time to execute sensor command frames + time to transmit sensor response frames +time to unload sensor response

(Eqn 4) output= time to load actuator commands + time to transmit actuator command frames + time to execute actuator command frames

Time to transmit sensor command frames:

= # input command frames X time to transmit word X words/input command frame

Time to execute sensor command frames:

= # input command frames X DIU overhead + total # of sensor reads X sensor read time + total # of actuator status reads X actuator status read time + total # actuator position reads X actuator position read time

Time to transmit sensor response frames:

= total # words in all sensor response frames X time to transmit word

Time to unload response:

= total # of response words in all sensor response frames X data exchange unload rate

Time to load actuator commands:

= total # of actuator commands to load X data exchange load rate

Time to transmit actuator command frame:

= (# actuator command frames X overhead words/actuator command frame) + (total # of actuator commands + total # of actuator safe commands) X time to transmit word

Time to execute actuator command frame:

= # actuator command frames X DIU + (total # of actuator commands X actuator command time + total # of actuator safe commands X actuator safe command time

Figure 4.1.4-3. Input-Output Activity Timing for Initial AIPS Timing

Rate	DIU operations					Sensor command frame (words)		Sensor response frame (words)		Data exchange of response data (words)	Actuator command frame (words)		Data exchange of output command data (words)
100 Hz	SR	AC	ASC	AS	AP	Overhead	Data	Overhead	Data		Overhead	Data	
S1	6					5		5	6	11 1/2			
S2	6					5		5	6	11 1/2			
OFL	1	1	1	1	1	5		5	3	8 1/2	5	2	5 1/2
OFR	1	1	1	1	1	5		5	3	8 1/2	5	2	5 1/2
IFL	2	1	1	1	1	5		5	4	9 1/2	5	2	5 1/2
IFR	2	1	1	1	1	5		5	4	9 1/2	5	2	5 1/2
TEL	1	1	1	1	1	5		5	3	8 1/2	5	2	5 1/2
TER	2	1	1	1	1	5		5	4	9 1/2	5	2	5 1/2
Totals	21	6	6	6	6	40		40	33	77	30	12	33
50 Hz													
S1	2					5		5	2	7 1/2			
S2	2					5		5	2	7 1/2			
CP1	3					5		5	3	8 1/2			
CP2	3					5		5	3	8 1/2			
CDL		1	1	1	1	5		5	2	7 1/2	5	2	5 1/2
CDR		1	1	1	1	5		5	2	7 1/2	5	2	5 1/2
RL		1	1	1	1	5		5	2	7 1/2	5	2	5 1/2
RR		1	1	1	1	5		5	2	7 1/2	5	2	5 1/2
N		1	1	1	1	5		5	2	7 1/2	5	2	5 1/2
LER		3	3	3	3	5		5	6	11 1/2	5	6	9 1/2
Totals	10	8	8	8	8	50		50	26	81	30	16	37
25 Hz													
S1	1					5		5	1	4 1/2			
CP1	1					5		5	1	4 1/2			
Totals	2					10		10	2	9			

Legend:

SR – Sensor read
AC – Actuator command
ASC – Actuator safe command
AS – Actuator status
AP – Actuator position

Figure 4.1.4-4. Flight Control Computer Initial AIPS Applicatior Timing

Rate	Network input (µs)	DX of sensor data (µs)	Compute (µs)	DX of actuator data (µs)	Network output (µs)
100	1579	1232	1723	396	476
50	1618	1296	3016	444	628
25	266	144	7100	–	–

Resulting utilization:

CP	50.06%
IOP	25.84%
Network	32.28%

Figure 4.1.4-5. Flight Control Computer Initial AIPS Application Timing - Rate Values

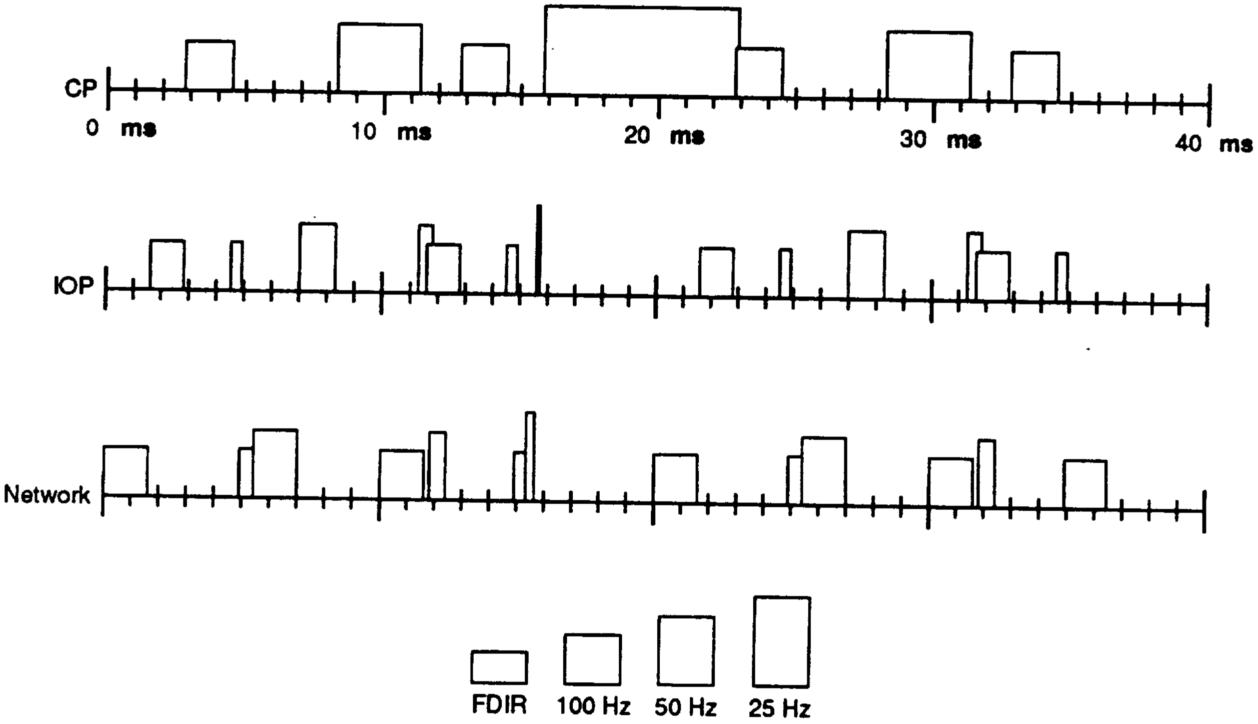


Figure 4.1.4-6. Flight Control Computer Initial AIPS Application Timing

	DIU operations					Sensor command frame (words)		Sensor response frame (words)		Data exchange of response data (words)	Actuator command frame (words)		Data exchange of output command data (words)
Rate	SR	AC	ASC	AS	AP	Overhead	Data	Overhead	Data		Overhead	Data	
100 Hz	3	3	3	3	3	5		5	9	14 1/2	5	6	9 1/2
50 Hz		3	3	3	3	5		5	6	11 1/2	5	6	9 1/2
25 Hz	7	8	4	8	8	5		5	23	28 1/2	5	16	19 1/2

Legend:

SR – Sensor read
AC – Actuator command
ASC – Actuator safe command
AS – Actuator status
AP – Actuator position

Figure 4.1.4-7. Engine Control Computer Initial AIPS Application Timing

Rate	Network input (µs)	DX of sensor data (µs)	Compute (µs)	DX of actuator data (µs)	Network output (µs)
100	287	116	158	57	148
50	218	92	94	57	148
25	335	228	1293	117	278

Resulting utilization:

CP	5.28%
IOP	3.34%
Network	7.71%

Figure 4.1.4-8. Engine Control Computer Initial AIPS Application Timing – Rate Values

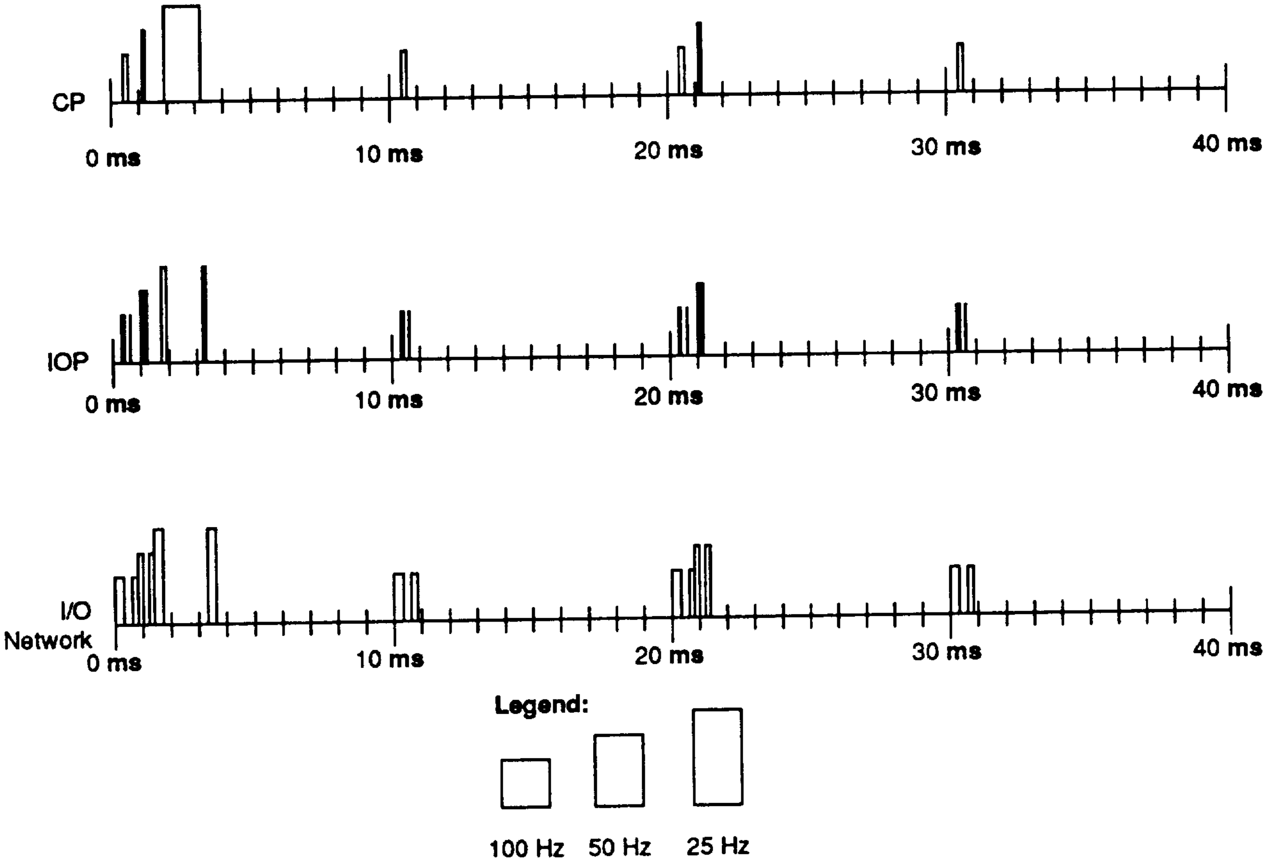


Figure 4.1.4-9. Engine Control Computer Initial AIPS Application Timing

4.1.5 Revised AIPS Timing

In the revised AIPS timing phase of the performance model development the overall I/O activity is reorganized by grouping the sensor read input I/O activity and actuator command output I/O activity into a single network activity per rate group. This I/O organization is referred to as grouped I/O. In this strategy, all sensor read operations and actuator write operations within a single DIU are combined into one transaction per rate group. The transmission of the actuator commands from the previous control cycle is combined with the transmission of the sensor read commands for the current control cycle. This reduces the loading on the I/O network because DIUs that have both sensors and actuators are now only accessed once per application cycle. As a consequence, control law time delay will increase. The equation used to calculate the duration of the I/O activity is shown in figure 4.1.5-1. The sequencing and control function now only keeps track of one I/O activity per rate group. A control cycle begins with the grouped I/O activity that transmits the commands from the previous cycle and requests sensor data for the current cycle. Since the I/O activity starts with loading actuator commands, the IOP activity is usually the first action in a control cycle.

The final change to the model is the assumption of a single CP and IOP that must be shared by the different rate groups. It is assumed that the control function allocates the CP or the IOP to the fastest rate group needing service.

Flight Control Group

The sensor and actuator data requirements for the flight control group, reflecting the new I/O organization, are illustrated in figure 4.1.5-2. Note that the data that previously appeared in the actuator command frame now appears in the sensor command frame column. Also the total number of frames is reduced by the number of actuator command frames. The revised I/O activity duration data are summarized in figure 4.1.5-3. A timeline is presented in figure 4.1.5-4 for the flight control group. The effect of

(Eqn 5) network activity = time to load actuator commands + time to transmit grouped sensor/actuator commands + time to execute grouped sensor/actuator command frames + time to transmit response frames + time to unload sensor responses

Time to load actuator commands:

= total # of actuator commands to load X data exchange load rate

Time to transmit grouped sensor/actuator commands:

= ((# command frames X words/command frame) + total # actuator commands + total number actuator safe commands) X time to transmit word

Time to execute grouped sensor/actuator command frame:

= # command frames X DIU overhead + Total # sensor reads X sensor read time + total # actuator command X actuator command time + total # actuator safe commands X actuator safe command time + Total # actuator status reads X actuator status time + total # actuator positions X actuator position time

Time to transmit response frames:

= ((total # command frames X words/response frame) + total # actuator positions + total # actuator status + total # sensor) X time to transmit word + total # command frames X transaction turnaround time

Time to unload response:

= total # of response words to data exchange X data exchange unload rate

Figure 4.1.5-1. Input-Output Activity Execution

Rate	DIU operations					Sensor command frame (words)		Data exchange of output command data (words)	Sensor response frame (words)		Data exchange of response data (words)
100 Hz	SR	AC	ASC	AS	AP	Overhead	Data		Overhead	Data	
S1	6					5			5	6	11 1/2
S2	6					5			5	6	11 1/2
OFL	1	1	1	1	1	5	2	5 1/2	5	3	8 1/2
OFR	1	1	1	1	1	5	2	5 1/2	5	3	8 1/2
IFL	2	1	1	1	1	5	2	5 1/2	5	4	9 1/2
IFR	2	1	1	1	1	5	2	5 1/2	5	4	9 1/2
TEL	1	1	1	1	1	5	2	5 1/2	5	3	8 1/2
TER	2	1	1	1	1	5	2	5 1/2	5	4	9 1/2
Totals	21	6	6	6	6	40	12	33	40	12	77
50 Hz											
S1	2					5			5	2	7 1/2
S2	2					5			5	2	7 1/2
CP1	3					5			5	3	8 1/2
CP2	3					5			5	3	8 1/2
CDL		1	1	1	1	5	2	5 1/2	5	2	7 1/2
CDR		1	1	1	1	5	2	5 1/2	5	2	7 1/2
RL		1	1	1	1	5	2	5 1/2	5	2	7 1/2
RR		1	1	1	1	5	2	5 1/2	5	2	7 1/2
N		1	1	1	1	5	2	5 1/2	5	2	7 1/2
LER		3	3	3	3	5	6	9 1/2	5	6	11 1/2
Totals	10	8	8	8	8	50	16	37	50	16	81
25 Hz											
S1	1					5			5		4 1/2
CP1	1					5			5		4 1/2
Totals	2					10			10		9

Legend:
SR – Sensor read
AC – Actuator command
ASC – Actuator safe command
AS – Actuator status
AP – Actuator position

Figure 4.1.5-2. Flight Control Computer Revised AIPS Application Timing

Rate	Load command	Network command	Unload response	Compute
100	396	1735	1232	1723
50	444	1842	1296	3016
25	–	266	144	7100

Resulting utilization:

CP	50.06%
I/O	25.34%
Network	27.23%

Figure 4.1.5-3. Flight Control Computer Revised AIPS Application Timing - Rate Values

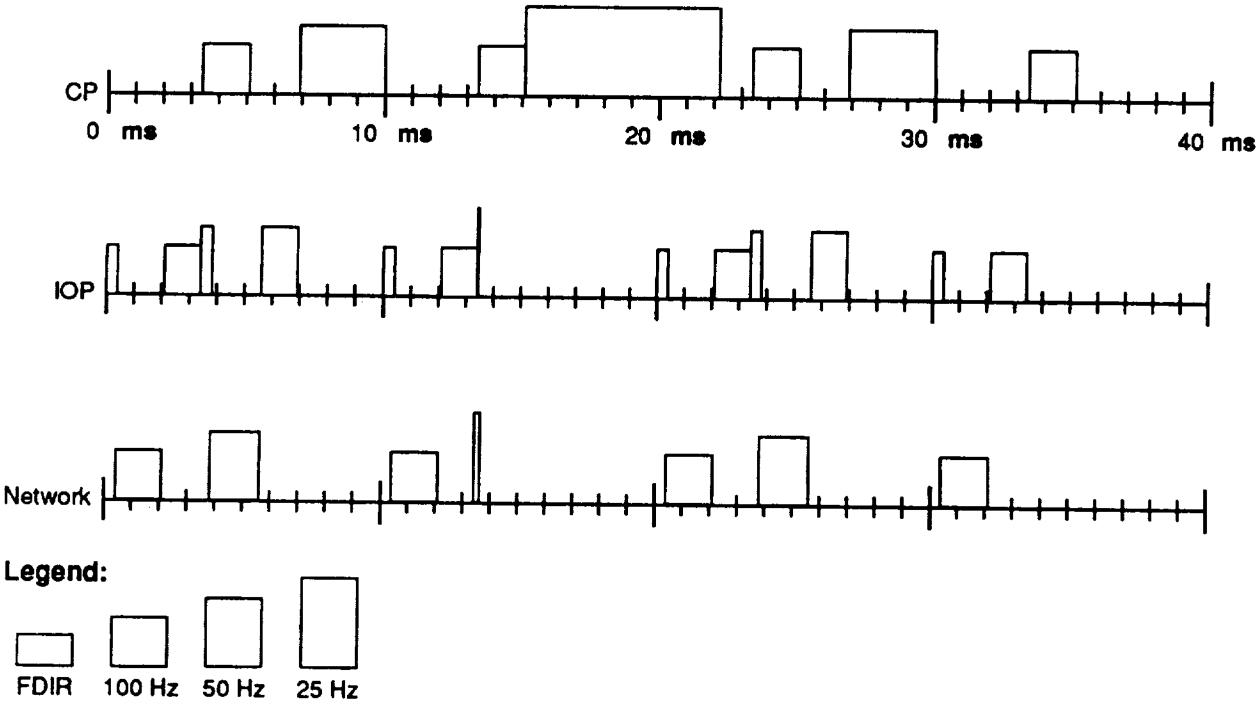


Figure 4.1.5-4. Flight Control Computer Revised AIPS Application Timing

the change to grouped I/O is to reduce somewhat the network use and to increase the system time delay. As a result of the change, the time delay is now approximately one update period.

The performance data developed will be used to form the basis for the simulation model. Other details of the application timing requirements must now be considered. The basic detail of the application workload has been determined, but certain key interactions due to resource contention, fault processing, etc., require more sophisticated analysis methods. At this point in the development we are ready to build a simulation model. But first, based on the initial assessment of the architecture configuration, we must address critical areas for validation and what experiments will be run with the simulation.

4.2 Critical Validation Issues

This section describes some high-level critical performance-related validation issues for the candidate architecture. These critical issues involve ways in which timing performance can prevent safe operation. Special situations or operating circumstances can be a key factor. These critical validation issues are identified to drive the development of the simulation model so that they can then be studied early in the design cycle when the cost benefit impact of improvements is very favorable.

Two performance-related issues are to be studied in this effort. The first is the effect on performance of the relative phasing of the application activity and the system failure detection, identification and reconfiguration (FDIR) activity. The second is the effect on the application activity of the I/O network repair actions.

4.2.1 Application/FDIR Coordination

The execution of the application cycle (computing processes in the CP and I/O activity processing in the IOP) will be affected by the execution of the system FDIR processes. FDIR protects the computing integrity of the

FTP channel. Certain FTP failures are catastrophic if not handled within a few application cycles. FDIR executes at the rate of the fastest application cycle to guarantee rapid fault reaction. Each FTP channel has a watchdog timer for failure protection; FDIR must reset this timer within narrow tolerances. In temporary application overload situations, then, FDIR has the most stringent timing requirement and must therefore have priority over the application processing. The concern to be analyzed is whether certain phasings between scheduled application execution and the FDIR process can degrade performance unacceptably. If performance can be significantly affected, a mechanism to control the relative phasing will be required.

4.2.2 I/O Network Repair

Each major group in the reference configuration has two reconfigurable I/O networks. The sensors and actuators are distributed between these networks for fault-tolerant operation. When a failure brings down one network, the application continues using the sensors and actuators accessible via the other network. The major concern is the interactions between the repair activity and the application activity.

Application I/O activity is terminated on the faulty network for the duration of the repair activity. The system is therefore vulnerable if a fault occurs on the second network before the first network is repaired. The likelihood of this catastrophic event is dependent on the duration of the network repair. Thus the network repair duration is a critical issue.

The repair activity interacts with the nominal execution of the application functions. The repair algorithms involve many processing and I/O activity steps using the IOP and the network. Timely execution of repair is affected by the application's need to communicate over the unfaulted network, which also involves IOP processing. The timing performance of the application must be acceptable during the repair activity. Because of the complexity of the interaction of the repair activity and the application, the duration and effect of the repair activity is addressed most effectively with simulation techniques.

Two I/O network repair strategies are evaluated: one-shot and full regrow. The one-shot strategy is characterized by rapid diagnosis and specific repair actions. Full regrow is the same process used to grow the I/O network at power on. It uses a robust sequence of steps to grow a path to all good devices reachable with the unfailed network elements.

In the one-shot repair strategy, when a communication fault occurs, a chain that accesses all the nodes on the failed network is executed. Based on the nodes that did not respond, the network manager assumes that an I/O link has failed and attempts to repair the failed link by enabling a spare link that is connected to the lost nodes. If communication has been restored to all the lost nodes, the link failure assumption was correct and the network is returned to service. However, if some of the lost nodes remain unreachable, the link failure assumption was wrong and a process associated with a node failure hypothesis is invoked. If at any point in the one-shot process it becomes clear that progress is impossible, for any reason, a robust regrow algorithm is begun. It should be noted that the repair algorithm implemented in the model for the current experiments does not support the repair of a node failure or the reversion to a regrow strategy.

The full regrow repair strategy makes no attempt to determine the type of failure; it uses a robust algorithm to establish a new I/O network topology as if the power had just been applied. Unlike the one-shot repair strategy, the regrow algorithm can establish a path to all reachable nodes (and therefore DIUs) regardless of the type of failure.

4.3 Define Experiments

This section describes the experiments to be performed with the IAPSA II simulation model. It should be noted that experiment 1 was conducted earlier in the program to develop a common experimentation interface. Therefore the experiment numbering for the preliminary simulation experiments begins with experiment 2.

4.3.1 Experiment Configurations

The reference configuration comprises three major groups: a flight control group and an engine control group for each engine. The composition of these groups is described in section 3. The experimental concept is to perform experiments on two separate models: a model of the flight control group and a model of the engine control group. This allows comparison between a large, heavily loaded system and a relatively small, lightly loaded system.

4.3.2 Experiment Procedure

The variability of the processing needs of the application and the system services was captured by modeling elements as stochastic processes. To address the statistical characteristics of the reference configuration, multiple samples of the experiment variables were collected. The experiment data were then plotted on histograms to assess the statistical aspects of the system behavior. Data from 100 major frames were used to evaluate the general characteristics of the application's behavior.

The I/O network repair runs were intended to determine how the application's behavior changes during the repair of a fault. Fifty runs of each failure case were conducted to randomly cover the time of fault occurrence.

4.3.3 I/O Network Repair Time (Experiment 2)

The objective of experiment 2 was to measure the time needed to successfully return a network to service after it experiences a network fault. This experiment also evaluated the effect of the additional network repair processing on the timing performance of the application. This experiment evaluated both configurations, flight control and engine control. Each active link in the network was failed passively in this experiment. The link is failed at a random time during the second major frame and the experiment run terminated one major frame after the completion of the I/O repair activity.

The experiments were run with both the rapid one-shot repair strategy and a full regrow repair strategy. This scoped the minimum and maximum possible repair times for single faults. Each link failure case was repeated 50 times with the fault occurring at a random time with respect to the major frame. This allowed a coarse assessment of the relationship between fault occurrence time and network repair time.

4.3.4 I/O Scheduling (Experiment 3)

The purpose of experiment 3 was to evaluate the effect of the I/O scheduling mechanism on the performance of the application during normal operation. The system services software has two mechanisms available to the application designer to schedule the application I/O activity, scheduled I/O and on-demand I/O. In on-demand I/O, at the beginning of each cycle, the application executing in the CP makes an I/O request and then suspends itself. When the I/O has completed, the application process resumes. In scheduled I/O, the system software executing in the IOP makes the I/O request periodically. The application computing in the CP is scheduled every cycle by the completion of the I/O activity.

Two I/O activity organization schemes were also considered: separated I/O and grouped I/O, which were briefly discussed in section 4.1. In separated I/O, the I/O activity for the rate group is separated into two parts, the input I/O activity and the output I/O activity. In the grouped I/O scheme, all of the I/O activity is grouped together and performed at the same time.

This experiment evaluated the effect of these options on the application performance. There is a single experiment run for each of the I/O scheduling mechanism and I/O activity organization combinations. The duration of each run was 102 major application frames.

4.3.5 FDIR/Application Phasing (Experiment 4)

The objective of experiment 4 was to evaluate the effect of the relative phasing of the application activity and the system FDIR process. The FDIR and application demands were evaluated during normal operation. The system time scheduler assumed for this study has a granularity of one millisecond. That is, time-scheduled tasks can only be specified to the nearest even millisecond. The 10 specific relative phasing situations possible because of the 10 millisecond minor frame period were analyzed. In each simulation it is assumed that the independent FDIR processes in the CP and the IOP are scheduled to run at the same time. The experiment was defined for both the flight control and engine control configurations, for both I/O scheduling mechanisms, and for both I/O activity organizations.

4.3.6 CP, IOP, I/O System, I/O Network Utilization (Experiment 5)

The purpose of experiment 5 was to estimate the use of the key candidate system resources during normal operation. Major areas of resource contention were modeled for this experiment. This includes contention between the different application rate groups, as well as the previously described contention between the application rate groups and the time-critical FDIR function. A preemptive priority sequencing and control algorithm is modeled to control processor allocation. Accounting for contention produced a more realistic growth capability estimate and allowed an evaluation of the effect of execution variability. Measurements representative of each of the application performance requirements defined in section 4.1 are made for normal operation and the network failure cases. The application performance data for failure cases are only collected during the fault recovery action. Thus comparison of failed and non-failed cases will show how the operation changes during the fault recovery process.

4.3.7 Experiment Execution Strategy

The experiment configurations for data collection are illustrated in table 4.3.7-1. The experiments were performed in the following order: (1) FDIR/ application phasing (experiment 4); (2) I/O scheduling mechanism (experiment 3); and (3) I/O link repair (experiment 2). The CP, IOP, I/O system, I/O network utilization experiment 5 data were collected during the running of the previous experiments. Table 4.3.7-2 defines the number of

Table 4.3.7-1. Experiment Configuration

Experiment No.	Configuration ID	Layout	FDIR Coordination	I/O Scheduling	I/O Grouping
3	1	Flight control	Yes	On demand	Grouped
	2	Flight control	Yes	On demand	Separated
	3	Flight control	Yes	Scheduled	Grouped
4	4	Flight control	No	On demand	Grouped
	5	Flight control	No	On demand	Separated
	6	Flight control	No	Scheduled	Grouped
3	7	Engine control	Yes	On demand	Grouped
	8	Engine control	Yes	On demand	Separated
	9	Engine control	Yes	Scheduled	Grouped
2	10	Flight control regrow repair strategy	Yes	Scheduled	Grouped
	11	Flight control one-shot repair strategy	Yes	Scheduled	Grouped
	12	Engine control regrow repair strategy	Yes	On demand	Grouped
	13	Engine control one-shot repair strategy	Yes	On demand	Grouped
4	14	Engine control	No	On demand	Grouped
	15	Engine control	No	On demand	Separated
	16	Engine control	No	Scheduled	Grouped

Table 4.3.7-2. Configuration for Experiment Execution

Experiment ID	Configuration ID	No. of phase or faults	Range of run IDs	Total run potential
4	6	10	1	10
	14	10	1	10
	15	10	1	10
	16	10	1	10
2	10	18	1..50	900
	11	18	1..50	900
	12	4	1..50	200
	13	4	1..50	200

of possible faults or relative FDIR phasings considered for each experiment configuration, as well as the number of repetitions for each link fault experiment.

An actual application's demands will not necessarily grow uniformly across all the system resources. Therefore, to assess growth capability utilization was measured for four key resources: the CP, IOP, I/O system, and I/O network. The CP utilization measures its use in computing the application control laws and the system FDIR. The IOP utilization measures its use in the loading and unloading of application I/O activity, network manager processing and system FDIR. The I/O system utilization measures the end to end operation of the I/O networks for application activity. The I/O system utilization starts when an application I/O request is made and ends when all application activity is complete and the system can immediately respond to a new request. This utilization figure is intended to measure the ability of the I/O system to handle additional I/O requests. The I/O network utilization measures the time from the beginning to the end of IOS execution of I/O activity.

Deadline margin is a figure of merit that indicates how well the system is meeting its periodic control cycle requirements, that is, how close the system is to missing a time-critical action. The critical actions include updating an input or an output set of I/O data to complete the processing for one control cycle before the scheduled starting time for the next cycle.

The time delay figure of merit is an overall indicator of time delay for a particular rate group. The value is computed as the difference between the start of an I/O activity in one cycle and the conclusion of an I/O activity in the next cycle. Deadline margin and time delay were illustrated in figure 4.1.1-1.

4.4 Build Model

This section describes the development of a simulation that models the behavior of an implementation concept for the reference configuration.

4.4.1 Performance Tool

Several tools were evaluated for developing the IAPSA II performance model. A major shortcoming of most tools evaluated was their inability to represent algorithms. This capability is necessary since it allows some key timing parameters to be established using prototype algorithms.

Boeing Advanced Systems selected the Discrete Event Network (DENET) simulation language for the development of the simulation model for the IAPSA II reference configuration. DENET was developed at the University of Wisconsin's computer science department by Dr. Miron Livny. It is a discrete event simulation language based on the Discrete Event System Specification modeling methodology. This methodology is complemented with the MODULA II programming language, which results in a simulation language capable of developing modular system simulations at any level of detail. This feature allows the DENET tool to represent algorithms.

DENET simulations are composed of discrete event modules (DEVM) and arcs, which connect outputs of one DEVM to inputs of another. Each DEVM is programmed to contain some function of the system; the function can be either a high-level abstraction or a very detailed emulation. DEVMs receive input and generate output through ports. The ports of DEVMs are interconnected with arcs. A simulation model consists of a group of DEVMS connected together with arcs. Each instance of a DEVM is characterized with input parameters. The input parameters allow the module to parameterize the specific DEVM's behavior so that modular building blocks can be supported.

Discrete Event Module

A fundamental feature of a DEVM is its connections with other DEVMs. The connections are implemented with input ports and output ports. A port is created on a DEVM when a connection is made to another DEVM with an ARC. The orientation of the ARC defines whether a port is an input port or an output port. DEVM ports are associated with either an INPUT event or an output variable. Each INPUT event and OUTPUT variable has an associated data type for transferring information when the variable is assigned. The data type can be a simple Boolean or an integer, or can be a composite type such as a record. Output from a DEVM is generated with an assignment statement to an output variable. The output port is generally connected to an input port on another DEVM. At the time an output is assigned, an input event is triggered in the destination module. The input event is the mechanism by which the modeler controls the processing in the DEVM. The resulting processing can change the state of the DEVM based on the received data, perform a complex algorithm, and/or schedule an output port assignment to take place at a defined future time.

An example of an input event and output variable for a DEVM that models an AIPS node (AIPSNODE) is shown in figure 4.4.1-1. The input event has the name NodeCommandFrame and the output variable is named NodeResponseFrame. When an input event occurs, the AIPSNODE processing is initiated. An example of the event-driven processing is shown in the following AIPSNODE DEVM description. The first action taken, if the input event message corresponds to an enabled node port, is to assign the received message to the output variables corresponding to all other enabled node ports. The second action is to examine the message to see if it contained a command for that specific node. If true, the directed action is taken, and a response message is generated and assigned to the NodeResponseFrame output variables corresponding to the node's enabled ports.

ARC Definition

An ARC definition is used to connect output variables of a DEVM to the input events of other DEVMs, as illustrated in figure 4.4.1-2. This figure shows half of the connection between two AIPSNODE DEVMs in a network. The ARC definition, NodeToNode, defines the output port NodeResponseFrame to be connected to the input port NodeCommandFrame.

Parameter Characterization

The parameters of a DEVM are used to fully characterize the behavior of a DEVM. One use is the definition of a specific instance of a generic DEVM

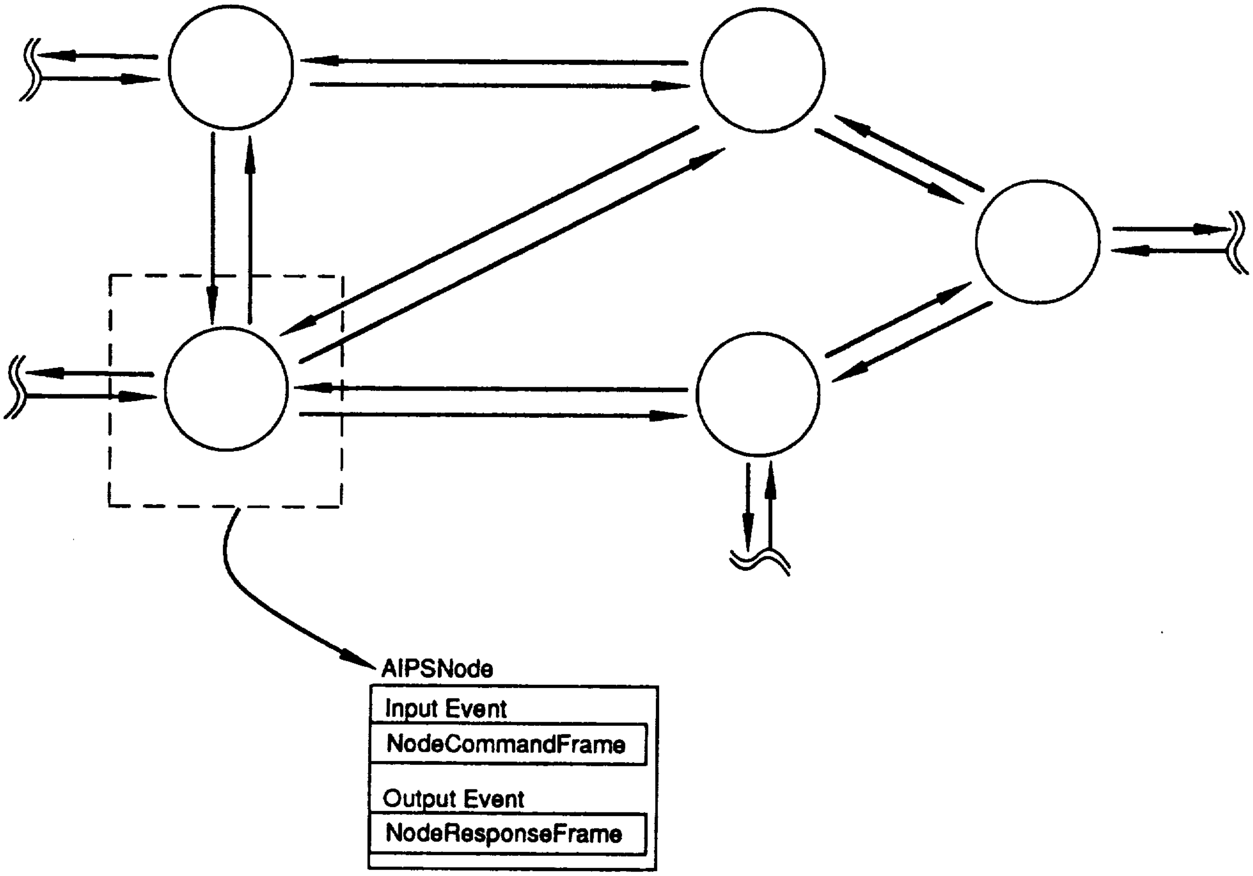
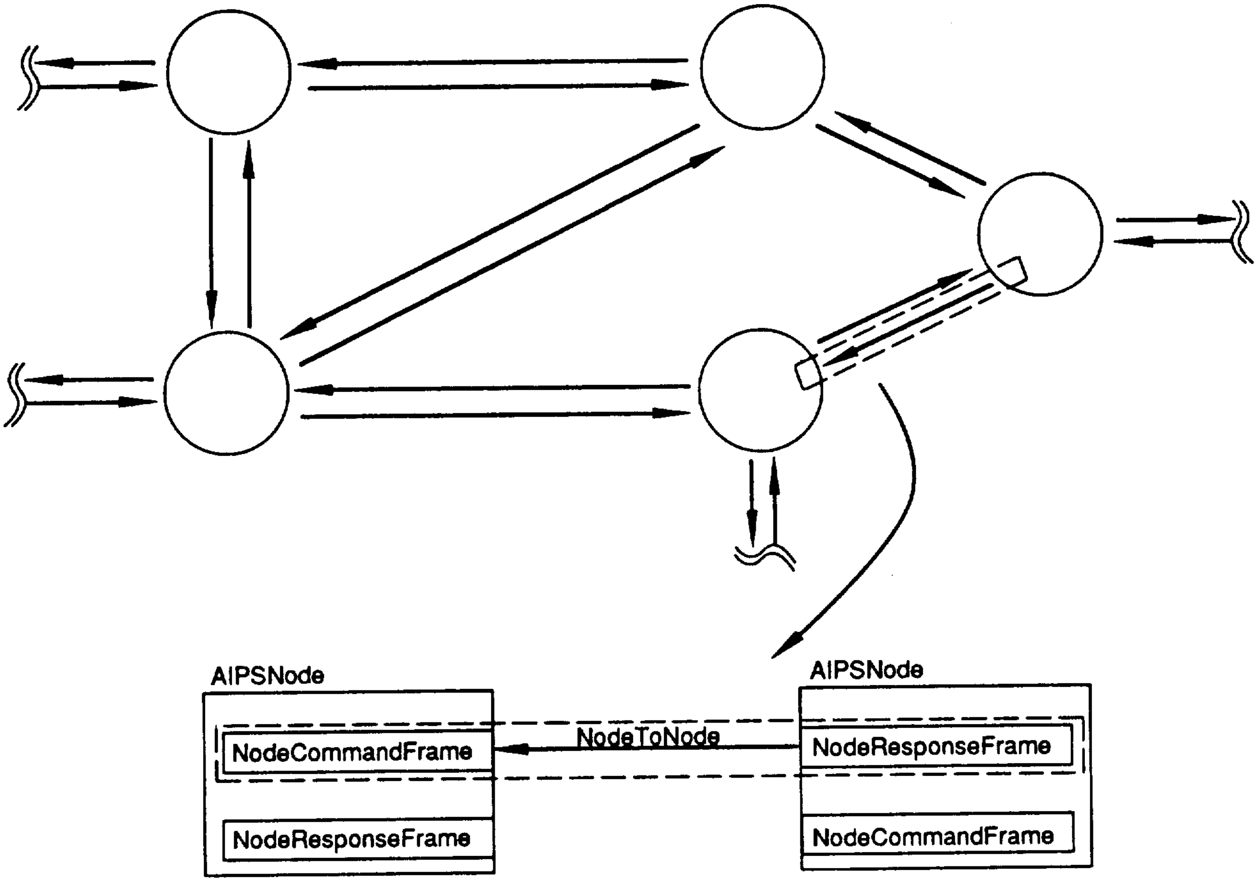


Figure 4.4.1-1. DEVM I/O Definition for AIPS Node



NodeToNode = (NodeResponseFrame NodeCommandFrame)

Figure 4.4.1-2. ARC Definition for Inter-AIPS Node Connections

such as the definition of an AIPSNODE illustrated by figure 4.4.1-3. The generic AIPSNODE DEVM has five parameters; the first two integer values, NetworkID and NodeNumber, are used to define the specific I/O network node interconnections. The SequencerTimeUpper and SequencerTimeLower parameters are real numbers that define the limits of the uniform distribution that the AIPSNODE uses to determine the time interval between the reception of a valid command and the transmission of a response. The InitialConfiguration parameter is used by the AIPSNODE to set the configuration of its ports at simulation initialization time.

Simulation Topology File

The DENET simulation is set up with a topology file that defines DEVMs, their parameter values, and their interconnections. An example topology file illustrating each of these features is shown in figure 4.4.1-4. The top section indicates that there are five instances of AIPSNODE in this simulation and that their DEVM numbers are 70 through 74. The next section shows the inter-DEVM connections with the NodeToNode ARC definition. An input/output port combination is created for each connection made with an ARC connection. Finally, the configuration parameters of each AIPSNODE are shown. By implementing other functions in DEVMs and defining ARC definitions, a complete simulation of the reference configuration was developed.

4.4.2 Simulation Model

The following paragraphs describe the functionality of the DEVMs used in modeling the reference configuration.

Some problems were encountered during the development of the models for the AIPS elements because the architecture implementation concept was still in development. As the model development progressed, certain assumptions were made about how functions were implemented and how much time each required. The intent in these cases was to err on the optimistic side. This has resulted in simulation models that are AIPS-like in overall behavior but

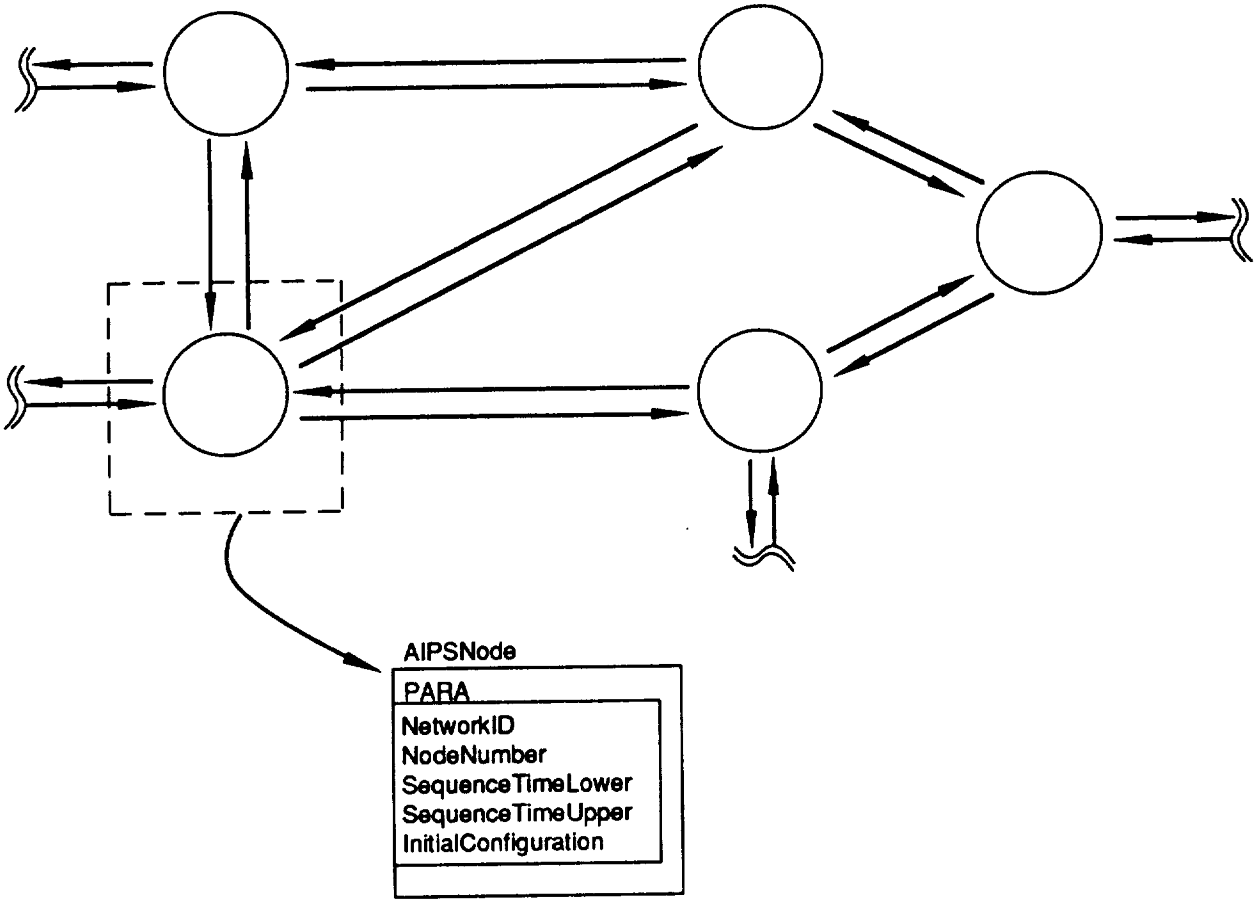
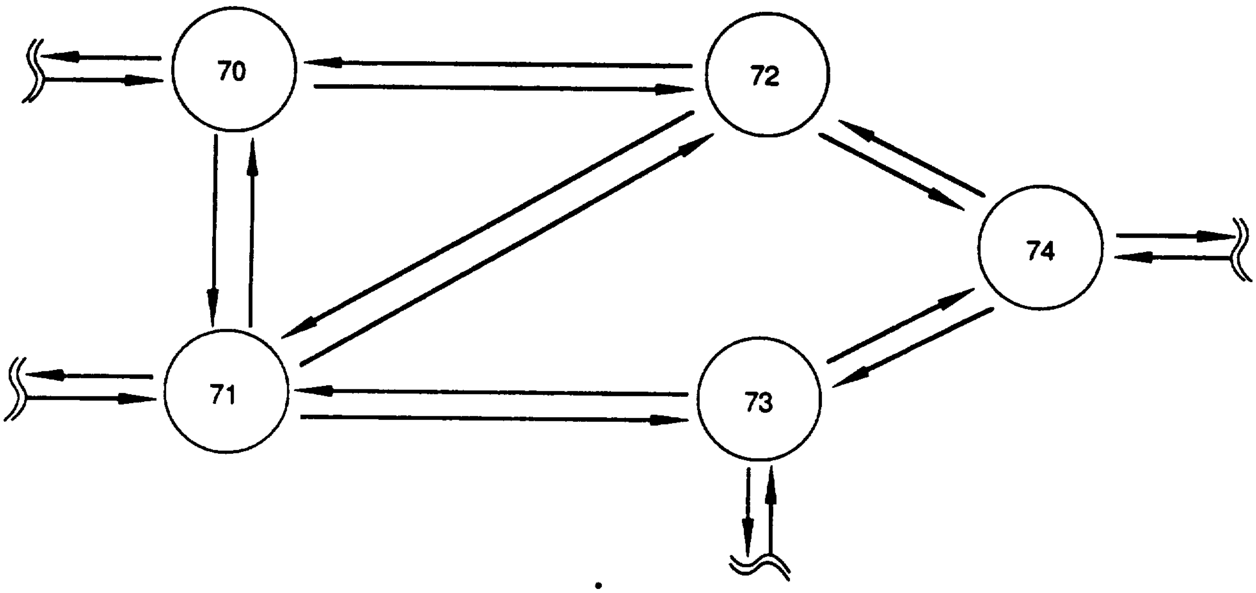


Figure 4.4.1-3. Parameter Definitions for AIPS Node



```
                                   •
                                   •
                                   •
                     70..74: = AIPSNode; creates 5 nodes
                                   •
                                   •
                                   •
                  [70|71,72]: = NodeToNode;
               [71|70,72,73]: = NodeToNode;
               [72|70,71,74]: = NodeToNode;  Defines internode connections shown above
                  [73|71,74]: = NodeToNode;
                  [74|72,73]: = NodeToNode;
                                   •
                                   •
                                   •
70: = {2 1  0.000315  0.000385  { T T T T F }};
71: = {2 2  0.000315  0.000385  { T T T T F }};
72: = {2 3  0.000315  0.000385  { T T T T F }};  Assigns values to parameters of each node
73: = {2 4  0.000315  0.000385  { T T T T F }};
74: = {2 5  0.000315  0.000385  { T T T T F }};
                                   •
                                   •
                                   •
```

Figure 4.4.1-4. Example Topology File

not guaranteed to match any current or future AIPS implementation. Before describing the specific DEVMs, the steps involved in accomplishing an application I/O request will be reviewed.

Application I/O Activity

Application I/O activity involves the interaction of many system elements. The process resulting from a request made by the application computing is illustrated in figure 4.4.2-1. The figure shows the significant activity by system element. The IOP responds to the I/O request by loading the necessary data and transferring control to the IOS. The IOS sends the individual command frames to the DIUs over the network and collects the response frames from the DIUs. When the IOS is finished, the IOP collects the DIU responses and makes them available to the CP via the data exchange. The application can then process the DIU responses. An overview of the interaction of the DEVMs is provided by the following description of how this sequence is implemented in the simulation model.

The DEVMs that model the above sequencing are illustrated in figure 4.4.2-2. One application rate group is modeled by one instance of the application DEVM. When it is time for the application to execute, it makes a request to the processor DEVM representing the CP. The processor notifies the application when the application's processing is complete for the current cycle. The application may then make an I/O request to the I/O service DEVM, as illustrated in figure 4.4.2-1.

To execute the I/O request, the IO service must acquire the IOP to load the data needed for the current I/O cycle. The IO service then makes a processing request to the instance of the processor DEVM that models the IOP sequencing and control function. The processor notifies the IO service when it has completed the IO service's processing request. At this point the IO service commands an IOS for each network to execute the I/O activity.

The IOS DEVM sends messages to the network and collects the responses. All the DEVMs modeling the network (nodes and DIUs) receive the command frames.

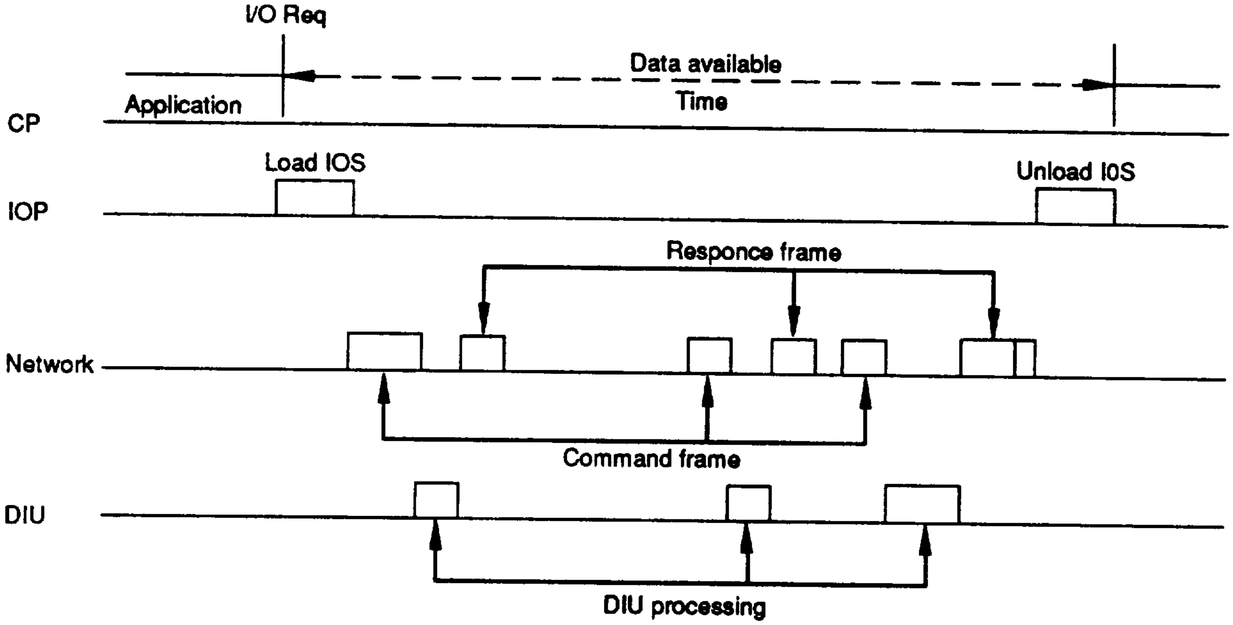


Figure 4.4.2-1. Application Cycle

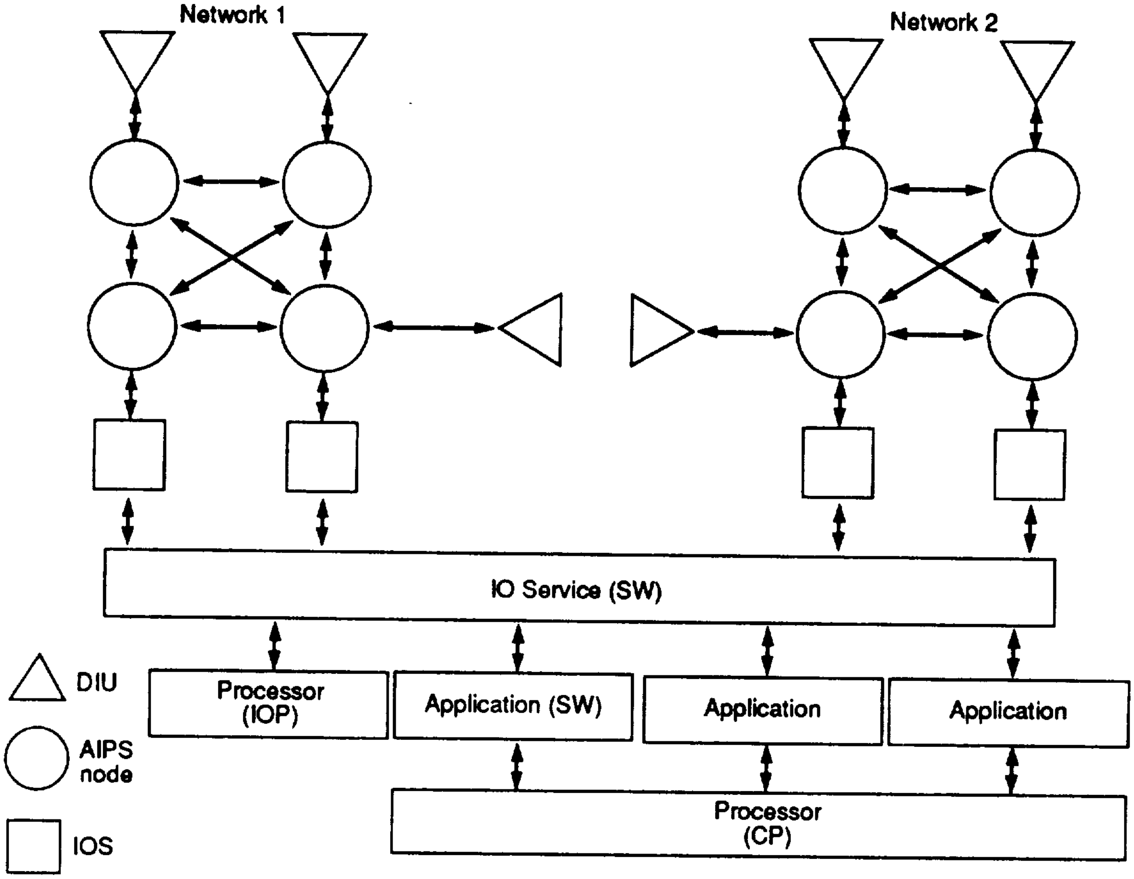


Figure 4.4.2-2. Example Application Simulation Topology

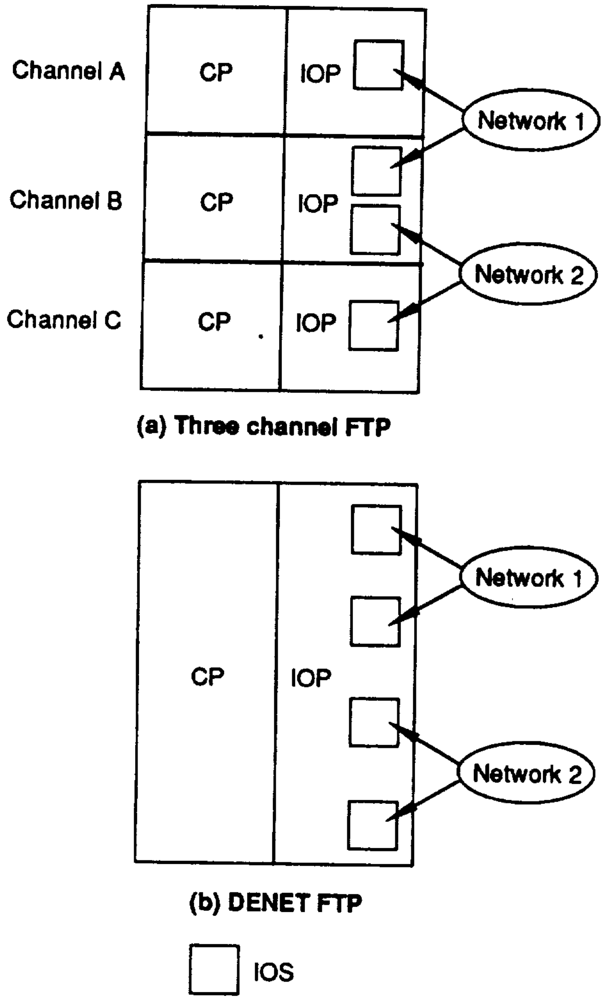
The DEVM addressed in the command frame transmits a response to the network. When the I/O activity is complete, the IO service makes another processing request to the processor DEVM to unload the networks. When the IO service is notified that the IOP has completed its request, the IO service notifies the application DEVM that its I/O request has been completed.

At this point the application DEVM has sensor data to process. It makes a request to the instance of the processor DEVM modeling the CP control function for the appropriate processing time. The following sections describe these DEVMs in detail.

Processor

Each FTP channel is composed of two CPUs: an I/O processor (IOP) and a computational processor (CP). All CPs and all IOPs in an FTP execute in instruction synchronization. This organization is depicted in figure 4.4.2-3(a). For the defined experiments, it can be assumed that the CPs and the IOPs remain in synchronization. With this assumption, a multiple-channel FTP can be modeled as a single-channel FTP, as illustrated in figure 4.4.2-3(b). This modeling simplification is critically dependent on the operation of multiple I/O networks and the process that ensures that a multiple-channel FTP remains in synchronization.

As depicted in figure 4.4.2-2, two different instances of the processor DEVM support the processing requirements of the I/O system and the application. The key processor DEVM characteristics are listed in figure 4.4.2-4. The processor DEVM models the sequencing and control functions that execute on either the CP or IOP. This sequencing is based on a preemptive priority model in which the highest priority process ready for execution acquires the processor and retains it until it completes or until a higher priority process becomes ready. The processor maintains a priority queue of processes waiting to use the processor. When a process completes, the first element of this queue acquires the processor. If a process makes a request to use the processor, it is inserted into this



(a) Three channel FTP

(b) DENET FTP

Figure 4.4.2-3. FTP Model

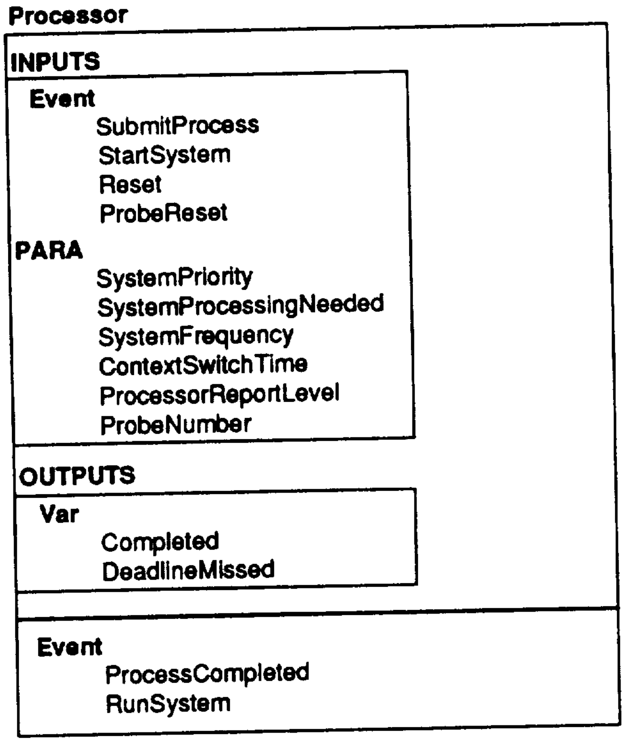


Figure 4.4.2-4. Processor DEVM

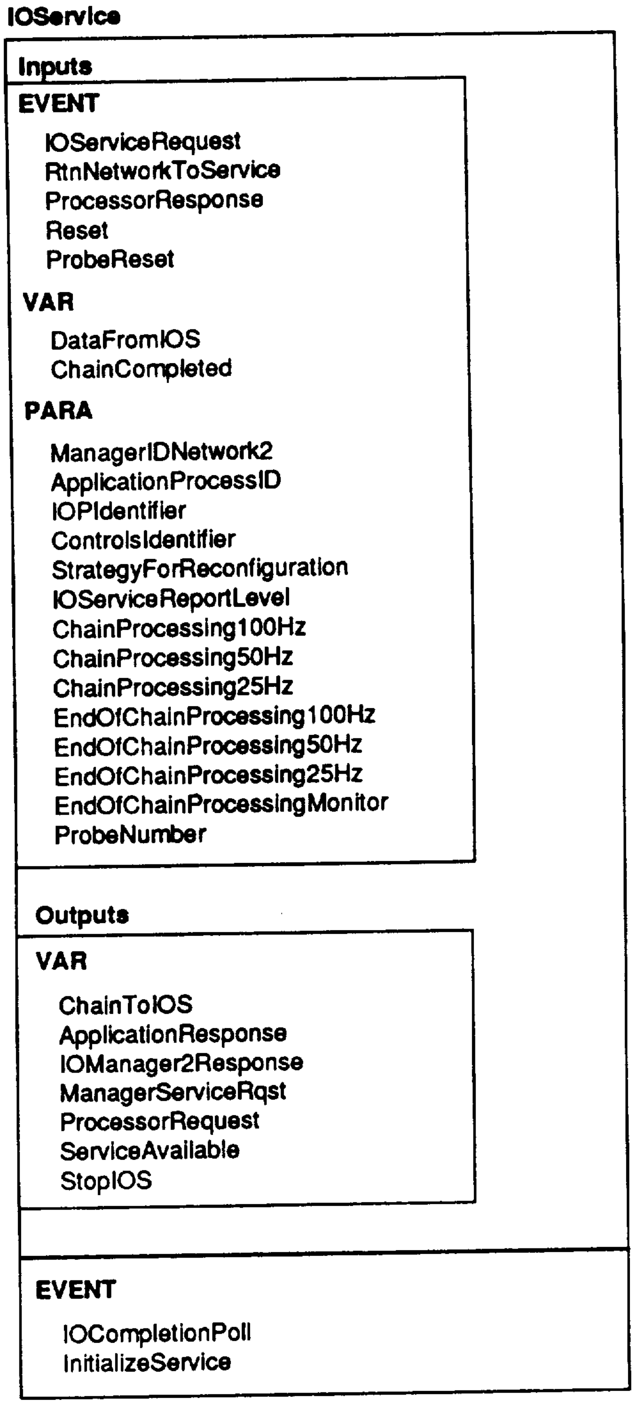
priority queue if its priority is not higher than the process currently running on the processor. If its priority is higher, it acquires the processor and the currently running process is inserted into the queue.

Each application process is assigned a priority, with the faster rates having higher priority. The FDIR process is assigned a priority higher than the application processes and executes at the rate of the fastest application. In the IOP, the processing related to an application I/O activity is assumed to have the same priority as the application computing. Again, the FDIR is assigned a higher priority than all the application priorities and executes at the fastest application rate. The processes related to the repair of the I/O network are assigned a lower priority than all the application priorities.

Efficiency is a concern for all system services functions. Overhead processing is required in preemptive priority systems to handle the different processes. This overhead can dominate a processor's activity, depending on the sequence in which processes become ready to execute and the amount of time needed to switch processes. A value of 0.300 millisecond was used to model the time needed for sequencing and control overhead in the processor DEVM. A process switch is assumed to be an uninterruptable operation. In some situations a process can become ready while another context switch is in progress. In this case the processor performs two switches. At the completion of the first, the system recognizes the new higher priority request and immediately switches processes again.

IO Service

The IO service DEVM models the software functions that execute primarily in the IOP. This software communicates with the IOS DEVM and the application DEVM. The key IO service DEVM characteristics are listed in figure 4.4.2-5. The model focuses on the software that controls the sequencing of pre- and post-processing activity in response to an I/O request. Some IO service functions reside in the CP for interface reasons. These were assumed to take negligible processing time in the model.



IOService

Inputs

EVENT

IOServiceRequest
RtnNetworkToService
ProcessorResponse
Reset
ProbeReset

VAR

DataFromIOS
ChainCompleted

PARA

ManagerIDNetwork2
ApplicationProcessID
IOPIdentifier
ControlsIdentifier
StrategyForReconfiguration
IOServiceReportLevel
ChainProcessing100Hz
ChainProcessing50Hz
ChainProcessing25Hz
EndOfChainProcessing100Hz
EndOfChainProcessing50Hz
EndOfChainProcessing25Hz
EndOfChainProcessingMonitor
ProbeNumber

Outputs

VAR

ChainToIOS
ApplicationResponse
IOManager2Response
ManagerServiceRqst
ProcessorRequest
ServiceAvailable
StopIOS

EVENT

IOCompletionPoll
InitializeService

Figure 4.4.2-5. IOService DEVM

The IO service process depicted in figure 4.4.2-6 controls access to the I/O networks. This model was developed as a result of initial simulation efforts and differs from the preliminary AIPS design. There are 4 steps to the execution of an I/O request: load command frame data, start IOS, I/O completion poll, and unload the response frame data. The IOSs execute the individual transactions of the I/O activity without IOP involvement. Note that the IO service was modeled as separate tasks, one dedicated to each application rate group and one for the network manager. This model was created when a previous version, based on a preliminary AIPS concept, could not satisfy the flight control application update requirements. The separate task model allowed some preemptive activity on the IOP without affecting the IOS execution.

A semaphore was employed to protect the execution of the application IOS activity. The I/O network is protected when the IOSs are commanded to start and released after the I/O completion poll. The modeled approach minimizes the time that the I/O networks are in a nonpreemptable state and relies on the preemptive priority scheduler to resolve the contention between I/O requests from the different application rates and the network manager. Assigning high priority to faster application rate groups allows the application activity with the closest deadline to be serviced first.

In the IO service model the IOP is free for lower priority processing after it starts the IOSs. The IO service schedules an I/O completion poll using the underlying system time schedule function. The I/O completion poll checks for the completion of the network activity for an I/O request and begins to unload the collected data. Each I/O request has a defined completion poll interval that guarantees that the activity is complete when the I/O networks are operating normally. The I/O completion poll occurs at integer milliseconds of system time. This is due to the assumed underlying system time scheduling granularity of one millisecond.

An example of the contention between application rates in the IO service is shown in figure 4.4.2-7. The situation is that the 50 Hz rate has an I/O request that has progressed to its execute state. At this point, the 100 Hz rate initiates an I/O request. Because the IOP is free, it can

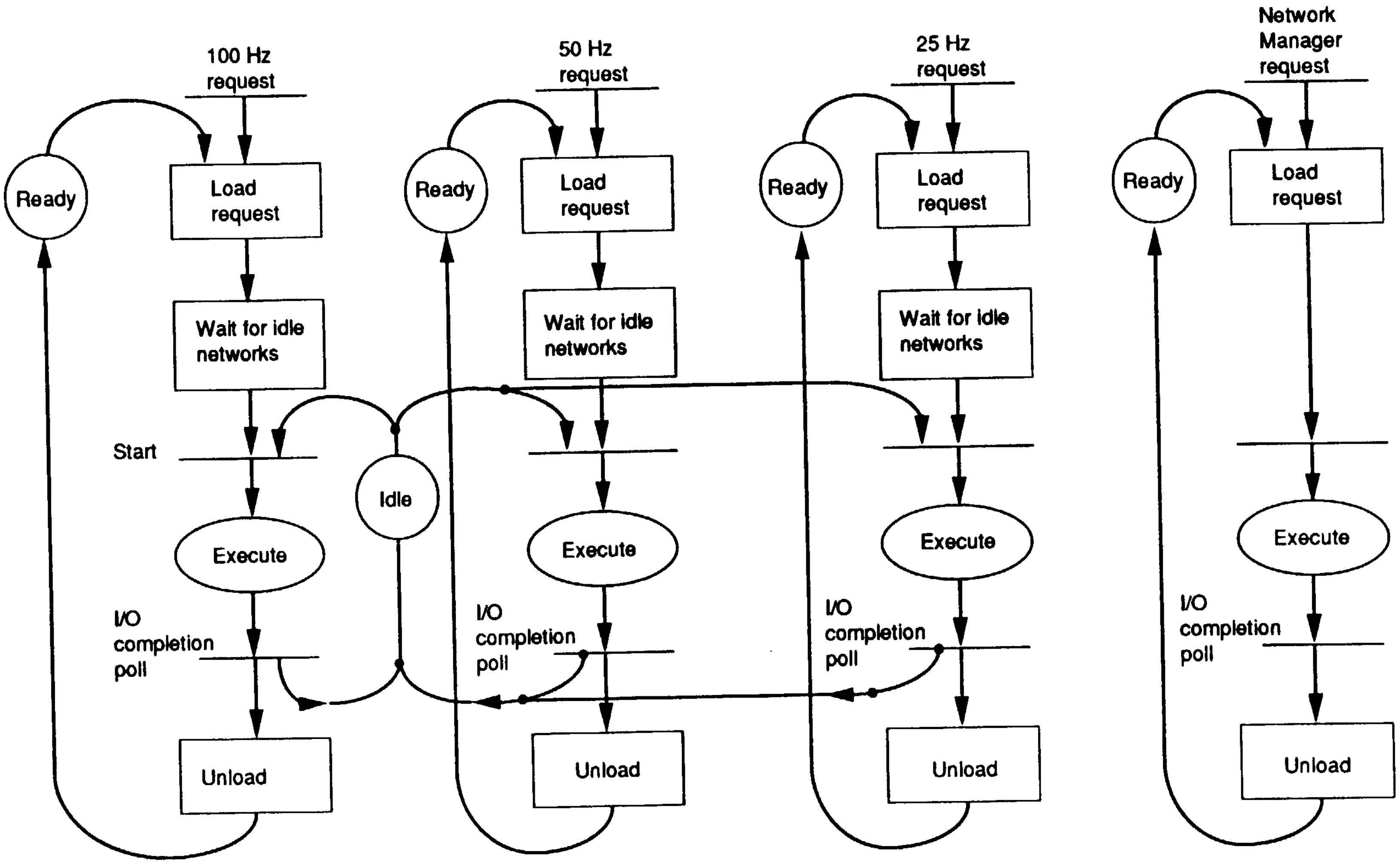


Figure 4.4.2-6. I O Service Access Contention

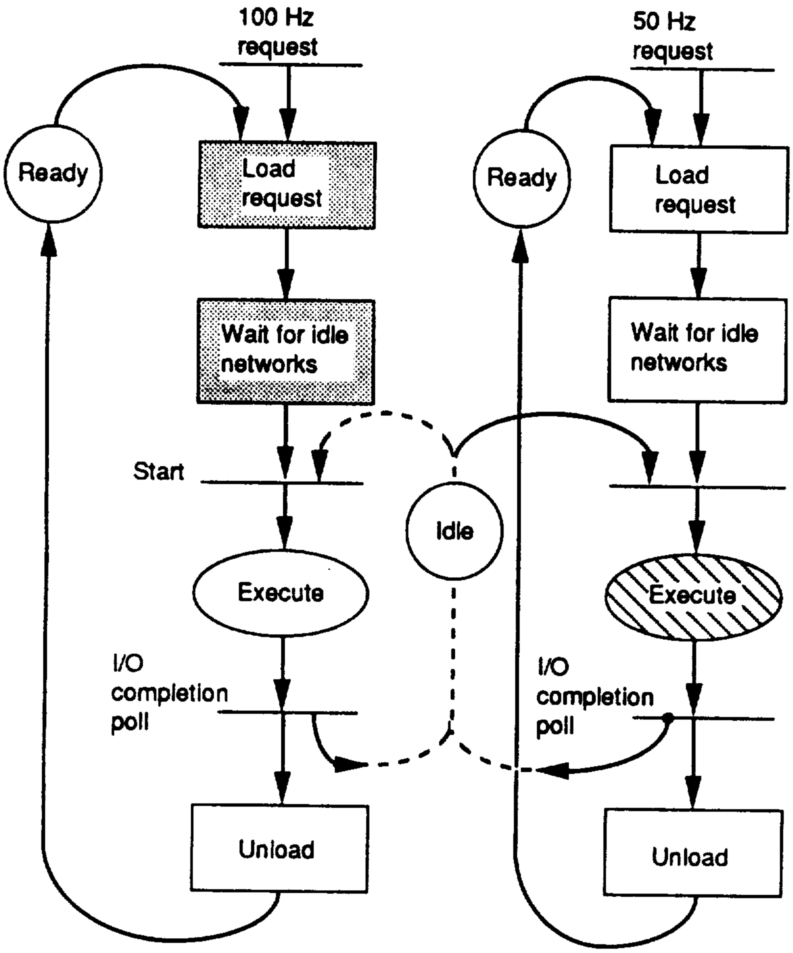


Figure 4.4.2-7. IO System Contention Example

process the 100 Hz I/O request until it is time to start the IOSs. The 100 Hz I/O request is then suspended, because the 50 Hz I/O process controls the semaphore. When the 50 Hz I/O completion poll occurs, the 50 Hz I/O process releases the semaphore making 100 Hz I/O process ready. The 100 Hz I/O process starts the IOS with the 100 Hz activity, schedules its own completion poll, and relinquishes the IOP. The 50 Hz data is then unloaded from the network interfaces because it is the highest priority job waiting for the IOP.

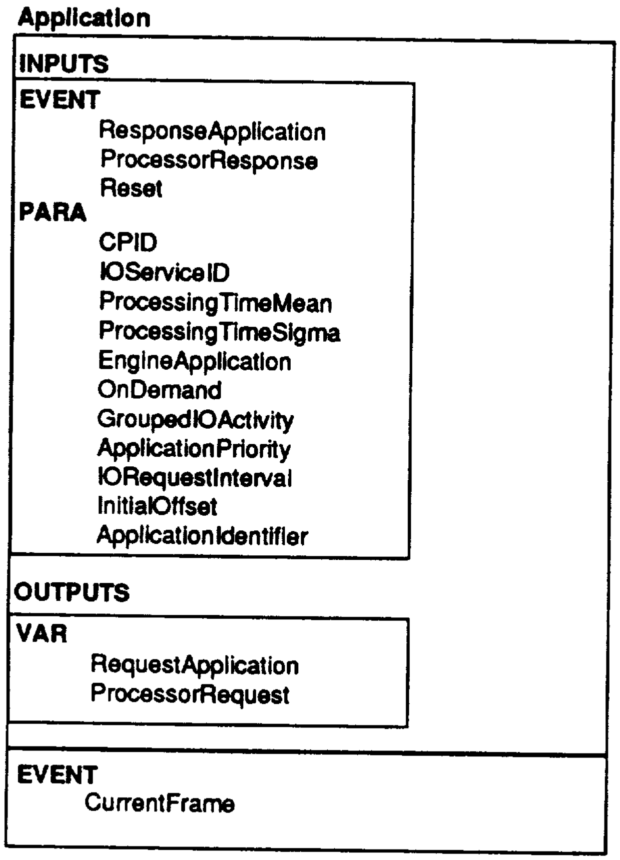
The IOS execution of network manager I/O activity can take place at the same time as application I/O activity. When a communication fault occurs on a network, no further application I/O activity takes place on that network until the I/O network manager repairs the fault. Thus, during failure recovery, the application I/O activity will be restricted to one network and the network manager activity will be restricted to the other network. There will be contention between the application and the network manager for the IOP, but the network manager I/O activity is unconstrained on the network being repaired.

Application

The application is a generic DEVM that models the functionality of a single application rate group. The application DEVM, whose key characteristics are listed in figure 4.4.2-8, can be configured to perform the workload of any flight control or engine rate group. Its execution sequence can be configured for either on demand or scheduled I/O. The DEVM models data-dependent processing requirements using a normal workload distribution for the needs of each application cycle.

Input/Output Sequencer

The IOS DEVM executes chains requested by the IO service DEVM and collects data resulting from chain execution. It provides the IO service with a status variable that corresponds to the state of IOS execution. This status variable indicates whether the IOS has completed the chain it is executing. The I/O service also has the ability to stop the IOS's



Application

INPUTS

EVENT

ResponseApplication
ProcessorResponse
Reset

PARA

CPID
IOServiceID
ProcessingTimeMean
ProcessingTimeSigma
EngineApplication
OnDemand
GroupedIOActivity
ApplicationPriority
IORequestInterval
InitialOffset
ApplicationIdentifier

OUTPUTS

VAR

RequestApplication
ProcessorRequest

EVENT

CurrentFrame

Figure 4.4.2-8. Application DEVM

execution of a chain at anytime. When commanded to start a chain, the IOS DEVM sends command frames to the adjacent node and waits for the response frames as needed until the I/O activity is finished. Its key characteristics are listed in figure 4.4.2-9.

Node

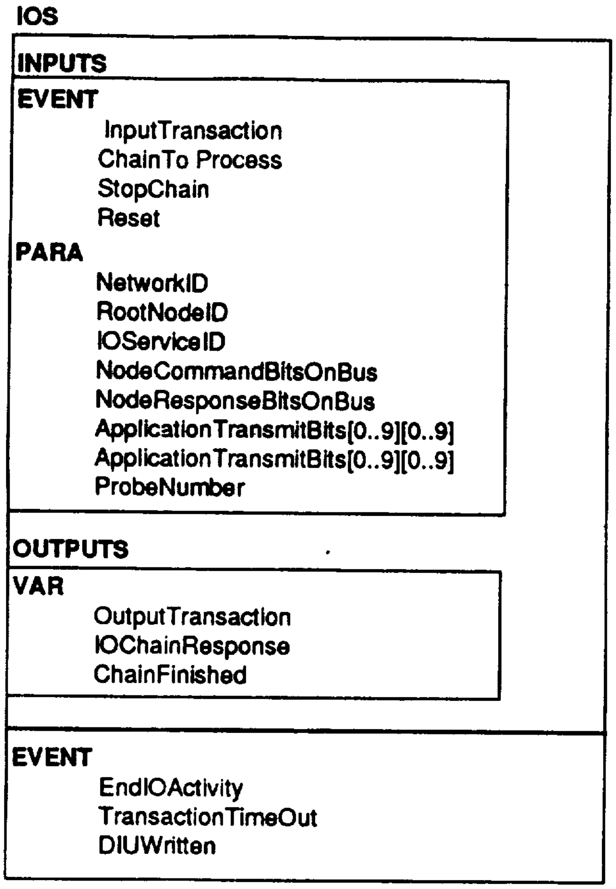
The I/O network node is the element used to construct the mesh configurable network. During normal operation, the AIPSNODE DEVM node acts a rebroadcast element. Any activity received on an enabled port is immediately retransmitted out all the other enabled ports. The AIPSNODE responds to reconfiguration commands addressed to it by changing its port configuration and then sending a response frame that contains the node's status. This allows detailed modeling of the network repair activity directed by the network manager to reconfigure around faults. A null command to a node results in the node transmitting its status. AIPSNODE DEVMs interface with other AIPSNODE DEVMS, IOS DEVMs, or DIU DEVMS. Their key characteristics are listed in figure 4.4.2-10.

DIU

The DIU DEVM models the network device to which the application sensors and actuators are connected. The DIU DEVM models the receipt of messages from the application. The DIU model schedules the transmission of a response message at a time that reflects the DIU processing time described in section 4.1. The DIU DEVM, whose key characteristics are listed in figure 4.4.2-11, models the statistical variation of the duration of DIU processing time.

Network Manager

The network manager is responsible for maintaining communications between the application process and the DIUs. The IO service DEVM notifies the network manager DEVM when the application process encounters a communication fault. From this point on, the IO service does not execute any application chains on the faulty I/O network until the network manager notifies the IO service that the repair is complete.



IOS

INPUTS

EVENT

InputTransaction
ChainTo Process
StopChain
Reset

PARA

NetworkID
RootNodeID
IOServiceID
NodeCommandBitsOnBus
NodeResponseBitsOnBus
ApplicationTransmitBits[0..9][0..9]
ApplicationTransmitBits[0..9][0..9]
ProbeNumber

OUTPUTS

VAR

OutputTransaction
IOChainResponse
ChainFinished

EVENT

EndIOActivity
TransactionTimeOut
DIUWritten

Figure 4.4.2-9. IOS DEVM

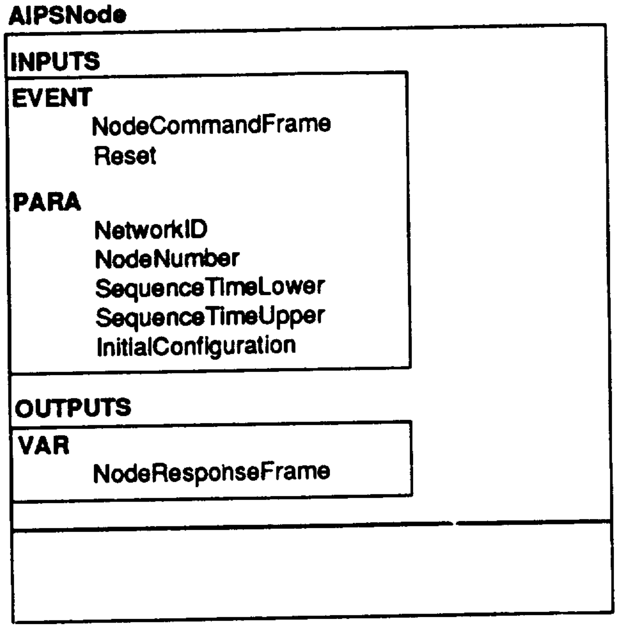


Figure 4.4.2-10. AIPS Node DEVM

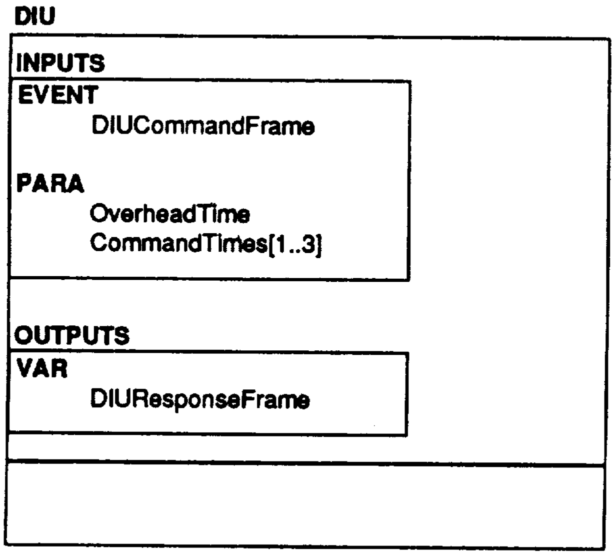


Figure 4.4.2-11. DIU DEVM

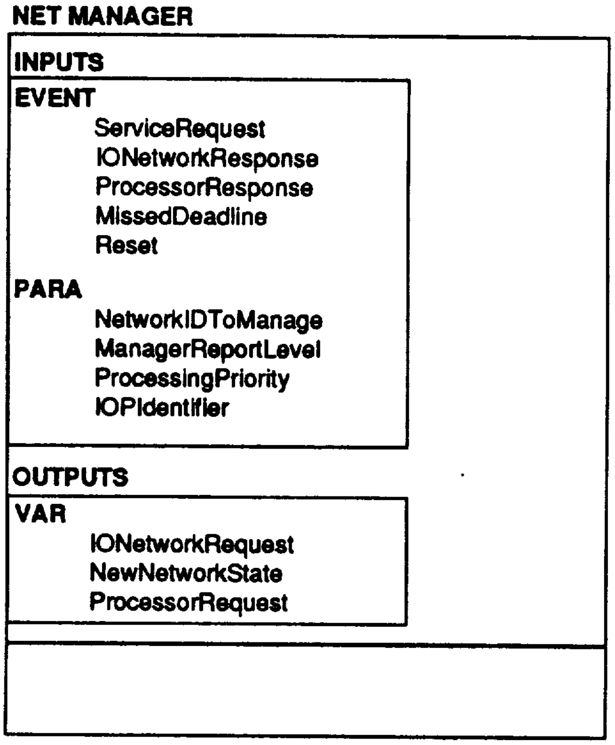
The network manager DEVM implements two different types of strategies for repairing I/O network faults, one-shot and regrow. Prototype algorithms for both of these strategies are implemented in the DEVM, whose key characteristics are listed in figure 4.4.2-12. A configuration item in the IO service DEVM dictates which type of strategy will be used in the current experiment to repair the I/O network. This allows a common DEVM to be used for both sets of experiments.

4.4.3 Development Problems with Performance Model

The IO service DEVM presented extreme difficulty during the development of the simulation model. This was primarily due to the complex functionality of this process. Modeling difficulty early in the development is probably a good sign of impending difficulty in the implementation phase. The IO service was not only a complex function but also had demanding performance requirements. Performance problems in the early simulation steps led to the separate task structure for IO service shown in figure 4.4.2-6. As improbable as it seems, this structure was actually simpler than the original design. This structure still has some remaining adverse performance characteristics. This is a good example of the benefit of performance modeling in exposing concept difficulties early in the design cycle.

4.4.4 Simulation Input Values

The values used by Boeing for certain simulation timing input parameters are shown in figures 4.4.4-1 through 4.4.4-3. The AIPS parameters that do not vary with simulation configuration are illustrated in figure 4.4.4-1. Most of these parameters relate to time needed by system overhead functions to support the application activity. Parameters unique to the flight control configurations are shown in figure 4.4.4-2. Parameters unique to the engine control configurations are shown in figure 4.4.4-3.



NET MANAGER

INPUTS

EVENT

- ServiceRequest
- IONetworkResponse
- ProcessorResponse
- MissedDeadline
- Reset

PARA

- NetworkIDToManage
- ManagerReportLevel
- ProcessingPriority
- IOPIdentifier

OUTPUTS

VAR

- IONetworkRequest
- NewNetworkState
- ProcessorRequest

Figure 4.4.2-12. Network Manager DEVM

DIU processing overhead		20
Process switch		300
Node processing time		Uniform (335, 385)
Transaction turn around time		10
Node transaction transmission time	Command	40
	Response	56
Powerup initialize IO service processing		100
On demand request delay		Uniform (20, 35)
Request processing		25
Chain processing overhead		50
End of chain processing overhead		50
Request completion processing		25
Network manager chain loading time (one network)		
Network manager		
1 transaction chain		18
2 transaction chain		36
4 node monitor chain (one network)		72
18 node monitor chain (one network)		324
Network manager end of chain processing time (one network)		
1 transaction chain		52
2 transaction chain		104
4 node monitor chain (one network)		208
18 node monitor chain (one network)		936
Network manager processing elements		
Grow initialize (assume 25/node)		
Flight control		450
Engine control		100
Compute one transaction chain		75
Compute two transaction chain		150
Network response computation		50
Change network status		25
Analyze error report		75 + 5 μs/port decision
Switch root link processing		25

Figure 4.4.4-1. AIPS Time Elements (μs)

Application computing

i) 100 Hz	Normal ($\mu = 1723, \sigma = 86$)
ii) 50 Hz	Normal ($\mu = 3016, \sigma = 150$)
iii) 25 Hz	Normal ($\mu = 7100, \sigma = 355$)

DIU command execution
Grouped I/O

100Hz		
	S1	90 + Uniform (0, 10)
	S2	90 + Uniform (0, 10)
	OFL	45 + Uniform (0, 10)
	OFR	45 + Uniform (0, 10)
	IFL	60 + Uniform (0, 10)
	IFR	60 + Uniform (0, 10)
	TEL	45 + Uniform (0, 10)
	TER	60 + Uniform (0, 10)
50 Hz		
	S1	30 + Uniform (0, 10)
	S2	30 + Uniform (0, 10)
	CP1	45 + Uniform (0, 10)
	CP2	45 + Uniform (0, 10)
	CDL	30 + Uniform (0, 10)
	CDR	30 + Uniform (0, 10)
	RL	30 + Uniform (0, 10)
	RR	30 + Uniform (0, 10)
	N	30 + Uniform (0, 10)
	LER	90 + Uniform (0, 10)
25 Hz		
	S1	15 + Uniform (0, 10)
	CP1	15 + Uniform (0, 10)

Chain loading time (2 networks in service)

i) 100Hz	396
ii) 50Hz	444
iii) 25Hz	0

End of chain processing time (2 networks in service)

i) 100Hz	1232
ii) 50Hz	1296
iii) 25Hz	208

Transaction transmission time

	Command		Response
100Hz			
S1	40	88	
S2	40	88	
OFL	56	64	
OFR	56	64	
IFL	56	72	
IFR	56	72	
TEL	56	64	
TER	56	72	
50Hz			
S1	40	56	
S2	40	56	
CP1	40	64	
CP2	40	64	
CDL	56	56	
CDR	56	56	
RL	56	56	
RR	56	56	
N	56	56	
LER	88	88	
25Hz			
S1	40	48	
CP1	40	48	

Figure 4.4.4-2. Flight Control Time Elements (μs)

Application computing

i) 100 Hz	Normal ($\mu = 175, \sigma = 9$)
ii) 50 Hz	Normal ($\mu = 100, \sigma = 5$)
iii) 25 Hz	Normal ($\mu = 1300, \sigma = 65$)

DIU command execution

Grouped I/O

i) Inlet	135 + Uniform (0, 10)
ii) Nozzle	90 + Uniform (0, 10)
iii) Engine	325 + Uniform (0, 10)

Separated I/O (input only)

i) Inlet	105 + Uniform (0, 10)
ii) Nozzle	60 + Uniform (0, 10)
iii) Engine	265 + Uniform (0, 10)

Chain processing time (2 networks in service)

i) 100 Hz	114
ii) 50 Hz	114
iii) 25 Hz	186

End of chain processing time (2 networks in service)

a) 100Hz	232
b) 50Hz	184
c) 25Hz	456

Transaction transmission time

Grouped I/O

	Command	Response
Inlet	88	112
Nozzle	88	88
Engine	136	224

Separated I/O

	Input		Output	
	Command	Response	Command	Response
Inlet	40	88	88	-
Nozzle	40	112	88	-
Engine	40	224	136	-

Figure 4.4.4-3. Engine Control Time Elements (μs)

4.5 Evaluate

This section presents a high-level discussion of the results obtained from each experiment using DENET. The steps taken to analyze the simulation data from each experiment are discussed individually. The conclusions drawn from these results are discussed in section 4.6.

4.5.1 FDIR/Application Phasing (Experiment 4)

This experiment evaluates the effect of the relative phasing of the high-priority system FDIR process and the application activity. For each configuration shown in table 4.3.7-1, data were collected from execution of 100 major frames.

Flight Control Group

For two of the configurations (4 and 5), the application was unable to meet any control cycle deadlines. These two unsuccessful configurations correspond to the on-demand I/O scheduling option and the separated I/O configuration options for the flight control group. The simulation result showed that the flight control group was overloaded to the point that the application could not perform its function using either of these organization options. Consequently, configuration 4 and configuration 5 were eliminated as possible candidates.

A summary of the experimental data for configuration 6 (scheduled grouped I/O organization) is presented in table 4.5.1-1. The minimum deadline margin is a strong indicator of how close the application timing demands are to being violated. (The minimum value in the table is the smallest value observed during the experiment runs.) The 50 Hz rate group misses computing deadlines with five of the ten possible phasings (phase 1, 2, 3, 7, and 8), which is unacceptable. Examination of the data reveals that the deadline margin distribution is very close to zero for the five unacceptable phasings. Furthermore, the data indicated that these cases correlated with frames having somewhat larger than average 50 Hz computing demands. (Recall that application computing demands vary from frame to

Table 4.5.1-1. Experiment 4 Configuration 6 Summary - Flight Control Group

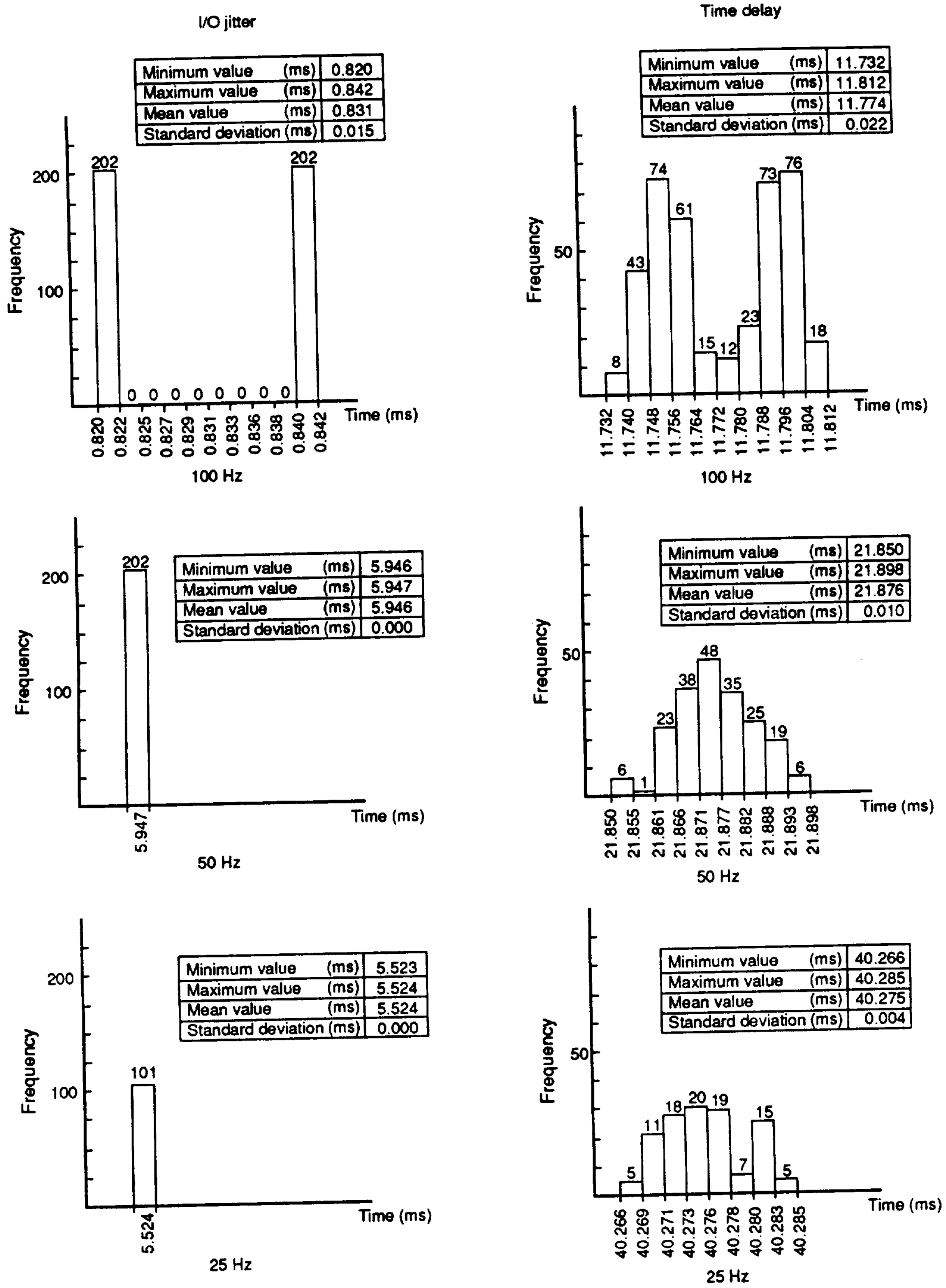
Phase/ID	100 Hz minumum deadline margin(ms)	50 Hz minimum deadline margin(ms)	25 Hz minimum deadline margin(ms)	CP utilization	IOP utilization	I/O system utilization	I/O network utilization
0	2.934	7.007	15.538	86%	72%	80%	28%
1	2.704	Missed 7 deadlines	10.188	86%	72%	93%	28%
2	2.704	Missed 7 deadlines	10.188	86%	72%	88%	28%
3	2.704	Missed 7 deadlines	10.188	86%	72%	83%	28%
4	3.370	0.319	10.320	86%	75%	78%	28%
5	0.508	1.287	10.267	86%	75%	84%	28%
6	0.653	7.848	9.896	86%	75%	79%	28%
7	0.551	Missed 7 deadlines	10.188	86%	75%	92%	28%
8	0.851	Missed 7 deadlines	10.188	86%	72%	91%	28%
9	1.905	0.625	11.529	86%	73%	87%	28%

frame as the loop duration times are modeled with a normal distribution.) In the other frames, the 50 Hz rate deadline is being met with a very small margin, illustrating how sensitive the specific bad relative phasing situations are.

Of the remaining cases, phase 4 has the largest deadline margin for the 100 Hz rate. However, the 50 Hz rate is very close to missing a deadline. While this configuration may meet the minimum acceptable requirements, the other configurations indicate that the overall performance can be improved. A desirable goal is to improve the minimum deadline margin for the 50 Hz rate without a significant reduction in the 100 Hz minimum deadline margin. The best improvement for the least reduction seems to occur with phase 0. The reduction in minimum deadline margin for the 100 Hz rate is less than 0.5 milliseconds. Additionally, the 50 Hz rate's minimum deadline margin has increased nearly 7 milliseconds and the 25 Hz rate has increased nearly 5 milliseconds.

The time delay and I/O jitter data for the phase 0 case are illustrated in figure 4.5.1-1. Two spikes, separated by 20 microseconds, are observed in the 100 Hz jitter. The two peaks are a result of the completion of the processing related to the unloading of the network interfaces for the 50 Hz rate in minor frames 1 and 3. This unloading completes nearly at the beginning of minor frames 2 and 4, causing a double context switch at the beginning of these frames that slightly delays the execution of the 100 Hz activity every other frame. However, the resulting 100 Hz rate jitter is a fraction of the update period. There is no variation in the start of the I/O activity from frame to frame for either the 50 Hz or 25 Hz rate groups.

The effect of the jitter at the start of the 100 Hz I/O activity is observable in the time delay statistics. It should be pointed out that the DIU response time is modeled as a statistical process that results in some variation in the I/O completion time and therefore contributes to the time delay statistics. The first contribution to the 100 Hz time delay is the variation for the start time of the 100 Hz I/O. The two distinct peaks are a result of the two distinct starting times for the 100 Hz I/O. The I/O activity for the 100 Hz rate is composed of eight transactions. Each



Minimum value (ms)	0.820
Maximum value (ms)	0.842
Mean value (ms)	0.831
Standard deviation (ms)	0.015

Minimum value (ms)	11.732
Maximum value (ms)	11.812
Mean value (ms)	11.774
Standard deviation (ms)	0.022

Minimum value (ms)	5.946
Maximum value (ms)	5.947
Mean value (ms)	5.946
Standard deviation (ms)	0.000

Minimum value (ms)	21.850
Maximum value (ms)	21.898
Mean value (ms)	21.876
Standard deviation (ms)	0.010

Minimum value (ms)	5.523
Maximum value (ms)	5.524
Mean value (ms)	5.524
Standard deviation (ms)	0.000

Minimum value (ms)	40.266
Maximum value (ms)	40.285
Mean value (ms)	40.275
Standard deviation (ms)	0.004

Figure 4.5.1-1. Experiment 4 Configuration 6 Phase 0 Application Performance Parameters

transaction is modeled as a statistical process with uniform distribution ranging in value between 0 and 10 microseconds. The overall contribution of these eight sources in the model contributes to the bell-shaped distribution about the two peaks in the figure. The time delay for the 50 Hz and 25 Hz rates exhibits no unexpected behavior.

The application sequencing for one major frame of the preferred configuration (phase 0) is illustrated in figure 4.5.1-2. This figure illustrates how the application activity aligns with the FDIR to provide a configuration that has better performance than the others evaluated in this experiment. In phase 0, the FDIR is scheduled to execute 1 millisecond into the minor frame. This allows a small degree of parallel activity in our model because the 100 Hz I/O activity can be executing using the IOSs and DIUs while the FDIR is executing on the IOP.

Engine Control Group

The workload on the engine control group is substantially less demanding than that on the flight control functions. Consequently, the engine control computer is able to meet the deadlines of the engine control functions in all the I/O scheduling and I/O grouping alternatives. Details for each configuration will be presented in the following paragraphs.

Configuration 14 (On Demand Grouped I/O Organization). The deadline margins and utilization values for configuration 14 are illustrated in table 4.5.1-2. The minimum deadline margin data from this experiment indicates that all the phasing options have adequate reserve margins. These values are significantly better than those of the flight control group, reflecting the difference in the workload demands.

No configuration has clearly superior performance characteristics. Therefore, the phase 4 case is selected as the preferred configuration based on the largest minimum deadline margin for all the application rates. The I/O jitter and time delay characteristics for this configuration are illustrated in figure 4.5.1-3. The 100 Hz rate exhibits what appear to be uniformly varying start times over an approximate range of 15 microseconds.

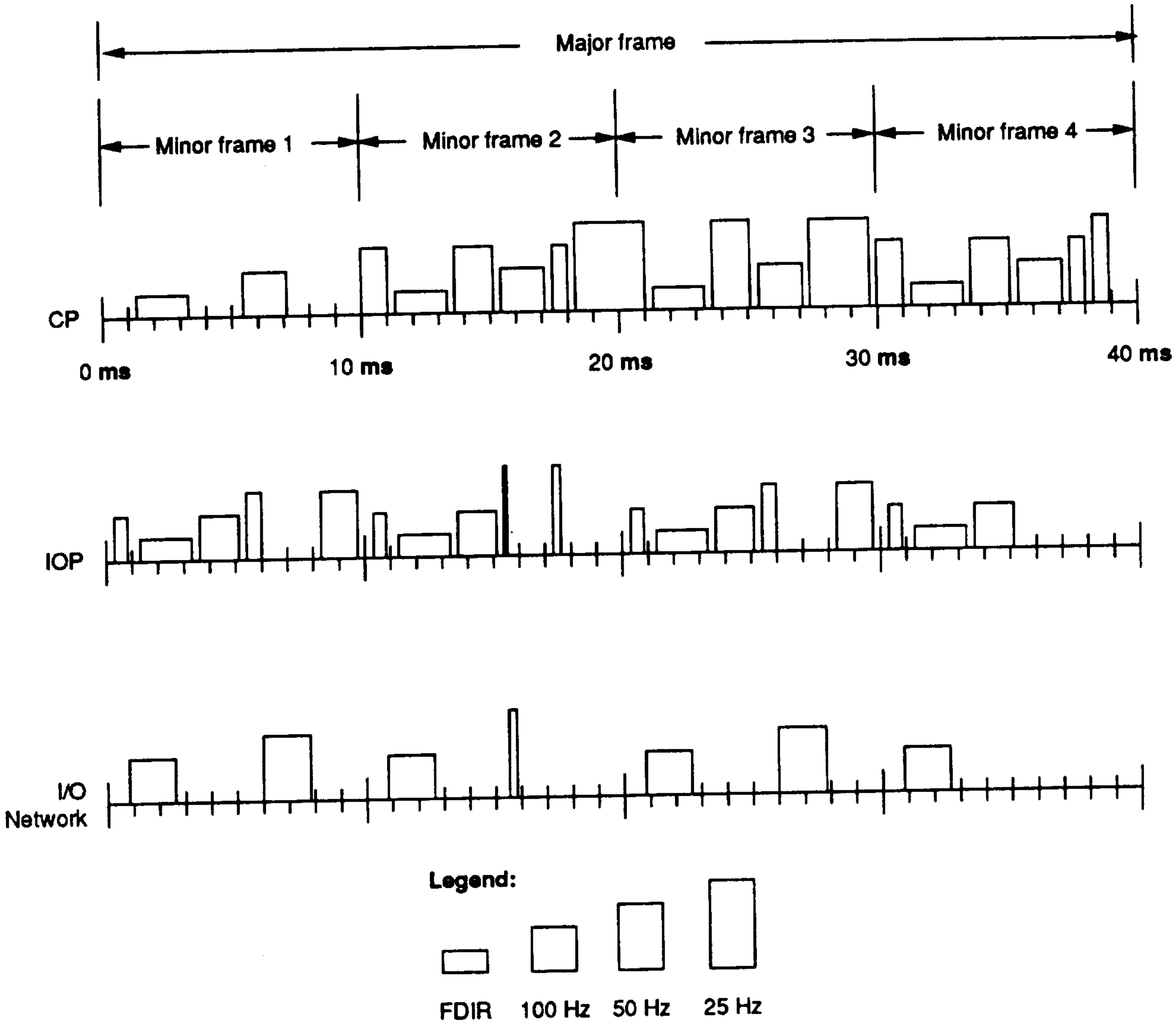
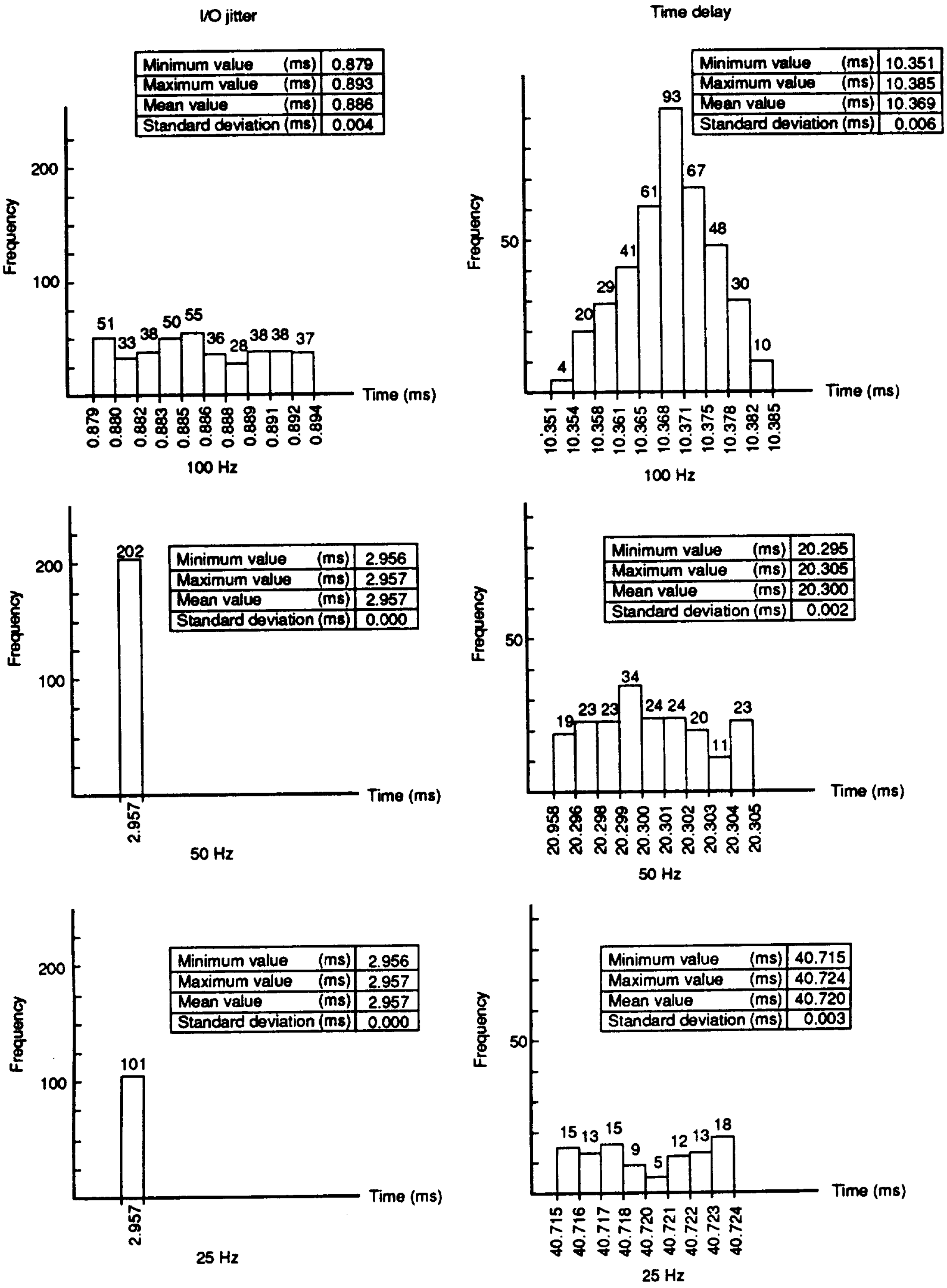


Figure 4.5.1-2. Experiment 4 Flight Control Computer Preferred Configuration

Table 4.5.1-2. Experiment 4 Configuration 14 Summary

Phase/ID	100 Hz minimum deadline margin(ms)	50 Hz minimum deadline margin(ms)	25 Hz minimum deadline margin(ms)	CP utilization	IOP utilization	I/O system utilization	I/O network utilization
0	5.657	13.598	31.941	51%	53%	57%	7%
1	6.440	14.898	33.241	48%	58%	46%	7%
2	6.440	14.898	33.241	48%	58%	46%	7%
3	6.440	14.898	33.241	48%	58%	46%	7%
4	7.140	15.560	33.241	51%	58%	39%	7%
5	7.140	15.560	31.341	51%	58%	39%	7%
6	7.140	12.998	31.341	51%	58%	39%	7%
7	7.140	13.298	31.641	51%	56%	56%	7%
8	4.540	14.041	32.384	51%	56%	50%	7%
9	4.840	12.598	30.941	51%	55%	67%	7%



Minimum value	(ms)	0.879
Maximum value	(ms)	0.893
Mean value	(ms)	0.886
Standard deviation	(ms)	0.004

Minimum value	(ms)	10.351
Maximum value	(ms)	10.385
Mean value	(ms)	10.369
Standard deviation	(ms)	0.006

Minimum value	(ms)	2.956
Maximum value	(ms)	2.957
Mean value	(ms)	2.957
Standard deviation	(ms)	0.000

Minimum value	(ms)	20.295
Maximum value	(ms)	20.305
Mean value	(ms)	20.300
Standard deviation	(ms)	0.002

Minimum value	(ms)	2.956
Maximum value	(ms)	2.957
Mean value	(ms)	2.957
Standard deviation	(ms)	0.000

Minimum value	(ms)	40.715
Maximum value	(ms)	40.724
Mean value	(ms)	40.720
Standard deviation	(ms)	0.003

Figure 4.5.1-3. Experiment 4 Configuration 14 Phase 4 Application Performance Parameters

This is characteristic of the uniformly distributed models for the response to an interprocessor event used in the on demand I/O option. This characteristic is not visible in either the 50 Hz or 25 Hz I/O because the 100 Hz I/O is executing when these requests are made. The time delay data for the preferred configuration are acceptable.

A timeline of one major frame for the phase 4 case is illustrated in figure 4.5.1-4. Note that the 50 Hz or 25 Hz I/O activity is loaded during the IOP idle time before unloading the 100 Hz data during the appropriate minor frames. This "preloading" does not occur in the preferred flight control configuration because the FDIR executes during the 100 Hz IOS execution.

The phase 4 case could have problems when future growth is considered. In this situation, the FDIR executes at the end of every minor frame, after the completion of all application activity. The simple minimum deadline margin figure of merit of 7.140 milliseconds may be misleading in this situation. Because the FDIR executes after the application activity, the application cannot expand to fill the 7.140 milliseconds without being preempted by the FDIR. A more sophisticated measure of deadline margin might be necessary to allow better comparison in such situations.

Configuration 15 (On Demand Separated I/O Organization). The deadline margins and utilization values for configuration 15 are shown in table 4.5.1-3. No data are available for phases 1, 2, 3, 7, and 8 because a simulation error prevented correct modeling of the overrun policy. The minimum deadline margins for this configuration are smaller than those from configuration 14.

The phase 6 case is selected as the preferred configuration based on the criterion that it maximizes the minimum deadline margin for all application rates. The jitter characteristics for the input and output I/O activity and the time delay data are shown in figure 4.5.1-5.

The output jitter histogram for the 100 Hz rate shows two peaks separated by approximately 3 milliseconds. Unlike previous situations, the variation is a significant portion of the minor frame period. A closer look at each

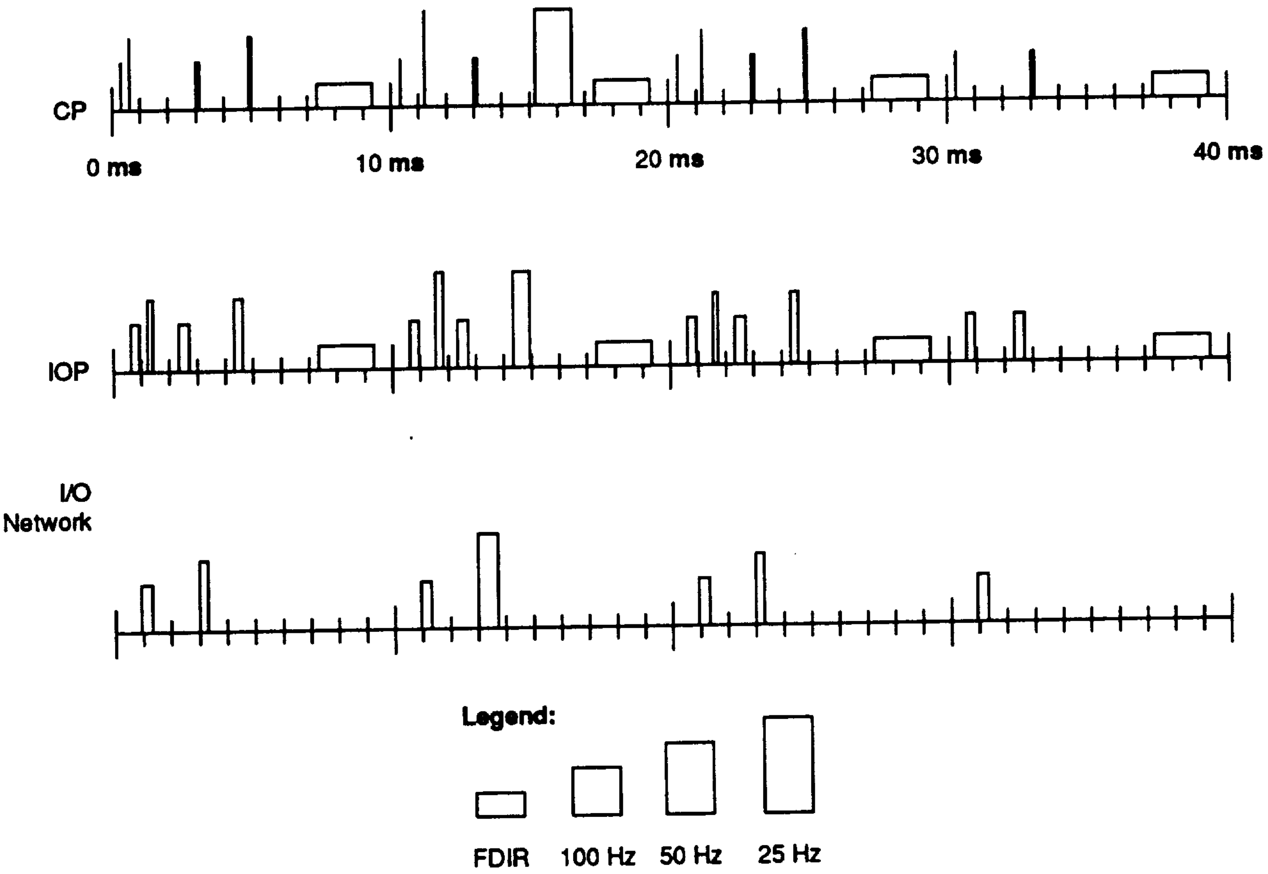
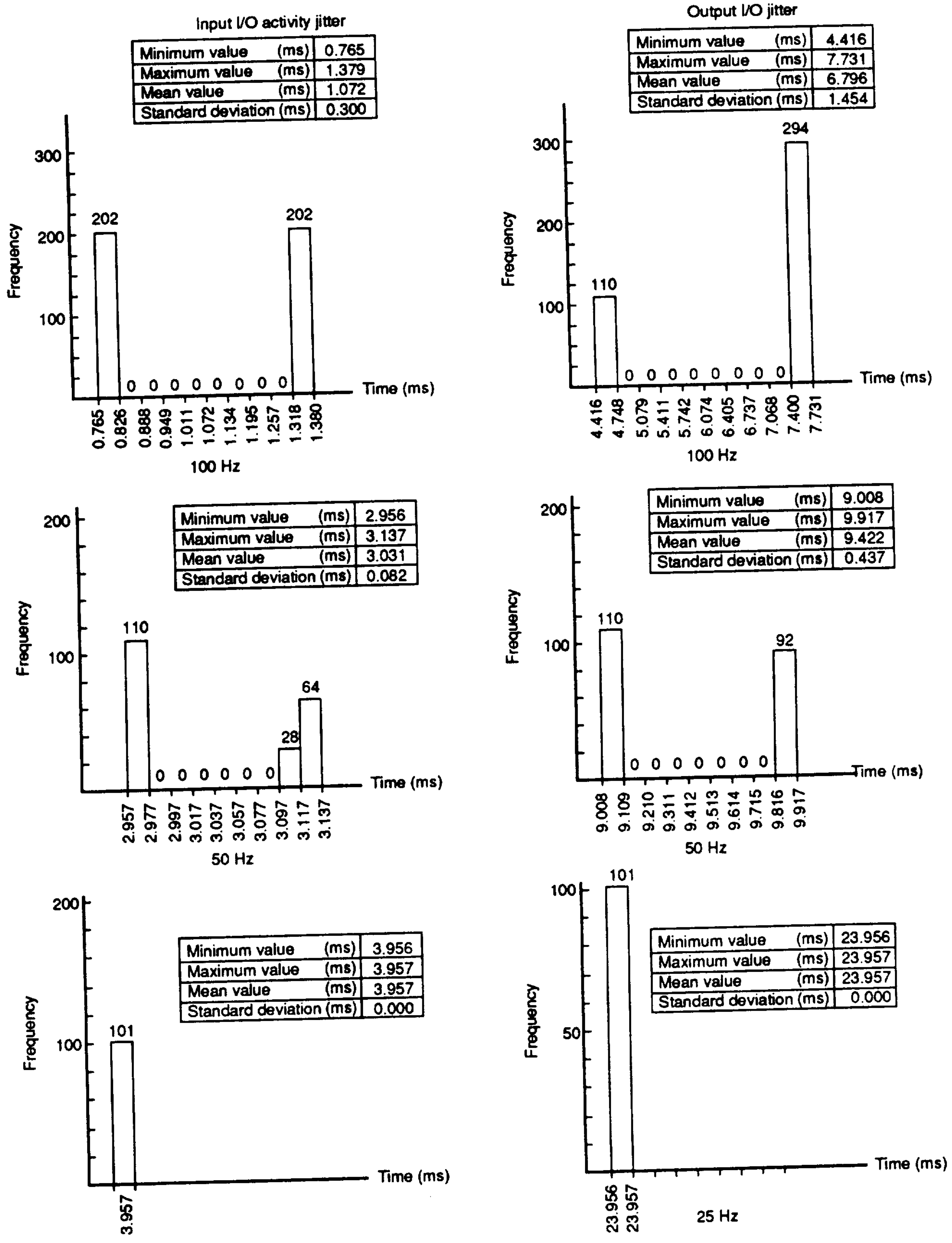


Figure 4.5.1-4. Experiment 4 Engine Control Computer Phase 4 Configuration 14

Table 4.5.1-3. Experiment 4 Configuration 15 Summary

Phase/ID	100 Hz minimum deadline margin(ms)	50 Hz minimum deadline margin(ms)	25 Hz minimum deadline margin(ms)	CP utilization	IOP utilization	I/O system utilization	I/O network utilization
0	4.134	9.996	19.867	51%	84%	96%	7%
1	No data available						
2							
3							
4	5.146	9.889	29.037	50%	87%	93%	7%
5	6.144	11.647	28.612	51%	86%	94%	7%
6	6.144	11.998	29.337	51%	86%	95%	7%
7	No data available						
8							
9	4.743	12.300	30.680	51%	87%	97%	7%



Input I/O activity jitter

Minimum value	(ms)	0.765
Maximum value	(ms)	1.379
Mean value	(ms)	1.072
Standard deviation	(ms)	0.300

Output I/O jitter

Minimum value	(ms)	4.416
Maximum value	(ms)	7.731
Mean value	(ms)	6.796
Standard deviation	(ms)	1.454

Minimum value	(ms)	2.956
Maximum value	(ms)	3.137
Mean value	(ms)	3.031
Standard deviation	(ms)	0.082

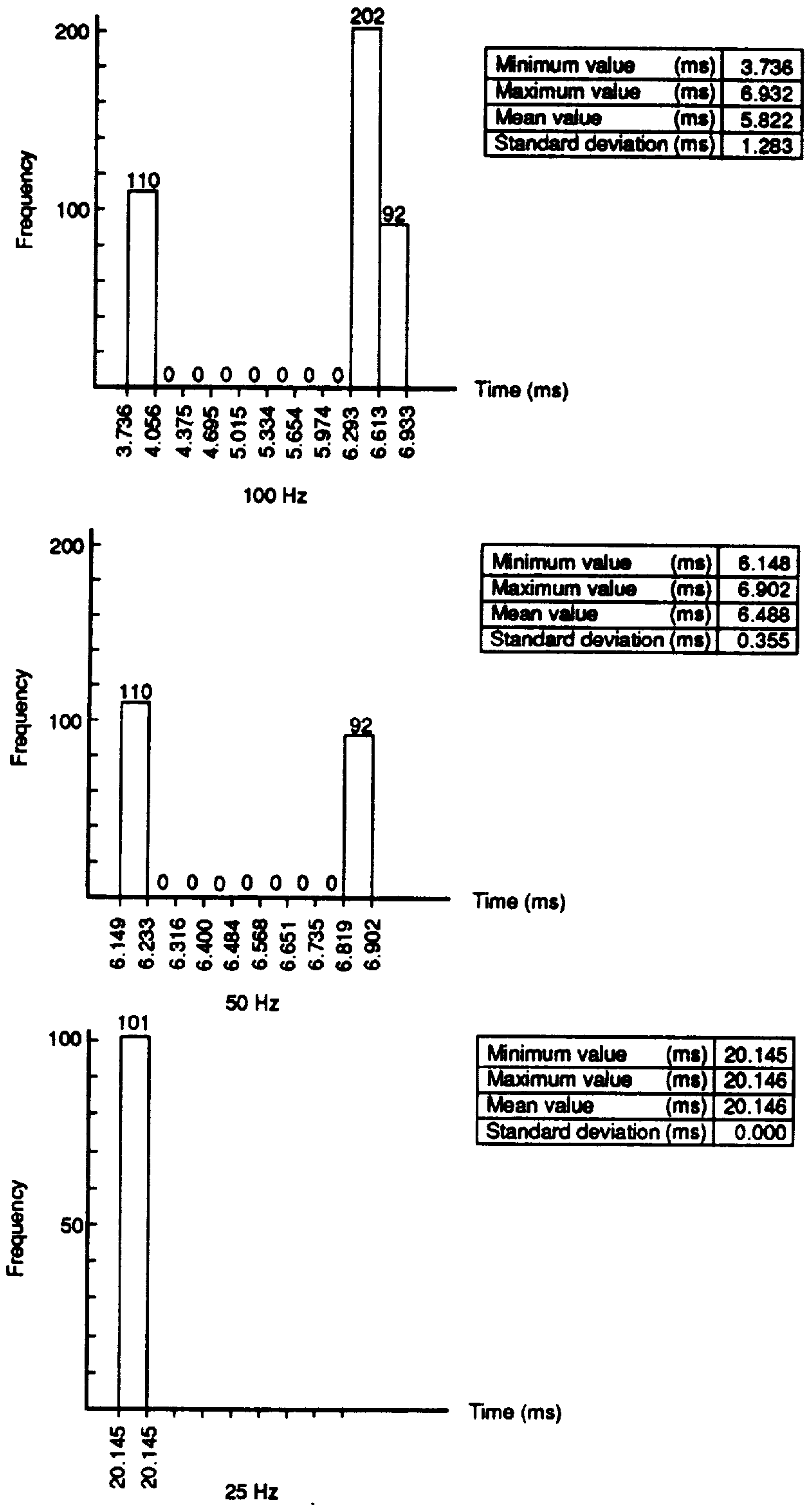
Minimum value	(ms)	9.008
Maximum value	(ms)	9.917
Mean value	(ms)	9.422
Standard deviation	(ms)	0.437

Minimum value	(ms)	3.956
Maximum value	(ms)	3.957
Mean value	(ms)	3.957
Standard deviation	(ms)	0.000

Minimum value	(ms)	23.956
Maximum value	(ms)	23.957
Mean value	(ms)	23.957
Standard deviation	(ms)	0.000

Figure 4.5.1-5 Experiment 4 Configuration 15 Phase 6 Application Performance Parameters (Sheet 1 of 2)

Time delay



100 Hz

Minimum value	(ms)	3.736
Maximum value	(ms)	6.932
Mean value	(ms)	5.822
Standard deviation	(ms)	1.283

50 Hz

Minimum value	(ms)	6.148
Maximum value	(ms)	6.902
Mean value	(ms)	6.488
Standard deviation	(ms)	0.355

25 Hz

Minimum value	(ms)	20.145
Maximum value	(ms)	20.146
Mean value	(ms)	20.146
Standard deviation	(ms)	0.000

Figure 4.5.1-5. Experiment 4 Configuration 15 Phase 6 Application Performance Parameters (Sheet 2 of 2)

peak of the 100 Hz output jitter is shown in figure 4.5.1-6. The first peak reduces to what appears to be a uniform distribution with a range of 16 microseconds. This shape and range are attributed to the system being idle when this I/O request is made. This situation happens most often in minor frame 4, when only the 100 Hz process executes. A more detailed investigation of the second group shows that it is composed of two separate distributions. One distribution is clustered at 7.600 milliseconds and the other appears to be uniformly distributed between 7.715 milliseconds and 7.731 milliseconds. The first cluster is associated with I/O requests that have been preloaded, while the second distribution is associated with I/O requests that are serviced by an idle I/O system. A timeline of the selected configuration is shown in figure 4.5.1-7.

Configuration 16 (Scheduled Grouped I/O Organization). The deadline margins and utilization values for configuration 16 are shown in table 4.5.1-4. These values do not differ significantly from the on demand-grouped I/O configuration. One minor difference is in the minimum deadline margins for phasing cases 1, 2, and 3. These values are 1 millisecond or more larger than the on demand grouped I/O configuration. This is because the start of frame occurs in the IOP instead of the CP. This eliminates the extra overhead step of waking up the CP just to start the I/O activity in the IOP. The earlier start of the I/O activity causes the I/O completion poll to be scheduled 1 millisecond earlier in the frame.

The phase 3 case is selected as the preferred configuration because it has the lowest IO service utilization. The I/O jitter and the time delay characteristics for the phase 3 case are illustrated in figure 4.5.1-8. The I/O jitter and the time delay histograms reveal nothing unexpected. The timeline for one major frame of the preferred configuration is illustrated in figure 4.5.1-9.

Experiment 4 shows that the phasing of the FDIR and the application is critical, especially in heavily loaded cases. An implementation technique will be needed to control the relative execution phasing of the FDIR process and the application. Additionally, the simulation showed that the

Minimum value	(ms)	4.416
Maximum value	(ms)	7.731
Mean value	(ms)	6.796
Standard deviation	(ms)	1.454

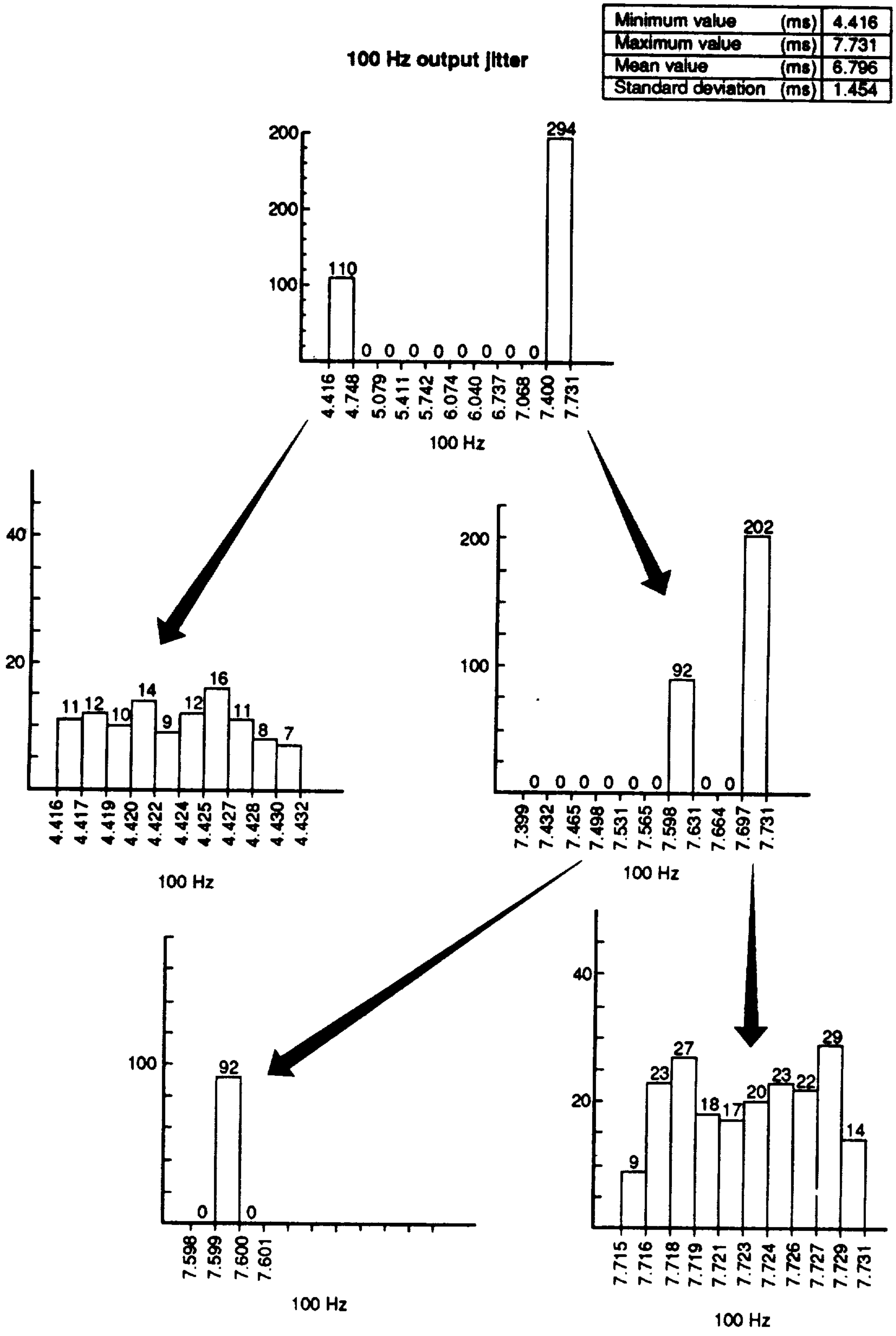


Figure 4.5.1-6. Detailed View of 100 Hz Output Jitter

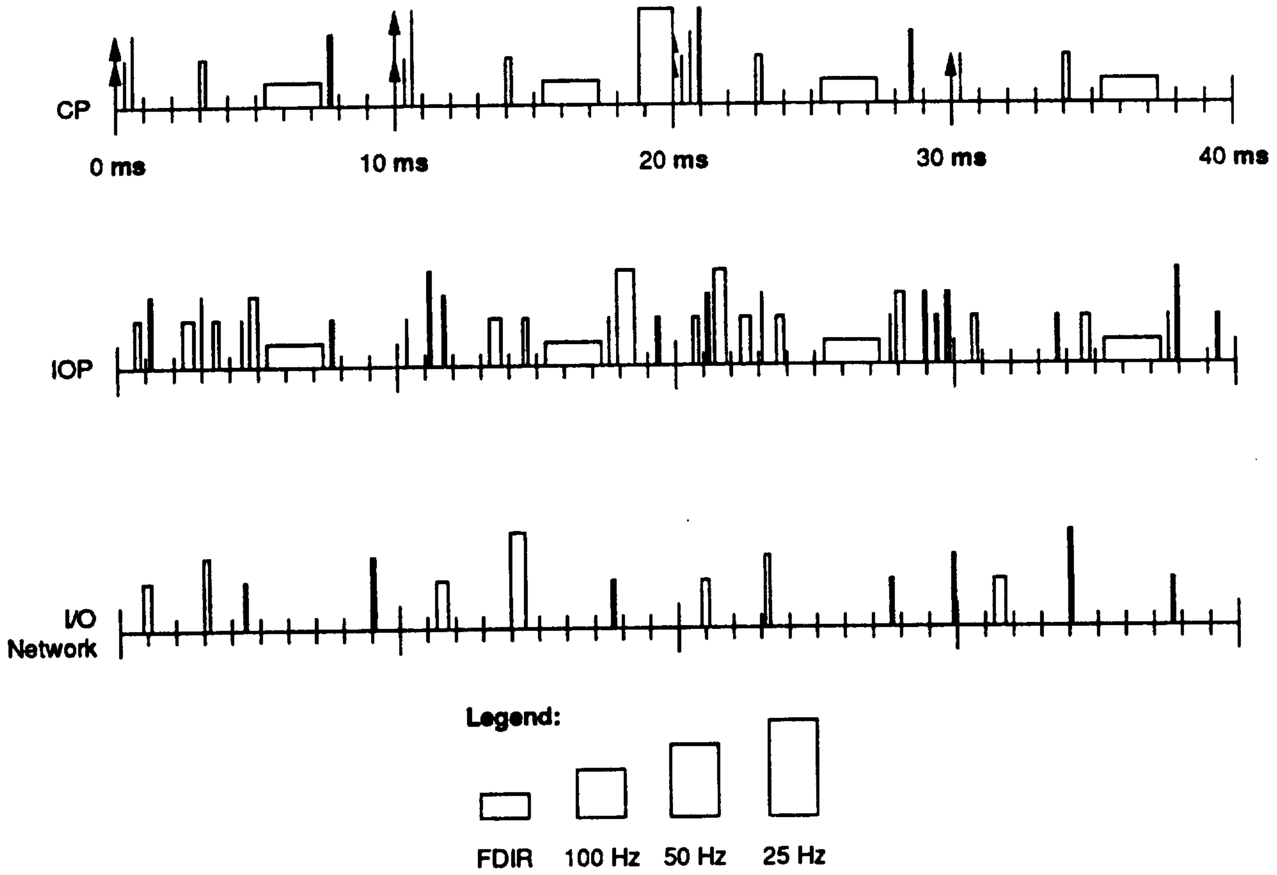
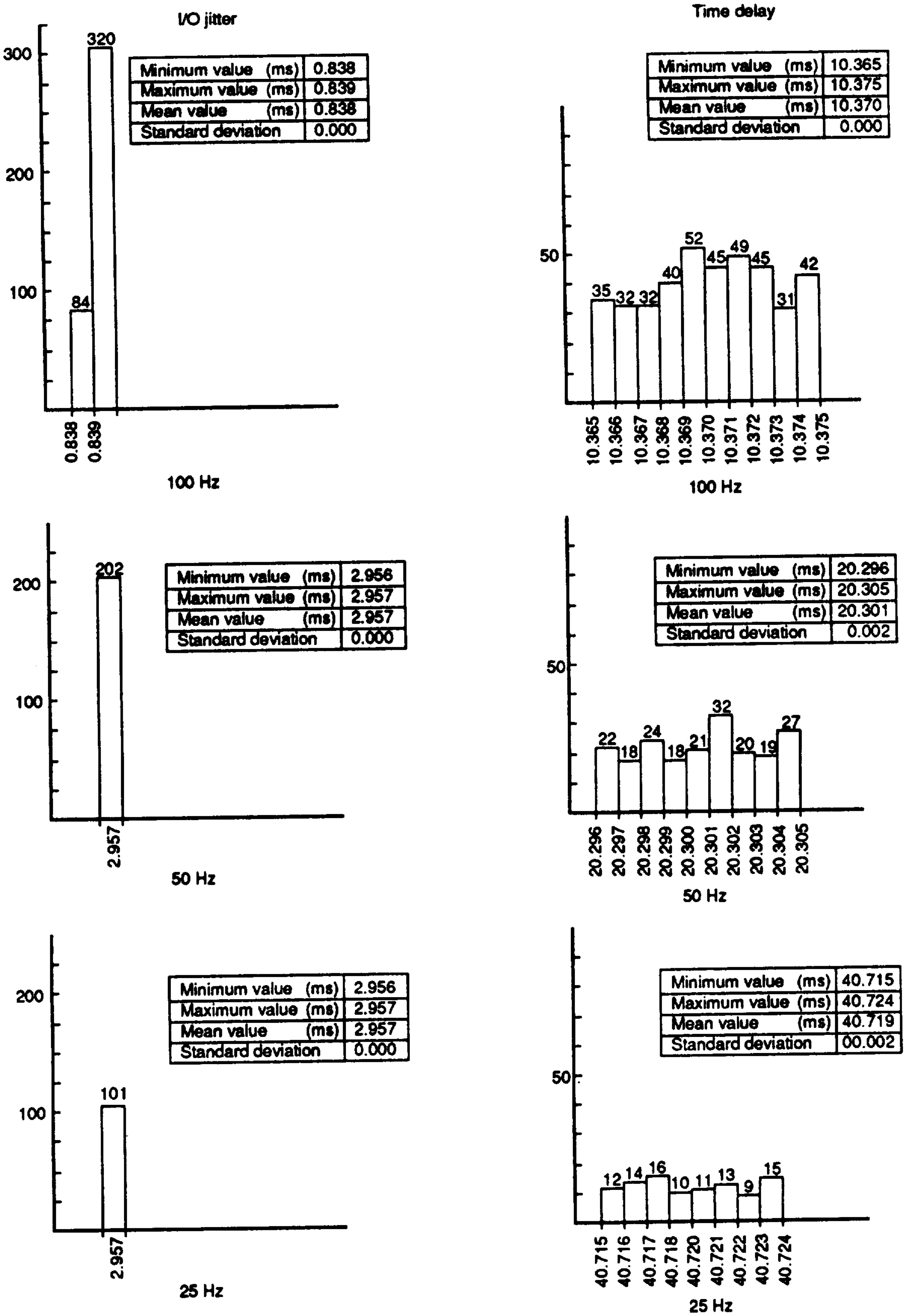


Figure 4.5.1-7. Experiment 4 Configuration 15 Phase 6 Case

Table 4.5.1-4. Experiment 4 Configuration 16 Summary

Phase/ID	100 Hz minumum deadline margin(ms)	50 Hz minimum deadline margin(ms)	25 Hz minimum deadline margin(ms)	CP utilization	IOP utilization	I/O system utilization	I/O network utilization
0	5.841	13.815	32.192	42%	53%	60%	7%
1	7.440	16.115	34.492	42%	55%	62%	7%
2	7.441	16.115	34.492	42%	55%	52%	7%
3	7.141	16.115	34.492	42%	55%	42%	7%
4	7.140	15.815	34.192	42%	58%	42%	7%
5	7.140	15.815	31.592	42%	58%	42%	7%
6	7.140	13.215	31.592	42%	58%	42%	7%
7	7.140	13.515	31.892	42%	56%	59%	7%
8	4.540	14.258	32.635	42%	56%	54%	7%
9	4.840	12.815	31.192	42%	55%	70%	7%



Minimum value (ms)	0.838
Maximum value (ms)	0.839
Mean value (ms)	0.838
Standard deviation	0.000

Minimum value (ms)	10.365
Maximum value (ms)	10.375
Mean value (ms)	10.370
Standard deviation	0.000

Minimum value (ms)	2.956
Maximum value (ms)	2.957
Mean value (ms)	2.957
Standard deviation	0.000

Minimum value (ms)	20.296
Maximum value (ms)	20.305
Mean value (ms)	20.301
Standard deviation	0.002

Minimum value (ms)	2.956
Maximum value (ms)	2.957
Mean value (ms)	2.957
Standard deviation	0.000

Minimum value (ms)	40.715
Maximum value (ms)	40.724
Mean value (ms)	40.719
Standard deviation	00.002

Figure 4.5.1-8. Experiment 4 Configuration 16 Phase 3 Application Performance Parameters

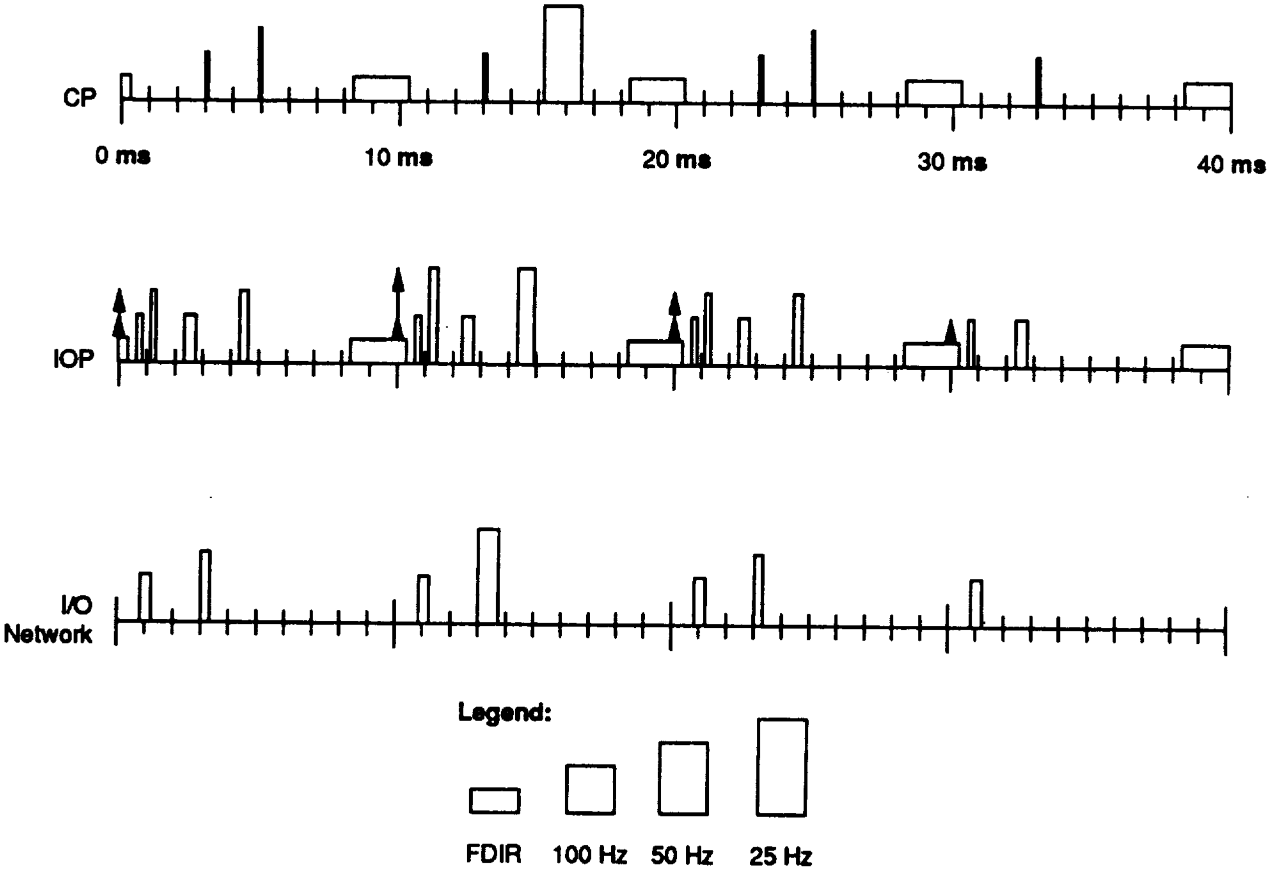


Figure 4.5.1-9. Experiment 4 Configuration 16 Phase 3 Case

system loading was more severe than indicated by the manual estimates, which could not cover resource contention and neglected task sequencing and control overhead.

4.5.2 Utilization (Experiment 5)

The utilization of four key resources was measured during the experiment 4 runs. The resources were the CP, IOP, I/O system, and the I/O network. In this way the objectives of experiment 5 were met without making any dedicated experiment runs. Before discussing the results, some peculiarities of the I/O system and the I/O network utilization parameters are described in greater detail.

The I/O system utilization is a measure of the availability of the I/O system to accommodate additional I/O demands. The I/O system values can vary somewhat according to how utilization is accounted for. Some FDIR application phasings result in scheduling an I/O completion poll (see section 4.4.2) at the same time the FDIR is scheduled to execute. When this happens the FDIR executes in the IOP, because it has higher priority. The I/O completion poll processing is delayed until the completion of the FDIR, which results in busy time counted against the I/O system even though no I/O activity is being accomplished. With slightly different phasing alignments this I/O completion delay does not occur and is not counted as utilization.

The I/O network utilization measures the time that the network is dedicated to performing application I/O. This includes the time from the beginning of the transmission of the first transaction in the I/O activity until the IOS has completed processing the last transaction in the activity. The IAPSA study results revealed that a small utilization value for the I/O network is not necessarily a good indicator of spare I/O capacity. This is because the I/O network use is only a small portion of the end-to-end time requirement for performing an I/O request. The I/O system utilization value appears to be a better indicator of the effect of the application workload on the I/O resources for this implementation.

Flight Control Group

The utilization of the system elements for configuration 6 is shown in table 4.5.1-1. The values are generally high, with the exception of the I/O network. The utilization of the CP is 86%. The utilization of the IOP is between 72% and 75%. The utilization measures for the preferred candidate (phase 0) do not meet the 100% growth capability requirement.

Engine Control Group

Configuration 14 (On Demand Grouped I/O Organization). The utilization figures are shown in table 4.5.1-2. Three phasing cases result in the lowest CP utilization (phase 1, 2, and 3). This is because these phasings have one less process switch per minor frame than all the other phasings. The IOP utilization ranges from 53% to 58%. As the scheduled time for FDIR moves through the minor frame, different sequences of processing occur. The wide range in I/O system utilization is attributed to the accounting method characteristics described previously. That is, if the FDIR interferes with the processing for I/O activity, the FDIR processing time is counted against the I/O system. The I/O network utilization is well within the system capacity limits.

Configuration 15 (On Demand Separated I/O Organization). The data for this configuration are presented in table 4.5.1-3. The utilization of the CP is essentially unchanged from other configurations. However, the IOP utilization has jumped significantly. The cause is the separation of the I/O activity into two activities per rate group per control cycle. When the I/O is divided into two activities per rate group, the overhead associated with I/O request execution doubles. This has an adverse effect on I/O system utilization resulting in essentially no reserve capacity remaining in this configuration.

Configuration 16 (Scheduled Grouped I/O Organization). The utilization values for configuration 16 are shown in table 4.5.1-4. Because the start of frame event occurs in the IOP, the CP utilization values are smaller than in the on demand grouped I/O configuration (configuration 14).

4.5.3 I/O Scheduling (Experiment 3)

The purpose of this experiment is to evaluate the effect of the I/O scheduling mechanism on the performance of the application during normal operation. In addition a single representative configuration will be selected for the engine and flight control groups to serve as a baseline for the I/O link fault experiments. No special runs were made to obtain data for this experiment. all needed information was gathered during experiment 4 runs.

Flight Control Group

As discussed in section 4.5.1, only the scheduled grouped I/O organization option (configuration 6) could support the workload of the flight control group. The resulting performance for the preferred phasing option is summarized in table 4.5.3-1.

Engine Control Group

The performance of the preferred configurations for the engine control group from experiment 4 are summarized in table 4.5.3-1. All engine control configurations appear to have adequate deadline margin. The CP utilization is satisfactory in all configurations. The IOP utilization is satisfactory except for the on demand separated I/O option (configuration 15). For experiment 2, a representative single configuration is chosen for the engine control group. The on demand grouped I/O option (configuration G4) is selected as the I/O scheduling mechanism for the engine control computer.

4.5.4 I/O Link Failure (Experiment 2)

The purpose of experiment 2 is to measure the time needed to successfully return a faulty network to service. In addition, the experiment evaluates the effect of the repair processing on application performance. The out-of-service time measures the time between a network being taken out-of-service for repair and the time the network manager returns a

Table 4.5.3-1. Summary I/O Scheduling

Engine Control Computer						
	100 Hz Minimum Deadline Margin (ms)	50 Hz Minimum Deadline Margin (ms)	25 Hz Minimum Deadline Margin (ms)	CP Utilization	IOP Utilization	I/O System Utilization
On demand/grouped	7.140	15.598	33.941	51%	58%	39%
On demand/separated	6.140	11.998	29.337	51%	86%	95%
Scheduled/grouped	7.440	16.115	34.492	42%	55%	42%
Flight Control Computer						
	100 Hz Minimum Deadline Margin (ms)	50 Hz Minimum Deadline Margin (ms)	25 Hz Minimum Deadline Margin (ms)	CP Utilization	IOP Utilization	I/O System Utilization
Scheduled/grouped	2.934	7.007	15.538	86%	72%	80%

network to service. A key concern in this experiment is the change in the application timing characteristics due to the I/O network repair activity. Second, the out-of-service time must be compared to the 1 second fault recovery period used in the initial reliability analysis. Experiment 2 faults were inserted at a random time relative to the major frame for each run. For each link the experiment was repeated 50 times.

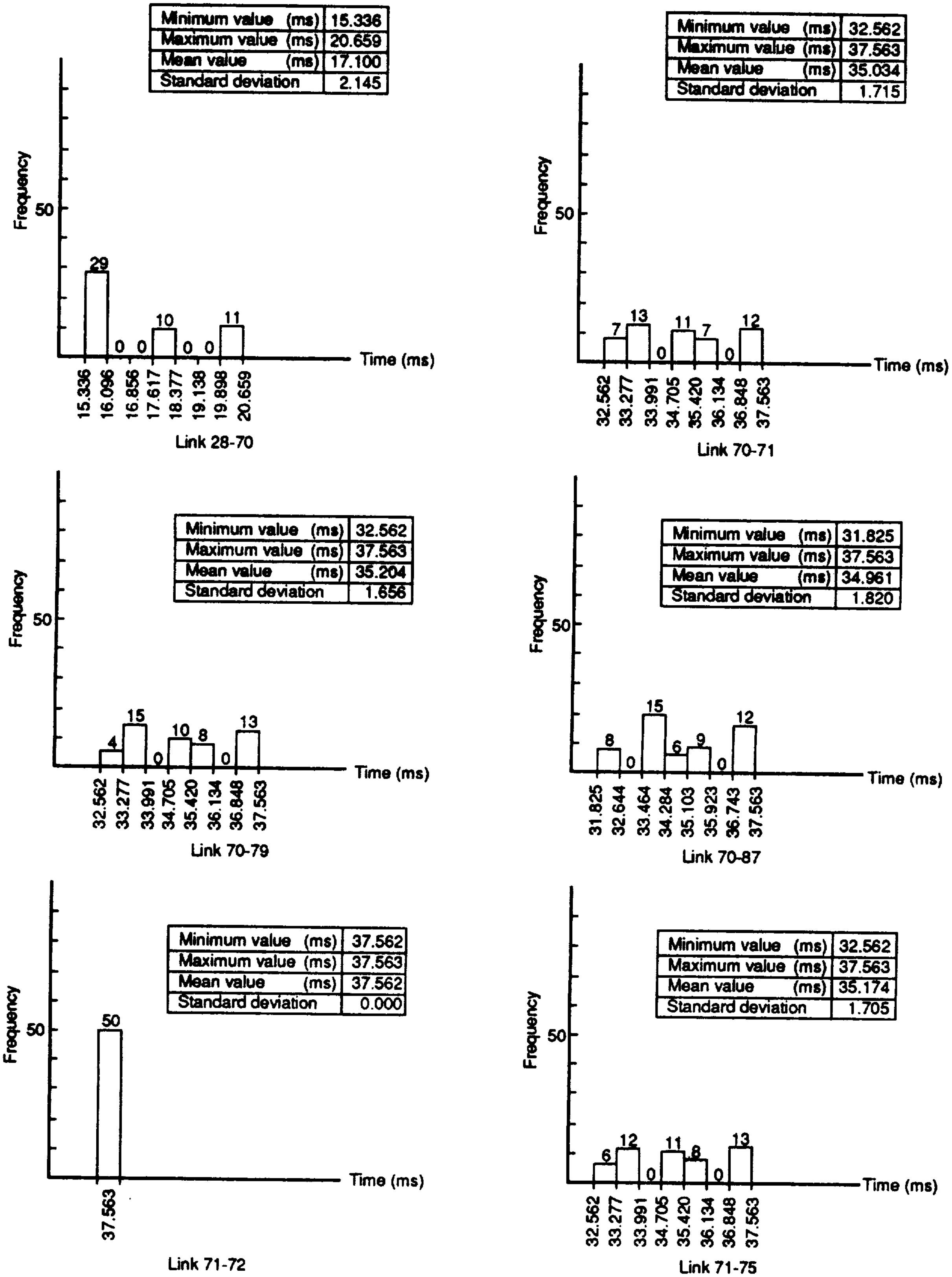
One-Shot Repair Strategy Experiments

Flight Control Group (Configuration 11). The I/O network out-of-service time distributions for this experiment are summarized in figure 4.5.4-1. Two factors drive the resulting distribution. First, the detection of a fault is dependent on the topology of the network. Second, the detection of the fault by the application I/O activity is dependent on the time of application I/O execution. The network topology aspect is illustrated in figure 4.5.4-2. Network 2 is shown as it is grown at power-up with no faults. The stars contained in each node indicate the application I/O activity rate groups that are dependent on the inboard link on that node.

Note that several of the link failures can be detected by communication faults appearing in more than one application I/O activity. This is because either the attached DIU or the downstream DIUs are accessed by more than one application rate group's I/O activity. One example is DIU CP3, which is accessed by both the 50 Hz rate and the 25 Hz rate I/O activity. Another example is DIU RR, which is accessed by the 50 Hz rate, while its downstream node, TER, is accessed by the 100 Hz I/O activity. Therefore, when the link that connects RR to FC4 fails, either the 100 Hz I/O or the 50 Hz I/O activity will result in communication failures.

There are seven application I/O completion polls per major frame for the grouped I/O organization configurations. Therefore there are seven discrete times in a major frame that a fault can be detected. A particular fault may only be discovered by the 50 Hz I/O activity and thus have only two opportunities for detection per major frame. A fault that is detected by both the 100 Hz and the 50 Hz I/O activities has six detection opportunities per major frame.

Out of service time



Link 28-70

Minimum value (ms)	15.336
Maximum value (ms)	20.659
Mean value (ms)	17.100
Standard deviation	2.145

Link 70-71

Minimum value (ms)	32.562
Maximum value (ms)	37.563
Mean value (ms)	35.034
Standard deviation	1.715

Link 70-79

Minimum value (ms)	32.562
Maximum value (ms)	37.563
Mean value (ms)	35.204
Standard deviation	1.656

Link 70-87

Minimum value (ms)	31.825
Maximum value (ms)	37.563
Mean value (ms)	34.961
Standard deviation	1.820

Link 71-72

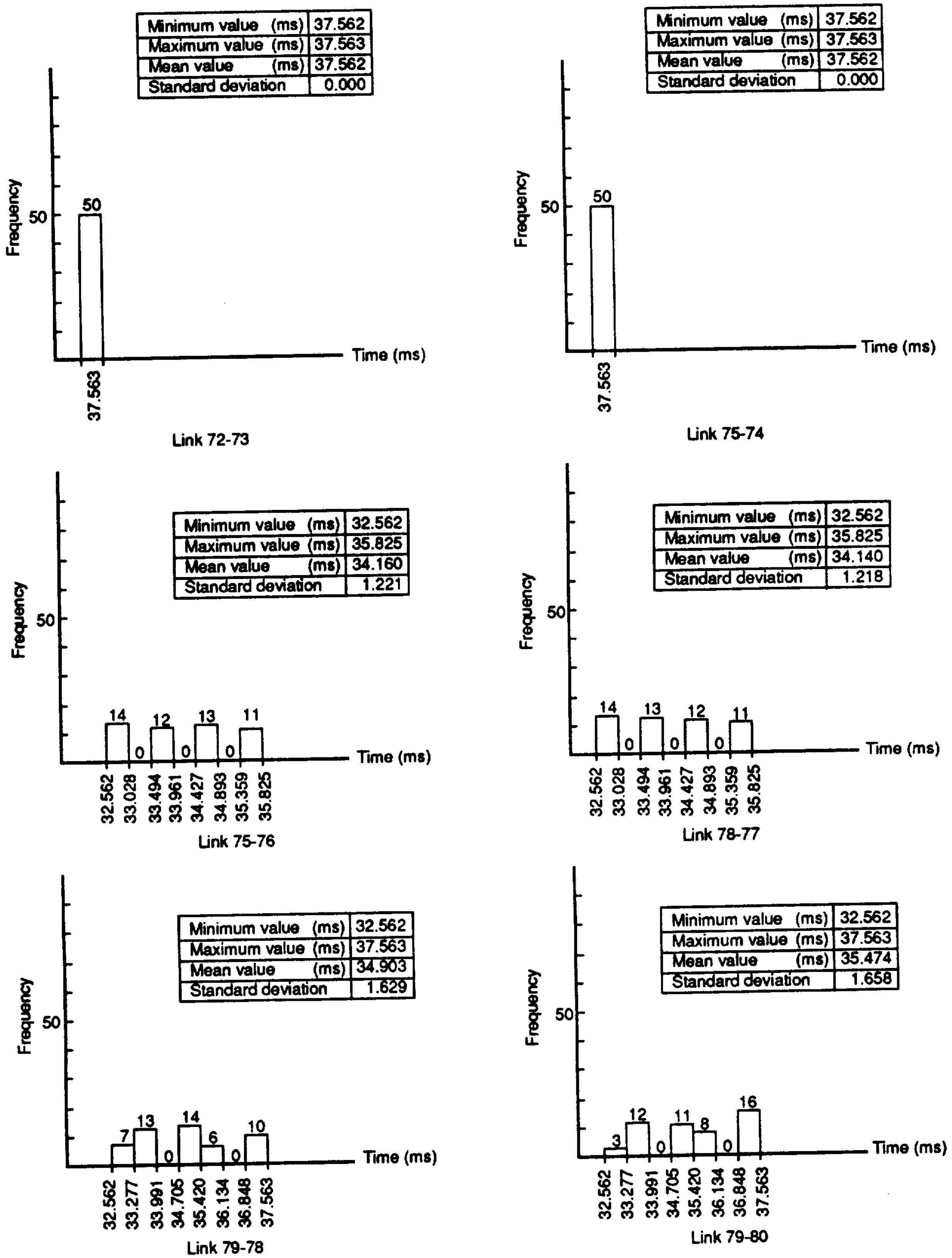
Minimum value (ms)	37.562
Maximum value (ms)	37.563
Mean value (ms)	37.562
Standard deviation	0.000

Link 71-75

Minimum value (ms)	32.562
Maximum value (ms)	37.563
Mean value (ms)	35.174
Standard deviation	1.705

Figure 4.5.4-1. Out of Service Time For One Shot Repair – Flight Control Computer (Sheet 1 of 3)

Out of service time



Link 72-73

Minimum value (ms)	37.562
Maximum value (ms)	37.563
Mean value (ms)	37.562
Standard deviation	0.000

Link 75-74

Minimum value (ms)	37.562
Maximum value (ms)	37.563
Mean value (ms)	37.562
Standard deviation	0.000

Link 75-76

Minimum value (ms)	32.562
Maximum value (ms)	35.825
Mean value (ms)	34.160
Standard deviation	1.221

Link 78-77

Minimum value (ms)	32.562
Maximum value (ms)	35.825
Mean value (ms)	34.140
Standard deviation	1.218

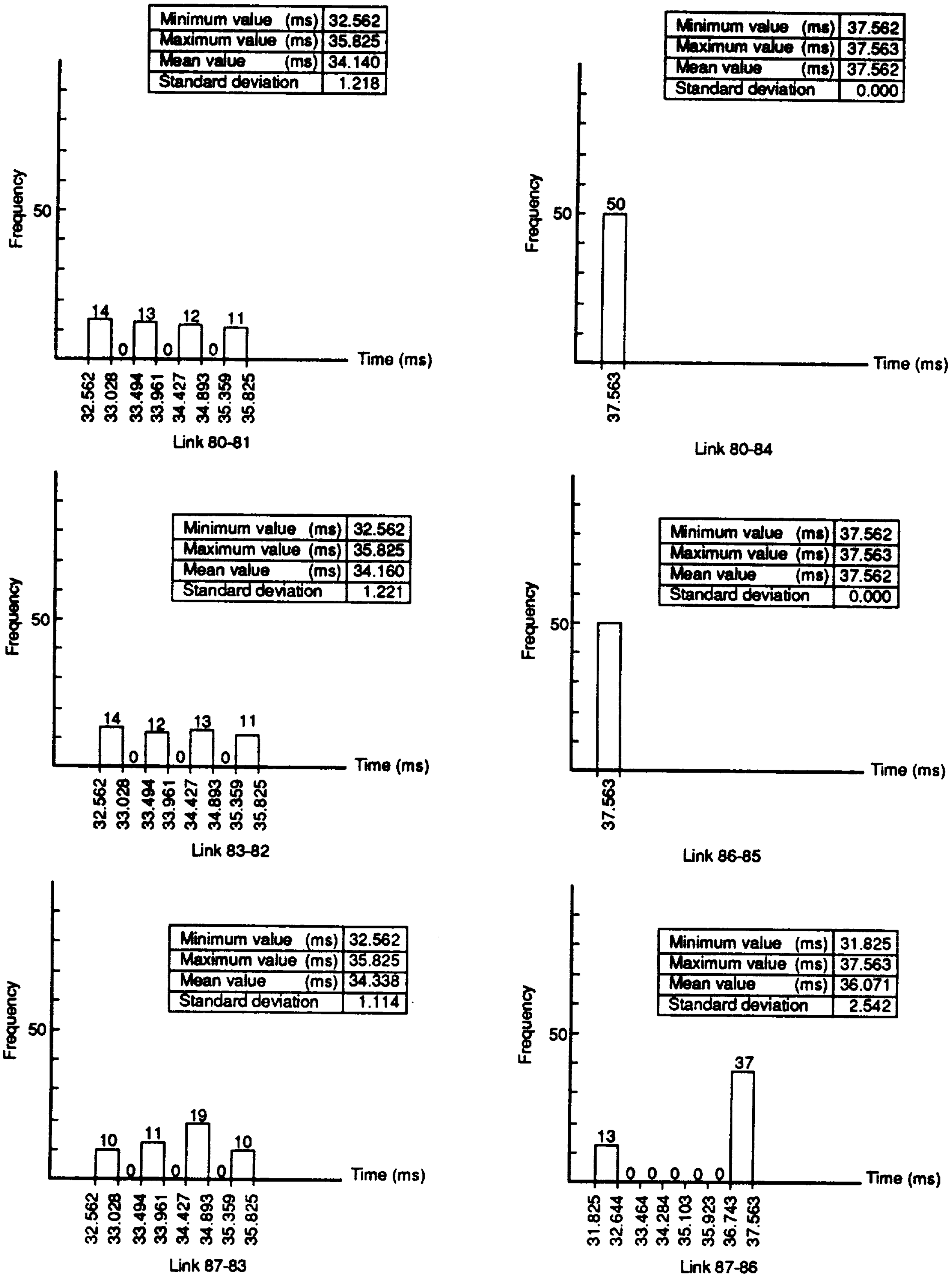
Link 79-78

Minimum value (ms)	32.562
Maximum value (ms)	37.563
Mean value (ms)	34.903
Standard deviation	1.629

Link 79-80

Minimum value (ms)	32.562
Maximum value (ms)	37.563
Mean value (ms)	35.474
Standard deviation	1.658

Figure 4.5.4-1. Out of Service Time For One Shot Repair – Flight Control Computer (Sheet 2 of 3)



Out of service time

Link 80-81

Minimum value (ms)	32.562
Maximum value (ms)	35.825
Mean value (ms)	34.140
Standard deviation	1.218

Link 80-84

Minimum value (ms)	37.562
Maximum value (ms)	37.563
Mean value (ms)	37.562
Standard deviation	0.000

Link 83-82

Minimum value (ms)	32.562
Maximum value (ms)	35.825
Mean value (ms)	34.160
Standard deviation	1.221

Link 86-85

Minimum value (ms)	37.562
Maximum value (ms)	37.563
Mean value (ms)	37.562
Standard deviation	0.000

Link 87-83

Minimum value (ms)	32.562
Maximum value (ms)	35.825
Mean value (ms)	34.338
Standard deviation	1.114

Link 87-86

Minimum value (ms)	31.825
Maximum value (ms)	37.563
Mean value (ms)	36.071
Standard deviation	2.542

Figure 4.5.4-1. Out of Service Time For One Shot Repair – Flight Control Computer (Sheet 3 of 3)

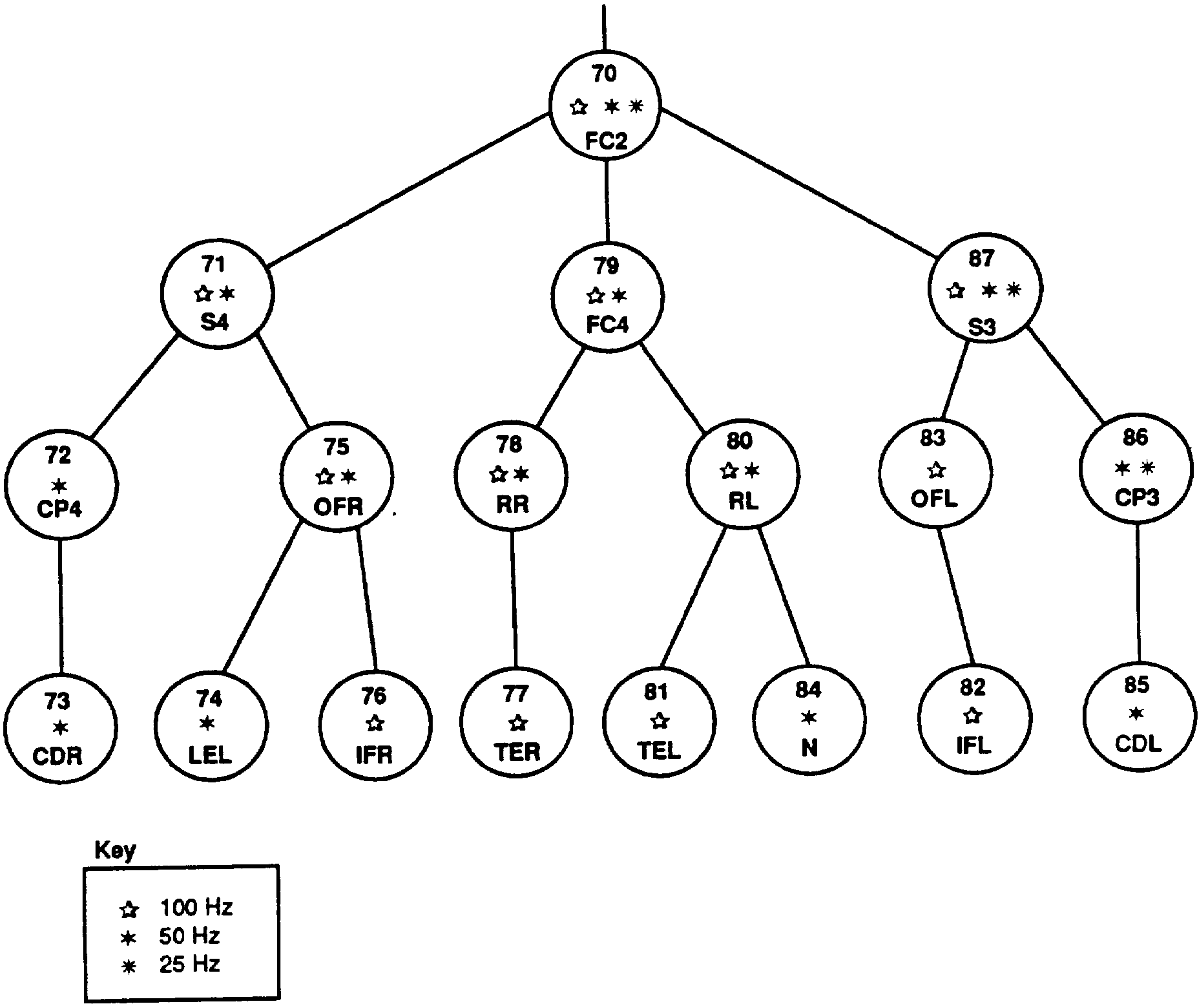


Figure 4.5.4-2. Network 2 With No Faults

Network repair activity consists of a sequence of processing and I/O request steps to repair a passive I/O link failure. The network manager is responsible for implementing the repair strategy. It executes in idle time on the IOP since its priority is less than the application I/O activity. The application activity in the IOP is relatively constant; the idle time slots therefore align themselves to positions in the major frame. The repair steps are accomplished during these idle periods. The I/O completion poll activity that detects the communication fault determines the initial phasing of the I/O repair activity relative to the major frame. The initial phasing, coupled with the sequence of idle time slots available in the IOP control, governs the out-of-service time.

The number of distinct out-of-service times depends primarily on the unique combinations of initial phasings and sequences of IOP idle time. Variability of IOP processing can also have an effect. A link failure between node 75 and 76 is only detected by the 100 Hz I/O activity. There are four unique repair times corresponding to faults detected by 100 Hz I/O activity. However, there is only one unique time, longer than any of the 100 Hz repair times for link failures detected by the 50 Hz rate. This is seen in the histogram for link failures between node 71 and node 72 on sheet 1 of figure 4.5.4-1. There is also a unique repair time, shorter than any other, for failures detected by the 25 Hz activity.

The link failures that are detected by more than one I/O activity result in out-of-service times that are a combination of the results for the individual rates. This is observed in the data for link failures between nodes 71 and 75, which are detected by the 100 Hz and 50 Hz application I/O activity, and failures between nodes 87 and 86, which are detected by the 50 Hz and 25 Hz I/O activity. This discussion illustrates that the results of the random testing can often be related to the details of system operation.

A summary of the application performance measures for each link failure sequence is illustrated in table 4.5.4-1. The CP utilization is not included in the summary because it is not affected by the network repair activity. In addition, the I/O network utilization is not included because

Table 4.5.4-1. Experiment 2 Configuration 11 Summary

Failed link	100 Hz minimum deadline margin(ms)	50 Hz minimum deadline margin (ms)	25 Hz minimum deadline margin(ms)	IOP utilization	IO system utilization
28-70	2.927	2.549	11.355	78%	68%
70-71	2.959	2.523	11.561	78%	68%
70-79	2.897	2.618	11.189	78%	68%
70-87	2.835	2.531	11.387	78%	68%
71-72	3.003	2.286	11.153	78%	68%
71-75	2.914	2.531	11.267	78%	68%
79-78	2.945	2.559	11.134	78%	68%
79-80	2.851	2.488	11.146	78%	68%
87-83	2.972	2.607	11.418	79%	68%
87-86	2.930	2.558	11.167	78%	68%
72-73	2.689	2.570	10.946	78%	69%
75-74	2.884	2.577	10.946	78%	69%
75-76	2.950	2.410	11.164	78%	68%
78-77	2.948	2.471	11.309	78%	68%
80-81	2.948	2.471	11.309	79%	68%
80-84	2.967	2.485	10.946	78%	68%
83-82	2.950	2.410	11.164	79%	68%
86-85	2.925	2.649	10.946	78%	68%

it is not affected on the "good" network, while the repair activity has exclusive use of the failed network. Some significant differences are observed when comparing table 4.5.4-1 to the normal flight control group results in table 4.5.3-1. First, the minimum deadline margin of the 50 Hz rate has been reduced from more than 7 milliseconds to approximately 2.5 milliseconds. Second, the utilization of the IOP has increased slightly while the I/O service utilization has been reduced from 80 percent to 68 percent.

Figure 4.5.4-3 is a timeline of processes for the CP, IOP, and I/O networks as they occurred in the 33 repetitions of the experiment simulating a link failure between nodes 72 and 73. The second major frame shown in the top part of the figure illustrates the alignment of processes during normal operation. At 67.1 milliseconds the link between nodes 72 and 73 is failed. This link is used exclusively by the 50 Hz I/O activity. The failure occurs during the second 50 Hz I/O activity in this major frame but after the transaction that uses the failed link, so the failure is not [discovered until the next 50 Hz I/O activity.

The I/O link failure is detected in the third major frame illustrated in the bottom part of the figure. Network 2 is taken out of service when the 50 Hz I/O completion poll processing discovers the communication faults. The effects of this action are immediately observed in the second minor frame. The load time for the I/O activity is reduced by 50 percent because network 2 is out of service. When a network is taken out of service, the reduction in pre- and post-processing needs in the IOP results in a new alignment application I/O processing.

As a result of network 2 being taken out of service, the 25 Hz I/O activity starts on the network earlier. This earlier start has two causes. First, loading activity for the 25 Hz data rate fits into the new idle time slot just before the FDIR processing (preloading). Second, the 100 Hz I/O activity unloading time is reduced.

At approximately 95 milliseconds, the first IOP idle slot occurs and the repair activity for network 2 starts. The first step in this process is to

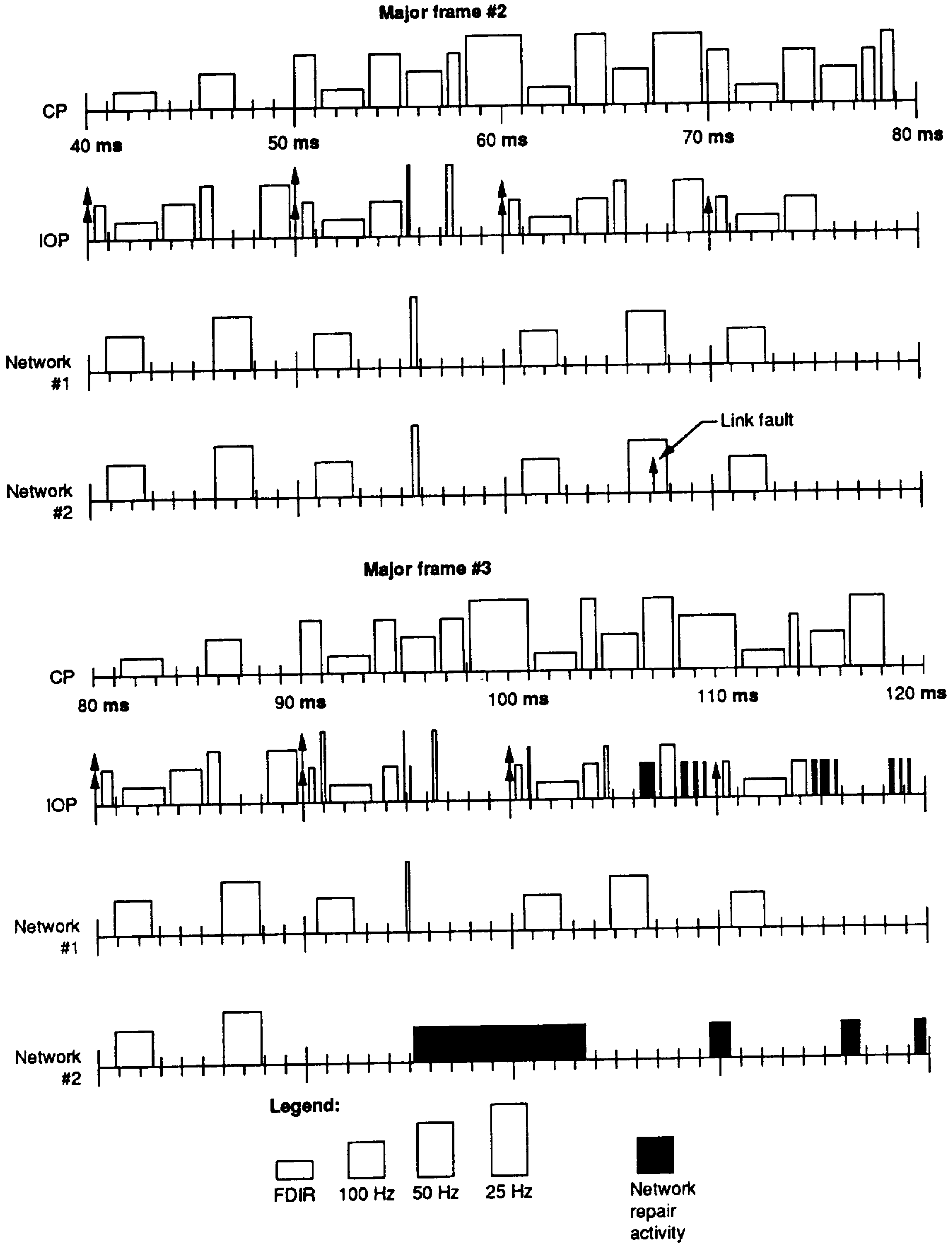


Figure 4.5.4-3. Experiment 2 Configuration 11, Link 72-73, Run 33 (1 of 2)

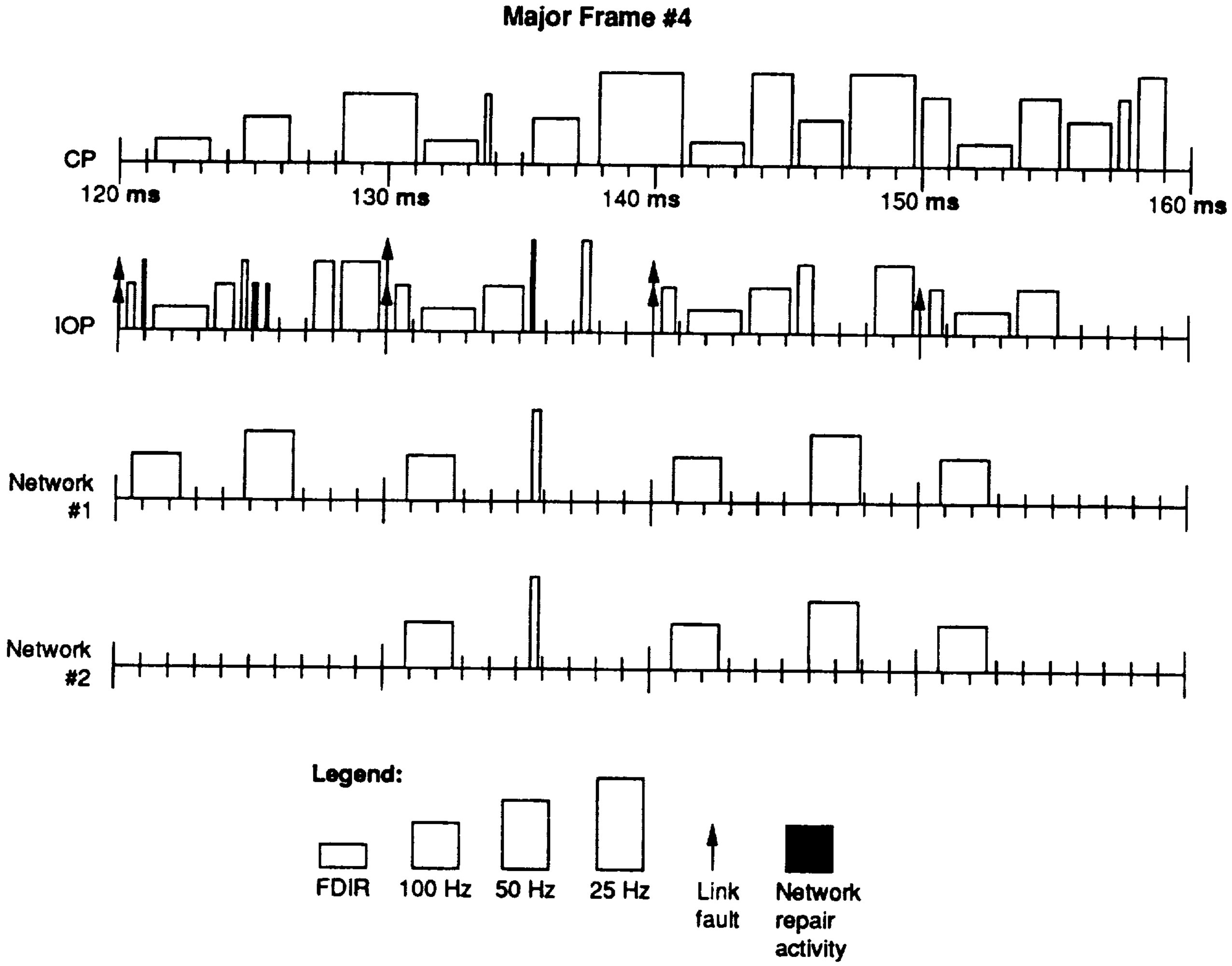


Figure 4.5.4-3. Experiment 2 Configuration 11, Link 72-73, Run 33 (2 of 2)

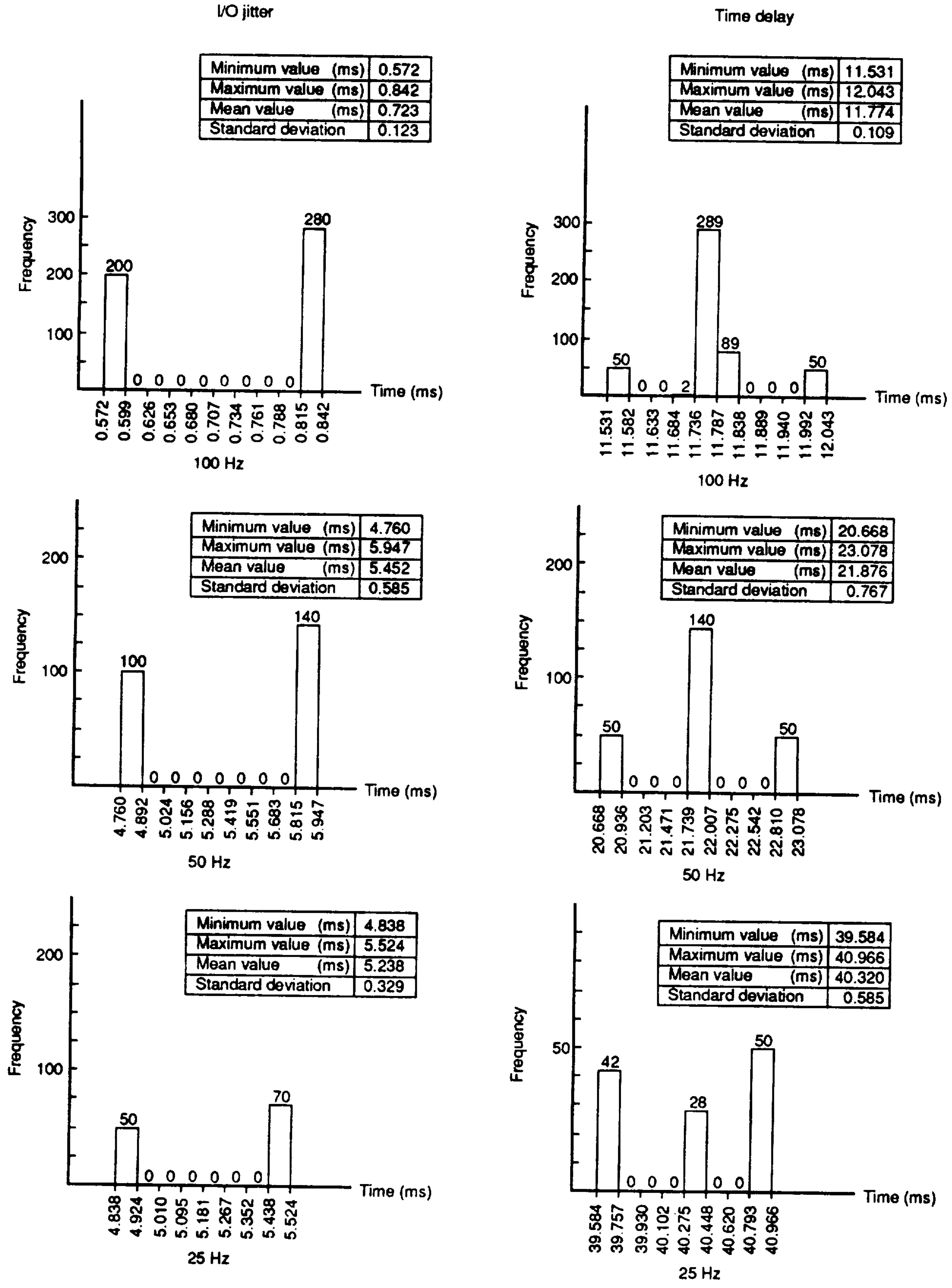
attempt communication with all the nodes in network 2, as seen by the long-duration I/O activity on the network 2 line.

The reduced minimum deadline margin for the 50 Hz rate group is a result of the new idle time slot in the IOP after the completion of the 100 Hz I/O loading. When this new idle time slot occurs in the third minor frame, the 50 Hz data loading starts during the gap. The deadline for the 50 Hz rate group is the start of I/O data loading. One effect of taking the network out of service is a forward shift of the 50 Hz computing deadline by 5 milliseconds.

The overall 50 Hz deadline margin effect is a reduction for the one transition frame when the deadline moves earlier by 5 milliseconds. The new minimum deadline margin is 2.5 milliseconds for the 50 Hz rate group. However, because the computing completion immediately moves forward as the activity in the CP adjusts itself to the new alignment of events in the IOP, the deadline margin jumps to a larger value for the remainder of the repair period. Thus the minimum value sample occurs in the transition frame in all of the failure simulations.

The repair activity in the IOP and on the network continues throughout the remainder of this major frame and completes in the first minor frame shown in major frame 4. The last processing element of the repair activity returns network 2 to service during the execution of the 50 Hz I/O. When network 2 is returned to service, the processing elements in the IOP realign to the positions they occupied before the fault. The return of these processing elements to their former positions triggers a new CP processing alignment which returns the processing elements to their positions before the network failure.

Figure 4.5.4-4 illustrates the other two application performance measures during I/O network repair. Data are presented for all 50 repetitions of the experiment. The data for both measures include samples when both networks are in service as well as when only one network is in service. Some of the I/O jitter samples reflect the earlier start of activity when only one network is in service. The time delay data include samples from



Minimum value (ms)	0.572
Maximum value (ms)	0.842
Mean value (ms)	0.723
Standard deviation	0.123

Minimum value (ms)	11.531
Maximum value (ms)	12.043
Mean value (ms)	11.774
Standard deviation	0.109

Minimum value (ms)	4.760
Maximum value (ms)	5.947
Mean value (ms)	5.452
Standard deviation	0.585

Minimum value (ms)	20.668
Maximum value (ms)	23.078
Mean value (ms)	21.876
Standard deviation	0.767

Minimum value (ms)	4.838
Maximum value (ms)	5.524
Mean value (ms)	5.238
Standard deviation	0.329

Minimum value (ms)	39.584
Maximum value (ms)	40.966
Mean value (ms)	40.320
Standard deviation	0.585

Figure 4.5.4-4. Experiment 2 Configuration 11, Link 72-73, Application Performance Parameters

the transition frames when a network is taken out of service and is put back in service. In both situations the relative frame time of the I/O activity event shifts, resulting in an unusually large or small value for time delay for the transition frame.

Engine Control Group (Configuration 13). The I/O network out of service times for this configuration are shown in figure 4.5.4-5. Network 2 is illustrated in figure 4.5.4-6 as it is grown with no link failures. The out-of-service times for this configuration are approximately 20 milliseconds with the exception of root link failures which take less than 10 milliseconds. These times are faster than the flight control configuration because there is more idle IOP capacity to perform repair activity.

A summary of the deadline margin and utilization data for each link failure is shown in table 4.5.4-2. One difference in the data for this configuration and the flight control computer is that the minimum deadline margins were not significantly affected by the I/O network repair activity. When a network is taken out of service, the application processing elements do not significantly realign. One reason for this is that the processing demands on the IOP are very light. A timeline of the repair activity from the 21st repetition of link failure between node 70 and 71 is shown in figure 4.5.4-7.

The remaining application performance measures during repair are illustrated in figure 4.5.4-8.

Regrow Repair Strategy Experiments

The one-shot repair strategy is able to diagnose and repair only a few active network failures. A check is made to uniquely identify failures such as an inboard babbling port and to select the proper repair actions. However, other failures such as an outboard babbling link will have the same symptoms as a passively failed link. The repair of an outboard babbler that is diagnosed as a passive link failure changes the babbler's

Out of service time

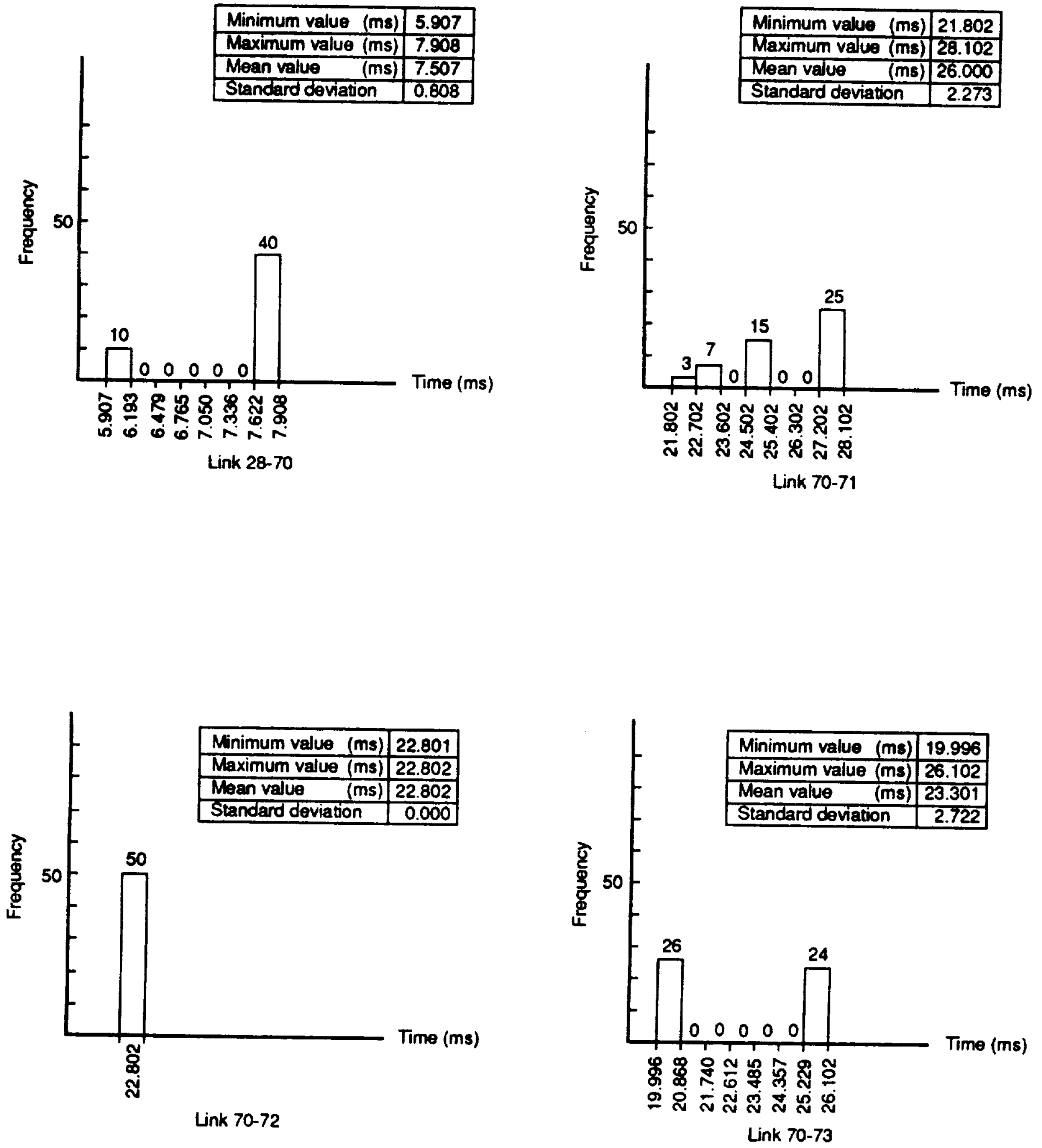


Figure 4.5.4-5. Out of Service Time for One Shot Repair Engine Control Computer

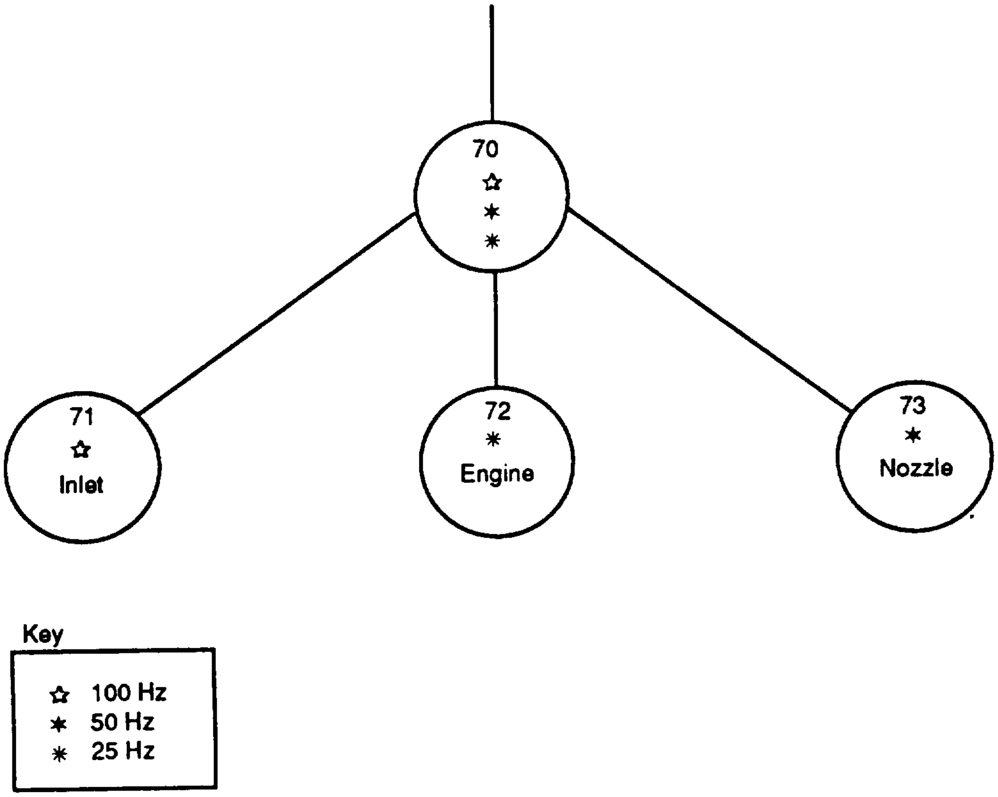


Figure 4.5.4-6. Network 2 With No Faults

Table 4.5.4-2. Experiment 2 Configuration 13 Summary

Failed link	100 Hz minimum deadline margin(ms)	50 Hz minimum deadline margin(ms)	25 Hz minimum deadline margin(ms)	IOP utilization	I/O system utilization
28-70	7.141	15.598	34.022	59%	39%
70-71	7.040	14.749	33.078	63%	38%
70-72	7.142	15.595	33.989	64%	38%
70-73	7.147	15.448	33.996	63%	39%

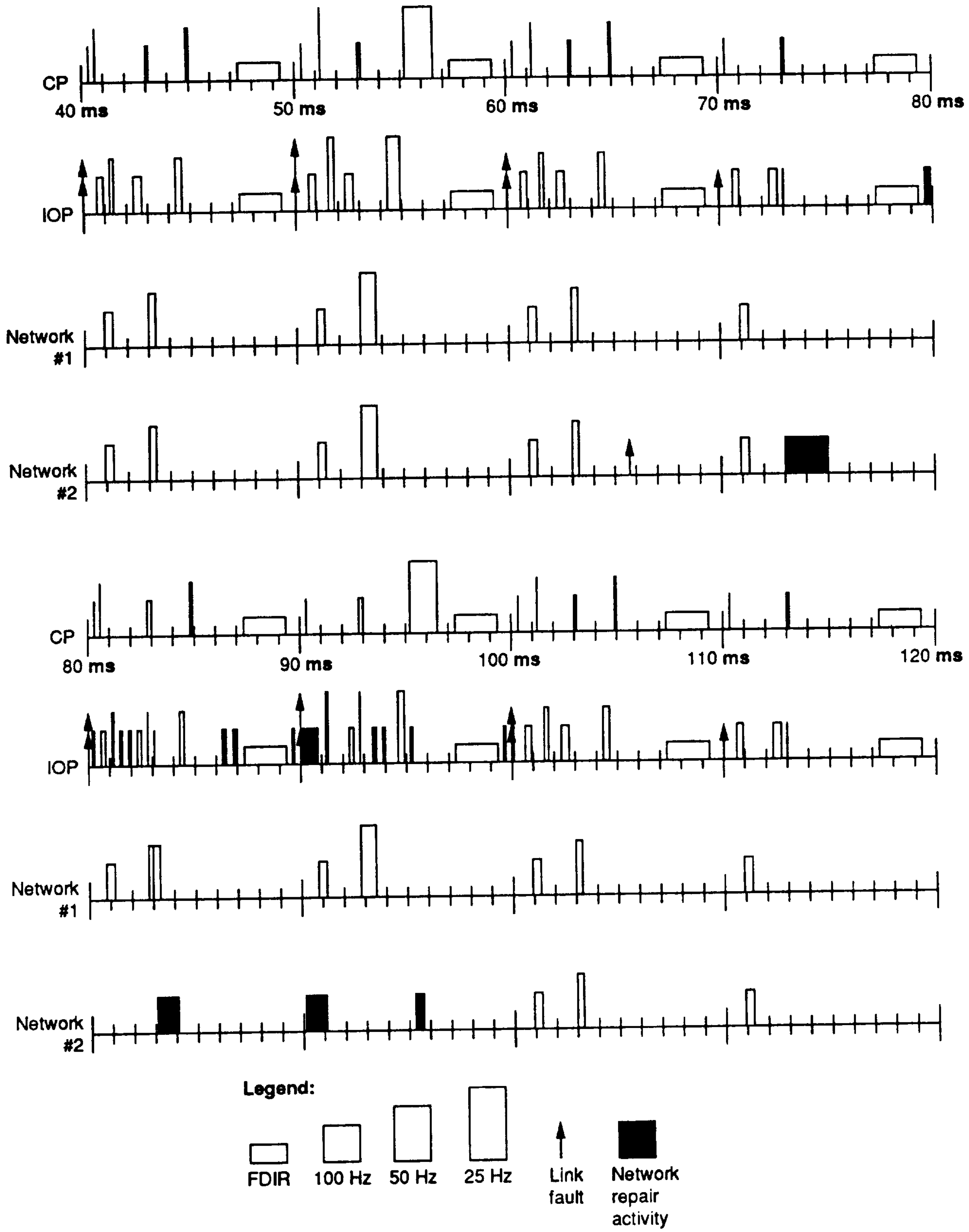


Figure 4.5.4-7. Experiment 2 Configuration 13, Link 70-71, Run 21

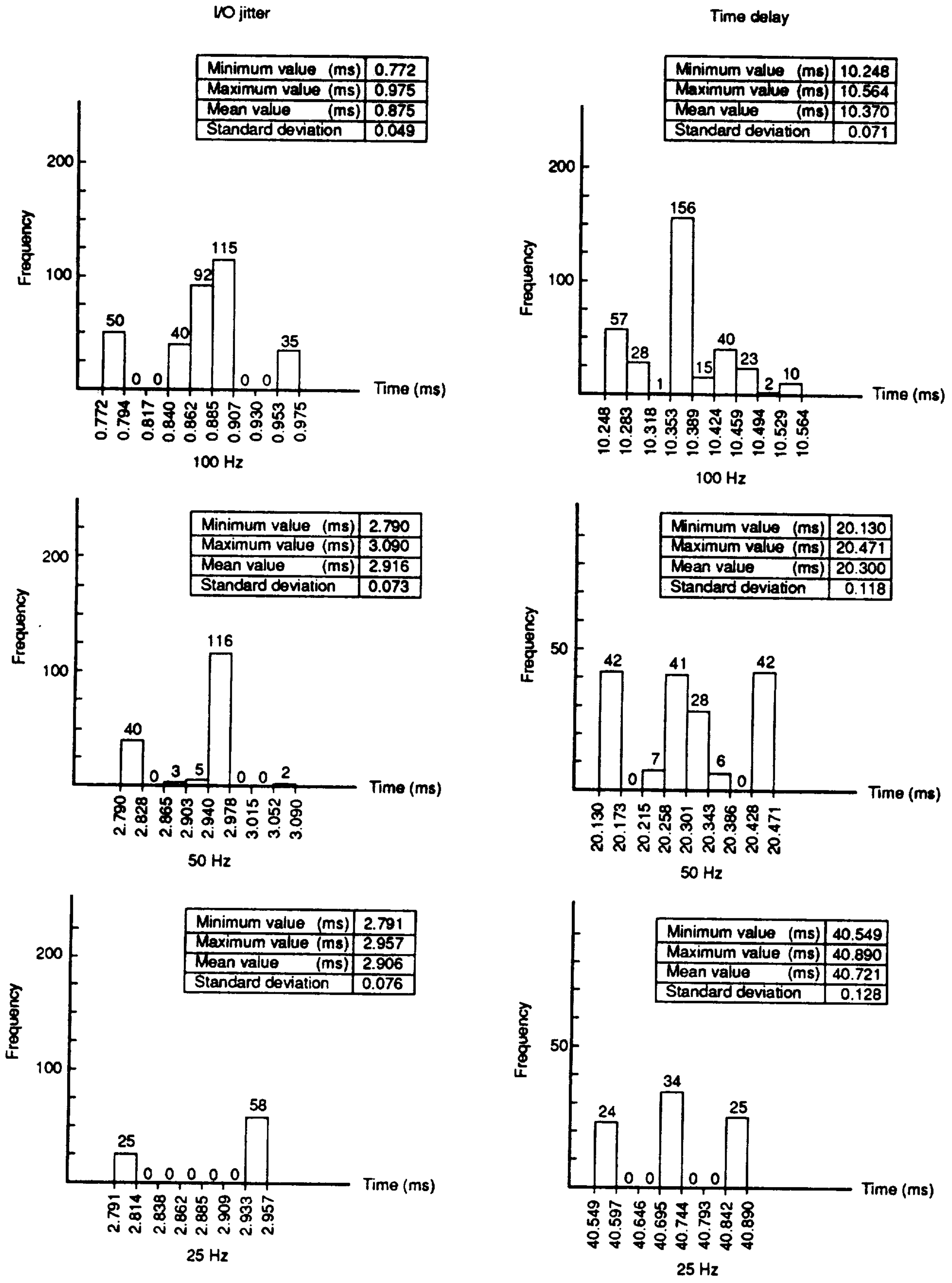


Figure 4.5.4-8. Experiment 2 Configuration 13, Link 70-71, Application Performance Parameters

orientation to inboard, causing loss of communication with the entire network. The inability to correctly identify this failure results in wasted repair actions and prolonged out-of-service times.

This experiment measures the duration of network repair activity when a full regrow strategy is invoked. This is a reasonable approximation of the repair time for active failures, because repair sequences that do not show progress must be abandoned and a more time-consuming regrow action used to guarantee a working network.

Flight Control Group (Configuration 10). The I/O network out-of-service times for this experiment are shown in figure 4.5.4-9. The significant difference between this repair strategy and the one-shot strategy is that the out-of-service times are substantially larger. The repair times for this experiment slightly exceed the 1 second value used in the reliability analysis.

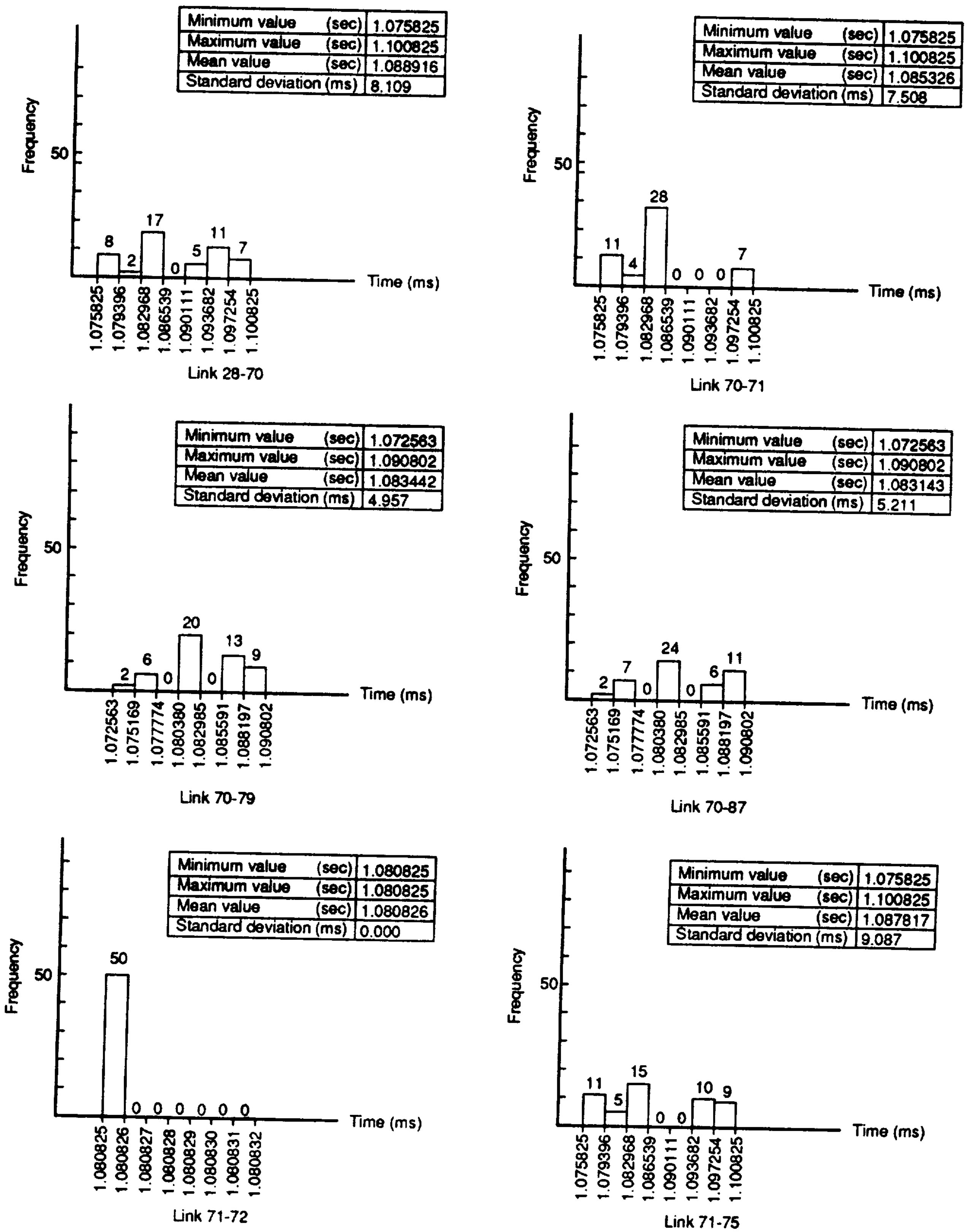
Some application performance measures are summarized in table 4.5.4-3, and the I/O jitter and time delay data are shown in figure 4.5.4-10. The shift in performance characteristics during repair is similar to those occurring in the one-shot experiment. The number of values in the bins corresponding to ongoing repair are greater because of the long duration of the full regrow repair.

Engine Control Group (Configuration 12). The out-of-service times for this experiment are shown in figure 4.5.4-11. The full regrow strategy requires significantly more time than the one-shot repair strategy for the engine control group. A summary of the application performance is presented in table 4.5.4-4. Histograms of the I/O jitter and time delay data are shown in figure 4.5.4-12.

4.6 Experiment Observations

During performance analysis a sequence of experiments is performed to evaluate normal and failed operation of a candidate system and to evaluate critical performance issues. The order of experiments is determined by the performance analyst and depends on factors such as modeling detail needed

Out of service time



Link 28-70

Minimum value (sec)	1.075825
Maximum value (sec)	1.100825
Mean value (sec)	1.088916
Standard deviation (ms)	8.109

Link 70-71

Minimum value (sec)	1.075825
Maximum value (sec)	1.100825
Mean value (sec)	1.085326
Standard deviation (ms)	7.508

Link 70-79

Minimum value (sec)	1.072563
Maximum value (sec)	1.090802
Mean value (sec)	1.083442
Standard deviation (ms)	4.957

Link 70-87

Minimum value (sec)	1.072563
Maximum value (sec)	1.090802
Mean value (sec)	1.083143
Standard deviation (ms)	5.211

Link 71-72

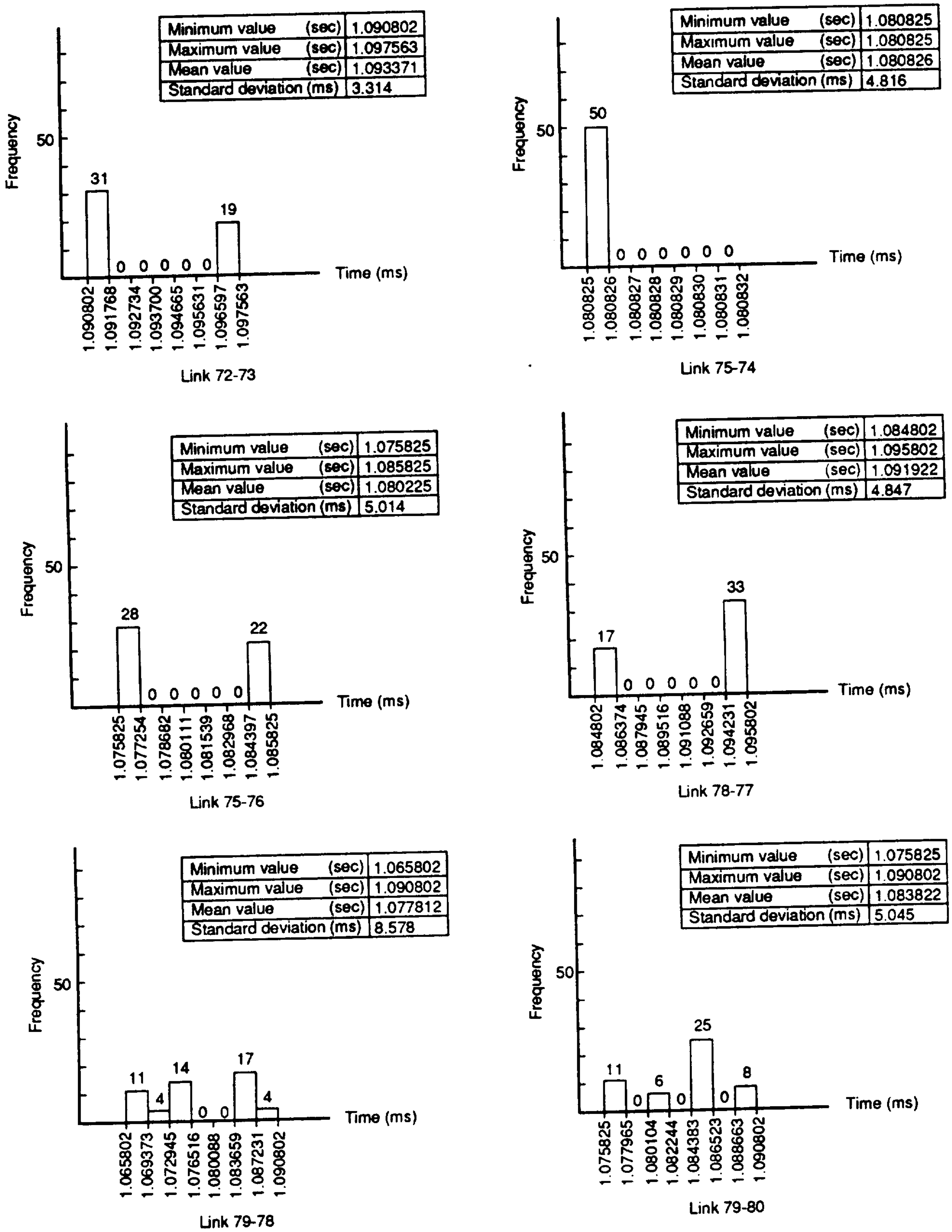
Minimum value (sec)	1.080825
Maximum value (sec)	1.080825
Mean value (sec)	1.080826
Standard deviation (ms)	0.000

Link 71-75

Minimum value (sec)	1.075825
Maximum value (sec)	1.100825
Mean value (sec)	1.087817
Standard deviation (ms)	9.087

Figure 4.5.4-9. Out of Service Time For Regrow Repair Flight Control Computer (Sheet 1 of 3)

Out of service time



Minimum value (sec)	1.090802
Maximum value (sec)	1.097563
Mean value (sec)	1.093371
Standard deviation (ms)	3.314

Link 72-73

Minimum value (sec)	1.080825
Maximum value (sec)	1.080825
Mean value (sec)	1.080826
Standard deviation (ms)	4.816

Link 75-74

Minimum value (sec)	1.075825
Maximum value (sec)	1.085825
Mean value (sec)	1.080225
Standard deviation (ms)	5.014

Link 75-76

Minimum value (sec)	1.084802
Maximum value (sec)	1.095802
Mean value (sec)	1.091922
Standard deviation (ms)	4.847

Link 78-77

Minimum value (sec)	1.065802
Maximum value (sec)	1.090802
Mean value (sec)	1.077812
Standard deviation (ms)	8.578

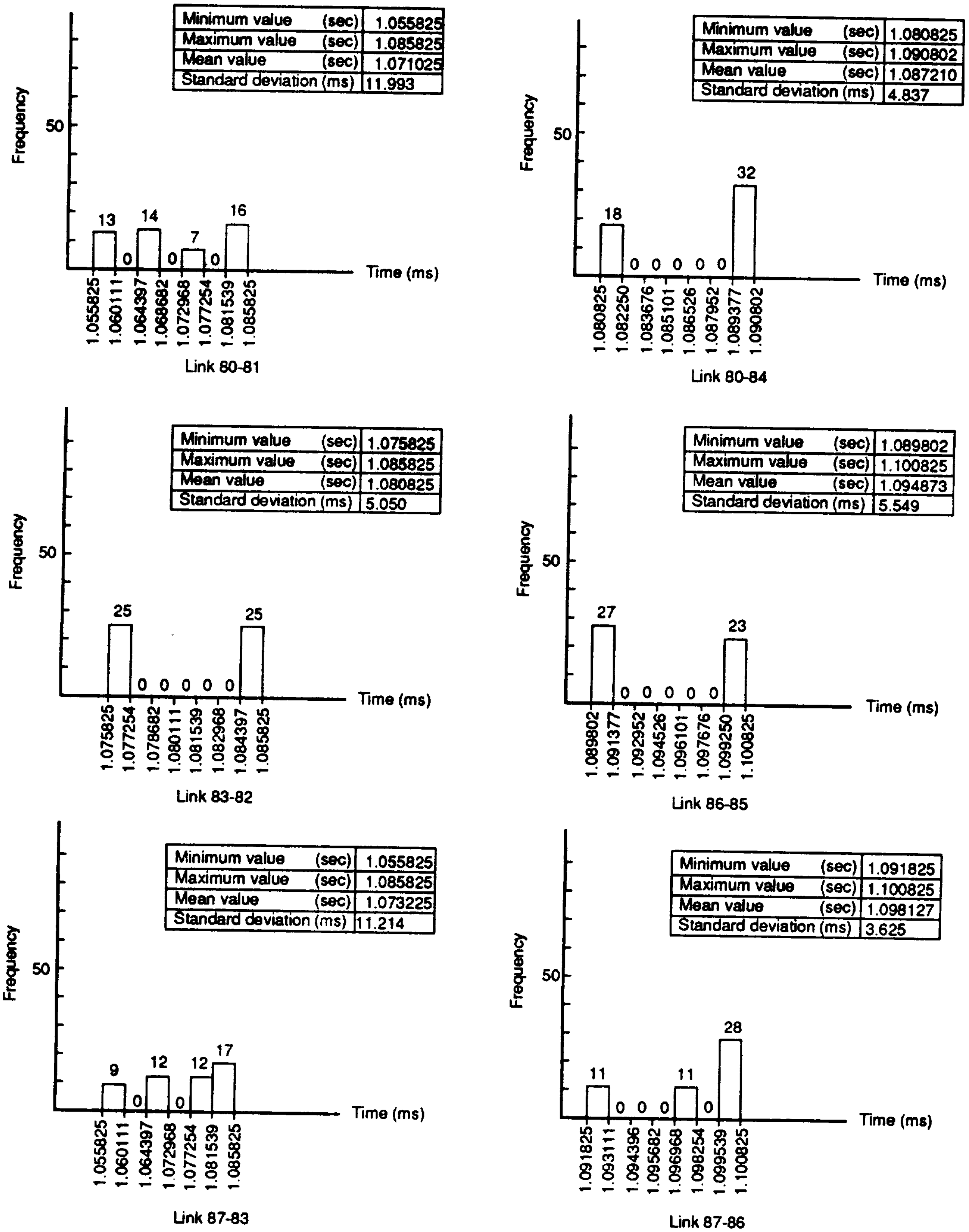
Link 79-78

Minimum value (sec)	1.075825
Maximum value (sec)	1.090802
Mean value (sec)	1.083822
Standard deviation (ms)	5.045

Link 79-80

Figure 4.5.4-9. Out of Service Time For Regrow Repair Flight Control Computer (Sheet 2 of 3)

Out of service time



Minimum value (sec)	1.055825
Maximum value (sec)	1.085825
Mean value (sec)	1.071025
Standard deviation (ms)	11.993

Minimum value (sec)	1.080825
Maximum value (sec)	1.090802
Mean value (sec)	1.087210
Standard deviation (ms)	4.837

Minimum value (sec)	1.075825
Maximum value (sec)	1.085825
Mean value (sec)	1.080825
Standard deviation (ms)	5.050

Minimum value (sec)	1.089802
Maximum value (sec)	1.100825
Mean value (sec)	1.094873
Standard deviation (ms)	5.549

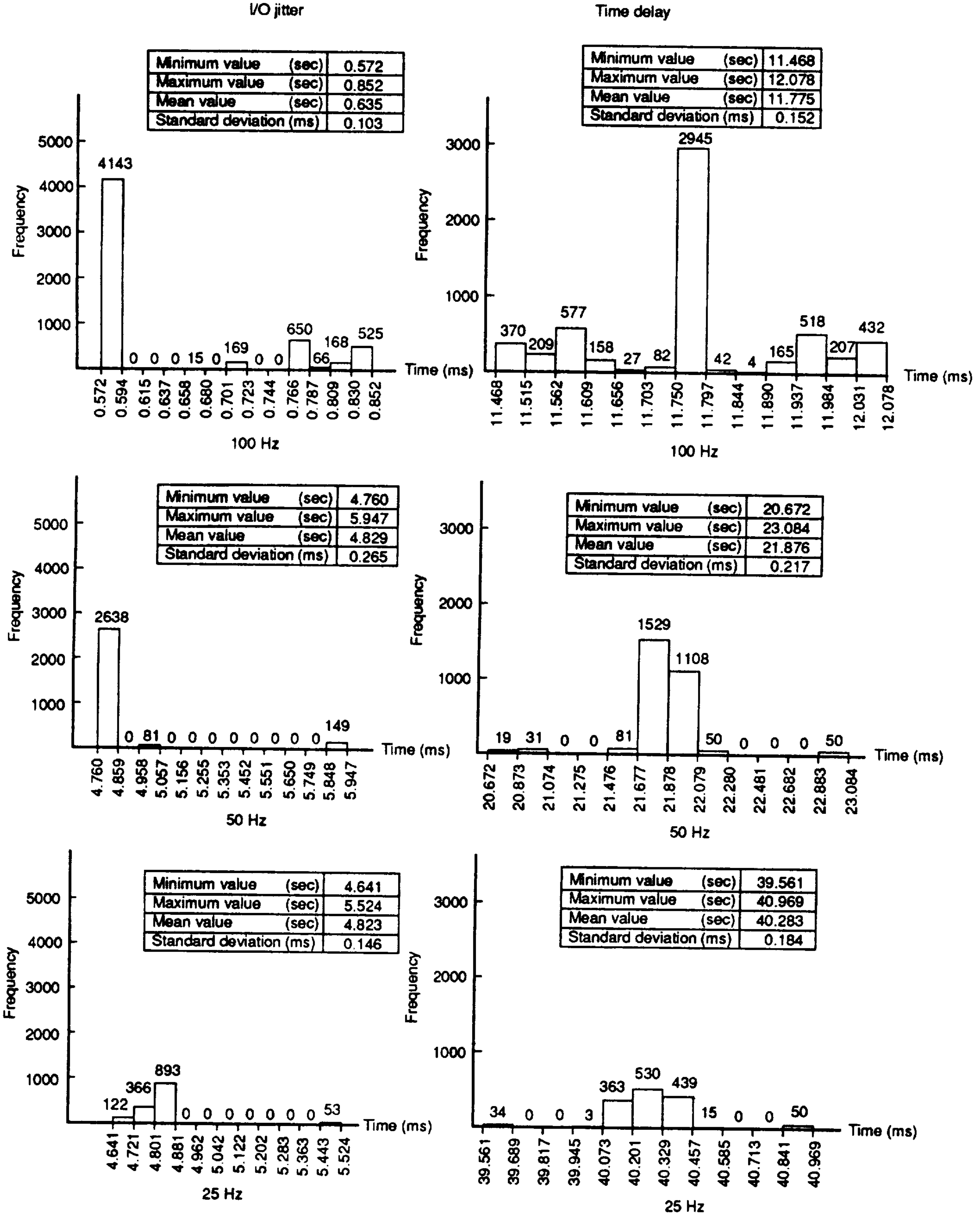
Minimum value (sec)	1.055825
Maximum value (sec)	1.085825
Mean value (sec)	1.073225
Standard deviation (ms)	11.214

Minimum value (sec)	1.091825
Maximum value (sec)	1.100825
Mean value (sec)	1.098127
Standard deviation (ms)	3.625

Figure 4.5.4-9. Out of Service Time for Regrow Repair Flight Control Computer (Sheet 3 of 3)

Table 4.5.4-3. Experiment 2 Configuration 10 Summary

Failed link	100 Hz minimum deadline margin(ms)	50 Hz minimum deadline margin(ms)	25 Hz minimum deadline margin(ms)	IOP utilization	I/O system utilization
28-70	2.995	2.582	11.124	79%	68%
70-71	2.691	2.703	10.914	78%	68%
70-79	2.893	2.584	11.206	78%	68%
70-87	2.928	2.560	11.257	78%	68%
71-72	2.905	2.603	11.328	78%	68%
71-75	2.742	2.723	10.887	78%	68%
79-78	2.965	2.342	10.964	78%	68%
79-80	2.868	2.667	11.291	78%	68%
87-83	2.928	2.597	11.571	79%	68%
87-86	2.779	2.561	10.930	78%	68%
72-73	2.977	2.432	11.167	78%	69%
75-74	2.916	2.456	11.103	78%	69%
75-76	2.963	2.371	11.548	78%	68%
78-77	2.974	2.641	11.526	78%	68%
80-81	2.765	2.539	11.787	79%	68%
80-84	2.935	2.754	10.914	78%	68%
83-82	2.880	2.454	10.944	79%	68%
86-85	2.954	2.192	11.052	78%	68%



Minimum value (sec)	0.572
Maximum value (sec)	0.852
Mean value (sec)	0.635
Standard deviation (ms)	0.103

Minimum value (sec)	11.468
Maximum value (sec)	12.078
Mean value (sec)	11.775
Standard deviation (ms)	0.152

Minimum value (sec)	4.760
Maximum value (sec)	5.947
Mean value (sec)	4.829
Standard deviation (ms)	0.265

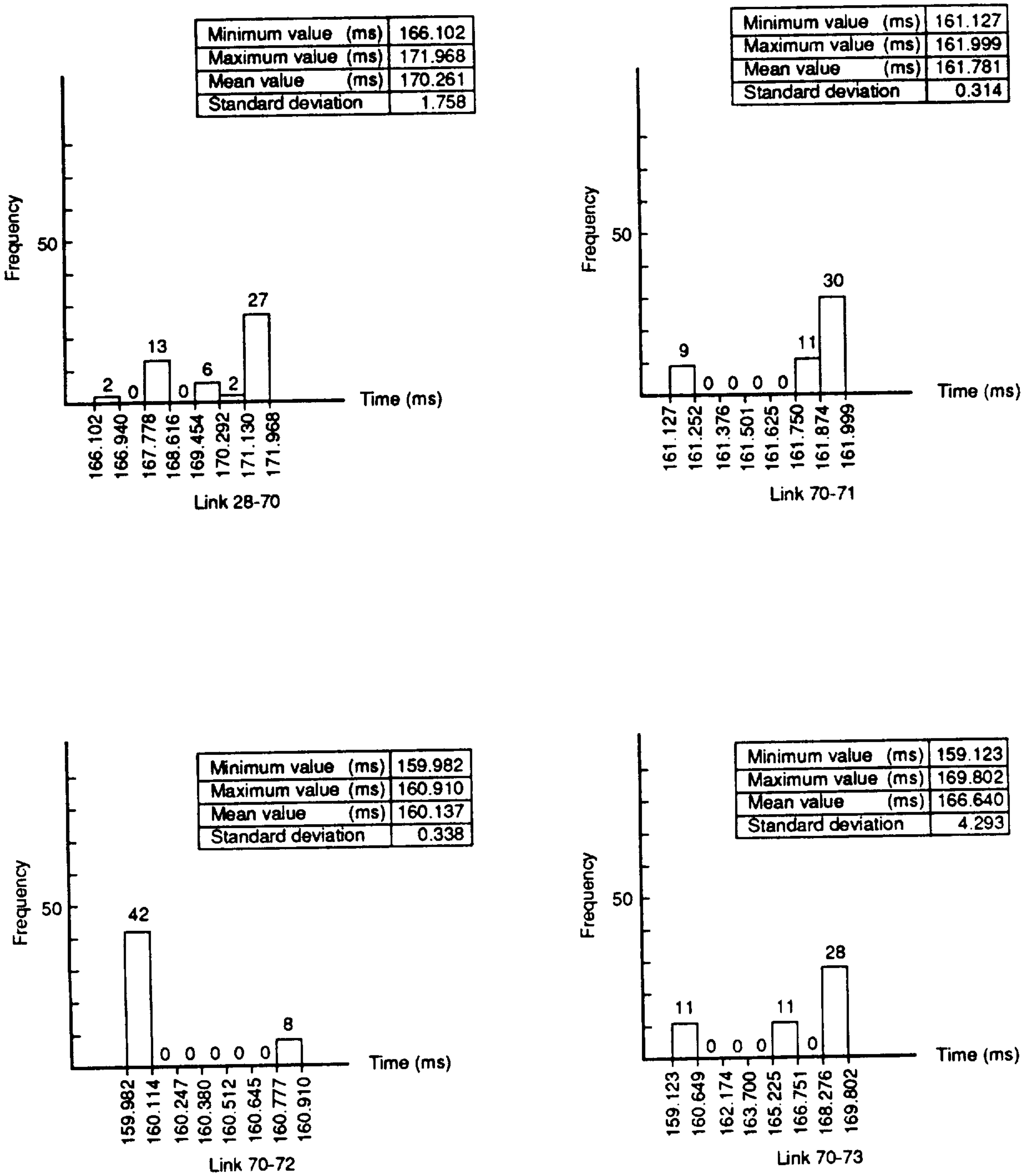
Minimum value (sec)	20.672
Maximum value (sec)	23.084
Mean value (sec)	21.876
Standard deviation (ms)	0.217

Minimum value (sec)	4.641
Maximum value (sec)	5.524
Mean value (sec)	4.823
Standard deviation (ms)	0.146

Minimum value (sec)	39.561
Maximum value (sec)	40.969
Mean value (sec)	40.283
Standard deviation (ms)	0.184

Figure 4.5.4-10. Experiment 2 Configuration 10, Application Performance Parameters

Out of service time



Link 28-70

Minimum value (ms)	166.102
Maximum value (ms)	171.968
Mean value (ms)	170.261
Standard deviation	1.758

Link 70-71

Minimum value (ms)	161.127
Maximum value (ms)	161.999
Mean value (ms)	161.781
Standard deviation	0.314

Link 70-72

Minimum value (ms)	159.982
Maximum value (ms)	160.910
Mean value (ms)	160.137
Standard deviation	0.338

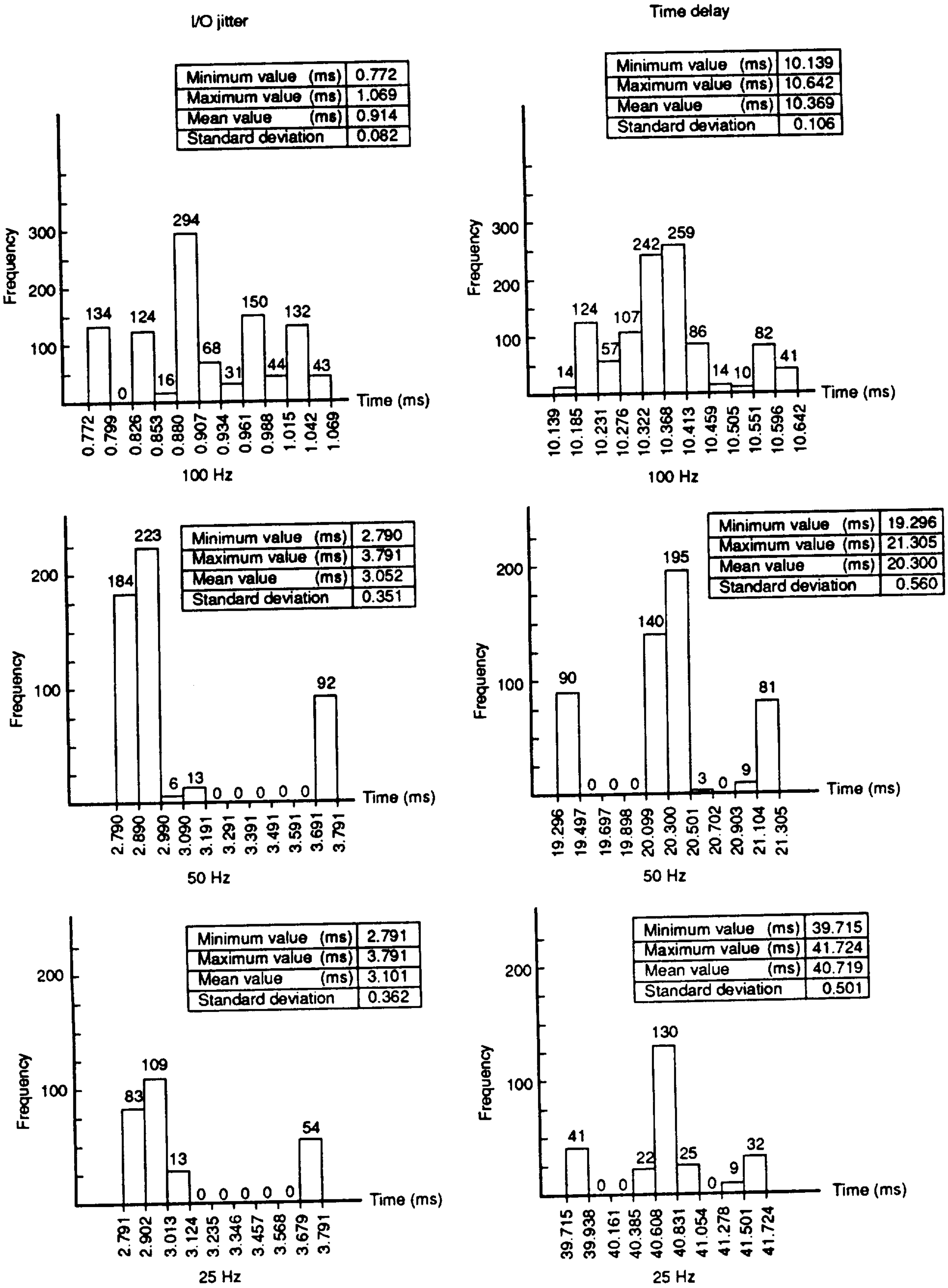
Link 70-73

Minimum value (ms)	159.123
Maximum value (ms)	169.802
Mean value (ms)	166.640
Standard deviation	4.293

Figure 4.5.4-11. Out of Service Time for Regrow Repair Engine Control Computer

Table 4.5.4-4. Experiment 2 Configuration 13 Summary

Failed link	100 Hz minimum deadline margin(ms)	50 Hz minimum deadline margin(ms)	25 Hz minimum deadline margin(ms)	IOP utilization	I/O system utilization
28-70	7.141	15.598	34.022	59%	39%
70-71	7.040	14.749	33.078	63%	38%
70-72	7.142	15.595	33.989	64%	38%
70-73	7.147	15.448	33.996	63%	39%



Minimum value (ms)	0.772
Maximum value (ms)	1.069
Mean value (ms)	0.914
Standard deviation	0.082

Minimum value (ms)	10.139
Maximum value (ms)	10.642
Mean value (ms)	10.369
Standard deviation	0.106

Minimum value (ms)	2.790
Maximum value (ms)	3.791
Mean value (ms)	3.052
Standard deviation	0.351

Minimum value (ms)	19.296
Maximum value (ms)	21.305
Mean value (ms)	20.300
Standard deviation	0.560

Minimum value (ms)	2.791
Maximum value (ms)	3.791
Mean value (ms)	3.101
Standard deviation	0.362

Minimum value (ms)	39.715
Maximum value (ms)	41.724
Mean value (ms)	40.719
Standard deviation	0.501

Figure 4.5.4-12. Experiment 2 Configuration 12, Link 70-71, Application Performance Parameters

for the experiment, how close the candidate is to satisfying the application requirements, time available to pursue alternate design options, etc. It is expected that during the evaluation of a candidate, some performance requirements will not be satisfied and refinements or a new design will be needed. Our approach in this situation was to complete the entire set of experiments and use all results before modifying the system.

It was obvious at the conclusion of experiment 4 that the candidate architecture could not meet the growth capability performance requirement. However, experiment 2 was completed before refining the candidate design to obtain insight into the candidate's operating characteristics under failure.

As discussed earlier, one characteristic of the io service model developed during the performance modeling, is that lower priority I/O data can be preloaded when higher priority I/O activity is being executed by the IOS. It is not clear whether the overall effect was beneficial. Additional complexity in the I/O system service software is required to support this capability. It was thought that preloading might improve application performance for heavily loaded configurations such as the flight control computer.

However, preloading did not occur during normal operation in the selected flight control configuration. It happened only when a network was taken out of service for repair. As a result, the deadline margin for the 50 Hz rate was substantially reduced for one cycle because preloading occurred in the new idle time slot before FDIR execution. Here the preloading effect was negative. Since the interactions can be subtle, there is a definite benefit to a detailed simulation when considering sequencing and control alternatives in a candidate architecture.

Preloading was used during normal operation of the selected engine control configuration. However, it was "lightly" loaded and the performance benefit was minimal.

The modeled priority scheme for the IO service processing in the IOP may inhibit better application performance in some configurations. There are several configurations in experiment 4 in which lower priority output data was preloaded before execution of a higher priority I/O completion poll. Since the preloaded I/O request is lower priority, the high-priority I/O data are unloaded before starting the preloaded I/O request. This occurred in the engine control group at the 100 Hz I/O completion poll, with a preloaded 50 Hz I/O request. Since the time to unload the 100 Hz data was approximately 1.4 milliseconds, the 50 Hz I/O execution was delayed by this amount.

An alternative policy would be to start the preloaded I/O activity and then begin to unload the higher priority I/O data. This would delay the unloading of the 100 Hz data by one task process switch and the time needed to start IOS execution, but would begin the execution of the 50 Hz I/O nearly 2 milliseconds earlier. This rule allows additional concurrency and would significantly improve the 50 Hz rate group deadline margin with minimal impact on the 100 Hz rate group. This illustrates the benefits of the performance simulation in that the direct effects and any unintended side effects on the application of operating rule changes can be determined easily. Furthermore, the interaction between the application and system features can be tuned early in the design cycle.

During normal operation, the preemptive priority scheduler proved useful by ensuring that the processor was allocated to the task with the smallest execution deadline. Additionally, during I/O network repair the preemptive priority scheduler allowed the IOP to perform network repair activity when no application activity was ready. This has the effect of coordinating the processing associated with the I/O network repair with the application-related processing on an as-needed basis.

One concern with preemptive priority schedulers that surfaced during the experiments is that they can introduce variability in the execution of some IOP processes whose effects propagated to other parts of the system. This occurred during repair in both the flight control computer and the engine control computer. The variability arose when two processes became ready to execute at nearly the same time and caused a double context switch. The variations did not cause any serious operational anomalies in the application in the selected flight control configuration or the engine control configuration. However, it did make the difference between meeting deadlines and failing in one of the marginally overloaded flight control configurations. This points out the need to thoroughly evaluate the interaction of the stochastically varying workload and the specific scheduler rules in all unusual system operating situations (failure recovery, temporary overload etc.) to ensure no unexpected side effects.

The results presented in this report present an optimistic picture of the candidate architecture performance. The system was assumed to operate near certain hardware limits. When more realistic values for the system overhead functions are available from proof-of-concept testing, the performance measures will probably suffer. Furthermore, some key performance interactions were not modeled in the performance simulation. These include ic network operation and the operation of the shared bus in each FTP channel.

The application sends time-critical data across the IC network. One concern is the ability of the IC network to meet the time-critical end-to-end data transfer requirements of the application during normal operation. Since the IC network operates with unsolicited messages, the network must be periodically polled to determine whether any communication has been received. This IC communications process executes on the IOP, which is also responsible for the application I/O operation. Therefore, another concern is the effect of IC network communications on the application I/O activity.

For the application to complete its I/O activity and IC activity, it must transfer data from the cp to the IOP through a shared bus. The shared bus has two states, locked and unlocked. When the shared bus is locked, access to other users is blocked. A major concern is whether a lack of coordination between the system processes in the CP and IOP can lead to shared bus utilization problems. This is a potential cause of serious degradation in the application performance.

5.0 REFINED ARCHITECTURE DESIGN

The candidate system evaluation effort described in section 3 for reliability and in section 4 for performance demonstrated that the candidate was not capable of meeting the system requirements. The predicted safety and mission reliability values exceeded the system constraints. Furthermore, the predicted timing needs of the major control functions did not leave adequate growth capability. The flight control group workload strained the system capacity in both computing and I/O activity. As a result, the IAPSA II candidate system concept was refined to improve its performance and reliability.

Three approaches were taken to refine the candidate architecture to better match the system needs. The first approach was to balance the computing and I/O workload between the engine and flight control groups. The preliminary timing estimates showed that the flight control group was heavily loaded, whereas the opposite was true for the engine control group. Shifting the system workload from the flight control group to the engine control groups should reduce the growth constraint problem.

The second approach was to improve the system failure protection. The reliability results for the candidate architecture show that the dominant safety failure sequences such as temporary exhaustion of body motion sensors involved two failures. A goal for the refined configuration, then, is to maintain flight safety with all two failure sequences. In the mission reliability area, the candidate architecture suffered from too many single failure situations. Again, the goal for the refined system is to maintain full mission capability with all single failures. Steps taken to achieve these goals will be discussed later in this section.

The final approach for refining the architecture was to reduce the number of communication elements in the system. Large networks have several disadvantages when compared to small networks. The preliminary DENET simulation experiments showed how the size of the individual networks dramatically affected the time needed to regrow a network. Another negative characteristic of large networks is that the probability of network repair action increases with the number of elements. Finally, from a performance standpoint, when a fixed number of sensors and actuators are spread across fewer interface devices, the number of transactions needed to access them is reduced. Because the transaction overhead time is a big contributor to I/O activity duration, reducing the number will decrease the I/O workload. In summary, reducing the number of communication elements should improve both the performance and reliability aspects of the system concept.

The resulting refined configuration is shown in figure 5.0-1. The most significant change is the organization of system components into two major groups instead of three. This configuration is physically similar to one of the options considered for the candidate architecture in reference 1. The two engine control groups of the candidate architecture are collapsed into one. Functions are reallocated to better balance the system. Details of these and other changes are presented in sections 5.1, 5.2, and 5.3. A reliability evaluation of this architectural concept is discussed in section 5.4, and some timing estimates are made in section 5.5.

5.1 Function Allocation Changes

The application computing functions were reallocated because the performance analysis showed that the flight control site was overloaded while the engine control sites were underused. This meant splitting the functions included in the flight control group in a way that did not cause communication bottlenecks on the system networks. Another consideration was the redundancy level match between the computing sites. In a system without function migration, critical functions that must tolerate two failures to meet reliability requirements must execute on quadruple redundant sites. Therefore, in order to move any critical functions, the destination site must be quadruple redundant.

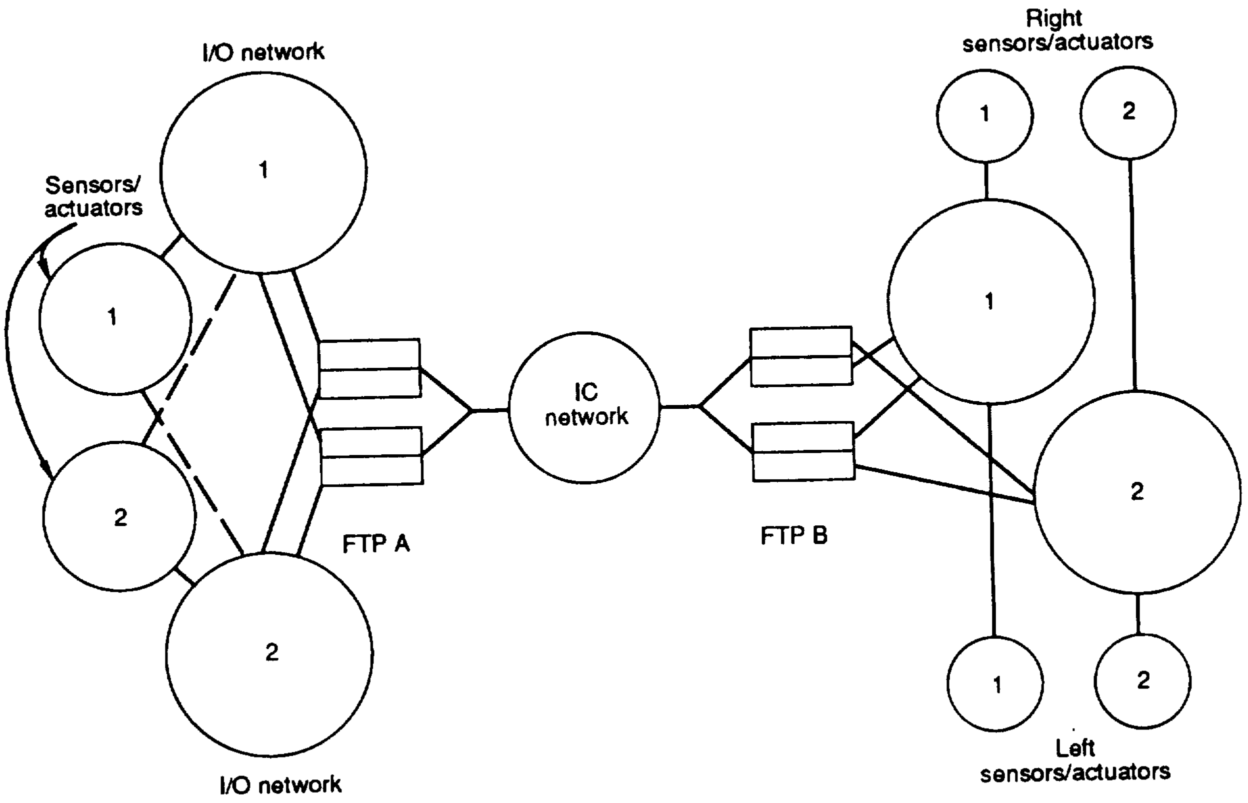


Figure 5.0-1. Refined Configuration Overview

Consideration must also be given to the awkwardness of adapting the candidate architecture configuration to a single-engine aircraft. With one engine, one site is responsible for flight-critical thrust control. Loss of that site is safety critical. Therefore, the engine control site must be quadruple redundant.

Rather than change the redundancy level of one of the two engine control sites, the refined configuration has two quadruple-redundant computing sites. This means that each site is suitable for safety-critical functions and that the function reallocation process can be relatively unconstrained. Furthermore, this new configuration will be more adaptable to a single engine vehicle.

The first step in changing the computing allocation was to combine the control for both propulsion systems in site B (fig. 5.0-1). The preliminary timing estimates indicated that this computing load could be easily handled by one site. Next the high-workload trajectory-following function was allocated to site B. This offloaded the application computing in site A. Since it has a relatively slow update rate, the relative time delay associated with any data transfer over the IC network is smaller than for the other flight control functions. Finally, the air data functions were moved to site B. The inlet control function has always required the highest update rates for air data; the associated move of the air data sensors to group B also helped to reduce the congestion on the group A I/O network. The disadvantage of the move was that it introduced some additional time delay in the air data for the manual control function. As a result of these changes, the refined system has more evenly loaded sites.

The configuration and functionality of the candidate propulsion system were reevaluated during the refined configuration effort. Some changes were made as a result of this study. The changes ranged from device nomenclature adjustments to revised ground rules for the mission capability and safety effects of propulsion subsystem failure conditions. The following subsection provides a brief overview of the differences in the refined propulsion system.

The nomenclature changes for the refined configuration reflect some mechanization changes assumed for the propulsion control functions. These changes for the most part had only a minor effect on the system configuration. The more significant changes were in the analysis of failure effects on the operational performance of the aircraft.

In the inlet, a forward ramp actuator, an aft ramp actuator, and an inlet bypass door are controlled instead of the upper ramp, lower ramp, and bypass ring assumed in the candidate system. The inlet sensors provide roughly the same measurements as in the candidate system except that the terminology now refers to forward ramp No. 3 static pressure sensor instead of duct static pressure sensor. The functionality changes for these sensors are reflected in the failure analysis results, which is summarized later.

A different actuation concept assumed for the nozzle devices leads to new nomenclature. The device that controls engine exit area is now called the nozzle area actuator instead of the converging nozzle actuator. The remaining two actuators now control thrust reversing and thrust vectoring separately instead of jointly. The new devices are the thrust reversing vane and thrust vectoring flap actuators, replacing the upper and lower nozzle actuators.

In the candidate system, the fuel flow meter was used along with fuel metering valve position in a fuel flow model. This model was assumed to allow perfect identification of all first failures and perfect detection of all second failures. The flow meter was not used in the refined configuration propulsion control concept.

The afterburner fuel control mechanization was changed for the refined configuration. The candidate system concept had a single flow metering valve that controlled fuel flow to five zones through zone flow control valves. The refined configuration had two metering valves, the afterburner core and duct metering valves controlling flow to the five afterburner fuel nozzles. Flow to the different zones was turned on and off in order by action of an afterburner segment sequence actuator.

5.2 Data Distribution

There are many possible alternatives to the candidate architecture data distribution approach. These alternatives range from minor changes in the candidate to entirely different concepts. Only two data distribution options were looked at in detail during the refinement effort. One of these incorporated a minimum change to the candidate data distribution concept, and the other replaced the mesh network with a set of buses. Even in the bus option the data distribution interface changes were minor. The candidate system data distribution problems will be discussed before these options are presented.

The first problem is that the I/O system growth capability for the flight control group is inadequate. The simplified estimates generated before the simulation experiments showed that the flight control group was slightly overloaded, while the engine groups easily satisfied the growth capability requirement. The more detailed estimates provided by the simulation experiments showed that the actual growth capability was much worse.

A timeline of the flight control I/O activity showed that there were three major time users. The biggest time requirement was due to the need to run the input and much of the output data through the data exchange hardware. The performance estimate assumed that data would be put through the data exchange at the hardware speed limit. (Our performance modeling assumes that the respective system software is extremely efficient.) Successful data exchange operation is tied intimately to fault-tolerant clock operation. It is also used to indicate whether the redundant channels are in instruction synchronism. For these and other reasons there are physical and conceptual limits to increasing speed in this area with the current FTP operating principles.

Bus activity and DIU turnaround time were the next largest time users for I/O activity. Bus activity refers to the time spent transmitting data on the bus, and DIU turnaround refers to the time used by the DIU to interpret incoming bus messages, take appropriate action, and begin the bus response message. Here again there are limits to the improvements possible. Bus speed could be increased easily by going to more advanced techniques, but a new node hardware concept may be required if virtual bus operation is to be maintained. Similarly, faster DIU response time requires either more powerful DIU hardware or a different system operational concept. Current operating assumptions are that all device activity (reading sensors or commanding actuators) takes place in response to command messages over the network from the FTP.

The time spent waiting to begin postprocessing in the IOP after the IOS has executed the chain of transactions is also a potentially large user of time. Since the chain execution occurs without IOP involvement, the mechanism it uses to detect when activity is completed can be important. The simplified timing estimate assumed that this time was zero, but a fairly coarse polling policy was modeled in the DENET simulation experiments, which showed this area to be a potential problem.

The I/O activity overload problem was addressed by reducing the number of transactions on the flight control network. The first step, moving the air data sensors off the flight control network, has already been mentioned. The next step, consolidating the system DIUs, will have several beneficial effects. From a performance standpoint, fewer DIUs means that fewer transactions are needed to communicate with the system sensors and actuators. The remaining DIUs are connected to more devices. The amount of data transferred that is directly associated with the devices is not changed. However, the data associated with each transaction (transaction overhead) is reduced. Since this transaction overhead data must go through the data exchange and be transmitted over the bus, a reduction helps in two of the big time-user categories.

There are reasons other than performance for reducing the number of communication elements in the system. One disadvantage of large networks is that there are more elements to fail. A system with many elements will have to undergo repair much more often than a system with only a few elements. Another disadvantage pointed out by the DENET simulation experiments is that larger networks have exponentially longer regrow repair times than smaller networks. The sheer size of the larger network can add to repair computation complexity and time. For these reasons, reducing network size provides a benefit in reducing both repair frequency and repair duration.

The candidate architecture uses two I/O networks per group, with the redundant elements divided between them. Having two networks allows the application to continue operating while one of the networks is being repaired. An interesting observation is that as the number of I/O networks increases the compelling need for reconfiguration decreases.

If all the system devices are located on a single network, communications must be reestablished very rapidly after network failures with at least a critical set of the devices. The time allowed to restore communication to this critical subset is limited by how long the application can be safely interrupted. For these types of systems, the network reconfiguration action naturally takes priority over the application computing. If the network repair can't be accomplished in the allowable safe interruption time, then a single network is unacceptable. In this situation a single failure can cause loss of safety.

A system with two networks is different in that a critical subset of devices is unaffected by the network loss, and therefore the repair time deadline is much less constrained. In the candidate system the constraint is set by the likelihood of nearly coincident faults that jeopardize safety. The repair action and continued application computing priority situation is reversed. Because network repair does not have to be completed to communicate with a critical subset of the sensors and actuators, it must have lower priority than the application function required for safe flight. It is allowable for network repair to take longer than the safe function interruption time. The reversal of priority means that the repair activity is accomplished in the application idle time, which causes its completion to be delayed.

As the number of networks increases, further dividing the redundant system devices, there is eventually no need for inflight repair. The aircraft can suffer the loss of an entire set of redundant devices and still meet system reliability requirements. For this reason only two options for data distribution were considered in the refined configuration study. These two options are shown in figure 5.2-1. The natural redundancy in the system tends to separate the sensors and actuators into four groups. Reliability concerns will keep redundant electronics elements in separate enclosures. Figure 5.2-1 shows how two sets of quadruple-redundant enclosures would be connected with the mesh network and the linear bus data distribution alternatives. The assignment of devices to enclosures is presented in tables 5.2-1 and 5.2-2. In the mesh network the devices are connected via two networks, whereas in the bus option they are connected by four linear buses. The key difference is that the mesh network has spare elements to allow reconfiguration, whereas the bus does not. These two options are discussed in more detail in the following sections.

5.2.1 I/O Network Option

The mesh network data distribution option is very similar to the candidate architecture. The key changes are the consolidation of network nodes and DIUs and the reallocation of system devices. There are still two networks per major group. Figure 5.2.1-1 shows the layout of one of the group A I/O

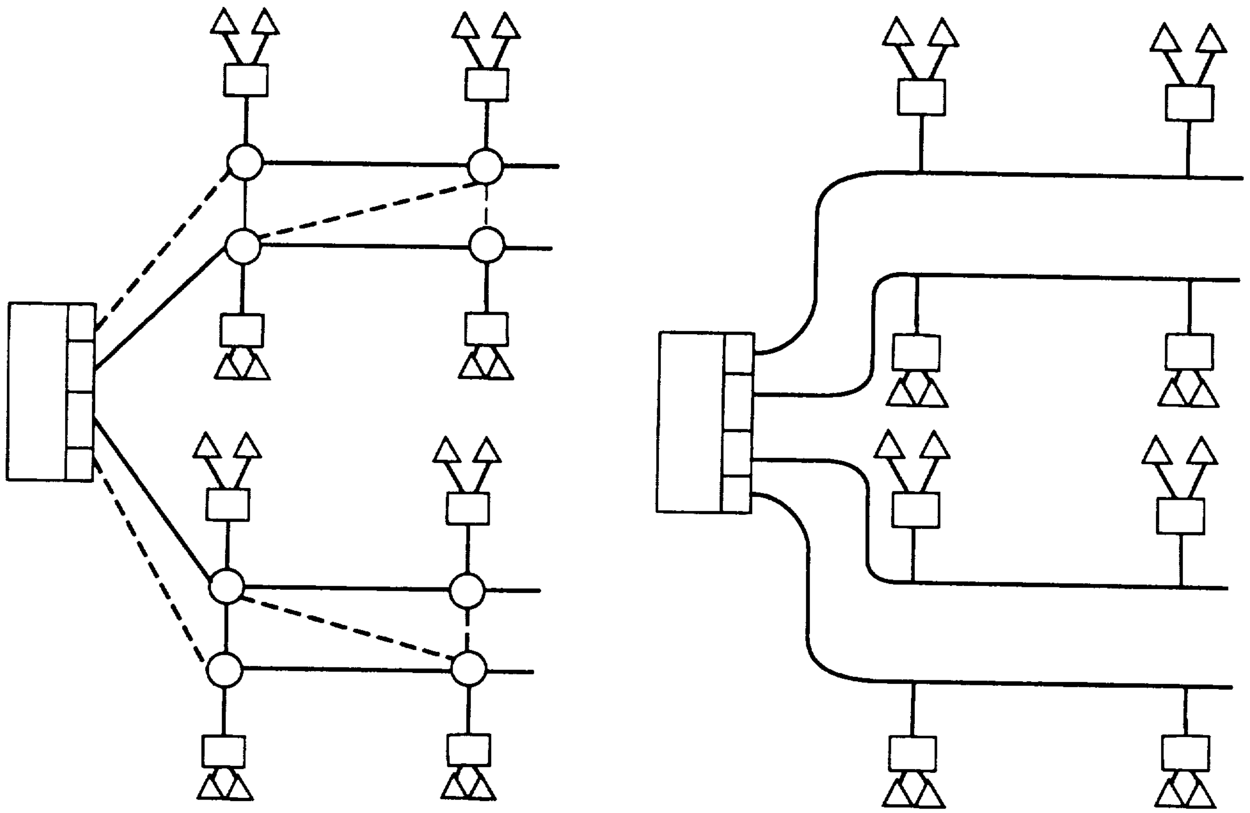


Figure 5.2-1. Mesh Network and Linear Bus Options

Table 5.2-1. Sensor/Actuator Computer Connection—Group A

Device	Node/DIU assignments																			
	Forward				Mid				Right wing				Left wing				Tail			
	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
Body accelerometers					2	2	2	2												
Body gyros					2	2	2	2												
Pitch stick	1	1	1	1																
Roll stick	1	1	1	1																
Rudder pedal	1	1	1	1																
Flap lever	1	1	1																	
Pitch trim	1		1	1																
Roll trim	1	1		1																
Yaw trim		1	1	1																
Left canard					1		1													
Right canard						1		1												
Nosewheel					1		1													
Leading edge						1		1												
L outboard flaperon													1		1					
R outboard flaperon									1		1									
L inboard flaperon														1		1				
R inboard flaperon										1		1								
L TE flap														1		1				
R TE flap									1		1									
L rudder																	1			1
R rudder																		1	1	
L outboard wing accel													1	1	1					
R outboard wing accel									1	1		1								
L midwing accel														1	1	1				
R midwing accel										1	1	1								
L inboard wing accel													1	1		1				
R inboard wing accel									1		1	1								
FTP channels (group A)	1	1	1	1																

Table 5.2-2. Sensor/Actuator Computer Connection—Group B

Device	Node/DIU assignments															
	Air				Inlet				Engine				Nozzle			
	1	2	3	4	L1	R1	L2	R2	L1	R1	L2	R2	L1	R1	L2	R2
Angle of attack	1	1	1	1												
Angle of sideslip	1	1	1	1												
Static pressure	1	1	1	1												
Total pressure	1	1	1	1												
Total temperature	1		1													
Left throttle	1	1														
Right throttle			1	1												
Forward ramp					1	1	1	1								
Aft ramp					1	1	1	1								
Inlet bypass door					1	1	1	1								
Forward ramp 3 static pressure					1	1	1	1								
Normal shock total pressure					1	1	1	1								
Normal shock static pressure					1	1	1	1								
Nozzle area													1	1	1	1
Thrust reversing vane													1	1	1	1
Thrust vectoring flap													1	1	1	1
Fan face static pressure									1	1	1	1				
Fan face temperature									1	1	1	1				
Fan speed									1	1	1	1				
Compressor speed									1	1	1	1				
Burner pressure									1	1	1	1				
Fan turbine inlet temperature									1	1	1	1				
Afterburner pressure									1	1	1	1				
Fan guide vane									1	1	1	1				
Compressor vane									1	1	1	1				
Fuel metering valve									1	1	1	1				
Afterburner core metering valve									1	1	1	1				
Afterburner duct metering valve									1	1	1	1				
Afterburner segment sequencer									1	1	1	1				
Afterburner light off detector									1	1	1	1				
Main fuel shutoff									1	1	1	1				

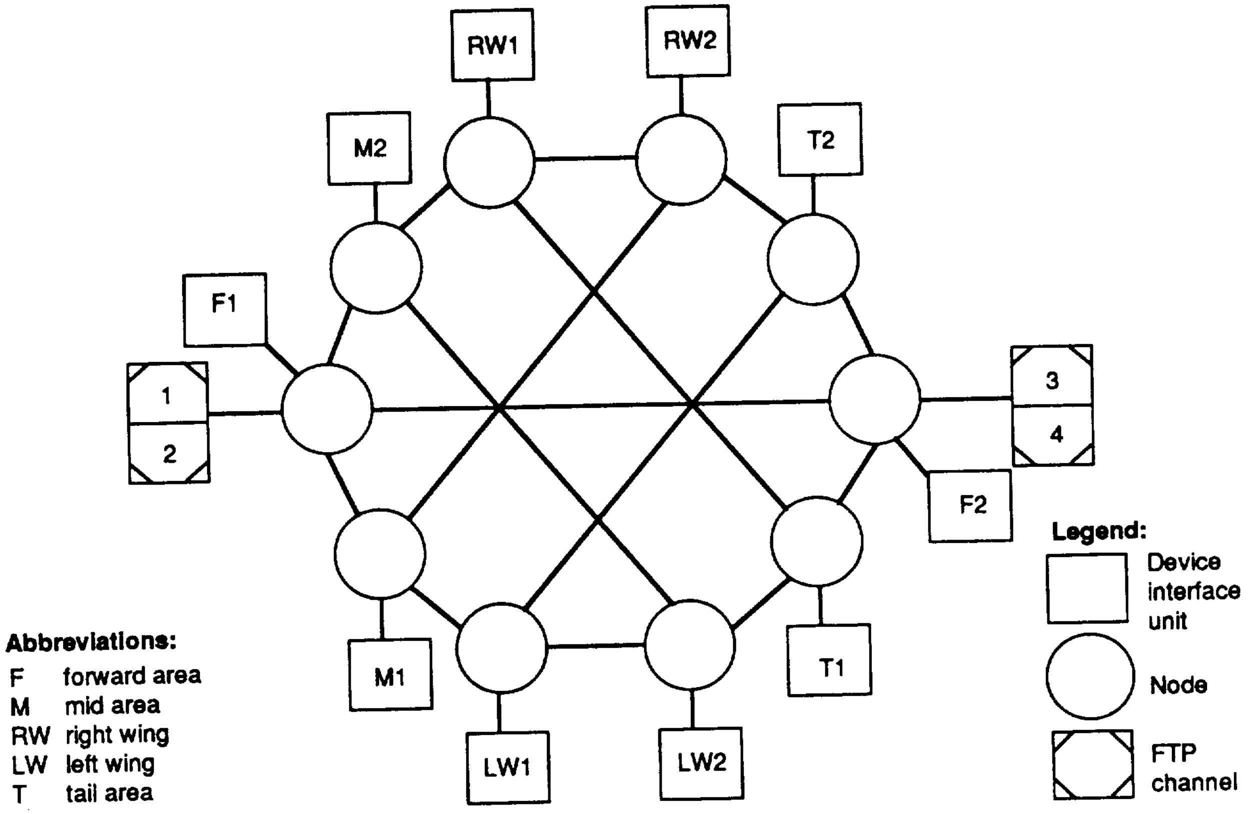


Figure 5.2.1-1. Group A I/O Network Layout

networks. Group A has two networks of 10 nodes and DIUs each. This is a reduction of eight nodes and six DIUs compared to the flight control network of the candidate architecture. The resulting allocation of devices to DIUs is shown in the reliability model section. There is one DIU per network node in the group A networks. There are no dedicated root nodes in this option; root links are connected to nodes that service DIUs.

The group B I/O network layout is shown in figure 5.2.1-2. Group B also has two networks, each containing eight nodes and DIUs. Each group B network is connected to elements on both engines. There is no reduction in the total number of nodes and DIUs compared to the candidate architecture, although they are now consolidated into two networks. Devices are allocated differently than the candidate architecture. The air data sensors and the throttle command sensors have been moved from the flight control networks to group B. Like group A, group B has no dedicated root nodes.

The number of transactions sent over the group A networks has been reduced due to the consolidation of DIUs. For comparison, the new 100 Hz rate group has six transactions per network versus the eight transactions of the candidate architecture. There are six 50 Hz transactions in the refined design compared to 10 in the candidate architecture. The 25 Hz transactions have been eliminated. However, the reallocation of devices and the placement of both engines on the same networks increases the number of transactions on the group B networks. Group B now has four 100 Hz transactions, two 50 Hz transactions and four 25 Hz transactions. In the candidate architecture each engine control rate group had a single transaction. Thus the relatively overloaded flight control networks have had a traffic reduction, while the use of the lightly loaded engine networks has increased. A performance assessment of these changes is discussed in section 5.5.

One change in the network configuration is due to the dominant reliability problem found in the candidate architecture. Recall that the layout of the candidate architecture had each device connected to only one DIU and one

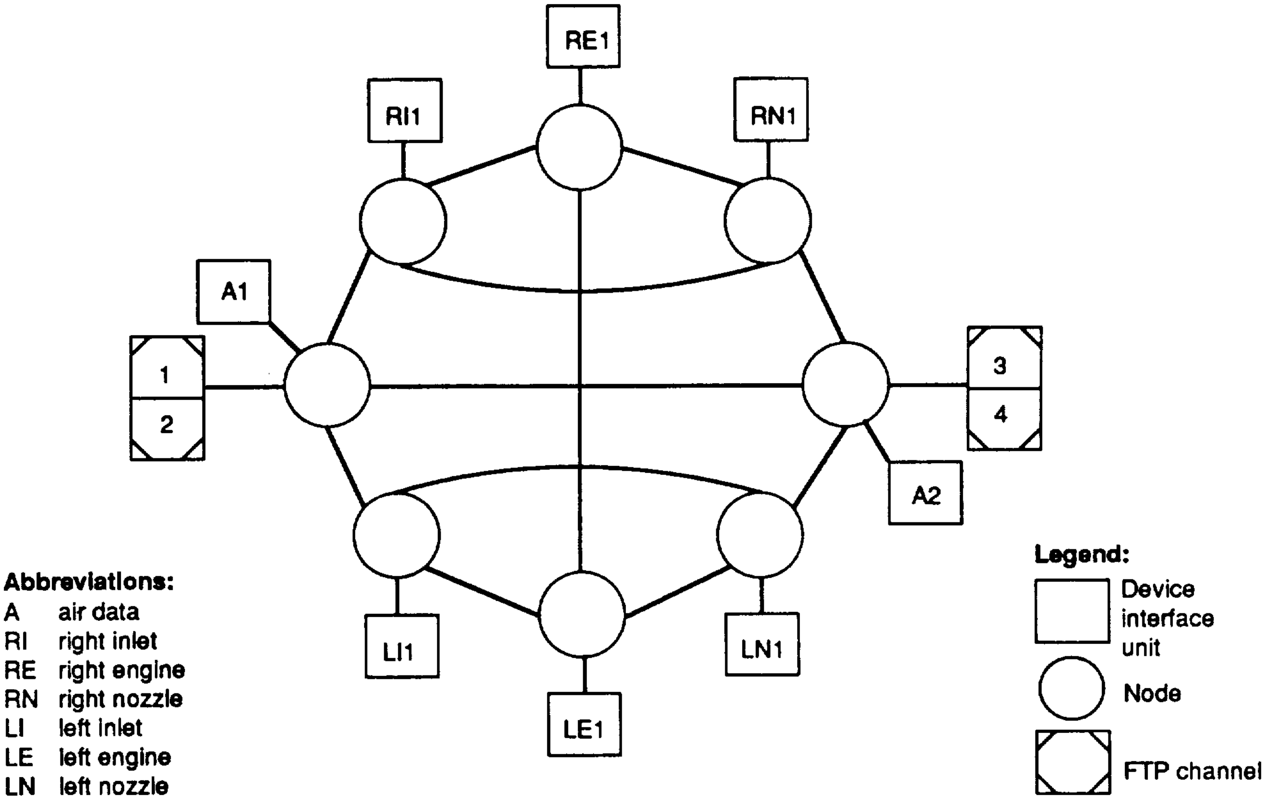


Figure 5.2.1-2. Group B I/O Network Layout

node. The reliability evaluation showed that this simple "brickwalled" scheme easily satisfied system requirements, with the exception of the body motion sensors. In the refined I/O network option, the critical body motion sensors are connected to both group A networks via dual-port DIUs. This is shown in figure 5.2.1-3. The purpose of this is to ensure that two failures are required before the system is vulnerable to temporary exhaustion failures. A critical design requirement generated by the cross-connection approach in the refined configuration is to ensure that no single failure in the dual-port DIU can cause simultaneous repair activities on both networks. Note that the cross connection is only used for the MID DIUs, where the reliability analysis showed it was required.

There are several system operation alternatives for this cross-connected configuration. The first alternative is to use the cross-network links always, transmitting the device data over both networks during normal operation. This is advantageous because it doesn't require any special operating modes to avoid temporary exhaustion vulnerability, but it causes additional transactions on networks that are already heavily loaded.

There are two operational alternatives if the cross-network links are used only when reacting to a fault. One alternative requires a fast reaction to a fault, while the other alternative can tolerate a much slower fault reaction.

The fast reaction alternative would take no action until temporary exhaustion is imminent. For example, a DIU or node fails followed by a failure, causing repair on the other network. In the fast reaction alternative no action is taken until the second failure. The situation must be recognized by the application and action taken to use the cross-connection link before a critical time period elapses. The need for special action must be discerned from the error information provided by the I/O system services. The special action would consist of (1) selecting transactions in the existing I/O request that are normally off to communicate with the devices over the cross-network links, (2) using an

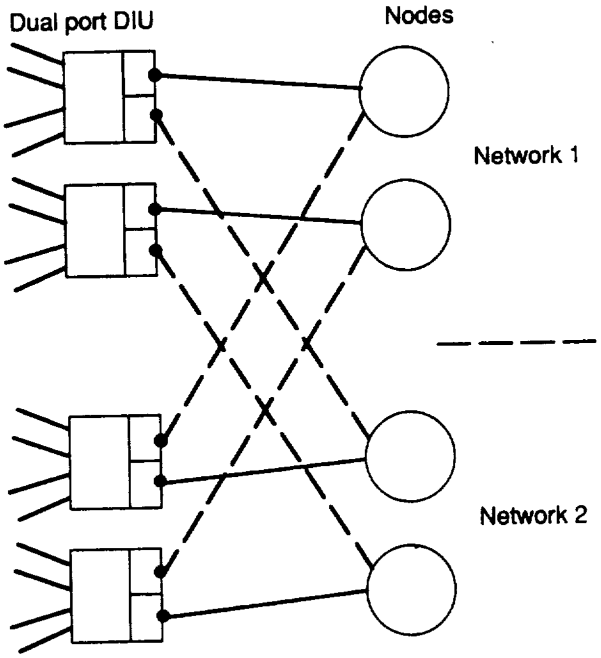


Figure 5.2.1-3. Body Motion Sensor Cross Connection

alternative I/O request for the remaining good network that contains the additional transactions for cross-network link communication, or (3) making an additional, emergency I/O request containing only the special transactions every compute cycle for the duration of the network repair. Each of these special action alternatives has certain drawbacks with the current version of I/O system services. It seems prudent, therefore, to plan an alternative with a non-time-critical fault reaction.

A slow fault reaction strategy only requires the application to recognize when the system is vulnerable to a subsequent failure causing temporary exhaustion. This means that the cross-connection links would be used after the first applicable failure instead of the second. In this case the action taken by the application is not time critical. The vulnerable situation is conveniently indicated when data from only two sensors are being sent on one of the networks due to previous failures. When the application detects this vulnerability it can change the I/O traffic loading via transaction selection to ensure that a subsequent network repair will not cause temporary exhaustion. Although this action seems straightforward, the application must react to a large number of possible situations (some of which cannot be corrected). This solution does require a substantial new application redundancy management process.

5.2.2 Redundant Bus Option

An alternative data transfer system outlined earlier consists of four non-reconfigurable linear buses. The number and arrangement of enclosures, DIUs, and devices is unchanged. All of the redundant devices are divided evenly across the four buses. Since the bus is not reconfigurable there are no network nodes in the system. Communication over the bus system is carried out just at it is over the mesh network. Recall that the mesh network operates like a virtual bus, with each device receiving all bus activity at approximately the same time. The same command/response protocol is assumed for the bus option. The data distribution interface and the IO system services are therefore assumed to be identical in this comparison.

The bus option illustrates the limiting case in which the number of networks in a system is increased to the point that reconfiguration after communication faults is not necessary. A major benefit of this step is the elimination of the complex I/O redundancy management software. Typically, validation of large, complex software processes is difficult and costly. Although this is a problem for the building block system provider to solve, the application designer must have confidence that validation is achieveable by the time the system is scheduled for first flight. Because the bus option does not include any of the network reconfiguration functionality, it sidesteps any associated validation issue.

To allow a more straightforward comparison between the mesh network and bus options, some configuration aspects were kept constant in both. Only four connections to the I/O devices were used in both options. This means that in the bus option each bus is singly connected to an FTP channel, whereas the mesh network option uses only one root link per FTP channel. Neither of these may be acceptable as a final configuration choice, but it keeps the comparison on more even terms. One implication of the bus option is that an active DIU failure can lead to the permanent loss of the entire bus and all connected devices. The normal I/O network recovery action will disable the DIU link and restore communications in such a situation on the mesh network. Special design precautions could be taken to make an active DIU failure unlikely for the bus option. Because this would result in an unsymmetric comparison, the same DIU design will be assumed for both options.

5.2.3 Electric Power Distribution

The candidate architecture analysis did not consider the details of the electrical power distribution. In the refined system study more attention was paid to the implications of the power distribution connection. The Fault Tolerant Electric Power (FTEP) system study configuration (reference 3) was used as a baseline for IAPSA II. In that study four distributed load centers (ELMC) provided electric power to the critical users. Main aircraft power buses were connected to the ELMCs, which monitored the airplane source and switch when necessary. Each load center had an uninterruptible battery bus for dc users that was tied to one of two aircraft batteries.

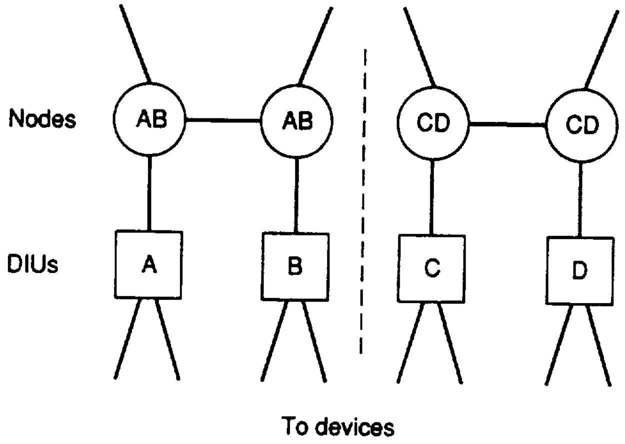
The simplest connection alternative for a system that is primarily quadruple redundant would have one ELMC source per enclosure. This alternative was broadly evaluated with satisfactory results in the candidate architecture reliability study. Each enclosure has a single local power supply that satisfies the bulk of all enclosure needs. With this single connection organization care must be taken when assigning electrical connections. All elements that have a dependency relationship (devices, DIUs, buses, FTP channels) must be connected to the same ELMC source. This guarantees that when a single source is lost only one level of redundancy for any device is affected. Otherwise, loss of a single source could bring down more than one redundant device via a dependency relationship. For example, a source could affect a DIU (and all attached devices) on one bus and a device of the same type on another bus.

The single power source alternative presents some special concerns for the mesh network option that were not evaluated in the candidate architecture analysis. When one power source fails, one fourth of the nodes fail. As a consequence, power source loss is another cause of network repair action in a singly connected system. The most general fix is regrowth of the network. In a two-network system, regrow must be restricted to a single network so that the application can continue operation with devices connected via the other network. Additionally, the power connection layout must ensure that the half of the network remaining after a power source loss is a viable configuration, with no isolated nodes.

Rather than accepting single power source losses as a cause of temporary I/O network failures, the refined design incorporated a dual power connection scheme. As a minimum, this delays any I/O network effect until the second electrical failure. The specific configuration is shown in figure 5.2.3-1. Two of the four power sources service all of the elements on a specific network. Each network node is connected to both of the network power sources. When both network power sources fail, it is unimportant that the nodes are lost since all serviced devices on that network are also lost. With this configuration, power source losses are irrelevant to I/O network operation.

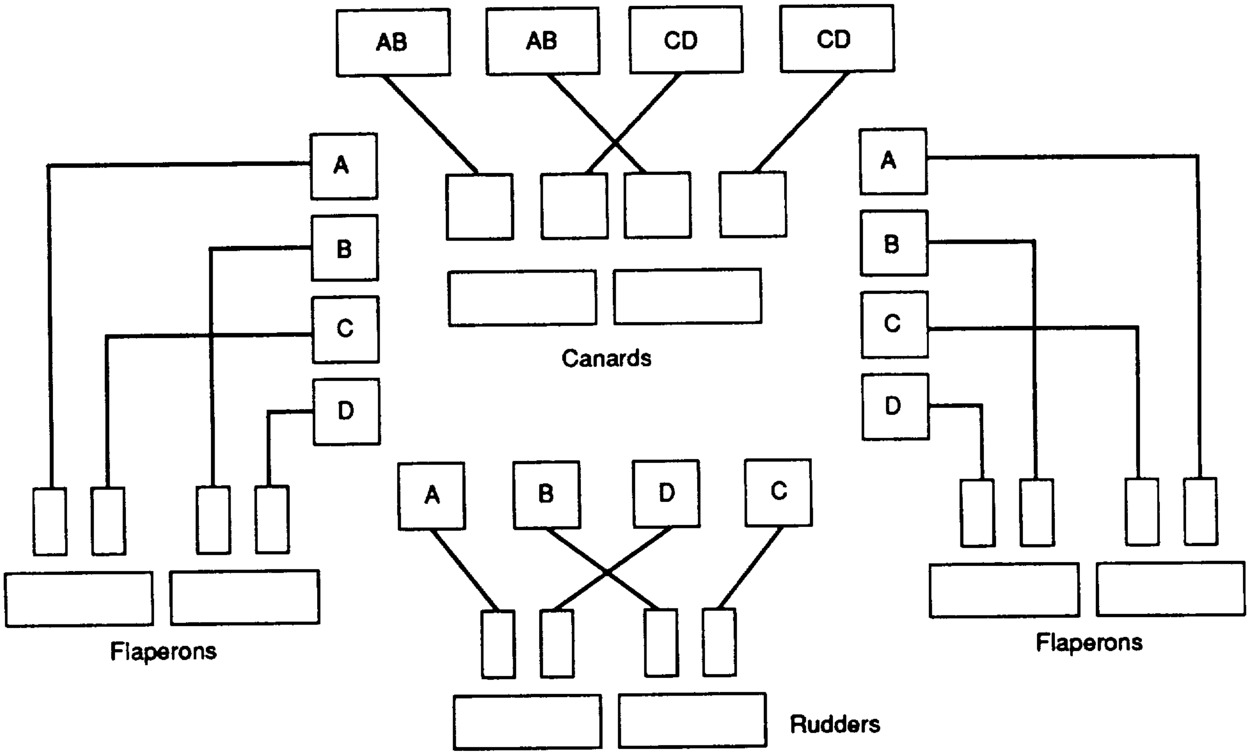
Another power connection concern is surface control. A hard requirement is that no two failures can cause degraded capability on more than one control axis. Otherwise this would be a dominant loss-of-safety situation for the refined configuration. For single power connection configurations this means that no two ELMC failures can be allowed to cause passive surfaces on more than one axis. In the baseline power configuration, each surface is driven by two channels, each of which is powered by a single source. If symmetrical flaperon pairs are connected identically, six surface types must tolerate both failure situations with only one surface-type failure. These are the right canard, right rudder, inboard flaperons, outboard flaperons, left canard and left rudder. Because there are six possible combinations of the four power sources taken two at a time, this seems feasible.

For the bus option the surface connection constraint can be met with no problems. However, in the mesh network option, the surface control constraint conflicts with the node power concept developed earlier. The node power concept basically means that all devices on one network are powered by the same two sources. Without a special fix, satisfying both constraints would mean that both actuation channels on some surfaces would be signaled over the same network. Thus, during temporary network outages for repair, that surface would switch to a passive operating mode. This would violate the network connection ground rule established during the candidate architecture layout. The solution used in the refined configuration is shown in figure 5.2.3-2. Instead of the single power source connection, the MID DIU is connected to two power sources.



Network Option

Figure 5.2.3-1. Refined Configuration—Node Power



Single Connection Alternative—Network Option

Figure 5.2.3-2. Surface Actuation Power Connection

Thus in the mesh network option there are several exceptional characteristics about the MID DIU and its devices. Its network cross connection and dual power sources are unique. This makes comparison with the bus option somewhat awkward. However, this comparison would be much more difficult if cross connection and dual power sources were used for all enclosures in the mesh network option.

5.3 Actuation Changes

One area of concern in the candidate architecture reliability study was surface actuation. The problems included two failure situations resulting in a loss of safety and single failure cases that caused loss of mission capability. Two major contributors were undetected actuation channel failures and active DIU failures.

The first contributor, undetected channel faults, was addressed by increasing the redundancy of the actuator processor and associated position sensor. The operating concept was changed to require two-processor agreement to drive the surface. The actuator channel now relies on comparison for failure detection rather than the self-test hardware and software. When the processors disagree, the channel must fail passive. This additional redundancy leads to fail-operational/fail-off capability for each surface.

Several candidate configurations were examined to define the interconnection of the actuator channels to the I/O network. Concepts where the actuation was invulnerable to temporary exhaustion and did not require dedicated cross actuator channel data paths were favored. The straightforward scheme used in the refined configuration requires that the concept be protected from active DIU faults.

The second major factor, active DIU faults, was addressed by changing the actuator communication concept. In the candidate architecture, the DIU has the responsibility for ensuring message integrity for its serviced devices. The postulated active failure mode causes the DIU to continue meeting the interface protocol while corrupting the commands sent to the actuator. The change for the refined configuration is the reassignment of responsibility for ensuring message integrity. The actuator processor now verifies the message that contains its position command. This end-to-end check guarantees that a good actuator channel will not use a corrupted command. This solution has the disadvantage of requiring additional verification processing in the actuator processor when the actuator position command is updated. In addition, the DIU must handle the transfer of a larger amount of data to each actuator interface. It must store the entire command message upon receipt and then write it to each actuator. Therefore DIU processing requirements are also increased.

5.4 Reliability Evaluation

The two data distribution options for the refined configuration, mesh network and bus, were evaluated to verify that the changes allow the system to meet its reliability requirements. The reliability measures evaluated included safe flight and landing, full mission capability, and sustained operational capability. The first two measures were used in the reliability evaluation of the candidate system described in section 4. The new sustained operational capability measure is described in the following paragraphs.

The sustained operational capability measure was used to compare the two options, emphasizing their ability to operate with failures. A key benefit of reconfigurable systems is their ability to restore operational performance after failures. The major Air Force emphasis in reliability and maintainability is operational performance over time. Maintenance aspects are very important in longer term measures, but they are beyond the scope of the IAPSA II study. Therefore a measure was needed that did not involve maintenance activity. The selected measure was based on a forward base operating scenario. The scenario involves rapid deployment to a forward base followed by immediate initiation of combat operations without any integrated control system spares. The combat scenario envisioned relaxed operating rules intended to maximize combat availability. Sustained operational capability is the probability that an aircraft can continue to fly combat missions without maintenance on any of the control system elements. Two conditions were assumed necessary for combat mission dispatch. First, the system must have full mission capability, and second, it must be able to absorb one more failure and continue to a safe landing.

The assumed operating rules are subject to discussion. In any case, this is a somewhat realistic measure of the two systems' abilities to operate with failures. The duration of this operational scenario is 50 operational hours. This reflects about a week of intensive operations. The results are the probability that a specific aircraft would be unable to maintain combat operations for the 50-hour period because of control system failures.

Some different reliability modeling techniques were used in the refined system evaluation. The first technique was explicit truncation of the models at a specified number of failures. Truncation greatly simplifies the reliability models. The technique is based on the fact that the dominant system failure sequences involve a small number of element failures. Contributions to system unreliability from sequences with a greater number of failures are less likely and therefore not significant. This allows all system states having a greater number of element failures to be modeled in an approximate manner. For our study, safety model truncation at the third failure level captured the dominant system failure sequences. The mission and sustained capability models were truncated at the second failure level.

The baseline truncation technique is shown in figure 5.4-1. The technique is based on the CSDL approach described in reference 4 used for the Computer Aided Markov Evaluator (CAME) program. The system states are

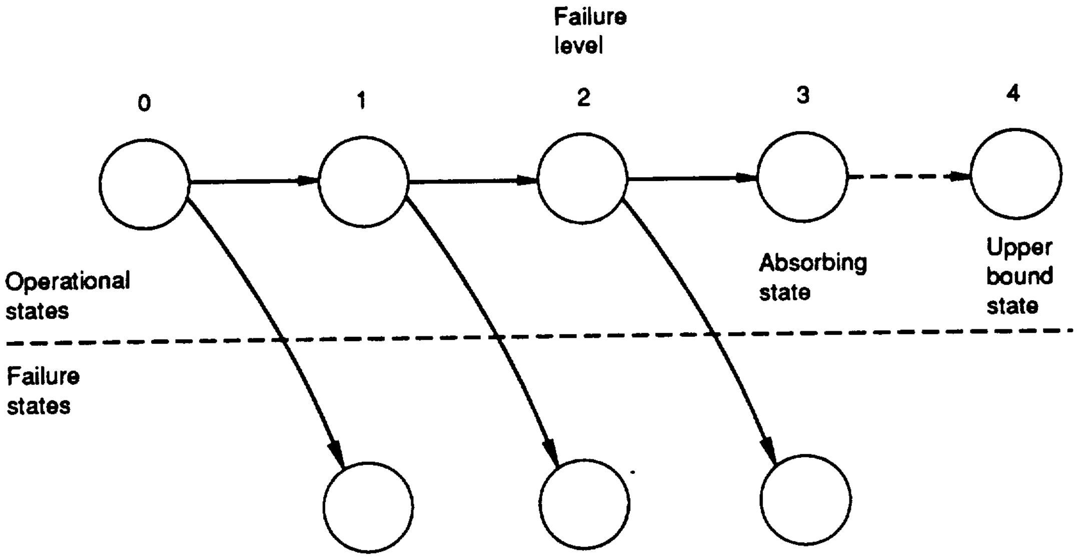


Figure 5.4-1 Safety Model Truncation

categorized by how many failures have occurred in the system and whether the system is operational or failed. The number of distinct states in each category rises dramatically with failure level. In the figure 5.4-1 example, the dominant system failure sequences involve three or fewer element failures.

Model construction is exact up to the third failure level. That is, all state information and all transition rates are exact for all sequences leading to the third failure level. The first simplification step is to aggregate all operational states that have three failures into a single state. All the state information is lost as a result of this aggregation. Next a conservative transition to the next failure level is built from the aggregated operational state. This transition is conservative because it is larger than the sum of the actual transition rates that it replaces. The destination of this conservative transition is the upper bound state, which represents all the system states that have four or more failures.

The upper bound state is conservatively treated as a system failure state even though it contains many operational states. When evaluated, the upper bound state provides a conservative estimate of the error introduced by the truncation technique. Viewed another way, an upper bound on the probability of system unreliability is provided by the sum of the probability of the system failure states and the upper bound state. A lower bound on the system unreliability is provided by the sum of the system failure states containing three or fewer failed elements.

Further simplifying techniques were used that amounted to modification of this baseline truncation technique. The justification for these techniques is that the relative likelihood of certain key system failure sequences is important to the evaluation of a system's strengths and weaknesses. Therefore it is not usually necessary to know the specific failure situation probability with more than one or two digit accuracy. The resulting simplifying technique ignores some sequences that contribute to the system's dominant failure situations when they contain more than a certain number of failures.

For example, failure of both hydraulic systems causes loss of safety. A large number of three-failure sequences involve failure of both hydraulic systems and an unrelated element. Taken together, the probability of all three-failure sequences is much smaller than the dominant two-failure sequence. Leaving out the three-failure sequences introduces only a small error in the likelihood of the specific sequence, and reduces the system lower bound for unreliability. The benefit is a much smaller model in terms of states and transitions.

Another simplifying technique was used with common element failures. Common elements are those that affect the functional success of many other system elements. Examples include communication elements, such as nodes and DIUs, and power sources such as hydraulic systems or ELMCs. From a reliability modeling standpoint the other elements are dependent on the common element. The dependency simplifying technique models only the most damaging transitions. These are the transitions that affect all the dependent elements. The transitions that are left out of the model are those that might occur after a previous failure has affected one or more of the dependent elements.

For example, if a common element affects two sensors, A and B, the most damaging transition will fail both sensors. If B has previously failed, however, the subsequent common element failure will only reduce the redundancy of the A elements. With the above simplifying techniques, the later transition would not be included in the reliability models. In this case the destination state of the unmodeled two-failure sequence has the same capability as the destination state of a much more likely single-failure sequence.

In all cases where this dependency simplifying technique was applied in the refined configuration evaluation, the unmodeled transitions would have taken the system into the aggregated operational state. In other words, none of the unmodeled transitions would have appeared in any of the dominant system failure sequences. This technique greatly reduces the number of transitions that must be modeled. However, the major advantage is the reduced effort needed to generate ASSIST statements to cover all the possible transitions. Effort is still required to derive the correct ASSIST expression for the remaining transition as a function of system state. A correct transition rate is needed to ensure that the system failure rates involving common element failures are correct.

As a final consequence of the dependency simplification, the states involving unmodeled transitions will not appear in the aggregated operational state of the baseline truncation model. Ultimately, the computed probability of the upper bound state will now be too low. It should be noted that the upper bound state is also low because of splitting the system into separate section models. The baseline truncation technique assumes that all system elements are included in the model. This is not true in our separate section approach. Other techniques are needed to estimate a usable probability for the upper bound state. For the refined configuration study, the lower reliability bound values were used to evaluate the system.

5.4.1 Critical Assumptions

The system elements in each physical enclosure were treated together for modeling purposes. This includes the local power supply and communication-related elements. A separate section model was created for each type of enclosure. The key system functions performed by the section model elements are shown in table 5.4.1-1. Three separate versions of the section models were created for this evaluation. These versions covered the safety, mission, and sustained operational capability failure conditions. Additionally, some of the section models had a version for each of the data distribution options, mesh network or bus.

The refined configuration models treated four new situations that were different than the candidate architecture. One difference in the mesh

Table 5.4.1-1. Section Modeling Allocation

Section	System function	Loss effect
Forward	Pitch command sensing	Safety
	Roll command sensing	Safety
	Yaw command sensing	Safety
	Flap command sensing	Mission
	FTP A computing	Safety
Mid	Body rate sensing	Safety
	Body acceleration sensing	Safety
	Canard actuation	Safety
	Leading edge actuation	Mission
	Nose wheel actuation	Mission
Tail	Rudder actuation	Safety
Right wing	Flaperon actuation	Safety
	Trailing edge flap actuation	Mission
	Outboard wing acceleration sensing	Mission
	Mid wing acceleration sensing	Mission
	Inboard wing acceleration sensing	Mission

network option was the likelihood of system failures involving single network operation. The refined configuration has fewer root links than the candidate system. Thus the probability of a permanent loss of an entire network is much higher than it was in the candidate architecture. Once an entire network becomes inoperative, failure of a critical sensor or a communication device on the remaining network causes a loss of safety. In the first case the two remaining sensors disagree, and in the second case no critical sensors or actuators are accessible during the subsequent network repair.

All of the elements involved in the single network operation are conveniently contained in the FORWARD model (table 5.4.1-1). The connection paths to the network depend on the FTP channels, the NI/root links, the root nodes, and the electric power sources. The failure of any one of these elements will affect one of the connections to the two networks. Certainly two-failure sequences will therefore eliminate an entire network. The probability of single-network operation is determined exclusively with the FORWARD model.

Another new modeling situation was operation of the mesh network system with MID enclosures cross connected to each network. The purpose of the cross connection is to allow the skewed sensors to be accessed from the other network to eliminate vulnerability to temporary exhaustion. It was assumed that I/O activity was reconfigured by the application immediately after failures, which made the system vulnerable to temporary exhaustion. With this kind of fault reaction in operation only a few two-failure conditions leave the system vulnerable to motion sensor temporary exhaustion. These situations occur when certain pairs of MID nodes or node and DIU combinations have failed. The MID section models handle these situations explicitly.

The mesh network option MID area enclosures are also unique because they are connected to two electrical power sources. Therefore power source failures have no effect on MID area devices until both sources for a set of

DIUs are lost. However, when this happens redundancy is dramatically reduced. For the mesh network power connection layout, this event coincides with the loss of a single network. Power source failures leading to this special situation are the only important ones for the MID devices. Therefore only electrical failures that fall in this category are modeled.

The final new situation involves changes in the surface actuator configuration and operation. The additional redundancy and interface operation changes were intended to produce fail-operational/fail-off capability for each surface. This means that the controller and communication device failures cannot affect safety until the fourth failure level except for temporary exhaustion situations. Similarly, these failures can't affect mission capability until the second failure level. The refined configuration modeling effort assumed that this fail-operational/fail-off capability was perfect and took advantage of model truncation at the third failure level to greatly simplify the safety models.

Propulsion system device criticality assumptions were different for the refined configuration. Some of the differences were due to the configuration changes between the candidate system and the refined configuration, while other differences were due to the functionality changes made during the propulsion system review effort. The following subsection outlines the major differences.

One change in functionality occurred in inlet sensing. The candidate evaluation assumed that a loss of inlet sensors reduced performance to the normal level as defined in section 3. For the refined evaluation, loss of the sensors degrades supersonic performance somewhat but supersonic operation is still possible. The assumption is that the system failure condition resulting from the degraded supersonic performance falls in the full mission capability category.

Another change was in the engine sensing area. In the candidate system, full performance capability was available with four of the five sensor types. The missing sensor value could be satisfactorily synthesized from the remaining sensor values. In the refined evaluation, the fan speed, the burner pressure, and the afterburner pressure are needed to provide normal performance capability. Compressor speed and fan turbine inlet temperature provide capability at the full performance level. The analytic redundancy method is still used for failure detection and identification among the engine core sensors.

The assumed consequences of guide vane actuation failures were different in the refined configuration. Loss of fan or compressor guide vane actuation led to loss of normal capability in the candidate evaluation. The same situation in the refined evaluation results only in a loss of full performance capability, which is less serious.

The failure assumptions for the nozzle area control actuator are changed for the refined evaluation. The candidate evaluation assumption was that failure resulted in normal capability, but in the refined evaluation failure reduced performance to the low capability level. Thus, nozzle area control is more critical in the refined evaluation.

Not all of the changes in the group B section models are due to the propulsion system reevaluation. In the refined configuration the group B FTP controls both engines. Failure of three of the four channels therefore leads to loss of safety. Failure of two of the four channels is a cause for loss of sustained operational capability. Because it accesses the safety-critical air data sensors, the group B network is vulnerable to the single network failure situation. It should be noted that in the refined evaluation the static and total air pressure sensors were explicitly modeled as mission critical instead of safety critical.

Another difference in the refined evaluation was that the propulsion devices and sensors were assumed to be invulnerable to mesh network temporary exhaustion. This means that the propulsion system control laws are mechanized so that commands can be interrupted for short periods. As a result, loss of commands to the propulsion actuators during network repair causes at most a slight hesitation in engine acceleration or deceleration. This is different from the flight control assumption, in which interruption of control commands to the unstable aircraft causes loss of safety.

5.4.2 Results

The results of the safety model evaluation are summarized in table 5.4.2-1. The loss of safety probability is dominated by group A device failures. Elements in group B have a smaller effect on safety. For this reason bus option versions were not created for several group B safety models. Nevertheless, the table shows that both refined option configurations meet the system safety requirement. Failure situations involving rare mechanical actuator jams and loss of both hydraulic systems are the largest contributors to unreliability. These results differ from those of the candidate architecture because of the absence of the special surface control failure sequences and a large reduction in the likelihood of body motion sensor temporary exhaustion. This is expected because the changes to the system are an attempt to alleviate precisely those problems.

For the mesh network option, the two largest contributors to unreliability are temporary exhaustion failure of the primary surface actuator channels and loss of FTP channels. (The FTP channel failure sequence has the same likelihood in both options). Both of these situations involve three-element failure sequences.

The new single network operation failure situation is not negligible, but doesn't appear to be a significant problem. As mentioned previously, it would be very easy to add a few root link connections to improve the network access redundancy. The main reason for four connections in the network option was to facilitate comparison with the bus option. Nevertheless, the system as defined easily satisfies the reliability requirement.

The table shows that the bus option is not vulnerable to any of the special network failure modes such as temporary exhaustion or nearly coincident sensor-network recovery. On the other hand, certain functional blocks in the bus option did have higher unreliability due to failure sequences involving permanent bus failures. These include the FORWARD area sensors and the MID area sensors.

The comparison between MID sensors in table 5.4.2-1 is affected by more than just the central bus failure dependency. The mesh network values are also better because of the dual power source connection to each DIU and the cross connection of each DIU to both networks. These special features further enhance the benefit of the mesh network option compared to the bus option in the table.

The unreconfigurable bus introduces a new central dependency aspect to the reliability model. However, note that even though the unreliability of some functional groups is worse in the bus option, the system requirement is still easily satisfied. The elements that play a role in the central bus reliability are the FTP channel, the FTP bus interface, the bus media (wires, production break connectors, etc.), and the individual DIUs. Of these elements, the bus media unreliability is insignificant. The key individual contributor to central bus failures is the FTP channel. Its unreliability overshadows the other elements. As mentioned previously, additional bus interfaces could be added to other FTP channels, just as additional network interfaces could be added in the mesh network option. This would greatly reduce the likelihood of central bus failures.

The DIUs or bus interface units (BIU) connect the devices to the central bus. Without special design features, certain single DIU failures will bring the entire bus down. An active DIU failure mode was modeled for the bus option, which causes the loss of all devices connected over that bus. Because of this hazard, practical bus designs for critical systems have special design features to make central failures very unlikely. However,

Table 5.4.2-1. Safety Model Results ($\times 10^{-7}$), 3-hr Flight

Sequence	Two network	Four bus
Exhaustion		
• Forward sensors	0.00034	0.0032
• Mid sensors	0.00040	0.029
• FTP	0.034	0.034
• Surface jam	0.24	0.24
• Hydraulic supply	0.18	0.18
• Air sensors	0.0012	*
• Engine-out throttle	0.0072	*
• Both engines	0.0016	*
• Surface pair safe	0.00014	0.00014
Nearly coincident		
• Like sensor	0.00027	0.00027
• FTP	0.000144	0.000144
• Sensor network	0.0058	–
• Dual network	0.0034	–
Temporary exhaustion		
• Forward sensors	0.00083	–
• Mid sensors	0.00012	–
• Surface controllers	0.013	–
• Air sensors	0.0012	–
Single network	0.0112	–
Total	0.501	*

* Not calculated

this study assumed no special design features in order to simplify the comparison between the mesh network and bus option. To assess the resulting hazard, a nominal value of 10 percent active DIU failures was assumed in the models. A sensitivity study showed that the table 5.4.2-1 results were not significantly affected when the active fault percentage was varied from 1 percent to 50 percent.

As mentioned earlier, group B elements have a small effect on system safety. The largest group B contributor is the failure of the FTP. This three-failure sequence results in a total loss of thrust control. An interesting failure condition is listed in the table as engine-out - throttle. This sequence is the shutdown of an engine for some reason followed by the failure of one of the two remaining throttle position sensors. The engine-out - throttle situation also results in a loss of thrust control. It should be noted that the group B network contributes to the single network failure situation. These kinds of failure conditions occur when access to an entire network is lost followed by a failure of one of the safety-critical airflow sensors.

The contribution to loss of safety due to failures in both engines was estimated by combining the results of several section models. The results of the separate section models that computed loss of normal performance capability in one engine were carefully combined to provide this figure. Correct combination of results was very difficult because much effort was required to adjust the section model results to cover only mutually exclusive situations. (Of course, the error introduced by ignoring this adjustment step turned out to be insignificant.)

The results of the full mission capability evaluation are presented in table 5.4.2-2. Unlike the loss of safety situation, the mission unreliability is dominated by the group B elements. Comparison of the mesh network and bus options shows that the network does better in mission reliability but that both systems meet the system requirements. A key assumption in this evaluation is that the mission can be continued after

Table 5.4.2-2. Mission Model Results ($\times 10^{-4}$), 1-hr Flight

Sequence	Two network	Four bus
Forward sensing	0.00022	0.00059
Mid actuation	0.0009	0.0038
Tail actuation	0.0015	0.0027
Wing actuation/sensing	0.0076	0.014
Air sensing	0.00016	0.00038
Inlet actuation	0.099	0.100
Nozzle actuation	0.099	0.100
Engine devices	0.205	0.212
Electric power supply	0.00015	–
Hydraulic power supply	0.00002	0.00002
Single network	0.0015	–
Central bus failure*	–	0.0061
Total	0.415	0.440

* Includes electric power and FTP channel

one of the two hydraulic systems fail because all actuators are dual. If loss of a single hydraulic system is a mission-abort condition, hydraulic supply failures would dominate the mission criteria.

The predominant mission failures were special single-failure situations involving the propulsion actuators. Specific causes were control valve jams and uncovered position sensor and valve drive failures. These failures prevent device control and result in the loss of full performance capability for its propulsion system.

A key assumption reflected in these results is that a mission can be continued after two safety-critical sensors or two FTP channels are lost. The system is still able to perform all functions, but, as in the hydraulic system situation, one more failure causes loss of the aircraft. If these situations are cause for mission abort, several more two-failure sequences will contribute to mission unreliability. The central effect of bus failures is more apparent in this model than in the safety model. Like the safety evaluation, the special features of the MID configuration make the network option MID sensors more reliable.

Table 5.4.2-3 summarizes the results of the sustained operational capability evaluation for the refined configuration options. The network option also has the advantage in this comparison. The dominant failure sequence, given the assumed operational rules described in section 5.4.1, is loss of a single hydraulic system. With only two systems, a second hydraulic failure causes loss of the aircraft. Therefore the aircraft requires maintenance before dispatch on another combat mission. This single failure situation masks somewhat the effects of other system failure sequences.

The ground rules for dispatch in this sustained capability model make the safety-critical sensors and FTP channels play a direct role. Unlike the mission model, failure of either two FTP channels or two safety-critical sensors is a system failure condition. It should be pointed out that in

Table 5.4.2-3. Sustained Capability Results ($\times 10^{-2}$), 50 hr

Sequence	Two network	Four bus
Forward sensing	0.019	0.053
FTP	0.184	0.184
Mid sensing/actuation	0.022	0.108
Wing sensing/actuation	0.190	0.350
Tail actuation	0.037	0.066
Air sensors	0.033	NC
Inlet actuation	0.102	NC
Nozzle actuation	0.102	NC
Engine devices	0.304	NC
Hydraulic power supply	0.450	0.450
Single network	0.015	–
Central bus failure (A only)	–	0.024
Total	1.46	NC

the sustained operational capability results the FTP channel and electric power failure combinations are listed as FTP failure sequences, whereas in the mission model they are shown as central bus failures. The central bus failure entry in the sustained capability table includes all failure sequences resulting in loss of two buses except for FTP channel and electric power source combinations.

Sensitivity Study

A limited study was performed to assess the sensitivity of the reliability evaluation results to the model parameters. The dominant failure sequences in the safety models and the mission models were examined to see how model parameters such as component failure rates, active failure fractions, uncovered failure fractions, etc., entered into the system unreliability. The limited assessment made use of the fact that the group A elements dominate the safety reliability and that the group B elements are most important to mission capability.

To judge whether a parameter was critical, its value was increased by an arbitrary multiple and the section model(s) recalculated to provide the effect on reliability. If the resulting change in reliability exceeded the system constraint the parameter was critical. The multiple assigned to the parameter was based on how well it was known. Parameters such as component total failure rate were increased by a factor of two. When the parameter was not as well known, such as an active failure fraction or percentage of uncovered failures, it was increased by an order of magnitude. Finally, for certain very unlikely failure modes, such as the probability of rare mechanical jams, the likelihood was increased by two orders of magnitude. The intent was to vary the parameters in a way that reflected the amount of uncertainty in the parameters. If variation in the model parameters causes a violation of reliability constraints, small variations in better known values are cause for the same level of concern as larger variations in less well known values.

A look at the most likely safety sequences using the above criteria showed two significant parameters. One was the fraction of surface actuation failures leading to a jammed surface. The assumed equivalent nominal failure rate for this mode corresponds to an extremely improbable event. A nominal value of one event in 10E+9 hours was used. The sensitivity study showed that this value could be increased by only a factor of 3 before the safety constraint was violated. This margin seems low for such a rarely occurring parameter.

The other critical parameter for safety was the failure rate of the hydraulic power system. There are two hydraulic power systems on the aircraft. The sensitivity study showed that the safety constraint was violated if the failure rate was increased by a factor of 2. The reduced margin on this parameter would indicate a need for close monitoring of the development of the system or perhaps alternative design provisions.

An analysis of the most likely mission failure sequences also pointed out two critical parameters. Both parameters contributed to the propulsion system single-failure situations. The first parameter was the fraction of propulsion actuator control valve failures leading to a jammed valve. Such a failure caused central redundancy management to safe the actuator. A nominal value for this failure mode was one event in 10E+6 hours. The sensitivity study showed that an increase in the value by a factor of 3.5 was enough to violate the mission capability criterion. Again this appears to be insufficient margin for a parameter that is not well known.

The second critical mission parameter was coverage of the actuator elements. A nominal coverage value of 0.99 was assumed for the position sensor and the drive elements. If this value were reduced below 0.95 the mission capability constraint criterion would be violated. Because this does not provide an order-of-magnitude margin, this parameter is therefore critical to the mission success for the refined configuration.

5.4.3 Transient Threat

The reliability evaluation described to this point deals only with the effects of permanent faults. Another concern for highly reliable systems is the effect of transient failures. A transient failure is characterized by faulty behavior of an element for some finite time followed by fault-free operation. The effect of a transient may persist after the fault disappears, for example, in an erroneous data value. For redundant systems, transients are primarily a redundancy management concern. If the effect of the transient failure were contained by the redundancy management process, the ideal action would be to wait for the transient to disappear. Unfortunately, it is not possible to tell whether a fault is transient or permanent.

The reaction of a specific redundancy management process to a transient is design peculiar. A major problem encountered when designing for transients is characterization of the threat. Because they disappear after a short time, solid operational data on transient rates and duration are scarce. A limited, parametric evaluation of the transient threat was performed during the refined configuration study. A benefit of this kind of transient study is that it shows the system designer the effectiveness of the redundancy management processes, including the effect of certain internal process parameters.

Sensor and actuator redundancy management processes must handle normal device mismatches to satisfy stringent missed alarm, false alarm, and failure transient criteria. Complex filtering processes are commonly used for this purpose. These characteristics make these processes less vulnerable to transients. On the other hand, the computing redundancy management requires the redundant channels to maintain strict instruction synchronism and bit-for-bit output agreement. For this reason computing transients were modeled in this limited evaluation of the IAPSA II system.

Since the evaluation revolves around the redundancy management behavior, some key features of the FTP FDIR process were modeled. These key features include exact voting of all redundant channel outputs every computing cycle, maintenance of instruction synchronism in all redundant channels, and periodic self-tests that ensure that each channel's memory agrees with the others. A key study assumption is that the transient affects only one of the redundant channels. If more than one computing channel is affected by the transient event, it is not possible to prevent errors from propagating to the rest of the system. The result is catastrophic system failure.

The transient fault-FDIR interaction model is shown in figure 5.4.3-1. A state diagram for the system as it responds to the transient event is shown at the top of the figure. A timeline showing when the FDIR processes take place during cyclic execution of the application is shown at the bottom of the figure. The transient event modeled in this study causes an error that does not disappear by itself. A transient event that changes a memory value corresponding to a program constant would cause this kind of behavior.

The model in figure 5.4.3-1 shows three possible transitions after the transient event occurs. The transition marked "scrub" indicates return to normal operation, which occurs if the periodic background FDIR process corrects the error before its use in the system computation. There are two possible outcomes if the error is used by the computation before this "memory scrub." One transition models the case in which the affected channel produces an output that disagrees with the other channels, while the other transition models a loss of synchronization by the affected channel. There is a big difference in the effect of these transitions on the system.

Loss of synchronization is critical for the IAPSA system because of the intense time-critical workload. The normal FTP FDIR reaction is to activate a resynchronization process. The resynchronization process brings

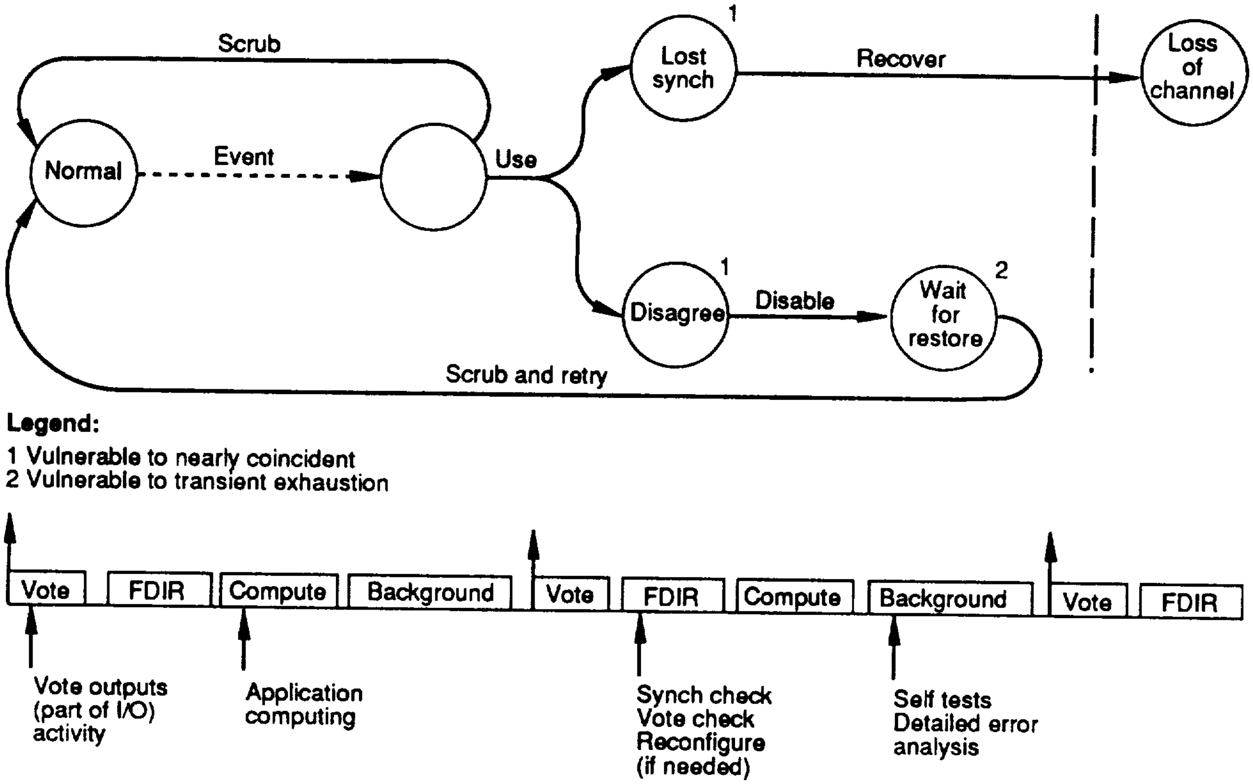


Figure 5.4.3-1. Simplified Model

the channels back into instruction synchronism and then aligns all volatile memory by voting it through the data exchange. This last step guarantees that the execution state of the channel being promoted is identical to that of the other channels.

The problem is that memory alignment must be completed before application execution can resume. Rough calculations indicate that even with very optimistic assumptions, such as data exchange operation at the hardware limit and a minimum system RAM implementation, the time required for channel restart is a major fraction of the IAPSA II minor frame. Thus it is assumed that channel restart cannot be accomplished in the available time. Loss of synchronization in a channel is just like a permanent channel fault for the rest of the flight. Resynchronization might be possible using a strategy of terminating the mission-critical functions at pilot discretion before attempting resynchronization. Sequential restart of the mission-critical functions might then be possible after resynchronization. This strategy and the resulting reliability trades between mission capability and safety were not investigated further.

The transient study therefore assumes that the recovery action for a lost synch fault is to retire the channel. Before recovery, the quad system is vulnerable to a nearly coincident fault in the same way that a channel is vulnerable while recovering from a permanent fault. System sensitivity data were generated using the percentage of faults that cause a loss of synch as a parameter. The other transient faults were assumed to cause output disagreement.

The modeled recovery sequence is different for faults that cause output disagreement. First, the "soft" fault may take longer to diagnose than loss of synch. During this period before disabling the channel, the system is vulnerable to another nearly coincident transient or permanent fault. Once the channel is disabled, the FDIR process periodically checks to see if the channel is still producing errors. If not, the channel is restored to operation. In our model the channel is restored once the periodic background process has corrected the error. When three FTP channels remain in operation and one of them is waiting for transient restoration, the system is vulnerable to a transient or permanent fault. This situation, termed transient exhaustion, results in a disagreement between the two voting channels. Since FDIR is unable to determine which channel is at fault, system loss is assumed.

The transient model for the loss of safety failure condition was evaluated using ASSIST and SURE. The model transitions were all approximated using exponential distributions. This is a fairly coarse but reasonable simplification for this sensitivity study. The nominal assumptions for the transient model are as follows. The mean time between the transient and the use of the corrupted data in the computation is 5 milliseconds. A nominal value of 50 percent of transient faults lead to a loss of synchronization. The ratio of the memory scrub period to the memory use period was 8. The mean channel recovery time was 20 milliseconds. Note that with these nominal model parameters, the probability of loss of safety due to FTP failures is significantly larger than the value shown in figure 5.4.2-1.

The sensitivity to the rate of transient faults is shown in figure 5.4.3-2. The figure shows that transients having the characteristics of our model can become the dominating failure sequence if their rate of occurrence is high. Figure 5.4.3-3 presents the sensitivity of the system to the fraction of transient faults that cause a channel to go out of synch. Failures that cause a loss of synch are naturally more critical because the channel is potentially lost for the remainder of the flight.

The effectiveness of the modeled memory scrub process is shown in figure 5.4.3-4. This process corrects the faulty data before they are used in the system computation. The results imply that the process is not very effective until its cycle rate approaches the cycle rate of the using process. This is a reasonable result, but the ability to achieve high background process cycle rates in a reasonably loaded system is highly questionable.

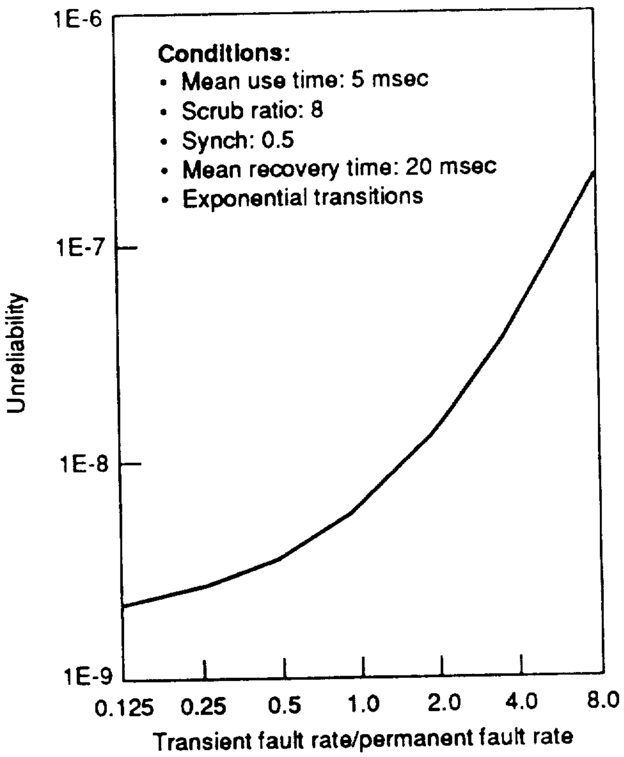


Figure 5.4.3-2. Transient Ratio Sensitivity

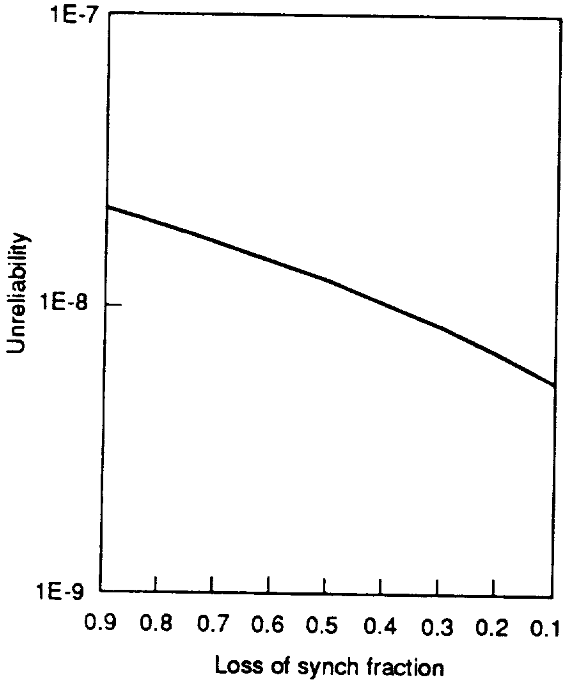


Figure 5.4.3-3. Loss of Synch Fraction Sensitivity

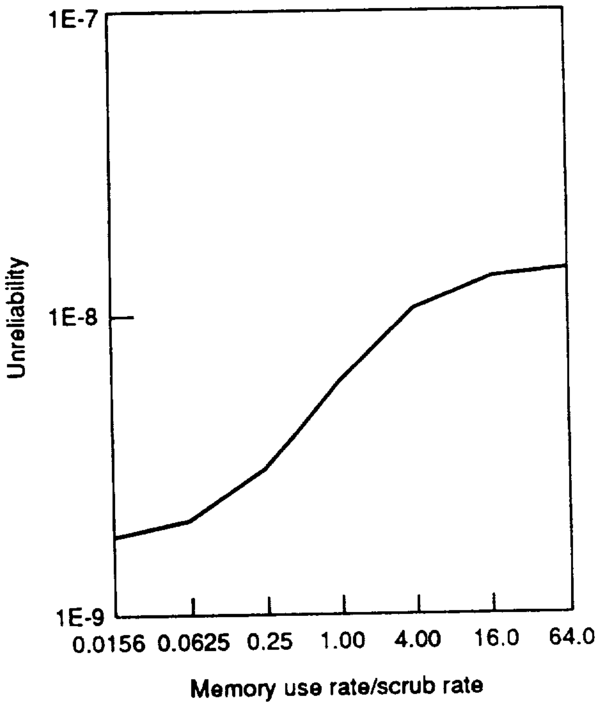


Figure 5.4.3-4. Relative Scrub Rate Sensitivity

Based on the nominal conditions assumed for the study, figure 5.4.3-5 shows that the increased likelihood of nearly coincident failures does not significantly affect overall unreliability until recovery times exceed about 1 second. (The figure shows the result for soft faults; the data for loss of synch faults are similar.) It should be noted that these results assume that the failed channel is not sending bad data on one of the two networks. The system can tolerate the latter situation for only a few application cycles.

These parametric results point out the need to characterize the transient environment and FDIR interaction in the system. Because the transient environment in flight operation is critical, this characterization will require inflight measurements on flight-type hardware. The study shows that certain key parameters dramatically affect system results. These key parameters values must eventually be measured in the system. With good environment data, more detailed modeling of the system FDIR processes could be used to more accurately define the threat. Determination or verification of the key parameters of the resulting models will be a difficult task.

5.5 Timing Prediction

In addition to the reliability analysis, a simplified performance estimate was made for the refined configuration. This estimate allows evaluation of the success of the changes made to the candidate architecture to improve growth capability. Ideally, a performance simulation such as that described in section 4.2 would be conducted for the refined configuration. However, time did not allow such an effort to be accomplished in this study.

The timing estimate is dependent on how the application computing and I/O activity is organized by the designer. Ideally, the application would be organized to keep the application functions independent. This independence

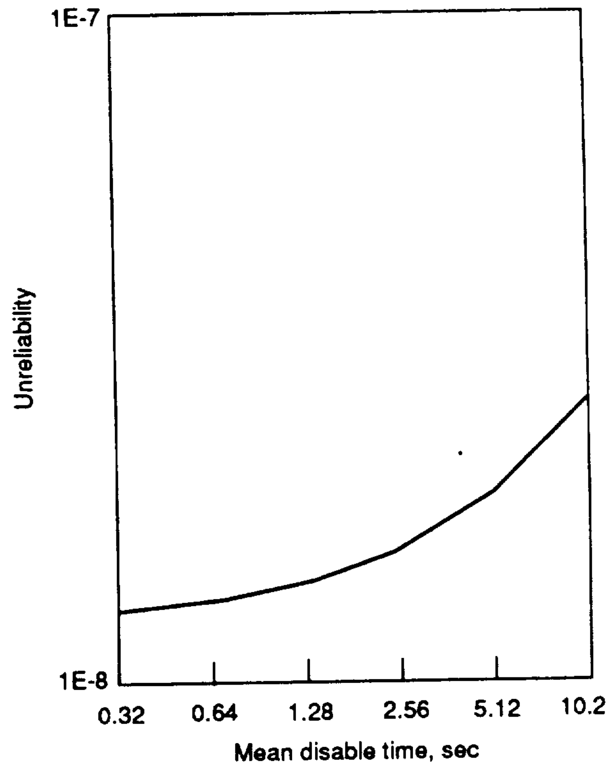


Figure 5.4.3-5. Soft Fault Disable Rate Sensitivity

eases development and verification effort, especially when the control functions are separated into less critical and more critical functions. This independence will, however, cause duplication of some of the processing when sensors or actuators are shared between functions. For this reason processes such as sensor redundancy management and computed state estimates are shared between functions. Preliminary timing estimates showed that further compromises were needed to meet the heavy loading needs of the IAPSA II application workload.

Certain ground rules were used to organize the application activity to minimize the effect of its demanding requirements on the candidate architecture. First, all the activity with a common execution rate requirement was grouped together in a rate group. This included both computing activity and I/O activity. Next the I/O activity for a single rate group was grouped into a single request containing both the input commands and output commands for a single cycle. As a result, there was an end-to-end time delay of approximately one cycle period from sensor read to actuator write. By comparison, when the I/O activity is organized into separate input and output requests, the time delay can be less than a cycle period, but the number of transactions per cycle, and hence I/O loading, nearly doubles.

A key problem was the limited I/O activity concurrency possible. Once execution of an I/O request reaches a certain point, it becomes impractical to suspend it for a higher priority request. Care must therefore be taken so that a low-priority request does not delay a higher priority request. A maximum of two I/O requests can be accomplished in each minor frame based on the preliminary estimates. For this reason, the activity in each minor frame of application execution was overtly organized so that any requested I/O activity would complete before the next request by a fast high-priority task. This was accomplished by controlling the initial phasing of the slower rate groups with respect to each other. For example, the medium rate group was started in minor frame one and repeated every other minor frame while the slow rate group started in minor frame two and repeated every fourth minor frame.

The organization ground rules used for the candidate system were used for the refined configuration. Although steps were taken to reduce the application loading, the demands on the system were still large. Key timing data for this configuration are shown in table 5.5-1. The changes in this data compared to the candidate architecture are due to fewer DIUs and the reallocation of computing and I/O activity between groups. Some minor changes were assumed in the command and response frame formats to slightly reduce bus overhead traffic and data exchange loading. On other hand, the DIU time required for sensor and actuator interface functions was increased slightly, to more realistic values. The results of these changes are shown in table 5.5-2.

Comparison of results with those generated manually for the candidate architecture shows that the changes were successful. For comparison, the candidate values were 59 percent for computing and 76 percent for I/O activity. It should be noted that these timing results are based on the same simplifying assumptions used in the candidate system estimate. Key assumptions are: (1) no chain completion delay, (2) slower rate processes can be evenly split into independent separate processes, and (3) growth capability measures how much the activity can expand uniformly before timing constraints are violated. Assumption 1 ensures that there is no time wasted between completion of the I/O activity on the network and the start of chain postprocessing in the IOP. Assumption 2 ignores the application-specific problems inherent in dividing up an application process.

Experience with the DENET simulation showed that the task switching overhead resulted in a big decrease in growth capability. The timing estimate produced previously was therefore adjusted in a simple way for the resulting overhead if the system utilization were increased to nearly 100 percent. This included an estimate of the effect of the task switches in the CP and the IOP that would occur in a system that was nearly fully loaded. The results shown in parentheses in table 5.5-2 were obtained when a fixed time of 0.3 milliseconds was allowed for this task switching.

Table 5.5-1. Refined Configuration Timing Data

Group	Rate, Hz	No. of transactions	Computing time, µsec
A	100.0	6	990
	50.0	6	4,793[1]
	12.5	–	267
B	100.0	4	1,050
	50.0	2	94
	25.0	4	9,687[2]

Organization ground rules like reference configuration

[1] Manual control fully active

[2] Trajectory following active

6.0 SMALL-SCALE SYSTEM TESTING

This section outlines the objectives and presents experiment definitions for the small-scale system testing effort. The small-scale system embodies key features of the IAPSA II design that will be evaluated in a limited experimentation effort. The limited effort will explore a set of critical aspects of the IAPSA II reference configuration architecture.

The small-scale system consists of a triple-channel FTP interfacing with a local I/O network made up of two subnetworks. Experimental data will be obtained for two purposes. First, certain performance estimates obtained during the detailed design effort will be verified. Performance simulation results will be directly compared with applicable measurements made on the small-scale system. Second, certain timing characteristics critical to successful operation in normal and faulted situations will be measured experimentally. A limitation of this small-scale system, of course, is that system level interactions (e.g., communication between the flight control group and the engine groups) cannot be tested.

The small-scale system effort is feasible because of the availability of fault-tolerant processors (FTP), network nodes, and interconnecting links in the hardware area as well as system services packages in the software area. A goal of this effort is to ensure that validation issues defined in the design/validation concept report and uncovered during the detailed design effort are evaluated fully.

The rest of this section contains a discussion of the small-scale system testing objectives, followed by a definition of experiments. The test configurations are described along with the test control strategy. The section concludes with a description of the data collection and analysis approach.

C-4

PRECEDING PAGE BLANK NOT FILMED

6.1 Testing Objectives

The general objective of this testing is to characterize the timing performance of the small-scale system under normal and faulted conditions. The resulting performance measurements, together with low-level time elements measured by the building block developer, will allow evaluation of the performance capability of the IAPSA II reference configuration.

Application computing and I/O activity workload simulation is used to evaluate the small-scale system. Preliminary workload estimates developed during the performance modeling effort are used in conjunction with the selected sequencing and control mechanisms to define a suitable application workload simulation for the small-scale system.

The simulated workload approach requires only a representative test system I/O environment. This means that the test facility need not perform a high-fidelity aircraft or engine simulation to support the experiments envisioned for the small-scale system. This greatly reduces the test facility requirements.

The experimental objectives fall into two major categories. The first category covers measurements dealing with the performance of key system services operations; the second category measures the overall performance of the application. The detailed objectives of the experiments are presented in the following subsections.

6.1.1 System Timing Characterization - Normal Conditions

The timing characteristics of the small-scale system will be measured while it executes the defined application workload. The application workload corresponding to the flight control configuration of the IAPSA II reference configuration is used to characterize key system timing behavior.

Input/Output Request Timing. The time needed to execute the application I/O activity is a key component of a control cycle. The performance model estimates may have been optimistic because they were based on operation near the hardware limit. The small-scale system measurements will provide a more realistic end-to-end time for this activity. One measurement will determine the total time required to perform the I/O activity. Time is measured from the beginning of the system call that starts the I/O activity until the data is available. This measures all intervening system overhead processing time.

Another set of measurements will capture the system overhead time needed to transfer output data in preparation for an I/O request or to transfer input data obtained as a result of an I/O request. These timing elements were not included in the performance model. Time is measured from the beginning of the system calls that starts the data transfer until the system function is complete and the data is available. The amount of data transferred by these calls will correspond to the reference configuration flight control I/O traffic.

Control Cycle Overhead. The total end-to-end system processing time needed to support the cyclic application execution will be measured. The measurement will be made in a controlled execution environment to determine allowable frame rates for slow-time testing. The key service operations are processing of I/O requests, scheduling and dispatch actions, and fast FDIR processing. The I/O request time components discussed previously are contained in this time. I/O processing will be measured in two situations: (1) when the application does no detailed error checking and (2) when the application checks the error status of every transaction.

Laboratory Environment Errors. The objective is to characterize the small-scale system testing environment in terms of its potential for naturally occurring errors. If the background environment causes errors in data transferred or voted via the data exchange, system FDIR actions will result. Similarly, naturally occurring errors due to noise, and so forth, in data transferred over the I/O network will cause I/O network manager actions. Unless errors of both types are very rare, they will interfere with small-scale system testing. These measurements will characterize only the normal execution environment in the laboratory. The critical issue of naturally occurring errors or transients in the flight environment can only be addressed by flight testing.

6.1.2 System Timing Characterization - Fault Conditions

Input/Output Network Faults. The time to recover from a fault on the I/O network will be measured from fault insertion until the network is back in service. While the network is being repaired, the application does not have access to half of the reference configuration sensors and actuators. In addition to the passive link failures run during the simulation model experiments, active link failures and active and passive node failures will be investigated. In addition, the AIPS network repair strategy has been further defined since the simulation experiments were defined. As a result, the small-scale system will incorporate a sophisticated repair strategy that combines some of the aspects of the strategies modeled in the simulation experiments. The network growth time predicted by the performance model will be compared to the small-scale system results allowing for improvements due to the more sophisticated strategy.

Fault-Tolerant Processor Faults. The recovery time from FTP faults is measured from fault insertion to reconfiguration completion. Rapid reconfiguration is important for two reasons. First, an FTP is vulnerable to a nearly coincident fault on another channel during an FTP channel failure-recovery period. In this situation, a failure that would otherwise result in a fully operational system leads to system failure. This special vulnerability to subsequent faults disappears after reconfiguration. Second, certain pathological channel failures can cause erroneous data to be sent over a network. In the IAPSA II design, the result is that all actuators will "freeze" near the last commanded position. It is therefore important for the FDIR to disable a faulty channel's outputs as soon as possible. A small set of FTP failures will be used to exercise this process.

6.1.3 Application Timing Characteristics - Normal Conditions

The key application timing requirements will be assessed using the small-scale system executing the application workload simulation. The three normal measurement categories are execution variability, time delay, and deadline margin.

Execution Variability. These execution variability measurements characterize the frame-to-frame regularity of key computing and I/O activity events. This will permit evaluation of the regular timing performance of the system scheduling and dispatch functions and the I/O system services processes.

Time Delay. Time delay measurements characterize the key application end-to-end timing performance. The performance of each major application function is affected by the overall time delay involved in one control cycle. Times representative of the sensor read and actuator write events at the device interface unit (DIU) will be recorded for each of the different application rate groups.

Deadline Margin. The deadline margin data indicate how well the system is keeping up with the periodic demands of the different application rate groups. The deadline is the latest time that the activity in one control cycle can complete and still satisfy the control cycle timing requirement. The time from the end of control cycle activity in one frame to the start of control cycle activity in the next frame marking the deadline will be measured.

6.1.4 Application Timing Characteristics - Input/Output Network Fault Conditions

The application timing measurements previously described will be made in the simulated failure experiments to see if the additional demands made on the system due to fault recovery adversely affect the continued application execution. The I/O network faults to be simulated in this testing are defined in section 6.4. The number of control cycles that the application processes must operate without access to the full complement of sensors and actuators will be recorded. Each application frame without full data due to repair actions will be marked.

Transaction Selection (Optional). This significant action eliminates any vulnerability to the temporary exhaustion system failure condition. Temporary exhaustion described in section 3 was the critical failure condition for the reference configuration. The refined configuration strategy for dealing with this threat requires the application to first determine that this special action is needed. Next, the application must make the appropriate transaction select and transaction deselect system calls to eliminate the vulnerability. The time from the simulated fault until execution of the reconfigured chain is the important parameter in this action.

6.1.5 Application Timing Characteristics - Fault-Tolerant Processor Fault Conditions

The application timing measurements described in section 6.1.3 will be made in the simulated failure experiments to see if the demands made on the system to handle faults adversely affect the continued application execution. In addition, the system is vulnerable if a faulty channel can send incorrect output commands over an I/O network. This measurement will determine how long it takes the system to disable a faulty channel's outputs. This will verify a key assumption that a bad channel is disabled and communication responsibility is transferred to a good channel within a few application cycles.

6.2 Experiment Descriptions

This section contains general descriptions of the small-scale system experiments. It should be noted that the experiment numbering begins with 10 to differentiate from the DENET experiments. Two different application organization options will be used in the testing; on-demand I/O and periodic I/O. These options refer to how the application control cycle is scheduled and organized. The periodic option starts the I/O requests each period using the IO service timing function in the IOP. The application computing executes when the I/O request completion is signalled.

The on-demand option starts the application computing cycles using the system services timing function in the CP. Input/output requests are executed when explicitly directed at the completion of the application computing activity. It should be noted that this organization is different from the on-demand option modeled in the simulation because the I/O activity occurs at the end of the computing instead of at the beginning. Additionally, there is only one cross processor (IOP - CP) event per control cycle instead of the two in the simulation model version. The simulation version effectively minimized the I/O activity jitter at the expense of additional overhead processing.

Experiment 10: Fault-Tolerant Processor Execution Environment Characterization. A special version of the pseudoapplication program will be run in a controlled environment to measure the time required by some key operations. A single application task will be executing in the FTP with no I/O activity. The key process to be timed will be executed a large number of times. Processes to be measured in this experiment include (1) the read and assign real-time clock value, (2) the background self-test FDIR, and (3) the application workload loop. The first measurement will characterize the overall impact of making timing measurements from the psuedoapplication program. The second set of measurements will enable an assessment of the effectiveness of the background FDIR process. This is because the effectiveness is dependent on how often it executes. The final set of measurements are needed to tailor the timing needs of the psuedoapplication so that the desired timing load can be simulated in the small-scale system.

Experiment 11: System Overhead Characterization. A special version of the pseudoapplication program will be used to time the end-to-end system overhead requirements. The total overhead for each application rate group will be stimulated in a controlled execution environment. A special test executive will control the execution of each application rate group activity to produce a sequence identical to that of an ideal major frame resulting from an on-demand scenario.

The special executive will wait for the I/O request completion flag to measure the time needed to complete the I/O activity before scheduling the next rate group. The experiment will be run for a large number of major frames to ensure adequate overhead assessment. The test will be run with and without error checking of each application transaction to determine the limits of its effect. The sequence will be run with and without the time-critical FDIR process to measure the additional overhead.

Experiment C/P IOP FDIR Phasing Investigation. In this experiment, a limited set of computational processor (CP) and I/O processor (IOP) FDIR phasing combinations will be run to assess the effect on key application timing parameters. The set of combinations will be run for each application scheduling option. This will be an all-up run with full operation of all system service processes and pseudoapplication processes except for the background FDIR self-test. Cyclic execution of application rate groups will be controlled by the defined system timing functions. The computing workload and the application group execution rates will be adjusted to produce a slow-time scenario for this test.

Experiment 13: Input/Output Network Faults. In this experiment, link faults will be inserted to simulate certain network element faults. After a few major frames of normal operation, a fault will be inserted at the desired network location at a predefined point in the test. Traffic on the good network and the faulty network will be monitored to evaluate the effect of the network repair activity. The FTP system services logs will be consulted to correlate the time of failure detection and return of the network to service. A passive failure mode and a special active failure mode will be simulated for both links and network nodes.

Experiment 14: Fault-Tolerant Processor Faults. At a predefined point in the test, the pseudoapplication program will perform special operations to cause faulty behavior in one FTP channel. Faults that result in output miscompares and faults that cause a channel to lose synchronization will be simulated. Network traffic and pseudoapplication task timing will be monitored for anomalous behavior during the failure recovery. Faults will be simulated in a channel driving a network and in a channel not driving a network. The system services logs will be consulted to correlate the time of failure detection and recovery to duplex operation. Power failure faults for a channel will be simulated using hardware methods and will be accomplished in a manner similar to the I/O network faults.

Experiment 15: Transaction Selection (Optional). This experiment will simulate a transaction selection scenario. It will cover the end-to-end application process from detection of the problem to reconfiguration of the application chains. A special version of the psuedoapplication program will contain code for the communications check and the fault-reaction process that selects and deselects transactions. This experiment will also use special chain definitions with an additional transaction that will be initially deselected. A fault will be simulated in the appropriate DIU link and the time of first execution of the reconfigured chains noted. After reconfiguration, the chains on one network will execute with one less transaction due to deselection, while the chains on the other network will run one extra transaction due to selection.

The detailed experiment needs will be defined in detailed test plans (DTP) before running any experiments. These DTPs will guide the actual execution of the experiments on the test facility. The DTP must describe the test sequence of events, the data configuration for the experiment, and the schedule for data acquisition by the test facility. Data acquisition includes the identification of parameters to be monitored and the applicable frequency and simulation time spans for data collection.

6.3 Experiment Test Configurations

The test configuration for the small-scale system experiments is shown in figure 6.3-1. This section describes the hardware and software elements of the test configuration organized into two categories, the system under test (SUT) and the test facility. The system-under-test elements represent components that ultimately are part of the flight system. Test facility elements are the hardware and software elements that enable the SUT operation to be simulated in the laboratory and that provide the development and analysis capabilities necessary to support testing.

6.3.1 System Under Test Elements

Fault-Tolerant Processor. The FTP is an AIPS triplex general purpose computer (GPC). The FTP version used in the small-scale system testing

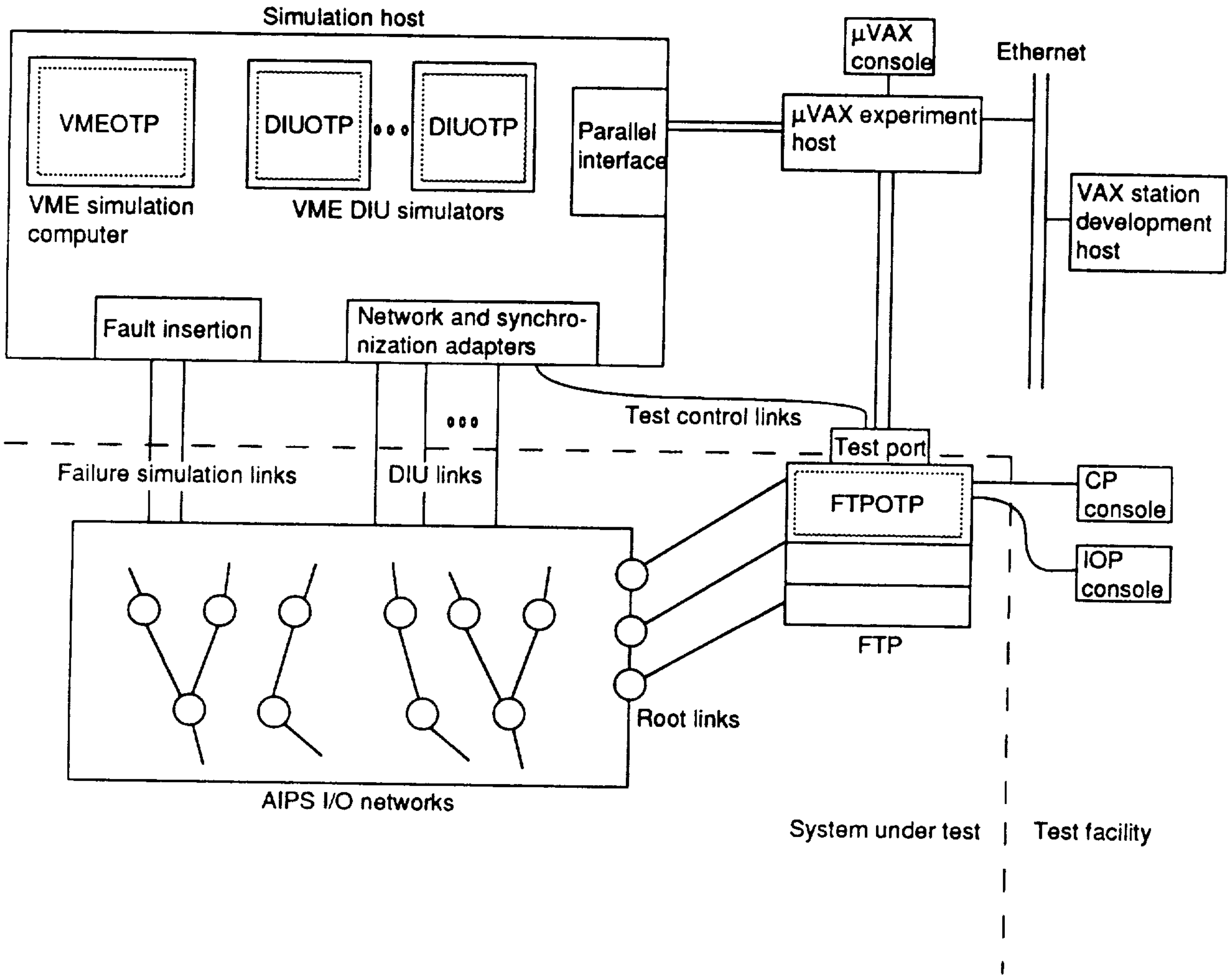


Figure 6.3-1. Experiment Test Configuration

uses two Motorola 68010 microprocessors running at 8 MHz. One is used as an IOP while the other is used as a CP. The CP is primarily used for application software execution. Each FTP channel has a shared bus, a data exchange interface, an interstage, and I/O network interface hardware. Operation of the FTP during experiment activities is controlled via the FTP test port that interfaces with the MicroVAX experiment host.

Fault-Tolerant Processor Operational Test Program. The fault-tolerant processor test program (FTPOTP) consists of two major elements, the pseudoapplication software and the AIPS system software. The pseudoapplication software has the responsibility for providing a computational and I/O activity workload simulation, collecting data in the FTP execution environment, and implementing the FTP test control function during the experiment runs. The system services building block elements are linked with the pseudoapplication elements to form the loadable FTPOTP.

Advanced Information Processing System Input/Output Network. Two advanced information processing system (AIPS) serial I/O networks are used to provide communications between an FTP and the sensors and actuators. The network is composed of prototype reconfigurable nodes and pairs of data links that support full duplex communications. The communications elements uses a modified HDLC protocol. The small scale system is configured to model the flight control group of the IAPSA II reference configuration. Network 1 is fully configured with all the nodes and simulated DIUs that would be used in the reference flight control configuration. All I/O network faults will be simulated on Network 1. Network 2 is simulated with two nodes interfacing with a full complement of DIU simulators. During operation, it behaves just like a fully configured network supporting the full I/O traffic load.

6.3.2 Test Facility Elements

The test facility must be able to establish the test conditions in the system-under-test elements to provide a representative I/O environment to the small-scale system during experiment runs and to collect experiment data from the system during the tests. In addition to these runtime activities, the test facility must support downloading of software into the system under test, debugging of the experimental setup, and analysis of the experimental data.

Simulation Host. The simulation host is a VMEbus-based system containing a 16.7 MHz 68020 CPU (referred to as the VME simulation computer), 16 MB of RAM for data storage, several intelligent serial I/O interface boards (referred to as VME DIU simulators), custom I/O network interface boards, a parallel I/O interface board for communications, and a fault insertion panel. During experiment runs, the simulation host is responsible for: (1) maintaining an experiment time reference, (2) providing a real-time DIU simulation capability, (3) controlling I/O network fault injection hardware, and (4) collecting data from I/O network activity.

VME Operational Test Program. A VME operational test program (VMWOTP) running on the VME simulation computer handles the test setup, initialization, test control, and runtime data collection functions for the simulation host. The VMEOTP fault control function commands the state of the I/O network fault insertion panel during experiment runs in accordance with a predefined fault script. This capability allows a wide range of network faults to be simulated. In cooperation with the DIU simulators, it manages the temporary storage of the I/O network activity data collected during experiment runs. Finally, in its test control function role, it coordinates the start of the experiment run with the FTP and the orderly termination of the experiment with the other simulation host elements.

Device Interface Unit Simulator Operational Test Program. In an actual system, DIUs connected to the I/O network provide an interface between application software executing in an FTP and the aircraft sensors and actuators. The small-scale system uses DIU simulators to support the I/O network transaction load representative of an actual system. The transactions contain dummy data used for test purposes and do not have values representative of actual sensors or actuators. The DIU simulator operational test program (DIUOTP) is responsible for initializing the DIU simulator hardware, checking the command frames received, collecting data about each command frame, generating any necessary response frames, and starting and stopping DIU operation during experiments.

MicroVAX Experiment Host. The MicroVAX experiment host computer controls the FTP using the VAX resident FTP interface program (VRIP) and the FTP resident AIPSDEBUG program. During experiment operations, the experiment host is responsible for (1) download of the FTP operational test program before experiment runs, (2) setup of the run peculiar data configuration in the FTP before experiment runs, and (3) start of the experiment run. The experiment host is also responsible for upload and temporary storage of the raw data collected in the simulation host and the FTP after experiment run termination.

The experiment host is also used to develop, compile, and link the FTP operational test program. The host contains the AIPS system services software library. When the pseudoapplication software is ready, this machine compiles and links the loadable FTP operational test program. The microvax experiment host converts the raw experiment data from the VME simulation host and the FTP to a common data analysis format. It supports data analysis and the archiving of processed experiment data. The MicroVAX II stores the VMEOTP and the DIUOTPs and downloads them to the VME simulation computer and the VME DIU simulators before running the experiments.

VAXstation Development Host. The VAXstation 2000 is used to develop the software and firmware targeted for the VME simulation host elements. This includes the VMEOTP and the DIUOTPS. The software elements are transferred to the experiment host for downloading into the simulation computer.

Laboratory Communication Links (Non-Runtime)

An Ethernet link provides a connection between the MicroVAX experiment host and the VAXstation development host. The link will be used to transfer developed VME operational test programs during software development.

A custom link is used to connect the FTP test port and the test port controller in the MicroVAX experiment host. This link is used to download the FTP operational test program before experiment runs, to start the operational test program in the FTP, and to upload raw experiment data after experiment runs.

A custom parallel interface connects the experiment host and the simulation host. It is used to download programs to the simulation host and to upload raw data after experiment runs.

Test Control Links. Three discrete links connect the FTP and the simulation host. Two links are used to coordinate the two main simulation elements at the start of the experiment run. The links also allow the time references in the simulation host to be synchronized at the start of the experiment in the FTP. A test clock link is used to distribute a common time reference between the FTP and the simulation host elements.

Experiment Peculiar Configurations. Most of the significant differences between the element configurations used for each experiment are in the operational test programs in the FTP and the VMEbus simulation computer. These differences are due to the different fault simulation needs, data collection needs, and simulated computing workload needs between the experiments. An exception is that only the FTP and the MicroVAX experiment host hardware elements are needed in experiment 10. The hardware configuration for the rest of the experiments is very similar.

6.4 Fault Insertion

The performance measurement under fault conditions is a key part of the small-scale system testing. Active and passive failures will be simulated in the following I/O network elements: (1) network links, (2) network nodes, (3) root links, and (4) DIU links. Passive faults will be simulated by a stuck logic 0 signal on the links while active faults will be simulated by a stuck logic 1 signal on the link. A node failure will be simulated by inserting the respective fault on all active node links simultaneously. Faults will also be simulated on the FTP. Special code in the FTPOTP will be used to force a CP channel out of sync or an IOP channel out of sync. Another version will be used to cause CP channel output disagreement. A final FTP channel failure will be loss of channel power.

6.4.1 General

The key elements involved in fault insertion are the I/O network link fault insertion panel, the VME operational test program, and the FTP operational test program. Patch cables to the I/O network link fault insertion panel provide the capability of inserting stuck logic 0 or stuck logic 1 signals into an I/O network link. The simulation host controls the introduction of I/O network faults via the I/O network fault insertion panel. The VME operational test program commands the fault insertion panel to initiate or terminate fault behavior.

FTP fault behavior is simulated in the FTP under control of the FTP operational test program. At the appropriate time in an experiment, special failure simulation code is used to cause the appropriate fault reaction from the AIPS FDIR process.

6.4.2 Experiment-Peculiar Strategy

Input/Output Network Faults. To set up an experiment 13 run, the specific links to be failed must be physically routed through the I/O network fault insertion panel hardware. The test system hardware and software must be set up in accordance with the fault script defined in the detailed test plan. This fault script defines the specific fault occurrence time, fault type, and fault insertion channel configuration used for the test.

Before an experiment 13 run, a fault setup subprogram running on the VME CPU in the simulation host sets up the fault insertion hardware. At the appropriate time during the experiment start, the fault insertion subprogram sends the appropriate control word to the I/O network fault insertion panel hardware. At the conclusion of the experiment run, the fault insertion panel is reset to its unfaulted state.

Fault-Tolerant Processor Faults. To set up most experiment 14 runs, a specific fault scenario must be defined to the FTP operational test program as defined in the fault script. The pseudoapplication program will incorporate a high priority one-shot task (higher than all tasks except

FDIR) that will be scheduled to execute at the defined fault occurrence time. The fault control task will execute special code designed to cause the specific fault behavior to occur. This method will be used for the loss of synchronization and output disagreement faults. The power fail fault will be accomplished using special hardware connected to the fault insertion panel. Power fail simulations will be setup and controlled in a similar manner to the I/O network faults previously described.

6.5 Test Control Strategy

The experiment run is coordinated via two test control discretes used to synchronize the experiment in the FTP and the simulation host. While either machine is being set up for a run, the sync discretes are set to the STOP state. When the simulation host has been set up for an experiment and the run time software has been started, the VME sync discrete is set to the RUN state. The simulation host then waits for the FTP test control discrete to change to the RUN state.

When all associated equipment is ready for experiment operation, the FTP operational test program is started via the VRIP interface. On completion of initialization, the FTP samples the VME sync discrete. If the input line is in the RUN state, the test control function in the FTPOTP schedules the application tasks and starts cyclic operation. At the appropriate time, it changes the FTP sync discrete to the RUN state.

All time reference clocks in the simulation host begin counting when the FTP sync discrete goes to the RUN state. On completion of an experiment run, the FTP sets the FTP sync discrete to STOP. The simulation host responds by terminating data collection, recording experiment ending time and changing the state of the VME sync discrete to STOP. Both computers are then free to transfer experiment raw data and set up for the next experiment.

The real-time clock in the FTP, the VME simulation computer time reference clock, and the VME DIU simulator time-reference clocks are synchronized at the start of an experiment. A common test clock with a 4.125 microsecond period drives all of the time reference elements in the system.

6.6 Data Collection and Analysis

The DIU simulator collects data about the I/O network transactions in real time. The raw data consists of information about all command frames sent by the FTP and processed by each DIU simulator. The information includes command frame error status information such as HDLC error, sumcheck error, etc. In addition to the error status information, the collected information includes an identifier, transmitted frame count, and the time of DIU simulator processing.

Data regarding FTP operation are collected during experiment runs by the pseudoapplication program and stored in CP and IOP local memory. After run completion, data are extracted from the FTP using the FTP test port interface and VRIP/AIPSDEBUG software and stored in raw data files on the MicroVAX experiment host. Data collected by the pseudoapplication includes the real time clock value at significant application events, indicators for certain I/O system and FTP errors, and the background program workload count at the beginning of each minor frame.

In addition to the data visible to the pseudoapplication, system services data are necessary to fully document some experiment runs. This information is recorded by system services in special logs that can be accessed via CRTs connected directly to the FTP coprocessors. The significant data from these logs are manually recorded in the experiment log after each experiment run.

The raw data generated after each experiment run will be transferred to the MicroVAX experiment host for analysis. The raw data is converted to a common format before use by the data analysis program. The FTP data recorded in the experiment log is entered manually for use by the data analysis program.

6.6.1 Standard Statistical Data

The data analysis program has a statistics package that generates a standard set of statistics for certain data sets. The standard set of statistics includes the mean, the standard deviation, and the extreme values found in the data set. The package also creates a histogram to display the data set values. The histogram range and number of intervals is set by default based on the extreme values in the data set and the number of data samples. The user can also manually set the histogram display parameters.

For all of the following statistical data requirements, the number of data points included in the statistical summary will be listed. Data samples associated with any abnormal frames are not included in the statistical summary. These abnormal frame conditions are defined in the event summary description.

Execution Variability Data. The statistical data in this category indicates the frame-to-frame variability of the application execution. Execution variability statistics are based on the frame relative time of a specific application event. The frame relative time of each occurrence of the event is based on the ideal frame start time for each application frame. This is determined based on the ideal start time of the very first frame and the frame repetition period.

Duration Data. Statistics in this category are produced on data derived from the raw application event information. The data are based on the difference in time of occurrence between two application events. Two examples are time delay and deadline margin.

Time delay data are indicative of the sensor-to-actuator time delay for each application update rate. A transaction used by a specific rate group is used as representative of the sensor read and actuator write events. The difference between the transaction time in one frame and its occurrence in the subsequent frame is recorded as the time delay value.

Standard statistics data are generated for the time delay data sets. The time delay values for abnormal frames, such as missed I/O update or frame overrun situations, are not used in the subsequent statistical analysis. Similarly, if there is an extraordinary case of long time delay because of a missing frame, the value is not used. The number of frames for which no time delay value was recorded due to exceptional conditions should be listed in the statistics output.

Deadline margin data indicates how close the system is to not satisfying the regular update requirements of the application processes. The processed data is derived based on the time the final activity completes in one frame and the deadline for that activity occurs in the subsequent frame. The final activity and the deadline are different depending on the selected scheduling mechanism.

For periodic I/O configurations, the deadline event occurs near the start of the I/O request processing in the IOP when the output data buffer is accessed. The final activity in a control cycle is the updating of the output data for the next I/O cycle. The final activity completes just before the end of the application computing.

For on-demand I/O configurations, the deadline is the reading of the input data from the preceding I/O cycle. This takes place just after the beginning of the the computing activity for a control cycle. The final activity in the frame is the completion of the update of the input data by the I/O request processing.

A problem for deadline margin measurements in the small-scale system is that one of the events cannot be recorded directly unless the system services software is instrumented. This was not done for these experiments. Therefore, another event closely related to the actual deadline or final activity is used. This will introduce a bias in the processed deadline margin values.

The statistical analysis is performed on the computed deadline margin data sets. The deadline margin values for abnormal frames such as those with missed I/O updates or frame overrun cases will not be used in the subsequent statistical analysis. The number of data samples rejected due to abnormal frames will be shown in the statistics output.

Time Reserve. Time reserve data will be produced for the CP. These data are indicative of the demand placed on the system by the application and the time critical system services (fast FDIR). The time reserve information for experiment 12 characterizes the steady-state demands on the system. For experiments 13 and 14, the data illustrates any change in the demand on the system during the fault recovery period.

The time reserve data for one frame is determined by subtracting the background counter value in a frame from its value in the subsequent frame. The background count is incremented during the idle or reserve time when there is no other processing to be run. The time reserve data sets will be divided into four groups corresponding to the specific minor frame of the major frame cycle (A,B,C, or D). Standard statistics are generated on each group.

Fault Recovery. The fault recovery information indicates how long the system takes to repair a fault for experiment 13 and 14 runs. Some of the raw data is recorded from CRT displays of the system services log and so must be manually entered for the data analysis program. The fault recovery time is based on the fault occurrence time and a corresponding recovery complete time for each run. The fault recovery times for all of the runs having the same configuration and fault type form a data set for which statistics are generated.

6.6.2 Event Summary Data

In addition to statistical data, the analysis program determines and presents event summary data for the experiment run. The summary information is based on data collected during each run as well as detailed information about certain situations occurring during the runs being evaluated. The summary is organized by run in the case of multiple-run evaluation, with the events in each run listed in chronological order. When an event is listed, associated data recorded with the event are presented.

The event summary for each run includes a run start entry and a run termination entry. A brief description of the other significant situations is presented in the following paragraphs.

An entry for the FTP fault insertion event is included for all experiment 13 runs. This entry will show all data associated with the specific fault condition. Similarly, there will be an entry for the VME fault insertion event for all experiment 14 runs. All data associated with the specific fault condition will be presented.

Experiment 13 and 14 runs will also contain entries for the end fault repair event. These entries will show the time that the system services completed the reconfiguration actions taken to repair the fault.

There will be entries for each application frame that experiences communication errors during I/O activity. These errors include (1) chain error, (2) all transactions bad, (3) chain not complete, (4) chain did not execute, or (5) network out of service. The entry will show the time, frame count, and application rate group identifier.

Any command frame received at a DIU with errors will appear in the summary. The command fame identifier, time, frame count, and error code will be presented.

The partial data summary shows the number of frames in which communications with the complete set of DIUs is interrupted because a network is out of service. The application partial data summary will show for each run, by application rate, the number of frames in which the application used a partial set of sensor and actuator data because a network was taken out of service.

The abnormal DIU data summary indicates when the DIU did not receive the expected periodic update from the application process. This occurs when a DIU command frame is repeated or skipped. For each DIU frame ID, the number of occurrences of repeated command frames and skipped command frames will be reported for each run.

Abnormal Frames. The abnormal frame entries document the occurrence of an incorrect application cycle. The three specific situations are missed I/O update, computing overrun, or IOR overrun.

Missed I/O updates occur when the I/O activity part of the control cycle and the computing part of the control cycle do not keep up with each other. In this situation, the deadline for the final activity in the frame is not met. In the on-demand I/O case, the IO service activity does not finish updating the input data buffer before the time it is read by the subsequent frame computing activity. In the periodic I/O case, the application computing does not complete updating the output data buffer before the subsequent frame I/O request.

A computing overrun refers to the situation in which the application CP processing for one frame is scheduled to start while the application process for the previous frame is still active. The system ignores the new request in this situation.

An IOR overrun occurs when the I/O system is unable to process a submitted I/O request. In this case, the I/O activity for a new frame is scheduled to occur before the I/O activity has completed for the previous frame. Special system calls are used to collect data about the occurrence of either type of overrun.

The analysis output data is determined based on the raw data from the experiment runs. Data in the execution variability and duration categories are based on a subset of the possible raw data produced. For experiment 12, 13, and 14, the analysis output data will be based on raw data from the start of the third application major frame through the end of the run.

7.0 CONCLUSIONS

During the detailed design effort for the IAPSA II contract, a candidate architecture design based on AIPS fault-tolerant system building blocks was evaluated for its ability to meet the demanding performance and reliability requirements of a flight-critical system. As a result of the preliminary evaluations some refinements were made to the candidate architecture. The refined configuration was evaluated for reliability and performance and a set of experiments defined for testing critical performance aspects with a small-scale system. This modeling effort provided several important results described below.

The major result was that several weaknesses in the candidate architecture became apparent as a result of using the prevalidation methodology. These shortcomings were not evident in the initial performance and reliability calculations. This is important because such concept weaknesses are often not uncovered until late in the system life cycle, for example at hardware and software integration. The IAPSA II effort shows that it is very important to perform detailed evaluation of concepts based on specifications before committing a project to a hardware and software build phase.

The candidate architecture was unable to meet either the reliability or performance requirements. Reliability was affected by uncovered element failures (a common finding in critical systems), rare mechanical failure modes, and an interaction of preexisting sensor failures with I/O network repair activity. As a consequence, there were several two-failure situations that resulted in loss of safety, and one-failure situations resulting in loss of full mission capability.

The performance simulation showed that part of the candidate system was overloaded and did not possess the needed growth capability. In fact, the loading was such that special coordination was needed between the application tasks and the time-critical system tasks to allow the application workload to complete in the allowable time. Alternative organizations of the workload were evaluated and only the most efficient could be fit into the allowable timeframe. However, even the optimum organization had inadequate growth capability.

Changes were made to the candidate architecture to better balance the workload, to improve the level of failure protection, and to reduce the number of communication elements. Although the refined system was shown to meet the reliability requirements, several concerns became evident during the evaluation. Our modeling approach was based on separate models reflecting the success of the key system functions. Modeling the dependency of the system functions on the central elements such as communication devices and electrical and hydraulic power distribution was difficult. It was easy to miss the implications of system interconnection alternatives, especially when the central elements also had dependency relationships. This was very true in the special power connection reliability concerns associated with the refined configuration mesh network option.

A solution that would make these central dependencies explicit in a large-system-level model still seems unattractive. A method is needed to more formally organize the combining of the separate section model results. The techniques used in the refined configuration analysis were more effective and easier to justify than those used during the candidate evaluation. However, the step of estimating error due to model truncation when a problem is split into submodels needs further development.

A sensitivity study of the susceptibility of the system to transient faults pointed out an area of concern. Because of the complex redundancy management strategies, the handling of transients is always challenging for flight critical systems. One problem is characterization of the transient threat. Solid operational data on the likelihood and duration of transients is scarce. In our limited study, the major concern was transients that can cause a channel to go out of synchronization. Because of the heavy application workload environment, there was not enough time for the system to bring a channel back into synchronization and align its memory with the remaining "good" channels. Therefore a transient that causes loss of synchronization has the same effect as a permanent failure. If the transient fault rate is significant in operational circumstances, then these failures will contribute significantly to system failure.

There were many situations in the study in which performance and reliability were closely interrelated. The above resynchronization situation is one example. Another was the strategy to eliminate vulnerability to "temporary exhaustion" failures. Due to the heavy application I/O workload, strategies of sending sensor and actuator data redundantly over both networks were impractical.

The use of a discrete event simulation tool, like DENET, is new in the analysis of a flight control system. Such tools appear very promising for determining critical performance requirements, but much application work is needed to define practical and effective modeling techniques. However, attempting to develop complex system configurations with multiple processing sites and intensive I/O needs without using such tools to uncover bottlenecks seems foolhardy based on our experience.

The rebalancing of application workload improved the overload situation significantly compared to the candidate system. However, the preliminary simulation results experiments indicates that the improvement is not enough. Using a simplified adjustment for task overhead processing was enough to show that the growth factor for the refined configuration is below requirements. To make matters worse, the proof-of-concept testing conducted before the small-scale system testing shows that the real system is not operating anywhere near the hardware limit. Because our performance models were based on very efficient operation, the small-scale system testing is likely to show that performance needs cannot be met with the current hardware and software design.

The AIPS building block hardware design is much more mature than the software design; the functionality assigned to the system services software is overwhelming. Validation of this complex software, which includes some nondeterministic behavior, will be extremely challenging. Efficiency considerations have only recently been addressed in the software development effort. Early in the IAPSA II design, performance calculations were based on the original AIPS system performance goals as expressed in the system specification. However, since these goals were not reflected in the performance requirements for the AIPS building blocks, a building block redesign will probably be needed.

In our system study, a set of redundant buses provided nearly the same general reliability as the reconfigurable I/O mesh networks. Because of the system software complexity associated with the management of the reconfigurable mesh I/O networks and the needed validation effort, the I/O mesh network would be eliminated from further consideration at this time. However, to meet the throughput needs of the application functions, an IC network between computing sites is essential. The IC network has even more demanding network management requirements than the I/O networks. The characteristics of communications over the IC network also generate some demanding performance needs. These needs and associated potential resource contention problems were not evaluated in our study.

REFERENCES

1. NASA Contractor Report 178084, "Design and Validation Concept for the Integrated Airframe/Propulsion Control System Architecture," G. C. Cohen, et al., June 1986.

2. NASA Technical Memorandum 83553, "A Real-Time Implementation of an Advanced Sensor Failure Detection, Isolation, and Accommodation Algorithm," J. C. Delaat and W. C. Merrill, January 1984.

3. AFWAL-TR-86-2084, "Fault Tolerant Electrical Power System, Phase II: Analysis and Preliminary Design," Boeing Military Airplane Company, December 1986.

4. CSDL-P-1945, "Evaluation Methodologies for an Advanced Information Processing System," R. S. Schabowsky Jr., et al., August 1984.

5. AIAA-88-4409, "Fault Tolerant System Performance Modeling," M. J. Strickland and D. L. Palumbo, AIAA/AMS/ASEE Aircraft Design, Systems and Operations Conference, September 1988.

APPENDIX A

SELECTED ASSIST MODELS

IAPSA II REFINED CONFIGURATION

```
(*** Forward Area Model - Safety Criteria - Network Option ***)
(*                      Group A                                *)
(*   _______________________________________________________  *)

SPACE    = (  FWNOD:    0..4,    (* FORWARD NODE                        *)
              FWREC:    0..2,    (* NETWORK RECOVERY INDICATOR          *)
              FWDIU:    0..4,    (* FORWARD DIU STATE INDICATOR         *)
              PITSTK:   0..4,    (* PITCH COMMAND SENSOR                *)
              PITREC:   0..2,    (* PITCH SENSOR RECOVERY INDICATOR     *)
              ROLSTK:   0..4,    (* ROLL COMMAND SENSOR                 *)
              ROLREC:   0..2,    (* ROLL SENSOR RECOVERY INDICATOR      *)
              YAWPED:   0..4,    (* YAW COMMAND SENSOR                  *)
              YAWREC:   0..2,    (* YAW SENSOR RECOVERY INDICATOR       *)
              FTP:      0..4,    (* FTP CHANNEL STATUS                  *)
              FTPREC:   0..2,    (* FTP CHANNEL RECOVERY IND.           *)
              ROOT:     0..4,    (* ROOT LINK STATUS                    *)
              ELMC:     0..4,    (* ELECTRIC POWER STATE                *)
              ONREC:    0..2,    (* OTHER NODES IN A SINGLE NETWORK     *)
              NFAIL:    0..4);   (* NO. OF FAILED ELEMENTS              *)

START  = ( 4, 0, 4, 4, 0, 4, 0, 4, 0, 4, 0, 4, 4, 0, 0 );

PRUNEIF  NFAIL > 2;

DEATHIF   PITREC + FWREC + ONREC  > 1 OR ROLREC + FWREC + ONREC > 1 OR
          YAWREC + FWREC + ONREC > 1;

DEATHIF   PITSTK-PITREC < 1 OR ROLSTK-ROLREC < 1 OR YAWPED-YAWREC < 1;

DEATHIF   FTP - FTPREC < 1;

DEATHIF   FTP < 2;

DEATHIF   ROOT < 3 AND ( PITREC > 0 OR ROLREC > 0 OR
          YAWREC > 0 OR FWREC > 0 OR ONREC > 0 );

LIST  = 2;
TIME  = 3.0;
PRUNE = 1.0E-16;
ECHO  = 0;

LAMPOS  = 10.0E-6;        (* POSITION SENSOR FAILURE RATE      *)
LAMNOD  = 17.0E-6;        (* NODE FAILURE RATE - CONDITIONED   *)
LAMDIU  = 15.0E-6;        (* DIU FAILURE RATE - CONDITIONED    *)
POSMEAN = 3.0E-4;         (* RECOVERY TIME MEAN              *)
POSSTD  = 1.0E-4;         (* RECOVERY TIME STD DEV           *)
NWMEAN  = 3.0E-4;         (* NW RECOVERY TIME MEAN             *)
NWSTD   = 1.0E-4;         (* NW RECOVERY TIME STD DEV          *)

LAMPS     =  10.0E-6;     (* LOCAL POWER SUPPLY RATE - CONDITIONED    *)
LAMEL     = 50.0E-6;      (* ELECTRIC POWER CENTER FAILURE RATE       *)
LAMFTP    = 200.0E-6;     (* FTP CHANNEL FAILURE RATE                 *)
LAMROOT   =  15.0E-6;     (* FTP NETWORK INTERFACE FAILURE RATE       *)
FTPMEAN   =   5.0E-6;     (* FTP RECOVERY TIME MEAN                   *)
```

```
FTPSTD    =  1.0E-6;        (* FTP RECOVERY TIME STD DEV                 *)

LAMON     = 289.0E-6;       (* OTHER NODES ON SINGLE NW FAIL RATE        *)


          (* PITCH COMMAND SENSOR FAILURES AND RECOVERIES *)

IF PITSTK > 0 AND PITREC = 0 AND ROLREC = 0 AND YAWREC = 0
     AND FTPREC = 0 AND FWREC = 0 AND ONREC = 0 THEN
     IF NFAIL = 0 OR PITSTK = 2 THEN
        TRANTO PITSTK = PITSTK-1, PITREC=PITREC+1, NFAIL=NFAIL+1
        BY PITSTK*LAMPOS;
     ELSE
        IF ROOT = 2 THEN    (* SINGLE NETWORK SUSCEPTIBILITY *)

           TRANTO  PITSTK=PITSTK-1, PITREC=PITREC+1, NFAIL=NFAIL+1
           BY (2/3)*LAMPOS;
           TRANTO PITSTK=PITSTK-1, NFAIL=NFAIL+1
           BY  ((PITSTK - 2) + 4/3 )*LAMPOS;
        ELSE
           TRANTO PITSTK = PITSTK-1, NFAIL=NFAIL+1
           BY PITSTK*LAMPOS;
        ENDIF;

     ENDIF;

ELSE         (* NEARLY COINCIDENT SENSOR-NETWORK *)

     IF FWREC > 0      (* SAME NODE SET *)
        TRANTO PITSTK = PITSTK-1, PITREC=PITREC+1, NFAIL=NFAIL+1
        BY (PITSTK+1)*LAMPOS /2;

     IF ONREC > 0      (* OTHER NODES *)
        TRANTO PITSTK = PITSTK-1, PITREC=PITREC+1, NFAIL=NFAIL+1
        BY PITSTK*LAMPOS /2;

     IF PITREC > 0 THEN  (* RECOVERY OR SYSTEM LOSS *)
        TRANTO PITREC = 0 BY < POSMEAN, POSSTD >;
        TRANTO PITSTK = PITSTK-1, PITREC=PITREC+1, NFAIL=NFAIL+1
        BY PITSTK*LAMPOS;
     ENDIF;

ENDIF;


          (* ROLL COMMAND SENSOR FAILURES AND RECOVERIES *)

IF ROLSTK > 0 AND PITREC = 0 AND ROLREC = 0 AND YAWREC = 0
     AND FTPREC = 0 AND FWREC = 0 AND ONREC = 0 THEN
     IF NFAIL = 0 OR ROLSTK = 2 THEN
        TRANTO ROLSTK = ROLSTK-1, ROLREC=ROLREC+1, NFAIL=NFAIL+1
        BY ROLSTK*LAMPOS;
     ELSE
        IF ROOT = 2 THEN
              (* SINGLE NETWORK SUSCEPTIBILITY *)
           TRANTO  ROLSTK=ROLSTK-1, ROLREC=ROLREC+1, NFAIL=NFAIL+1
           BY (2/3)*LAMPOS;
           TRANTO ROLSTK=ROLSTK-1, NFAIL=NFAIL+1
           BY  ((ROLSTK - 2) + 4/3 )*LAMPOS;
        ELSE
           TRANTO ROLSTK = ROLSTK-1, NFAIL=NFAIL+1
           BY ROLSTK*LAMPOS;
        ENDIF;

     ENDIF;

ELSE        (* NEARLY COINCIDENT SENSOR-NETWORK *)

     IF FWREC > 0      (* SAME NODE SET *)
        TRANTO ROLSTK = ROLSTK-1, ROLREC=ROLREC+1, NFAIL=NFAIL+1
        BY (ROLSTK+1)*LAMPOS /2;

     IF ONREC > 0      (* OTHER NODES *)
        TRANTO ROLSTK = ROLSTK-1, ROLREC=ROLREC+1, NFAIL=NFAIL+1
        BY ROLSTK*LAMPOS /2;

     IF ROLREC > 0 THEN   (* RECOVERY OR SYSTEM LOSS *)
        TRANTO ROLREC = 0 BY < POSMEAN, POSSTD >;
        TRANTO ROLSTK = ROLSTK-1, ROLREC=ROLREC+1, NFAIL=NFAIL+1
        BY ROLSTK*LAMPOS;
     ENDIF;

ENDIF;

         (* YAW COMMAND SENSOR FAILURES AND RECOVERIES *)

IF YAWPED > 0 AND PITREC = 0 AND ROLREC = 0 AND YAWREC = 0
     AND FTPREC = 0 AND FWREC = 0 AND ONREC = 0 THEN
     IF NFAIL = 0 OR YAWPED = 2 THEN
        TRANTO YAWPED = YAWPED-1, YAWREC=YAWREC+1, NFAIL=NFAIL+1
        BY YAWPED*LAMPOS;
     ELSE
        IF ROOT = 2 THEN
              (* SINGLE NETWORK SUSCEPTIBILITY *)
           TRANTO  YAWPED=YAWPED-1, YAWREC=YAWREC+1, NFAIL=NFAIL+1
           BY (2/3)*LAMPOS;
           TRANTO YAWPED=YAWPED-1, NFAIL=NFAIL+1
           BY  ((YAWPED - 2) + 4/3 )*LAMPOS;
        ELSE
           TRANTO YAWPED = YAWPED-1, NFAIL=NFAIL+1
           BY YAWPED*LAMPOS;
        ENDIF;

     ENDIF;

ELSE        (* NEARLY COINCIDENT SENSOR-NETWORK *)

     IF FWREC > 0      (* SAME NODE SET *)
        TRANTO YAWPED = YAWPED-1, YAWREC=YAWREC+1, NFAIL=NFAIL+1
        BY (YAWPED+1)*LAMPOS /2;

     IF ONREC > 0      (* OTHER NODES *)
```

```
        TRANTO YAWPED = YAWPED-1, YAWREC=YAWREC+1, NFAIL=NFAIL+1
        BY YAWPED*LAMPOS /2;

     IF YAWREC > 0 THEN  (* RECOVERY OR SYSTEM LOSS *)
        TRANTO YAWREC = 0 BY < POSMEAN, POSSTD >;
        TRANTO YAWPED = YAWPED-1, YAWREC=YAWREC+1, NFAIL=NFAIL+1
        BY YAWPED*LAMPOS;
     ENDIF;

ENDIF;

         (* FORWARD AREA NODE FAILURES *)

IF FWNOD > 0 AND FWREC = 0 AND PITREC = 0 AND ROLREC = 0 AND
   YAWREC = 0 AND FTPREC = 0 AND ONREC = 0 THEN
     IF PITSTK < 3 OR ROLSTK < 3 OR YAWPED < 3 THEN

               (* TEMPORARY EXHAUSTION FAILURE *)
        TRANTO FWNOD=FWNOD-1, FWDIU=FWDIU-1, PITSTK=PITSTK-1,
        ROLSTK=ROLSTK-1, YAWPED=YAWPED-1, ROOT=ROOT-1,
        FWREC=2, NFAIL=NFAIL+1 BY (2/3)*LAMNOD;

               (* TEMPORARY EXHAUSTION AVOIDED *)
        TRANTO FWNOD=FWNOD-1, FWDIU=FWDIU-1, PITSTK=PITSTK-1,
        ROLSTK=ROLSTK-1, YAWPED=YAWPED-1, ROOT=ROOT-1,
        NFAIL=NFAIL+1 BY (( FWNOD-2) + 4/3)*LAMNOD;

     ELSE       (* NO TEMPORARY EXHAUSTION VULNERABILITY *)

        IF ROOT = 2 THEN    (* SINGLE NETWORK VULNERABILITY *)
           TRANTO FWNOD=FWNOD-1, FWDIU=FWDIU-1, PITSTK=PITSTK-1,
           ROLSTK=ROLSTK-1, YAWPED=YAWPED-1, ROOT=ROOT-1,
           FWREC=FWREC+1, NFAIL=NFAIL+1 BY (2/3)*LAMNOD;

           TRANTO FWNOD=FWNOD-1, FWDIU=FWDIU-1, PITSTK=PITSTK-1,
           ROLSTK=ROLSTK-1, YAWPED=YAWPED-1, ROOT=ROOT-1,
           NFAIL=NFAIL+1 BY (( FWNOD-2) + 4/3)*LAMNOD;

        ELSE     (* NO SINGLE NETWORK VULNERABILITY *)

           IF ROOT > (FWNOD+ELMC-4) THEN
              (* CHECK FOR TWO FAULTS ON SAME "CHANNEL"  *)

              TRANTO FWNOD=FWNOD-1, FWDIU=FWDIU-1, PITSTK=PITSTK-1,
              ROLSTK=ROLSTK-1, YAWPED=YAWPED-1, ROOT=ROOT-1,
              NFAIL=NFAIL+1 BY ROOT* LAMNOD;

           ELSE
                (* FAULTS ARE ON SEPARATE "CHANNELS" *)
            (* FORWARD NODE FAILURE AFFECTS ROOT LINK AND DIU *)

              IF NFAIL = 0 THEN
                 TRANTO FWNOD=FWNOD-1, FWDIU=FWDIU-1, PITSTK=PITSTK-1,
                 ROLSTK=ROLSTK-1, YAWPED=YAWPED-1, ROOT=ROOT-1, FWREC=FWREC+1,

                 NFAIL=NFAIL+1 BY FWNOD* LAMNOD;
              ELSE
                 TRANTO FWNOD=FWNOD-1, FWDIU=FWDIU-1, PITSTK=PITSTK-1,
                 ROLSTK=ROLSTK-1, YAWPED=YAWPED-1, ROOT=ROOT-1,
                 NFAIL=NFAIL+1 BY ROOT* PITSTK*ROLSTK*YAWPED*LAMNOD/
                 ( (FWNOD+ELMC-4)* FWDIU**2 );
              ENDIF;

           ENDIF;
        ENDIF;

     ENDIF;

ELSE              (* NEARLY COINCIDENT NETWORK-SENSOR FAILURES  *)

     IF FWREC = 1 THEN    (* SAME NODE SET *)
                    (* NETWORK RECOVERY *)
        TRANTO FWREC = 0 BY < NWMEAN, NWSTD >;
     ENDIF;

     IF PITREC = 1
        TRANTO FWNOD=FWNOD-1, FWREC=FWREC+1, NFAIL=NFAIL+1 BY
        FWNOD*LAMNOD /2;

     IF ROLREC = 1
        TRANTO FWNOD=FWNOD-1, FWREC=FWREC+1, NFAIL=NFAIL+1 BY
        FWNOD*LAMNOD /2;

     IF YAWREC = 1
        TRANTO FWNOD=FWNOD-1, FWREC=FWREC+1, NFAIL=NFAIL+1 BY
        FWNOD*LAMNOD /2;
ENDIF;

         (* OTHER NODE FAILURES *)

IF  FWREC = 0 AND PITREC = 0 AND ROLREC = 0 AND
   YAWREC = 0 AND FTPREC = 0 AND ONREC = 0 THEN
      IF PITSTK < 3 OR ROLSTK < 3 OR YAWPED < 3 THEN
             (* TEMPORARY EXHAUSTION VULNERABLE *)

         TRANTO ONREC=2, NFAIL=NFAIL+1 BY LAMON/3;
      ELSE
         IF ROOT = 2 THEN   (* SINGLE NW VULNERABLE *)

            TRANTO ONREC=1, NFAIL=NFAIL+1 BY LAMON/3;
         ELSE
            IF NFAIL = 0  TRANTO ONREC=1, NFAIL=NFAIL+1 BY
               2*LAMON;
         ENDIF;
      ENDIF;
ELSE
         (* NEARLY COINCIDENT FAILURES- RECOVERIES *)
      IF ONREC = 1
         TRANTO ONREC = 0 BY < NWMEAN, NWSTD >;

      IF PITREC=1 OR ROLREC=1 OR YAWREC=1
         TRANTO ONREC=1, NFAIL=NFAIL+1 BY LAMON;
```

```
ENDIF;


        (* FORWARD DIU FAILURES *)

IF FWDIU > 0 AND FWREC = 0 AND PITREC = 0 AND ROLREC = 0 AND
   YAWREC = 0 AND FTPREC = 0 AND ONREC = 0 THEN
        TRANTO FWDIU=FWDIU-1, PITSTK=PITSTK-1, ROLSTK=ROLSTK-1, YAWPED=
        YAWPED-1, NFAIL=NFAIL+1 BY PITSTK*ROLSTK*YAWPED*( LAMDIU + LAMPS ) /
        FWDIU**2;

ENDIF;

       (* FTP CHANNEL FAILURES *)

IF FTP > 0 AND PITREC = 0 AND ROLREC = 0 AND YAWREC = 0
   AND FWREC = 0 AND FTPREC = 0 AND ONREC = 0 THEN
      IF NFAIL = 0 THEN
         TRANTO FTP=FTP-1, FTPREC=FTPREC+1, ROOT=ROOT-1, NFAIL=NFAIL+1
         BY FTP*LAMFTP;
      ELSE
         TRANTO FTP=FTP-1, ROOT=ROOT-1, NFAIL=NFAIL+1
         BY ROOT*LAMFTP;
      ENDIF;

ELSE
   IF FTPREC > 0 THEN
                  (* FTP CHANNEL RECOVERY *)
      TRANTO FTPREC=FTPREC-1 BY < FTPMEAN, FTPSTD >;
                   (* COINCIDENT FAULT *)
      TRANTO FTP=FTP-1, FTPREC=FTPREC+1, ROOT=ROOT-1, NFAIL=NFAIL+1
      BY FTP*LAMFTP;
   ENDIF;

ENDIF;


         (* NETWORK INTERFACE FAILURES *)

IF ROOT > 2 AND PITREC = 0 AND ROLREC = 0 AND YAWREC = 0 AND FTPREC = 0
       AND FWREC = 0 AND ONREC = 0
           TRANTO ROOT=ROOT-1, NFAIL=NFAIL+1 BY ROOT*LAMROOT;


          (*** ELECTRIC POWER DISTRIBUTION FAILURES ***)

IF ELMC > 0 AND PITREC = 0 AND ROLREC = 0 AND YAWREC = 0 AND FTPREC = 0
    AND FWREC = 0  AND ONREC = 0 THEN
                 (* ELMC FAILURE AFFECTS DIU, FTP AND ROOT LINK *)

    IF ROOT > (FWNOD+FTP-4) THEN
        (* CHECK FOR FAULTS ON SAME "CHANNEL" *)

         TRANTO FWDIU=FWDIU-1, PITSTK=PITSTK-1,
         ROLSTK=ROLSTK-1, YAWPED=YAWPED-1, FTP=FTP-1, ROOT=ROOT-1,
         ELMC=ELMC-1,
                NFAIL=NFAIL+1 BY ROOT* LAMEL;
          ELSE
                (* FAULTS ON DIFFERENT "CHANNELS" *)

                     TRANTO FWDIU=FWDIU-1, PITSTK=PITSTK-1,
                     ROLSTK=ROLSTK-1, YAWPED=YAWPED-1, FTP=FTP-1, ROOT=ROOT-1,
                     ELMC=ELMC-1,
                     NFAIL=NFAIL+1 BY ROOT* PITSTK*ROLSTK*YAWPED*
                     LAMEL / ( (FWNOD+ELMC-4)* FWDIU**2 );

          ENDIF;
    ENDIF;
```

```
(*** Mid Area Model - Safety Criteria - Network Option ***)
(*                       Group A                          *)
(*     __________________________________________________ *)


SPACE   = (  MIDNOD:    0..4,    (* MID NODE                         *)
             MIDREC:    0..2,    (* NETWORK RECOVERY INDICATOR       *)
             MIDDIU:    0..4,    (* MID DIU STATE INDICATOR          *)
             GYRO:      0..8,    (* GYROS                            *)
             GYREC:     0..2,    (* GYRO RECOVERY INDICATOR          *)
             ACCEL:     0..8,    (* ACCELEROMETERS                   *)
             ACCREC:    0..2,    (* ACCEL RECOVERY INDICATOR         *)
             CNDV:      0..4,    (* CND VALVE STATE                   *)
             HYD:       0..2,    (* HYDRAULIC SYSTEM STATE           *)
             ELMC:      0..4,    (* ELECTRIC SUPPLY SYSTEM STATE     *)
             ONREC:     0..2,    (* OTHER NODES IN A SINGLE NETWORK *)
             NFAIL:     0..4);   (* NO. OF FAILED ELEMENTS           *)

START   = ( 4, 0, 4, 8, 0, 8, 0, 4, 2, 4, 0, 0 );

PRUNEIF  NFAIL > 2;

DEATHIF   GYREC + ACCREC + MIDREC + ONREC > 1;

DEATHIF   GYRO-GYREC < 3 OR ACCEL-ACCREC < 3;

DEATHIF  CNDV = 0;

LIST  = 2;
TIME  = 3.0;
PRUNE = 1.0E-16;
ECHO  = 0;

LAMGYRO = 50.0E-6;        (* GYRO FAILURE RATE                 *)
LAMACC  = 30.0E-6;        (* ACCELEROMETER FAILURE RATE        *)
LAMNOD  = 17.0E-6;        (* NODE FAILURE RATE - CONDITIONED *)
LAMDIU  = 15.0E-6;        (* DIU FAILURE RATE - CONDITIONED  *)
GYRMEAN = 3.0E-4;         (* RECOVERY TIME MEAN                *)
GYRSTD  = 1.0E-4;         (* RECOVERY TIME STD DEV             *)
ACCMEAN = 3.0E-4;         (* ACC RECOVERY TIME MEAN            *)
ACCSTD  = 1.0E-4;         (* ACC RECOVERY TIME STD DEV         *)

LAMC   = 50.0E-6;            (* PROCESSOR FAILURE RATE            *)
LAMPOS = 10.0E-6;            (* POSITION SENSOR FAILURE RATE    *)
LAMV   = 15.0E-6;            (* VALVE GROUP FAILURE RATE          *)
VJAM   = 3.3333E-5;          (* ACTUATOR JAM FAILURE FRACTION *)
LAMHYD  = 45.0E-6;           (* HYDRAULIC SYSTEM FAIL RATE        *)

LAMPS    = 10.0E-6;       (* LOCAL POWER SUPPLY - CONDITIONED *)
LAMEL    = 50.0E-6;       (* ELMC FAILURE RATE                   *)
NWMEAN   = 3.0E-4;        (* NW RECOVERY TIME MEAN             *)
NWSTD    = 1.0E-4;        (* NW RECOVERY TIME STD DEV          *)
LAMON    = 289.0E-6;      (* OTHER NODES FAILURE RATE            *)
```

```
         (* GYRO SENSOR FAILURES AND RECOVERIES *)

IF GYRO > 0 AND GYREC = 0 AND ACCREC = 0 AND MIDREC = 0 AND ONREC = 0 THEN

     IF GYRO = 4 OR NFAIL = 0 THEN     (* EXHAUSTION *)
        TRANTO GYRO = GYRO-1, GYREC=GYREC+1, NFAIL=NFAIL+1
        BY GYRO*LAMGYRO;
     ELSE
           TRANTO GYRO = GYRO-1, NFAIL=NFAIL+1
           BY GYRO*LAMGYRO;
     ENDIF;
ELSE
       (* GYRO FAILURE RECOVERY *)
     IF GYREC = 1  TRANTO GYREC=0 BY < GYRMEAN, GYRSTD >;
       (* NCF:  GYRO - NETWORK *)
     IF MIDREC = 1 OR ONREC = 1
        TRANTO GYRO = GYRO-1, GYREC=GYREC+1, NFAIL=NFAIL+1
        BY GYRO*LAMGYRO/2;
ENDIF;

         (* ACCEL SENSOR FAILURES AND RECOVERIES *)

IF ACCEL > 0 AND GYREC = 0 AND ACCREC = 0 AND MIDREC = 0 AND ONREC = 0 THEN

     IF ACCEL = 4 OR NFAIL = 0 THEN     (* EXHAUSTION *)
        TRANTO ACCEL = ACCEL-1, ACCREC=ACCREC+1, NFAIL=NFAIL+1
        BY ACCEL*LAMACC;
     ELSE
           TRANTO ACCEL = ACCEL-1, NFAIL=NFAIL+1
           BY ACCEL*LAMACC;
     ENDIF;
ELSE
       (* ACCELEROMETER FAILURE RECOVERY *)
     IF ACCREC = 1  TRANTO ACCREC=0 BY < ACCMEAN, ACCSTD >;
       (* NCF:  ACCEL - NETWORK *)
     IF MIDREC = 1 OR ONREC = 1
        TRANTO ACCEL = ACCEL-1, ACCREC=ACCREC+1, NFAIL=NFAIL+1
        BY ACCEL*LAMACC/2;
ENDIF;

             (*** VALVE GROUP FAILURES ***)

IF CNDV > 0 AND GYREC=0 AND ACCREC=0 AND MIDREC = 0 AND ONREC = 0 THEN
        IF NFAIL = 0
              TRANTO CNDV=0, NFAIL=NFAIL+1 BY CNDV*VJAM*LAMV;
        TRANTO CNDV=CNDV-1, NFAIL=NFAIL+1 BY CNDV*(1.0-VJAM)*LAMV;
ENDIF; .

          (* HYDRAULIC SYSTEM FAILURES *)

IF HYD > 0 AND GYREC=0 AND ACCREC=0 AND MIDREC = 0 AND
   (HYD-NFAIL >= 0 ) AND ONREC = 0 THEN
     TRANTO CNDV=CNDV-2, HYD=HYD-1, NFAIL=NFAIL+1
     BY CNDV*(CNDV-1)*LAMHYD / ( 2* (2*HYD-1) );

     IF (2*HYD-CNDV) > 0
```

```
        TRANTO CNDV=CNDV-1, HYD=HYD-1, NFAIL=NFAIL+1
        BY (2*HYD-CNDV)*2*CNDV*LAMHYD / ( 2* (2*HYD-1) );

    IF (2*HYD-CNDV) > 1
        TRANTO HYD=HYD-1, NFAIL=NFAIL+1
        BY (2*HYD-CNDV)*(2*HYD-CNDV-1)*LAMHYD / ( 2* (2*HYD-1) );
ENDIF;

        (* MID AREA NODE FAILURES *)

IF MIDNOD > 0 AND MIDREC = 0 AND GYREC = 0 AND ACCREC = 0 AND ONREC = 0 THEN
    IF ( MIDNOD=2 AND MIDDIU=4 ) OR ( MIDNOD=3 AND MIDDIU=3) THEN

              (* TEMPORARY EXHAUSTION FAILURE *)
       TRANTO MIDNOD=MIDNOD-1,
       MIDREC=2, NFAIL=NFAIL+1 BY LAMNOD;

              (* TEMPORARY EXHAUSTION AVOIDED *)
       TRANTO MIDNOD=MIDNOD-1,
       NFAIL=NFAIL+1 BY ( MIDNOD-1 )*LAMNOD;

    ELSE       (* NO TEMPORARY EXHAUSTION VULNERABILITY *)

       IF MIDNOD = 3 AND MIDDIU = 4 THEN

              (* MID NODE FAILURE MIGHT AFFECT DIU *)

         IF GYRO = 8 AND ACCEL = 8 THEN
            TRANTO MIDNOD=MIDNOD-1, MIDDIU=MIDDIU-2, GYRO=GYRO-4,
            ACCEL=ACCEL-4, NFAIL=NFAIL+1
            BY LAMNOD;

         ELSE
            IF GYRO < 8
               TRANTO MIDNOD=MIDNOD-1, MIDDIU=MIDDIU-2, GYRO=GYRO-3,
               ACCEL=ACCEL-4, NFAIL=NFAIL+1
               BY LAMNOD / 2;

            IF ACCEL < 8
               TRANTO MIDNOD=MIDNOD-1, MIDDIU=MIDDIU-2, GYRO=GYRO-4,
               ACCEL=ACCEL-3, NFAIL=NFAIL+1
               BY LAMNOD / 2;

            TRANTO MIDNOD=MIDNOD-1, MIDDIU=MIDDIU-2, GYRO=GYRO-4,
            ACCEL=ACCEL-4, NFAIL=NFAIL+1
            BY LAMNOD / 2;
         ENDIF;

         TRANTO MIDNOD=MIDNOD-1, NFAIL=NFAIL+1 BY 2*LAMNOD;

      ELSE
         IF MIDNOD = 2 AND ELMC = 2 THEN
                 (* ELECTRIC SUPPLY EXHAUSTION *)
            TRANTO MIDNOD=MIDNOD-1, MIDDIU=MIDDIU-1, GYRO=GYRO-2,
            ACCEL=ACCEL-2, NFAIL=NFAIL+1 BY MIDNOD* LAMNOD;
         ELSE
```

```
              (* MID NODE FAILURE CAN'T AFFECT DIU *)

         IF NFAIL=0 THEN
           TRANTO MIDNOD=MIDNOD-1, MIDREC=1, NFAIL=NFAIL+1
           BY MIDNOD*LAMNOD;
         ELSE
           TRANTO MIDNOD=MIDNOD-1, NFAIL=NFAIL+1 BY MIDNOD*LAMNOD;
         ENDIF;
       ENDIF;
     ENDIF;

   ENDIF;

ELSE          (* NEARLY COINCIDENT NETWORK-SENSOR FAILURES  *)

   IF MIDREC = 1
              (* NETWORK RECOVERY *)
     TRANTO MIDREC = 0 BY < NWMEAN, NWSTD >;

   IF GYREC = 1 OR ACCREC = 1
              (* COINCIDENT FAULT *)
     TRANTO MIDNOD=MIDNOD-1, MIDREC=MIDREC+1, NFAIL=NFAIL+1 BY
     MIDNOD *LAMNOD /2;
ENDIF;

        (** OTHER NODE FAILURES **)

IF MIDREC = 0 AND GYREC = 0 AND ACCREC = 0 AND ONREC = 0 THEN
   IF ( MIDNOD=2 AND MIDDIU=4 ) OR ( MIDNOD=3 AND MIDDIU=3) THEN
        (* TEMPORARY EXHAUSTION VULNERABLE *)
     TRANTO ONREC = 2, NFAIL=NFAIL+1 BY LAMON/2;
   ELSE
     IF NFAIL = 0 TRANTO ONREC = 1, NFAIL=NFAIL+1 BY 2* LAMON;

   ENDIF;
ELSE
   IF ONREC = 1     (* NW RECOVERY *)
     TRANTO ONREC = 0 BY < NWMEAN, NWSTD >;

   IF GYREC = 1 OR ACCREC = 1
       (* NEARLY COINCIDENT FAULTS  SENSOR - NETWORK *)
       TRANTO ONREC=1, NFAIL=NFAIL+1 BY LAMON;
ENDIF;

      (* MID DIU FAILURES *)

IF MIDDIU > 0 AND MIDREC = 0 AND GYREC = 0 AND ACCREC = 0 AND ONREC = 0 THEN
     TRANTO MIDDIU=MIDDIU-1, GYRO=GYRO-2, ACCEL=ACCEL-2,
     NFAIL=NFAIL+1 BY GYRO* (GYRO-1)* ACCEL* (ACCEL-1)*
     (LAMDIU + LAMPS)/ ( 2* (2*MIDDIU)* (2*MIDDIU-1)**2 );

     IF 2*MIDDIU - GYRO > 0
       TRANTO MIDDIU=MIDDIU-1, GYRO=GYRO-1, ACCEL=ACCEL-2,
       NFAIL=NFAIL+1 BY 2*GYRO* (2*MIDDIU-GYRO)* ACCEL* (ACCEL-1)*
       (LAMDIU + LAMPS)/ ( 2* (2*MIDDIU)* (2*MIDDIU-1)**2 );
```

```
        IF 2*MIDDIU - ACCEL > 0
           TRANTO MIDDIU=MIDDIU-1, GYRO=GYRO-2, ACCEL=ACCEL-1,
           NFAIL=NFAIL+1 BY GYRO* (GYRO-1)* 2*ACCEL* (2*MIDDIU-ACCEL)*
           (LAMDIU + LAMPS)/ ( 2* (2*MIDDIU)* (2*MIDDIU-1)**2 );

ENDIF;


          (**** ELECTRICAL POWER DISTRIBUTION FAILURES ***)

IF ELMC > 2 AND MIDREC = 0 AND GYREC = 0 AND ACCREC = 0 AND ONREC = 0 THEN

     IF ELMC = 4 AND NFAIL < 2
        TRANTO ELMC=ELMC-1, NFAIL=NFAIL+1 BY ELMC*LAMEL;

     IF ELMC = 3 THEN
        IF NFAIL = 1
                (* TRANSITION TO SINGLE NW OPERATION *)
           TRANTO MIDNOD=MIDNOD-2, MIDDIU=MIDDIU-2, GYRO=GYRO-4,
           ACCEL=ACCEL-4,
           ELMC=ELMC-1, NFAIL=NFAIL+1 BY LAMEL;

        IF MIDNOD = 3
                 (* TRANSITION TO ELECTRIC SOURCE EXHAUSTION *)
           TRANTO MIDNOD=MIDNOD-2, MIDDIU=MIDDIU-3, GYRO=GYRO-6,
           ACCEL=ACCEL-6,
           ELMC=ELMC-1, NFAIL=NFAIL+1 BY LAMEL/2;

        IF MIDDIU = 3
                 (* TRANSITION TO ELECTRIC SOURCE EXHAUSTION *)
           TRANTO MIDNOD=MIDNOD-2, MIDDIU=MIDDIU-2, GYRO=GYRO-4,
           ACCEL=ACCEL-4,
           ELMC=ELMC-1, NFAIL=NFAIL+1 BY LAMEL/2;

     ENDIF;
ENDIF;
```

```
(*** Wing and Tail Area Model - Safety Criteria - Network Option ***)
(*                          Group A                                 *)
(*   ____________________________________________________________  *)


SPACE  = ( NWREC:   0..2,     (* NETWORK RECOVERY INDICATOR      *)
           RWP:     0..4,     (* RW CHANNEL STATE                *)
           RWV:     0..4,     (* RW VALVE STATE                  *)
           LWP:     0..4,     (* LW CHANNEL STATE                *)
           LWV:     0..4,     (* LW VALVE STATE                  *)
           TLP:     0..4,     (* TL CHANNEL STATE                *)
           TLV:     0..4,     (* TL VALVE STATE                  *)
           HYD:     0..2,     (* HYDRAULIC SYSTEM STATE          *)
           ELMC:    0..4,     (* ELECTRIC SYSTEM SUPPLY STATE    *)
           ONREC:   0..2,     (* OTHER NODES FAILURE RECOVERY    *)
           NFAIL:   0..4);    (* NO. OF FAILED ELEMENTS          *)

START  = ( 0, 4, 4, 4, 4, 4, 4, 2, 4. 0, 0 );

PRUNEIF  NFAIL > 2;

DEATHIF  RWV = 0;

DEATHIF  LWV = 0;

DEATHIF  TLV = 0;

DEATHIF   NWREC + ONREC > 1;


LIST    = 2;
TIME    = 3.0;
PRUNE   = 1.0E-16;
ECHO    = 0;

LAMC    = 50.0E-6;            (* PROCESSOR FAILURE RATE           *)
LAMPOS  = 10.0E-6;            (* POSITION SENSOR FAILURE RATE     *)
LAMSD   = 20.0E-6;            (* SERVODRIVE GROUP FAILURE RATE    *)
LAMV    = 15.0E-6;            (* VALVE GROUP FAILURE RATE         *)
VJAM    = 3.3333E-5;          (* ACTUATOR JAM FAILURE FRACTION    *)

LAMNODH = 42.5E-6;            (* NODE FAILURE RATE - HARSH        *)
LAMDIUH = 37.5E-6;            (* DIU FAILURE RATE - HARSH         *)
LAMHYD  = 45.0E-6;            (* HYDRAULIC SYSTEM FAIL RATE       *)

LAMPSH  = 25.0E-6;            (* LOCAL POWER SUPPLY RATE - HARSH   *)
LAMEL   = 50.0E-6;            (* ELMC FAILURE RATE                *)
NWMEAN  = 3.0E-4;             (* NW RECOVERY TIME - MEAN          *)
NWSTD   = 1.0E-4;             (* NW RECOVERY TIME - STD DEV       *)
LAMON   = 68.0E-6;            (* OTHER NODE FAILURE RATE          *)

           (***  RW PROCESSOR  GROUP / DIU FAILURE TRANSITIONS  ***)

IF NWREC = 0 AND ONREC = 0 THEN
```

```
    IF RWP > 0  THEN
               TRANTO RWP=RWP-1, NFAIL=NFAIL+1  BY RWP* ( 2*(LAMC + LAMPOS)
               + (LAMDIUH + LAMPSH) );
    ENDIF;
 ENDIF;

              (*** RW VALVE GROUP FAILURES ***)

 IF RWV > 0 AND NWREC = 0 AND ONREC = 0 THEN
         IF NFAIL = 0
               TRANTO RWV=0, NFAIL=NFAIL+1 BY RWV*VJAM*LAMV;
         TRANTO RWV=RWV-1, NFAIL=NFAIL+1 BY RWV*(1.0-VJAM)*LAMV;
 ENDIF;


           (* RW NODE FAILURES *)

 IF NWREC = 0 AND ONREC = 0 THEN
    IF  RWP < 3 OR LWP < 3 OR TLP < 3 THEN

              (* TEMPORARY EXHAUSTION FAILURE *)
           TRANTO RWP=RWP-1, NWREC=2, NFAIL=NFAIL+1
           BY 2*LAMNODH /3;
    ELSE
              (* TEMPORARY EXHAUSTION NOT POSSIBLE *)

       IF RWP > 0 THEN
          IF NFAIL=0 THEN
             TRANTO RWP=RWP-1, NWREC=1, NFAIL=NFAIL+1
             BY RWP*LAMNODH;
          ELSE
             TRANTO RWP=RWP-1, NFAIL=NFAIL+1
             BY RWP*LAMNODH;
          ENDIF;

       ENDIF;
    ENDIF;

 ENDIF;


           (***  LW PROCESSOR  GROUP / DIU FAILURE TRANSITIONS  ***)

 IF NWREC = 0 AND ONREC = 0 THEN
    IF LWP > 0  THEN
               TRANTO LWP=LWP-1, NFAIL=NFAIL+1  BY LWP*( 2*(LAMC + LAMPOS)
               + (LAMDIUH + LAMPSH) );
    ENDIF;
 ENDIF;

             (*** LW VALVE GROUP FAILURES ***)

 IF LWV > 0 AND NWREC = 0 AND ONREC = 0 THEN
         IF NFAIL = 0
               TRANTO LWV=0, NFAIL=NFAIL+1 BY LWV*VJAM*LAMV;
         TRANTO LWV=LWV-1, NFAIL=NFAIL+1 BY LWV*(1.0-VJAM)*LAMV;
```

```
ENDIF;

           (* LW NODE FAILURES *)

IF NWREC = 0 AND ONREC = 0 THEN
   IF  RWP < 3 OR LWP < 3 OR TLP < 3 THEN

              (* TEMPORARY EXHAUSTION FAILURE *)
          TRANTO LWP=LWP-1, NWREC=2, NFAIL=NFAIL+1
          BY 2*LAMNODH /3;
   ELSE
           (* TEMPORARY EXHAUSTION NOT POSSIBLE *)

      IF LWP > 0 THEN
         IF NFAIL=0 THEN
            TRANTO LWP=LWP-1, NWREC=1, NFAIL=NFAIL+1
            BY LWP*LAMNODH;
         ELSE
            TRANTO LWP=LWP-1, NFAIL=NFAIL+1
            BY LWP*LAMNODH;
         ENDIF;

      ENDIF;
   ENDIF;

ENDIF;

           (***  TL PROCESSOR  GROUP / DIU FAILURE TRANSITIONS  ***)

IF NWREC = 0 AND ONREC = 0 THEN
   IF TLP > 0  THEN
              TRANTO TLP=TLP-1, NFAIL=NFAIL+1  BY TLP* ( 2*(LAMC + LAMPOS)
              + (LAMDIUH + LAMPSH) );
   ENDIF;
ENDIF;

             (*** TL VALVE GROUP FAILURES ***)

IF TLV > 0 AND NWREC = 0 AND ONREC = 0 THEN
        IF NFAIL = 0
              TRANTO TLV=0, NFAIL=NFAIL+1 BY TLV*VJAM*LAMV;
        TRANTO TLV=TLV-1, NFAIL=NFAIL+1 BY TLV*(1.0-VJAM)*LAMV;
ENDIF;

           (* TL NODE FAILURES *)

IF NWREC = 0 AND ONREC = 0 THEN
   IF  RWP < 3 OR LWP < 3 OR TLP < 3 THEN

              (* TEMPORARY EXHAUSTION FAILURE *)
          TRANTO TLP=TLP-1, NWREC=2, NFAIL=NFAIL+1
          BY 2*LAMNODH /3;
   ELSE
              (* TEMPORARY EXHAUSTION NOT POSSIBLE *)
        IF TLP > 0 THEN
           IF NFAIL=0 THEN
              TRANTO TLP=TLP-1, NWREC=1, NFAIL=NFAIL+1
              BY TLP*LAMNODH;
           ELSE
              TRANTO TLP=TLP-1, NFAIL=NFAIL+1
              BY TLP*LAMNODH;
           ENDIF;

        ENDIF;
     ENDIF;

  ENDIF;

         (* NEARLY COINCIDENT FAULTS - RECOVERIES *)

     IF NWREC = 1  THEN
         (* NETWORK RECOVERY *)
         TRANTO NWREC = 0 BY < NWMEAN, NWSTD >;

               (* SAME NODE SET *)
         TRANTO NWREC=NWREC+1, NFAIL=NFAIL+1
         BY 6*LAMNODH;
     ENDIF;

     IF ONREC = 1 THEN     (* OTHER NODE SET *)
         TRANTO NWREC=NWREC+1, NFAIL=NFAIL+1
         BY 6*LAMNODH;
     ENDIF;


           (** OTHER NODE FAILURES **)

IF NWREC = 0 AND ONREC = 0 THEN
   IF  RWP < 3 OR LWP < 3 OR TLP < 3 THEN
           (* VULNERABLE TO TEMPORARY EXHAUSTION *)
      TRANTO ONREC=2, NFAIL=NFAIL+1 BY LAMON/3;
   ELSE
      IF NFAIL = 0
         TRANTO ONREC = 1, NFAIL=NFAIL+1 BY 2*LAMON;
   ENDIF;
ELSE       (* NEARLY COINCIDENT NW FAULT *)

   IF NWREC > 0
      TRANTO ONREC=1, NFAIL=NFAIL+1 BY LAMON;

   IF ONREC > 0 THEN
             (* NETWORK RECOVERY *)
      TRANTO ONREC = 0 BY < NWMEAN, NWSTD >;
             (* NCF - OTHER NODES *)
      TRANTO ONREC=ONREC+1, NFAIL=NFAIL+1 BY LAMON;
   ENDIF;

ENDIF;

          (* HYDRAULIC SYSTEM FAILURES *)
```

```
IF HYD > 0 AND NWREC = 0 AND ONREC = 0
     AND (HYD-NFAIL >= 0 ) THEN
     TRANTO RWV=RWV-2, LWV=LWV-2, TLV=TLV-2, HYD=HYD-1, NFAIL=NFAIL+1
     BY RWV* (RWV-1) *LWV* (LWV-1) *TLV* (TLV-1) *LAMHYD /
     ( 2* (2*HYD) **2 * (2*HYD-1) **3 );

ENDIF;

             (**** ELECTRIC POWER DISTRIBUTION ***)

IF ELMC > 0 AND NWREC = 0 AND ONREC = 0 THEN
     TRANTO RWP=RWP-1, LWP=LWP-1, TLP=TLP-1, ELMC=ELMC-1,
     NFAIL=NFAIL+1 BY RWP* LWP* TLP* LAMEL / ( ELMC**2);

ENDIF;
```

```
(***       Air Model - Safety Criteria - Network Option          ***)
(*                    Group B                                       *)
(*                                                                  *)

SPACE    = (  AIRNOD:     0..4,    (* AIR   NODE                        *)
              AIRREC:     0..3,    (* NETWORK RECOVERY INDICATOR        *)
              AIRDIU:     0..4,    (* FORWARD DIU STATE INDICATOR       *)
              AOA:        0..4,    (* AOA SENSOR                        *)
              AOAREC:     0..2,    (* AOA RECOVERY INDICATOR            *)
              AOS:        0..4,    (* AOS SENSOR                        *)
              AOSREC:     0..2,    (* AOS RECOVERY INDICATOR            *)
              THROT:      0..4,    (* THROTTLE POSITION SENSOR          *)
              THROTREC:   0..1,    (* THROTTLE FAILURE INDICATOR        *)
              FTP:        0..4,    (* FTP CHANNEL STATUS                *)
              FTPREC:     0..2,    (* FTP CHANNEL RECOVERY IND.         *)
              ROOT:       0..4,    (* ROOT LINK STATUS                  *)
              ELMC:       0..4,    (* ELECTRIC POWER STATE              *)
              ONREC:      0..3,    (* OTHER NODES IN A SINGLE NETWORK   *)
              NFAIL:      0..4);   (* NO. OF FAILED ELEMENTS            *)

START    = ( 4, 0, 4, 4, 0, 4, 0, 4, 0, 4, 0, 4, 4, 0, 0 );

PRUNEIF  NFAIL > 2;

DEATHIF   FTP < 2;

DEATHIF   FTPREC > 1;

DEATHIF   AOAREC > 1 OR AOSREC > 1;

DEATHIF   AOA-AOAREC < 1 OR AOS-AOSREC < 1;

DEATHIF   THROT-THROTREC < 1;

DEATHIF   AIRREC = 3 OR ONREC = 3;

DEATHIF   AIRREC + ONREC > 1;

DEATHIF   AIRREC + AOAREC + AOSREC > 1;

DEATHIF   ONREC + AOAREC + AOSREC > 1;

DEATHIF   ROOT < 3 AND ( AOAREC > 0 OR AOSREC > 0 OR THROTREC > 0
                         OR AIRREC > 0 OR ONREC > 0 );

LIST  = 3;
TIME  = 3.0;
PRUNE = 1.0E-16;
ECHO  = 0;

LAMANG   = 33.0E-6;        (* FLOW ANGLE SENSOR FAILURE RATE          *)
LAMPRESS= 20.0E-6;         (* PRESSURE SENSOR FAILURE RATE            *)
LAMPOS   = 10.0E-6;        (* POSITION SENSOR FAILURE RATE            *)
```

```
LAMNOD  = 17.0E-6;        (* NODE FAILURE RATE - CONDITIONED            *)
LAMDIU  = 15.0E-6;        (* DIU FAILURE RATE - CONDITIONED             *)
RECMEAN = 3.0E-4;         (* RECOVERY TIME MEAN                         *)
RECSTD  = 1.0E-4;         (* RECOVERY TIME STD DEV                      *)
NWMEAN  = 3.0E-4;         (* NW RECOVERY TIME MEAN                      *)
NWSTD   = 1.0E-4;         (* NW RECOVERY TIME STD DEV                   *)

LAMPS    =  10.0E-6;      (* LOCAL POWER SUPPLY RATE - CONDITIONED      *)
LAMEL    = 50.0E-6;       (* ELECTRIC POWER CENTER FAILURE RATE         *)
LAMFTP   = 200.0E-6;      (* FTP CHANNEL FAILURE RATE                   *)
LAMROOT  =  15.0E-6;      (* FTP NETWORK INTERFACE FAILURE RATE         *)
FTPMEAN  =  5.0E-6;       (* FTP RECOVERY TIME MEAN                     *)
FTPSTD   =  1.0E-6;       (* FTP RECOVERY TIME STD DEV                  *)

LAMON    = 255.0E-6;      (* OTHER NODES ON SINGLE NW FAIL RATE         *)


         (* AOA SENSOR FAILURES AND RECOVERIES *)

IF AOA > 0 AND AOAREC = 0  AND AOSREC = 0
     AND FTPREC = 0 AND AIRREC = 0 AND ONREC = 0 THEN
     IF NFAIL = 0 OR AOA = 2 THEN
        TRANTO AOA = AOA-1, AOAREC=AOAREC+1, NFAIL=NFAIL+1
        BY AOA*LAMANG;
     ELSE
        IF ROOT = 2 THEN
              (* SINGLE NETWORK SUSCEPTIBILITY *)
           TRANTO  AOA=AOA-1, AOAREC=AOAREC+1, NFAIL=NFAIL+1
           BY (2/3)*LAMANG;
           TRANTO AOA=AOA-1, NFAIL=NFAIL+1
           BY  ((AOA - 2) + 4/3 )*LAMANG;
        ELSE
           TRANTO AOA = AOA-1, NFAIL=NFAIL+1
           BY AOA*LAMANG;
        ENDIF;

     ENDIF;

ELSE        (* NEARLY COINCIDENT SENSOR-NETWORK *)

     IF AIRREC > 0      (* SAME NODE SET *)
        TRANTO AOA = AOA-1, AOAREC=AOAREC+1, NFAIL=NFAIL+1
        BY (AOA+1)*LAMANG /2;

     IF ONREC > 0      (* OTHER NODES *)
        TRANTO AOA = AOA-1, AOAREC=AOAREC+1, NFAIL=NFAIL+1
        BY AOA*LAMANG /2;

     IF AOAREC > 0 THEN  (* RECOVERY OR SYSTEM LOSS *)
        TRANTO AOAREC = 0 BY < RECMEAN, RECSTD >;
        TRANTO AOA = AOA-1, AOAREC=AOAREC+1, NFAIL=NFAIL+1
        BY AOA*LAMANG;
     ENDIF;

ENDIF;
```

```
          (* AOS SENSOR FAILURES AND RECOVERIES *)

IF AOS > 0 AND AOAREC = 0  AND AOSREC = 0
     AND FTPREC = 0 AND AIRREC = 0 AND ONREC = 0 THEN
     IF NFAIL = 0 OR AOS = 2 THEN
        TRANTO AOS = AOS-1, AOSREC=AOSREC+1, NFAIL=NFAIL+1
        BY AOS*LAMANG;
     ELSE
        IF ROOT = 2 THEN
              (* SINGLE NETWORK SUSCEPTIBILITY *)
           TRANTO  AOS=AOS-1, AOSREC=AOSREC+1, NFAIL=NFAIL+1
           BY (2/3)*LAMANG;
           TRANTO AOS=AOS-1, NFAIL=NFAIL+1
           BY  ((AOS - 2) + 4/3 )*LAMANG;
        ELSE
           TRANTO AOS = AOS-1, NFAIL=NFAIL+1
           BY AOS*LAMANG;
        ENDIF;

     ENDIF;

ELSE         (* NEARLY COINCIDENT SENSOR-NETWORK *)

     IF AIRREC > 0      (* SAME NODE SET *)
        TRANTO AOS = AOS-1, AOSREC=AOSREC+1, NFAIL=NFAIL+1
        BY (AOS+1)*LAMANG /2;

     IF ONREC > 0      (* OTHER NODES *)
        TRANTO AOS = AOS-1, AOSREC=AOSREC+1, NFAIL=NFAIL+1
        BY AOS*LAMANG /2;

     IF AOSREC > 0 THEN  (* RECOVERY OR SYSTEM LOSS *)
        TRANTO AOSREC = 0 BY < RECMEAN, RECSTD >;
        TRANTO AOS = AOS-1, AOSREC=AOSREC+1, NFAIL=NFAIL+1
        BY AOS*LAMANG;
     ENDIF;

ENDIF;

         (* THROTTLE SENSOR FAILURES AND RECOVERIES *)

IF THROT > 0 AND AOAREC = 0  AND AOSREC = 0
     AND FTPREC = 0 AND AIRREC = 0 AND ONREC = 0 THEN
     IF THROT = 2 THEN
        TRANTO THROT=THROT-1, THROTREC=THROTREC+1, NFAIL=NFAIL+1
           BY THROT*LAMPOS;
     ELSE
        IF ROOT = 2 THEN
              (* SINGLE NETWORK SUSCEPTIBILITY *)
           TRANTO  THROT=THROT-1, THROTREC=THROTREC+1, NFAIL=NFAIL+1
           BY (2/3)*LAMPOS;
           TRANTO THROT=THROT-1, NFAIL=NFAIL+1
           BY  ((THROT - 2) + 4/3 )*LAMPOS;
        ELSE
           TRANTO THROT = THROT-1, NFAIL=NFAIL+1
           BY THROT*LAMPOS;
        ENDIF;
     ENDIF;
ENDIF;

          (* AIR AREA NODE FAILURES *)

IF AIRNOD > 0 AND AIRREC = 0 AND
   AOAREC = 0 AND AOSREC = 0 AND FTPREC = 0 AND ONREC = 0 THEN
     IF AOA < 3 OR AOS < 3 THEN

                (* TEMPORARY EXHAUSTION FAILURE *)
        TRANTO AIRNOD=AIRNOD-1, AIRDIU=AIRDIU-1,
        AOA=AOA-1, AOS=AOS-1, THROT=THROT-1, ROOT=ROOT-1,
        AIRREC=2, NFAIL=NFAIL+1 BY (2/3)*LAMNOD;

                (* TEMPORARY EXHAUSTION AVOIDED *)
        TRANTO AIRNOD=AIRNOD-1, AIRDIU=AIRDIU-1,
        AOA=AOA-1, AOS=AOS-1, THROT=THROT-1, ROOT=ROOT-1,
        NFAIL=NFAIL+1 BY (( AIRNOD-2) + 4/3)*LAMNOD;

      ELSE       (* NO TEMPORARY EXHAUSTION VULNERABILITY *)

         IF ROOT = 2 THEN   (* SINGLE NETWORK VULNERABILITY *)
            TRANTO AIRNOD=AIRNOD-1, AIRDIU=AIRDIU-1,
            AOA=AOA-1, AOS=AOS-1, THROT=THROT-1, ROOT=ROOT-1,
            AIRREC=AIRREC+1, NFAIL=NFAIL+1 BY (2/3)*LAMNOD;

            TRANTO AIRNOD=AIRNOD-1, AIRDIU=AIRDIU-1,
            AOA=AOA-1, AOS=AOS-1, THROT=THROT-1, ROOT=ROOT-1,
            NFAIL=NFAIL+1 BY (( AIRNOD-2) + 4/3)*LAMNOD;

         ELSE    (* NO SINGLE NETWORK VULNERABILITY *)

                  (* IN THIS MODEL FAULTS ARE ON SEPARATE "CHANNELS" *)
              (* FORWARD NODE FAILURE AFFECTS ROOT LINK AND DIU *)

                IF NFAIL = 0 THEN
                   TRANTO AIRNOD=AIRNOD-1, AIRDIU=AIRDIU-1,
                   AOA=AOA-1, AOS=AOS-1, THROT=THROT-1, ROOT=ROOT-1,
                   AIRREC=AIRREC+1, NFAIL=NFAIL+1 BY AIRNOD* LAMNOD;
                ELSE
                   TRANTO AIRNOD=AIRNOD-1, AIRDIU=AIRDIU-1,
                   AOA=AOA-1, AOS=AOS-1, THROT=THROT-1, ROOT=ROOT-1,
                   NFAIL=NFAIL+1 BY ROOT* AOA*AOS* THROT* LAMNOD/
                   ( (AIRNOD+ELMC-4)* AIRDIU**2 );
                ENDIF;

          ENDIF;

       ENDIF;

  ELSE            (* NEARLY COINCIDENT NETWORK-SENSOR FAILURES  *)

       IF AIRREC = 1 THEN   (* SAME NODE SET *)
                       (* NETWORK RECOVERY *)
          TRANTO AIRREC = 0 BY < NWMEAN, NWSTD >;
          TRANTO AIRREC = 3, NFAIL=NFAIL+1 BY 2*LAMNOD;
       ENDIF;

       IF ONREC = 1
          TRANTO AIRREC = 3, NFAIL=NFAIL+1 BY 2*LAMNOD;

       IF AOAREC = 1 OR AOSREC = 1
          TRANTO AIRNOD=AIRNOD-1, AIRREC=AIRREC+1, NFAIL=NFAIL+1 BY
          AIRNOD*LAMNOD /2;

 ENDIF;

          (* OTHER NODE FAILURES *)

 IF  AIRREC = 0 AND
    AOAREC = 0 AND AOSREC = 0 AND FTPREC = 0 AND ONREC = 0 THEN
      IF AOA < 3 OR AOS < 3 THEN
              (* TEMPORARY EXHAUSTION VULNERABLE *)

         TRANTO ONREC=2, NFAIL=NFAIL+1 BY LAMON/3;
      ELSE
         IF ROOT = 2 THEN   (* SINGLE NW VULNERABLE *)

            TRANTO ONREC=1, NFAIL=NFAIL+1 BY LAMON/3;
         ELSE
            IF NFAIL = 0  TRANTO ONREC=1, NFAIL=NFAIL+1 BY
               2*LAMON;
         ENDIF;
      ENDIF;
ELSE
         (* NEARLY COINCIDENT FAILURES- RECOVERIES *)
      IF ONREC = 1 THEN
         TRANTO ONREC = 0 BY < NWMEAN, NWSTD >;
         TRANTO ONREC = 3, NFAIL=NFAIL+1 BY LAMON;
      ENDIF;

      IF AIRREC = 1
         TRANTO ONREC = 3, NFAIL=NFAIL+1 BY LAMON;

      IF AOAREC=1 OR AOSREC = 1
         TRANTO ONREC=1, NFAIL=NFAIL+1 BY LAMON;
ENDIF;


         (* FORWARD DIU FAILURES *)

IF AIRDIU > 0 AND AIRREC = 0 AND
   AOAREC = 0 AND AOSREC = 0 AND FTPREC = 0 AND ONREC = 0 THEN
        TRANTO AIRDIU=AIRDIU-1, AOA=AOA-1, AOS=AOS-1, THROT=THROT-1,
        NFAIL=NFAIL+1 BY AOA*AOS* THROT*( LAMDIU + LAMPS ) / AIRDIU**2;

ENDIF;

       (* FTP CHANNEL FAILURES *)

IF FTP > 0 AND AOAREC = 0 AND AOSREC = 0
```

```
   AND AIRREC = 0 AND FTPREC = 0 AND ONREC = 0 THEN
      IF NFAIL = 0 THEN
         TRANTO FTP=FTP-1, FTPREC=FTPREC+1, ROOT=ROOT-1, NFAIL=NFAIL+1
         BY FTP*LAMFTP;
      ELSE
         TRANTO FTP=FTP-1, ROOT=ROOT-1, NFAIL=NFAIL+1
         BY ROOT*LAMFTP;
      ENDIF;

ELSE
   IF FTPREC > 0 THEN
                   (* FTP CHANNEL RECOVERY *)
      TRANTO FTPREC=FTPREC-1 BY < FTPMEAN, FTPSTD >;
                    (* COINCIDENT FAULT *)
      TRANTO FTP=FTP-1, FTPREC=FTPREC+1, ROOT=ROOT-1, NFAIL=NFAIL+1
      BY FTP*LAMFTP;
   ENDIF;

ENDIF;


           (* NETWORK INTERFACE FAILURES *)

IF ROOT > 2 AND AOAREC = 0 AND AOSREC = 0
       AND FTPREC = 0 AND AIRREC = 0 AND ONREC = 0
           TRANTO ROOT=ROOT-1, NFAIL=NFAIL+1 BY ROOT*LAMROOT;


            (*** ELECTRIC POWER DISTRIBUTION FAILURES ***)

IF ELMC > 0 AND AOAREC = 0 AND AOSREC = 0
     AND FTPREC = 0 AND AIRREC = 0  AND ONREC = 0 THEN
                    (* ELMC FAILURE AFFECTS DIU, FTP AND ROOT LINK *)

           (* IN THIS MODEL FAULTS ON DIFFERENT "CHANNELS" *)

               TRANTO AIRDIU=AIRDIU-1,
               AOA=AOA-1, AOS=AOS-1, FTP=FTP-1, THROT=THROT-1,
               ROOT=ROOT-1, ELMC=ELMC-1,
               NFAIL=NFAIL+1 BY ROOT* AOA* AOS* THROT*
               LAMEL / ( (AIRNOD+ELMC-4)* AIRDIU**2 );

ENDIF;
```

```
(***    Forward Area Model - Safety Criteria - Bus Option    ***)
(*                   Group A                                   *)
(*      ___________________________________________________    *)

SPACE    = (  FWDIU:    0..4,     (* FORWARD DIU STATE INDICATOR     *)
              PITSTK:   0..4,     (* PITCH COMMAND SENSOR            *)
              PITREC:   0..2,     (* PITCH SENSOR RECOVERY INDICATOR *)
              ROLSTK:   0..4,     (* ROLL COMMAND SENSOR             *)
              ROLREC:   0..2,     (* ROLL SENSOR RECOVERY INDICATOR  *)
              YAWPED:   0..4,     (* YAW COMMAND SENSOR              *)
              YAWREC:   0..2,     (* YAW SENSOR RECOVERY INDICATOR   *)
              FTP:      0..4,     (* FTP CHANNEL STATUS              *)
              FTPREC:   0..2,     (* FTP CHANNEL RECOVERY IND.       *)
              ROOT:     0..4,     (* ROOT LINK STATUS                *)
              NFAIL:    0..4);    (* NO. OF FAILED ELEMENTS          *)

START  = ( 4, 4, 0, 4, 0, 4, 0, 4, 0, 4, 0 );

PRUNEIF  NFAIL > 2;

DEATHIF   PITREC > 1 OR ROLREC > 1 OR
          YAWREC > 1;

DEATHIF   PITSTK-PITREC < 1 OR ROLSTK-ROLREC < 1 OR YAWPED-YAWREC < 1;

DEATHIF   FTP - FTPREC < 1;


LIST   = 2;
TIME   = 3.0;
PRUNE  = 1.0E-16;
ECHO   = 0;

LAMPOS   = 10.0E-6;        (* POSITION SENSOR FAILURE RATE    *)
LAMNOD   = 15.0E-6;        (* NODE FAILURE RATE - CONDITIONED *)
LAMDIU   = 15.0E-6;        (* DIU FAILURE RATE - CONDITIONED  *)
POSMEAN  = 3.0E-4;         (* RECOVERY TIME MEAN            *)
POSSTD   = 1.0E-4;         (* RECOVERY TIME STD DEV         *)
NWMEAN   = 3.0E-4;         (* NW RECOVERY TIME MEAN           *)
NWSTD    = 1.0E-4;         (* NW RECOVERY TIME STD DEV        *)

LAMPS     =  10.0E-6;      (* LOCAL POWER SUPPLY RATE - CONDITIONED   *)
LAMEL     = 50.0E-6;       (* ELECTRIC POWER CENTER FAILURE RATE      *)
LAMFTP    = 200.0E-6;      (* FTP CHANNEL FAILURE RATE                *)
LAMROOT   =  15.0E-6;      (* FTP NETWORK INTERFACE FAILURE RATE      *)
FTPMEAN   =  5.0E-6;       (* FTP RECOVERY TIME MEAN                  *)
FTPSTD    =  1.0E-6;       (* FTP RECOVERY TIME STD DEV               *)

LAMGYRO   = 50.0E-6;       (* GYRO FAILURE RATE                       *)
LAMACC    = 30.0E-6;       (* ACCELEROMETER FAILURE RATE              *)
LAMOD     = 127.5E-6;      (* OTHER DIUS ON SINGLE BUS                *)
(*  DIUACT    = 0.10;    *)      (* DIU ACTIVE FAILURE FRACTION              *)
```

```
         (* PITCH COMMAND SENSOR FAILURES AND RECOVERIES *)

IF PITSTK > 0 AND PITREC = 0 AND ROLREC = 0 AND YAWREC = 0
     AND FTPREC = 0 THEN

     IF NFAIL = 0 OR PITSTK = 2 THEN
        TRANTO PITSTK = PITSTK-1, PITREC=PITREC+1, NFAIL=NFAIL+1
        BY PITSTK*LAMPOS;
     ELSE
           TRANTO PITSTK = PITSTK-1, NFAIL=NFAIL+1
           BY PITSTK*LAMPOS;
     ENDIF;

ELSE         (* NEARLY COINCIDENT FAILURES *)

     IF PITREC > 0 THEN   (* RECOVERY OR SYSTEM LOSS *)
        TRANTO PITREC = 0 BY < POSMEAN, POSSTD >;
        TRANTO PITSTK = PITSTK-1, PITREC=PITREC+1, NFAIL=NFAIL+1
        BY PITSTK*LAMPOS;
     ENDIF;

ENDIF;


         (* ROLL COMMAND SENSOR FAILURES AND RECOVERIES *)

IF ROLSTK > 0 AND PITREC = 0 AND ROLREC = 0 AND YAWREC = 0
     AND FTPREC = 0 THEN

     IF NFAIL = 0 OR ROLSTK = 2 THEN
        TRANTO ROLSTK = ROLSTK-1, ROLREC=ROLREC+1, NFAIL=NFAIL+1
        BY ROLSTK*LAMPOS;
     ELSE
           TRANTO ROLSTK = ROLSTK-1, NFAIL=NFAIL+1
           BY ROLSTK*LAMPOS;
     ENDIF;

ELSE         (* NEARLY COINCIDENT FAILURES *)

     IF ROLREC > 0 THEN   (* RECOVERY OR SYSTEM LOSS *)
        TRANTO ROLREC = 0 BY < POSMEAN, POSSTD >;
        TRANTO ROLSTK = ROLSTK-1, ROLREC=ROLREC+1, NFAIL=NFAIL+1
        BY ROLSTK*LAMPOS;
     ENDIF;

ENDIF;

         (* YAW COMMAND SENSOR FAILURES AND RECOVERIES *)

IF YAWPED > 0 AND PITREC = 0 AND ROLREC = 0 AND YAWREC = 0
     AND FTPREC = 0 THEN

     IF NFAIL = 0 OR YAWPED = 2 THEN
        TRANTO YAWPED = YAWPED-1, YAWREC=YAWREC+1, NFAIL=NFAIL+1
        BY YAWPED*LAMPOS;
     ELSE
```

```
            TRANTO YAWPED = YAWPED-1, NFAIL=NFAIL+1
            BY YAWPED*LAMPOS;
      ENDIF;

ELSE          (* NEARLY COINCIDENT FAILURES *)

      IF YAWREC > 0 THEN  (* RECOVERY OR SYSTEM LOSS *)
         TRANTO YAWREC = 0 BY < POSMEAN, POSSTD >;
         TRANTO YAWPED = YAWPED-1, YAWREC=YAWREC+1, NFAIL=NFAIL+1
         BY YAWPED*LAMPOS;
      ENDIF;

ENDIF;


         (*             *** FORWARD DIU FAILURES ***                       *)
         (*   PASSIVE FAILURE TRANSITIONS LISTED HERE.  ACTIVE             *)
         (*   FAILURE TRANSITIONS IN BUS AND INTERFACE FAILURE AREA.       *)

IF FWDIU > 0 AND PITREC = 0 AND ROLREC = 0 AND
   YAWREC = 0 AND FTPREC = 0 THEN
        TRANTO FWDIU=FWDIU-1, PITSTK=PITSTK-1, ROLSTK=ROLSTK-1, YAWPED=
        YAWPED-1, NFAIL=NFAIL+1 BY PITSTK* ROLSTK* YAWPED* ( LAMDIU*
        (1.0-DIUACT) + LAMPS ) / FWDIU**2;

        IF FWDIU > PITSTK    (* AFFECTED DIU HAS FAILED PITSTK *)
           TRANTO FWDIU=FWDIU-1, ROLSTK=ROLSTK-1, YAWPED=
           YAWPED-1, NFAIL=NFAIL+1 BY (FWDIU-PITSTK)*ROLSTK*
           YAWPED*( LAMDIU*(1.0-DIUACT) + LAMPS ) / FWDIU**2;

        IF FWDIU > ROLSTK    (* AFFECTED DIU HAS FAILED ROLSTK *)
           TRANTO FWDIU=FWDIU-1, PITSTK=PITSTK-1, YAWPED=
           YAWPED-1, NFAIL=NFAIL+1 BY PITSTK*(FWDIU-ROLSTK)*
           YAWPED*( LAMDIU*(1.0-DIUACT) + LAMPS ) / FWDIU**2;

        IF FWDIU > YAWPED    (* AFFECTED DIU HAS FAILED YAWPED *)
           TRANTO FWDIU=FWDIU-1, PITSTK=PITSTK-1, ROLSTK=ROLSTK-1,
           NFAIL=NFAIL+1 BY PITSTK*ROLSTK*(FWDIU-YAWPED)*
           ( LAMDIU*(1.0-DIUACT) + LAMPS ) /  FWDIU**2;
ENDIF;

       (* FTP CHANNEL FAILURES *)

IF FTP > 0 AND PITREC = 0 AND ROLREC = 0 AND YAWREC = 0
   AND FTPREC = 0 THEN

      IF NFAIL = 0 OR FTP = 2 THEN
         TRANTO FWDIU=FWDIU-1, PITSTK=PITSTK-1, ROLSTK=ROLSTK-1, YAWPED=
         YAWPED-1, FTP=FTP-1, FTPREC=FTPREC+1, ROOT=ROOT-1, NFAIL=NFAIL+1
         BY FTP*LAMFTP;
      ELSE
           (* FAILURE AFFECTS DIU, FTP AND ROOT LINK *)

         TRANTO FWDIU=FWDIU-1, PITSTK=PITSTK-1,
         ROLSTK=ROLSTK-1, YAWPED=YAWPED-1, FTP=FTP-1, ROOT=ROOT-1,
         NFAIL=NFAIL+1 BY PITSTK*ROLSTK*YAWPED*
```

```
          LAMFTP / ( FWDIU**2 );

          IF FWDIU > PITSTK    (* AFFECTED DIU HAS FAILED PITSTK *)
               TRANTO FWDIU=FWDIU-1,
               ROLSTK=ROLSTK-1, YAWPED=YAWPED-1, FTP=FTP-1, ROOT=ROOT-1,
               NFAIL=NFAIL+1
               BY (FWDIU-PITSTK)*ROLSTK*YAWPED* LAMFTP /
               ( FWDIU**2 );

          IF FWDIU > ROLSTK     (* AFFECTED DIU HAS FAILED ROLSTK *)
               TRANTO FWDIU=FWDIU-1, PITSTK=PITSTK-1,
               YAWPED=YAWPED-1, FTP=FTP-1, ROOT=ROOT-1,
               NFAIL=NFAIL+1
               BY PITSTK*(FWDIU-ROLSTK)*YAWPED* LAMFTP /
               ( FWDIU**2 );

          IF FWDIU > YAWPED      (* AFFECTED DIU HAS FAILED YAWPED *)
               TRANTO FWDIU=FWDIU-1, PITSTK=PITSTK-1,
               ROLSTK=ROLSTK-1, FTP=FTP-1, ROOT=ROOT-1,
               NFAIL=NFAIL+1
               BY PITSTK* ROLSTK*(FWDIU-YAWPED)*
               LAMFTP /  ( FWDIU**2 );

          IF ROOT > FWDIU
                (* AFFECTED CHANNEL HAS FAILED DIU *)

               TRANTO FTP=FTP-1, ROOT=ROOT-1, NFAIL=NFAIL+1
               BY (ROOT-FWDIU)* LAMFTP;

          IF FTP > ROOT (* AFFECTED CHANNEL HAS FAILED ROOT *)

               TRANTO FTP=FTP-1, NFAIL=NFAIL+1
               BY (FTP-ROOT)* LAMFTP;
       ENDIF;
ELSE

   IF FTPREC > 0 THEN
                  (* FTP CHANNEL RECOVERY *)
      TRANTO FTPREC=FTPREC-1 BY < FTPMEAN, FTPSTD >;
                   (* COINCIDENT FAULT *)
      TRANTO FTP=FTP-1, FTPREC=FTPREC+1, ROOT=ROOT-1, NFAIL=NFAIL+1
      BY FTP*LAMFTP;
   ENDIF;

ENDIF;


             (* BUS AND NETWORK INTERFACE FAILURES *)

IF ROOT > 0 AND PITREC = 0 AND ROLREC = 0 AND YAWREC = 0 AND FTPREC = 0
     THEN
            (* FAILURE AFFECTS DIU AND ROOT LINK *)

        TRANTO FWDIU=FWDIU-1, PITSTK=PITSTK-1,
        ROLSTK=ROLSTK-1, YAWPED=YAWPED-1, ROOT=ROOT-1,
        NFAIL=NFAIL+1 BY PITSTK*ROLSTK*YAWPED*
```

```
          ( LAMROOT + (LAMDIU + LAMOD) * DIUACT ) / ( FWDIU**2 );

          IF FWDIU > PITSTK    (* AFFECTED DIU HAS FAILED PITSTK *)
               TRANTO FWDIU=FWDIU-1,
               ROLSTK=ROLSTK-1, YAWPED=YAWPED-1, ROOT=ROOT-1,
               NFAIL=NFAIL+1
               BY (FWDIU-PITSTK)* ROLSTK* YAWPED* ( LAMROOT +
               (LAMDIU + LAMOD) * DIUACT ) / ( FWDIU**2 );

          IF FWDIU > ROLSTK     (* AFFECTED DIU HAS FAILED ROLSTK *)
               TRANTO FWDIU=FWDIU-1, PITSTK=PITSTK-1,
               YAWPED=YAWPED-1, ROOT=ROOT-1,
               NFAIL=NFAIL+1
               BY PITSTK* (FWDIU-ROLSTK)* YAWPED* ( LAMROOT +
               (LAMDIU + LAMOD) * DIUACT ) / ( FWDIU**2 );

          IF FWDIU > YAWPED      (* AFFECTED DIU HAS FAILED YAWPED *)
               TRANTO FWDIU=FWDIU-1, PITSTK=PITSTK-1,
               ROLSTK=ROLSTK-1, ROOT=ROOT-1,
               NFAIL=NFAIL+1
               BY PITSTK* ROLSTK* (FWDIU-YAWPED)*
               ( LAMROOT + (LAMDIU + LAMOD) * DIUACT ) /  ( FWDIU**2 );

          IF ROOT > FWDIU
                (* AFFECTED CHANNEL HAS FAILED DIU *)

               TRANTO ROOT=ROOT-1, NFAIL=NFAIL+1
               BY (ROOT-FWDIU)* ( LAMROOT + LAMOD*DIUACT );
ENDIF;


           (*** ELECTRIC POWER DISTRIBUTION FAILURES ***)

IF FTP > 0 AND PITREC = 0 AND ROLREC = 0 AND YAWREC = 0 AND FTPREC = 0
     THEN
           (* FAILURE AFFECTS DIU, FTP AND ROOT LINK *)

         TRANTO FWDIU=FWDIU-1, PITSTK=PITSTK-1,
         ROLSTK=ROLSTK-1, YAWPED=YAWPED-1, FTP=FTP-1, ROOT=ROOT-1,
         NFAIL=NFAIL+1 BY PITSTK*ROLSTK*YAWPED*
         LAMEL / ( FWDIU**2 );

         IF FWDIU > PITSTK    (* AFFECTED DIU HAS FAILED PITSTK *)
              TRANTO FWDIU=FWDIU-1,
              ROLSTK=ROLSTK-1, YAWPED=YAWPED-1, FTP=FTP-1, ROOT=ROOT-1,
              NFAIL=NFAIL+1
              BY (FWDIU-PITSTK)*ROLSTK*YAWPED* LAMEL /
              ( FWDIU**2 );

         IF FWDIU > ROLSTK     (* AFFECTED DIU HAS FAILED ROLSTK *)
              TRANTO FWDIU=FWDIU-1, PITSTK=PITSTK-1,
              YAWPED=YAWPED-1, FTP=FTP-1, ROOT=ROOT-1,
              NFAIL=NFAIL+1
              BY PITSTK*(FWDIU-ROLSTK)*YAWPED* LAMEL /
              ( FWDIU**2 );
```

```
               IF FWDIU > YAWPED      (* AFFECTED DIU HAS FAILED YAWPED *)
                    TRANTO FWDIU=FWDIU-1, PITSTK=PITSTK-1,
                    ROLSTK=ROLSTK-1, FTP=FTP-1, ROOT=ROOT-1,
                    NFAIL=NFAIL+1
                    BY PITSTK* ROLSTK*(FWDIU-YAWPED)*
                    LAMEL /   ( FWDIU**2 );

               IF ROOT > FWDIU
                     (* AFFECTED CHANNEL HAS FAILED DIU *)

                    TRANTO FTP=FTP-1, ROOT=ROOT-1, NFAIL=NFAIL+1
                    BY (ROOT-FWDIU)* LAMEL;

               IF FTP > ROOT (* AFFECTED CHANNEL HAS FAILED ROOT *)

                    TRANTO FTP=FTP-1, NFAIL=NFAIL+1
                    BY (FTP-ROOT)* LAMEL;

      ENDIF;
```

```
(***   Mid Area Model - Safety Criteria - Bus Option    ***)
(*                 Group A                                *)
(*     ____________________________________________       *)

SPACE   = (  DUMMY1:    0..1,
             MIDDIU:    0..4,    (* MID DIU STATE INDICATOR          *)
             GYRO:      0..8,    (* GYROS                            *)
             GYREC:     0..2,    (* GYRO RECOVERY INDICATOR          *)
             ACCEL:     0..8,    (* ACCELEROMETERS                   *)
             ACCREC:    0..2,    (* ACCEL RECOVERY INDICATOR         *)
             CNDP:      0..4,    (* CND CHANNEL STATE                 *)
             CNDV:      0..4,    (* CND VALVE STATE                   *)
             HYD:       0..2,    (* HYDRAULIC SYSTEM STATE           *)
             ELMC:      0..4,    (* ELMC STATE INDICATOR             *)
             NFAIL:     0..4);   (* NO. OF FAILED ELEMENTS           *)

START  = ( 0, 4, 8, 0, 8, 0, 4, 4, 2, 4, 0 );

PRUNEIF  NFAIL > 2;

DEATHIF   GYRO-GYREC < 3 OR ACCEL-ACCREC < 3;

DEATHIF  CNDV = 0;

LIST   = 2;
TIME   = 3.0;
PRUNE  = 1.0E-16;
ECHO   = 0;
POINTS  = 6;

LAMGYRO = 50.0E-6;        (* GYRO FAILURE RATE                *)
LAMACC  = 30.0E-6;        (* ACCELEROMETER FAILURE RATE       *)
LAMNOD  = 15.0E-6;        (* NODE FAILURE RATE - CONDITIONED  *)
LAMDIU  = 15.0E-6;        (* DIU FAILURE RATE - CONDITIONED   *)
GYRMEAN = 3.0E-4;         (* RECOVERY TIME MEAN               *)
GYRSTD  = 1.0E-4;         (* RECOVERY TIME STD DEV            *)
ACCMEAN = 3.0E-4;         (* ACC RECOVERY TIME MEAN           *)
ACCSTD  = 1.0E-4;         (* ACC RECOVERY TIME STD DEV        *)

LAMC    = 50.0E-6;           (* PROCESSOR FAILURE RATE          *)
LAMPOS = 10.0E-6;            (* POSITION SENSOR FAILURE RATE    *)
LAMV    = 15.0E-6;           (* VALVE GROUP FAILURE RATE        *)
VJAM    = 3.3333E-5;         (* ACTUATOR JAM FAILURE FRACTION   *)
LAMHYD  = 45.0E-6;           (* HYDRAULIC SYSTEM FAIL RATE      *)

LAMPS    = 10.0E-6;       (* LOCAL POWER SUPPLY - CONDITIONED *)
LAMEL    = 50.0E-6;       (* ELMC FAILURE RATE                *)
NWMEAN   = 3.0E-4;        (* NW RECOVERY TIME MEAN           *)
NWSTD    = 1.0E-4;        (* NW RECOVERY TIME STD DEV        *)
LAMON    = 255.0E-6;      (* OTHER NODES FAILURE RATE         *)
LAMFTP   = 200.0E-6;      (* FTP CHANNEL FAILURE RATE         *)
LAMROOT =  15.0E-6;       (* BUS INTERFACE FAILURE RATE       *)
LAMOD    = 127.5E-6;      (* OTHER DIUS ON BUS FAILURE RATE   *)
(* DIUACT  = 0.10; *)        (* DIU ACTIVE FAILURE FRACTION       *)
```

```
         (* GYRO SENSOR FAILURES AND RECOVERIES *)

IF GYRO > 0 AND GYREC = 0 AND ACCREC = 0 THEN

     IF GYRO = 4 THEN      (* EXHAUSTION *)
        TRANTO GYRO = GYRO-1, GYREC=GYREC+1, NFAIL=NFAIL+1
        BY GYRO*LAMGYRO;
     ELSE
           TRANTO GYRO = GYRO-1, NFAIL=NFAIL+1
           BY GYRO*LAMGYRO;
     ENDIF;
ENDIF;

         (* ACCEL SENSOR FAILURES AND RECOVERIES *)

IF ACCEL > 0 AND GYREC = 0 AND ACCREC = 0 THEN

     IF ACCEL = 4 THEN      (* EXHAUSTION *)
        TRANTO ACCEL = ACCEL-1, ACCREC=ACCREC+1, NFAIL=NFAIL+1
        BY ACCEL*LAMACC;
     ELSE
           TRANTO ACCEL = ACCEL-1, NFAIL=NFAIL+1
           BY ACCEL*LAMACC;
     ENDIF;
ENDIF;

           (***  ACTUATOR PROCESSOR  GROUP FAILURE TRANSITIONS  ***)

    IF CNDP > 0  THEN
               TRANTO CNDP=CNDP-1, NFAIL=NFAIL+1  BY 2*CNDP*(LAMC + LAMPOS);
    ENDIF;

             (*** VALVE GROUP FAILURES ***)

IF CNDV > 0 THEN
        IF NFAIL = 0
              TRANTO CNDV=0, NFAIL=NFAIL+1 BY CNDV*VJAM*LAMV;
        TRANTO CNDV=CNDV-1, NFAIL=NFAIL+1 BY CNDV*(1.0-VJAM)*LAMV;
ENDIF;

          (* HYDRAULIC SYSTEM FAILURES *)

IF HYD > 0 AND (HYD-NFAIL >= 0 ) THEN
     TRANTO CNDV=CNDV-2, HYD=HYD-1, NFAIL=NFAIL+1
     BY CNDV*(CNDV-1)*LAMHYD / ( 2*  (2*HYD-1) );

     IF (2*HYD-CNDV) > 0
         TRANTO CNDV=CNDV-1, HYD=HYD-1, NFAIL=NFAIL+1
         BY (2*HYD-CNDV)*2*CNDV*LAMHYD / ( 2*  (2*HYD-1) );

     IF (2*HYD-CNDV) > 1
         TRANTO HYD=HYD-1, NFAIL=NFAIL+1
         BY (2*HYD-CNDV)*(2*HYD-CNDV-1)*LAMHYD / ( 2*  (2*HYD-1) );
ENDIF;
```

```
          (* MID DIU FAILURES *)

    IF MIDDIU > 0 AND GYREC = 0 AND ACCREC = 0 THEN
          TRANTO MIDDIU=MIDDIU-1, GYRO=GYRO-2, ACCEL=ACCEL-2, CNDP=CNDP-1,
          NFAIL=NFAIL+1 BY GYRO* (GYRO-1)* ACCEL* (ACCEL-1)* CNDP*
          (LAMDIU + LAMPS)/ ( 2* (2*MIDDIU)*
          (2*MIDDIU-1)**2 *MIDDIU );

          IF 2*MIDDIU - GYRO > 0  (* DIU HAS FAILED GYRO *)
             TRANTO MIDDIU=MIDDIU-1, GYRO=GYRO-1, ACCEL=ACCEL-2, CNDP=CNDP-1,
             NFAIL=NFAIL+1 BY 2*GYRO* (2*MIDDIU-GYRO)* ACCEL* (ACCEL-1)* CNDP*
             (LAMDIU + LAMPS)/ ( 2* (2*MIDDIU)*
             (2*MIDDIU-1)**2 *MIDDIU );

          IF 2*MIDDIU - ACCEL > 0  (* DIU HAS FAILED ACCELEROMETER *)
             TRANTO MIDDIU=MIDDIU-1, GYRO=GYRO-2, ACCEL=ACCEL-1, CNDP=CNDP-1,
             NFAIL=NFAIL+1 BY GYRO* (GYRO-1)* 2*ACCEL* (2*MIDDIU-ACCEL)* CNDP*
             (LAMDIU + LAMPS)/ ( 2* (2*MIDDIU)*
             (2*MIDDIU-1)**2 *MIDDIU );

          IF MIDDIU - CNDP > 0  (* DIU HAS FAILED CND PROCESSOR *)
             TRANTO MIDDIU=MIDDIU-1, GYRO=GYRO-2, ACCEL=ACCEL-2,
             NFAIL=NFAIL+1 BY GYRO* (GYRO-1)* ACCEL* (ACCEL-1)* (MIDDIU-CNDP)*
             (LAMDIU + LAMPS)/ ( 2* (2*MIDDIU)*
             (2*MIDDIU-1)**2 *MIDDIU );
   ENDIF;

           (* BUS CENTRAL FAILURES - NOT COMMON TO FOR MODEL *)

   IF MIDDIU > 0 AND GYREC = 0 AND ACCREC = 0 THEN

         TRANTO MIDDIU=MIDDIU-1, GYRO=GYRO-2, ACCEL=ACCEL-2, CNDP=CNDP-1,
         NFAIL=NFAIL+1 BY GYRO* (GYRO-1)* ACCEL* (ACCEL-1)* CNDP*
         ( LAMROOT+ (LAMOD)*DIUACT )  /
         ( 2* (2*MIDDIU)* (2*MIDDIU-1)**2 *MIDDIU );

         IF 2*MIDDIU - GYRO > 0  (* DIU HAS FAILED GYRO *)

            TRANTO MIDDIU=MIDDIU-1, GYRO=GYRO-1, ACCEL=ACCEL-2, CNDP=CNDP-1,
            NFAIL=NFAIL+1 BY 2* GYRO* (2*MIDDIU-GYRO)* ACCEL* (ACCEL-1)*
            CNDP* ( LAMROOT+ (LAMOD)*DIUACT )  /
            ( 2* (2*MIDDIU)* (2*MIDDIU-1)**2 *MIDDIU );

         IF 2*MIDDIU - ACCEL > 0  (* DIU HAS FAILED ACCELEROMETER *)

            TRANTO MIDDIU=MIDDIU-1, GYRO=GYRO-2, ACCEL=ACCEL-1, CNDP=CNDP-1,
            NFAIL=NFAIL+1 BY GYRO* (GYRO-1)* 2* ACCEL* (2*MIDDIU-ACCEL)*
            CNDP* ( LAMROOT+ (LAMOD)*DIUACT )  /
            ( 2* (2*MIDDIU)* (2*MIDDIU-1)**2 *MIDDIU );

         IF MIDDIU - CNDP > 0  (* DIU HAS FAILED CND PROCESSOR *)

            TRANTO MIDDIU=MIDDIU-1, GYRO=GYRO-2, ACCEL=ACCEL-2,
            NFAIL=NFAIL+1 BY GYRO* (GYRO-1)* ACCEL* (ACCEL-1)* (MIDDIU-CNDP)*
            ( LAMROOT+ (LAMOD)*DIUACT )  /
          ( 2* (2*MIDDIU)* (2*MIDDIU-1)**2 *MIDDIU );
ENDIF;

         (* CENTRAL FAILURES - INCLUDES ELEMENTS COMMON TO FOR MODEL *)

IF ELMC > 0 AND GYREC = 0 AND ACCREC = 0 THEN

     TRANTO MIDDIU=MIDDIU-1, GYRO=GYRO-2, ACCEL=ACCEL-2, CNDP=CNDP-1,
     ELMC=ELMC-1, NFAIL=NFAIL+1 BY GYRO* (GYRO-1)* ACCEL* (ACCEL-1)* CNDP*
     ( LAMFTP + LAMEL ) /
     ( 2* (2*MIDDIU)* (2*MIDDIU-1)**2 *MIDDIU );

     IF 2*MIDDIU - GYRO > 0  (* DIU HAS FAILED GYRO *)

        TRANTO MIDDIU=MIDDIU-1, GYRO=GYRO-1, ACCEL=ACCEL-2, CNDP=CNDP-1,
        ELMC=ELMC-1, NFAIL=NFAIL+1 BY 2* GYRO* (2*MIDDIU-GYRO)* ACCEL*
        (ACCEL-1)* CNDP* ( LAMFTP + LAMEL ) /
        ( 2* (2*MIDDIU)* (2*MIDDIU-1)**2 *MIDDIU·);

     IF 2*MIDDIU - ACCEL > 0  (* DIU HAS FAILED ACCELEROMETER *)

        TRANTO MIDDIU=MIDDIU-1, GYRO=GYRO-2, ACCEL=ACCEL-1, CNDP=CNDP-1,
        ELMC=ELMC-1, NFAIL=NFAIL+1 BY GYRO* (GYRO-1)* 2* ACCEL*
        (2*MIDDIU-ACCEL)* CNDP* ( LAMFTP + LAMEL ) /
        ( 2* (2*MIDDIU)* (2*MIDDIU-1)**2 *MIDDIU );

     IF MIDDIU - CNDP > 0  (* DIU HAS FAILED CND PROCESSOR *)

        TRANTO MIDDIU=MIDDIU-1, GYRO=GYRO-2, ACCEL=ACCEL-2, ELMC=ELMC-1,
        NFAIL=NFAIL+1 BY GYRO* (GYRO-1)* ACCEL* (ACCEL-1)* (MIDDIU-CNDP)*
        ( LAMFTP + LAMEL ) /
        ( 2* (2*MIDDIU)* (2*MIDDIU-1)**2 *MIDDIU );

     IF ELMC - MIDDIU > 0  (* CHANNEL HAS FAILED DIU *)

        TRANTO ELMC=ELMC-1, NFAIL=NFAIL+1 BY (ELMC-MIDDIU)*LAMEL;
ENDIF;
```

```
(***   Wing and Tail Area Model - Safety Criteria - Bus Option   ***)
(*                          Group A                                *)
(*     _______________________________________________________    *)


SPACE   = ( DUMMY1: 0..1,
            DUMMY2: 0..1,
            RWP:    0..4,    (* RW CHANNEL STATE              *)
            RWV:    0..4,    (* RW VALVE STATE                *)
            LWP:    0..4,    (* LW CHANNEL STATE              *)
            LWV:    0..4,    (* LW VALVE STATE                *)
            TLP:    0..4,    (* TL CHANNEL STATE              *)
            TLV:    0..4,    (* TL VALVE STATE                *)
            HYD:    0..2,    (* HYDRAULIC SYSTEM STATE        *)
            BUS:    0..4,    (* CENTRAL BUS STATE             *)
            NFAIL:  0..4);   (* NO. OF FAILED ELEMENTS        *)

START   = ( 0, 0, 4, 4, 4, 4, 4, 4, 2, 4, 0 );

PRUNEIF  NFAIL > 2;

DEATHIF  RWV = 0;

DEATHIF  LWV = 0;

DEATHIF  TLV = 0;


LIST   = 2;
TIME   = 3.0;
PRUNE  = 1.0E-16;
ECHO   = 0;
POINTS = 6;

LAMC    = 50.0E-6;           (* PROCESSOR FAILURE RATE          *)
LAMPOS = 10.0E-6;            (* POSITION SENSOR FAILURE RATE    *)
LAMSD   = 20.0E-6;           (* SERVODRIVE GROUP FAILURE RATE   *)
LAMV    = 15.0E-6;           (* VALVE GROUP FAILURE RATE        *)
VJAM    = 3.3333E-5;         (* ACTUATOR JAM FAILURE FRACTION   *)

LAMNODH = 37.5E-6;           (* NODE FAILURE RATE - HARSH       *)
LAMDIUH = 37.5E-6;           (* DIU FAILURE RATE - HARSH        *)
LAMHYD  = 45.0E-6;           (* HYDRAULIC SYSTEM FAIL RATE      *)

LAMPSH   = 25.0E-6;          (* LOCAL POWER SUPPLY RATE - HARSH  *)
LAMEL    = 50.0E-6;          (* ELMC FAILURE RATE               *)
NWMEAN   = 3.0E-4;           (* NW RECOVERY TIME - MEAN         *)
NWSTD    = 1.0E-4;           (* NW RECOVERY TIME - STD DEV      *)
LAMON    = 60.0E-6;          (* OTHER NODE FAILURE RATE         *)
LAMOD    = 30.0E-6;          (* OTHER DIUS ON 1 BUS FAIL RATE   *)
LAMFTP   = 200.0E-6;         (* FTP CHANNEL FAILURE RATE        *)
LAMROOT = 15.0E-6;           (* BUS INTERFACE FAILURE RATE      *)
(* DIUACT = 0.10; *)            (* DIU ACTIVE FAILURE FRACTION   *)
```

```
          (***  RW PROCESSOR  GROUP FAILURE TRANSITIONS  ***)

   IF RWP > 0  THEN
              TRANTO RWP=RWP-1, NFAIL=NFAIL+1  BY 2*RWP*(LAMC + LAMPOS);
   ENDIF;

            (*** RW VALVE GROUP FAILURES ***)

IF RWV > 0 THEN
        IF NFAIL = 0
             TRANTO RWV=0, NFAIL=NFAIL+1 BY RWV*VJAM*LAMV;
        TRANTO RWV=RWV-1, NFAIL=NFAIL+1 BY RWV*(1.0-VJAM)*LAMV;
ENDIF;


       (* RW DIU FAILURES *)

    IF RWP > 0  THEN
        TRANTO RWP=RWP-1, NFAIL=NFAIL+1  BY RWP*(LAMDIUH*
        (1.0-DIUACT) + LAMPSH);
    ENDIF;

          (***  LW PROCESSOR  GROUP FAILURE TRANSITIONS  ***)

   IF LWP > 0  THEN
              TRANTO LWP=LWP-1, NFAIL=NFAIL+1  BY 2*LWP*(LAMC + LAMPOS);
   ENDIF;

            (*** LW VALVE GROUP FAILURES ***)

IF LWV > 0 THEN
        IF NFAIL = 0
             TRANTO LWV=0, NFAIL=NFAIL+1 BY LWV*VJAM*LAMV;
        TRANTO LWV=LWV-1, NFAIL=NFAIL+1 BY LWV*(1.0-VJAM)*LAMV;
ENDIF;


       (* LW DIU FAILURES *)

    IF LWP > 0  THEN
        TRANTO LWP=LWP-1, NFAIL=NFAIL+1  BY LWP*(LAMDIUH*
        (1.0-DIUACT) + LAMPSH);
    ENDIF;


          (***  TL PROCESSOR  GROUP FAILURE TRANSITIONS  ***)

   IF TLP > 0  THEN
              TRANTO TLP=TLP-1, NFAIL=NFAIL+1  BY 2*TLP*(LAMC + LAMPOS);
   ENDIF;

            (*** TL VALVE GROUP FAILURES ***)

IF TLV > 0 THEN
        IF NFAIL = 0
             TRANTO TLV=0, NFAIL=NFAIL+1 BY TLV*VJAM*LAMV;
```

```
      TRANTO TLV=TLV-1, NFAIL=NFAIL+1 BY TLV*(1.0-VJAM)*LAMV;
ENDIF;


      (* TL DIU FAILURES *)

    IF TLP > 0  THEN
       TRANTO TLP=TLP-1, NFAIL=NFAIL+1  BY TLP*(LAMDIUH*
       (1.0-DIUACT) + LAMPSH);
    ENDIF;


         (* HYDRAULIC SYSTEM FAILURES *)

IF HYD > 0 AND (HYD-NFAIL >= 0 ) THEN

    TRANTO RWV=RWV-2, LWV=LWV-2, TLV=TLV-2, HYD=HYD-1, NFAIL=NFAIL+1
    BY RWV*(RWV-1)*LWV*(LWV-1)*TLV*(TLV-1)*LAMHYD /
    ( 2* (2*HYD)**2 * (2*HYD-1)**3 );

    IF (2*HYD-RWV) > 0 THEN
       TRANTO RWV=RWV-1, LWV=LWV-2, TLV=TLV-2, HYD=HYD-1, NFAIL=NFAIL+1
       BY (2*HYD-RWV)*2*RWV*LWV*(LWV-1)*TLV*(TLV-1)*LAMHYD /
       ( 2* (2*HYD)**2 * (2*HYD-1)**3 );
    ENDIF;

    IF (2*HYD-LWV) > 0 THEN
       TRANTO RWV=RWV-2, LWV=LWV-1, TLV=TLV-2, HYD=HYD-1, NFAIL=NFAIL+1
       BY RWV*(RWV-1)* 2*LWV*(2*HYD-LWV)* TLV*(TLV-1)*LAMHYD /
       ( 2* (2*HYD)**2 * (2*HYD-1)**3 );
    ENDIF;

    IF (2*HYD-TLV) > 0 THEN
       TRANTO RWV=RWV-2, LWV=LWV-2, TLV=TLV-1, HYD=HYD-1, NFAIL=NFAIL+1
       BY RWV*(RWV-1)* LWV*(LWV-1)* 2*TLV*(2*HYD-TLV)*LAMHYD /
       ( 2* (2*HYD)**2 * (2*HYD-1)**3 );
    ENDIF;

ENDIF;

          (**** BUS CENTRAL AND ELECTRIC POWER DISTRIBUTION ***)

IF BUS > 0 THEN
    TRANTO RWP=RWP-1, LWP=LWP-1, TLP=TLP-1, BUS=BUS-1,
    NFAIL=NFAIL+1 BY RWP* LWP* TLP* (LAMROOT+LAMFTP+LAMEL+
    (LAMDIUH+LAMOD) * DIUACT) / ( BUS**2);

    IF BUS - RWP > 0
      TRANTO LWP=LWP-1, TLP=TLP-1, BUS=BUS-1,
      NFAIL=NFAIL+1 BY (BUS-RWP)* LWP* TLP* (LAMROOT+LAMFTP+LAMEL+
      (LAMOD) * DIUACT) / ( BUS**2);

    IF BUS - LWP > 0
      TRANTO RWP=RWP-1, TLP=TLP-1, BUS=BUS-1,
      NFAIL=NFAIL+1 BY RWP* (BUS-LWP)* TLP* (LAMROOT+LAMFTP+LAMEL+
      (LAMOD) * DIUACT) / ( BUS**2);
```

```
        IF BUS - TLP > 0
           TRANTO RWP=RWP-1, LWP=LWP-1, BUS=BUS-1,
           NFAIL=NFAIL+1 BY RWP* LWP* (BUS-TLP)* (LAMROOT+LAMFTP+LAMEL+
           (LAMOD) * DIUACT) / ( BUS**2);
ENDIF;
```

APPENDIX B

DENET SIMULATION MODEL

BUSMESSAG

```
DEFINITION DEVM BusMessag;

    EXPORT BusMessageType*, MessageType, NodeCommandType,
                PortNameType, IOActivityChoice,
                PortStateType, PortEnableRegisterType,
                NodeMessageCommandType,
                NodeMessageResponseType, DIUCommandType,
                NumberOfPortsPerNode, NumberOfNodes,
                MakeNodeConfigurationCommand, MakeMonitorCommand;

    CONST NumberOfNodes        = 18;
          NumberOfPortsPerNode = 5;

    TYPE

        NodeCommandType          = (ChangePortEnable, Null);
        PortNameType             = [1 .. NumberOfPortsPerNode];
        PortStateType            = (Enabled, Disabled);
        PortEnableRegisterType   = ARRAY PortNameType OF PortStateType;

        NodeMessageCommandType = RECORD

            CASE Command   : NodeCommandType OF

                ChangePortEnable   :

                    PortEnableRegister : PortEnableRegisterType; |

                Null :

            END;

        END;

        NodeMessageResponseType = RECORD

            PortEnableRegister    : PortEnableRegisterType;

        END;

        IOActivityChoice = (Input, Output, Grouped);

        DIUCommandType = RECORD

            Activity                : IOActivityChoice;
            CommandNumber           : INTEGER;

        END;

        MessageType = (NodeInput, NodeOutput, DIUInput, DIUOutput);

        BusMessageType = ENTITY

            Address      : INTEGER;
            CASE Message : MessageType OF

                NodeInput :

                        Input        : NodeMessageCommandType

                | NodeOutput :

                        Output       : NodeMessageResponseType;

                | DIUInput :

                        DIUCommand   : DIUCommandType;

                | DIUOutput:

            END;
        END;

    (*******************************************************************************)
    (* This procedure generates a node configurtion command so that it
       can be sent to the network to modify a node's port enable register. *)

    PROCEDURE MakeNodeConfigurationCommand(NodeAddress   : INTEGER;
                                           Configuration : PortEnableRegisterType)
                                           : BusMessageType;

    (*******************************************************************************)
    (* This procedure generates a node status command (i.e null command
       to the node) to be used by the Network Manager to collect the
       Network's Status.  *)

    PROCEDURE MakeMonitorCommand(NodeAddress : INTEGER)
                                 : BusMessageType;

    (*******************************************************************************)

END BusMessag.
```

BUSMESSAG

```
DEVM BusMessag;

    EXPOSE

        (************************************************************************)

        PROCEDURE MakeNodeConfigurationCommand(NodeAddress : INTEGER;
                                               Configuration : PortEnableRegisterType)
                                               : BusMessageType;

            VAR Command   : BusMessageType;
                PortIndex : PortNameType;

        BEGIN

            NEW(Command);

            Command^.Address       := NodeAddress;
            Command^.Message       := NodeInput;
            Command^.Input.Command := ChangePortEnable;

            FOR PortIndex := 1 TO NumberOfPortsPerNode DO

                Command^.Input.PortEnableRegister[PortIndex] := Configuration[PortIndex];

            END;

            RETURN (Command);

        END MakeNodeConfigurationCommand;

        (************************************************************************)

        PROCEDURE MakeMonitorCommand(NodeAddress : INTEGER)
                                     : BusMessageType;


            VAR Command : BusMessageType;

        BEGIN

            NEW(Command);

            Command^.Address       := NodeAddress;
            Command^.Message       := NodeInput;
            Command^.Input.Command := Null;

            RETURN (Command);

        END MakeMonitorCommand;

        (************************************************************************)

    END;

BEGIN

END BusMessag.
```

TYPECONST

```
DEFINITION MODULE TypeConst;

    FROM BusMessag IMPORT PortNameType, NumberOfNodes, NumberOfPortsPerNode;

    EXPORT QUALIFIED NetworkElementType, ChannelIDType, PortArrayType,
                NodeRecordType, PortRecord, NodeArrayType, StatusType,
                PortConfigurationType, PortStatusRecord, PortStatusArray,
                NodeStatusRecord, ChannelStatusRecord, NodeStatusArray,
                ChannelStatusArray, NumberOfNetworks, NumberOfIOSPerChannel;

    (* These declarations will need to be moved elsewhere when
       a more appropiate home has been found for them.  *)

    CONST NumberOfGPCS          = 1;
          NumberOfDIUS          = 4;
          NumberOfNetworks      = 2;
          NumberOfIOSPerChannel = 3;

    TYPE GPCAddressType = [1 .. NumberOfGPCS];
         ChannelIDType = (A, B, C);
         DIUAddressType = [1 .. NumberOfDIUS];

    (***************************************************************************)

    TYPE NetworkElementType = (GPC, Node, DIU, None);

         NetworkElementRecord  = RECORD

            CASE AdjacentElement : NetworkElementType OF

                GPC:
                    GPCAddress : INTEGER;
                    Channel    : ChannelIDType;

                | Node:

                    NodeNumber  : INTEGER;
                    NodeAddress : INTEGER;
                    Port        : PortNameType;

                | DIU:

                    DIUAddress : INTEGER;

                | None:

            END;

         END;

         PortRecord = RECORD

            Element : NetworkElementRecord;

         END;

         PortArrayType = ARRAY [1 .. NumberOfPortsPerNode] OF NetworkElementRecord;

         NodeRecordType = RECORD

            NodeAddress : INTEGER;
            PortArray   : PortArrayType;

         END;

         NodeArrayType = ARRAY [1 .. NumberOfNodes] OF NodeRecordType;

         (*************************************************************************)
         (* The types pertain to the Network Manager maintaining the status
            of the Network.  *)

         StatusType = (Idle, Active, Failed);
```

```
        PortConfigurationType = (Inboard, Outboard);

        PortStatusRecord = RECORD

            CASE Status : StatusType OF

                Active:

                    Direction : PortConfigurationType;

                | Idle, Failed:

            END;

        END;

        PortStatusArray  = ARRAY [1 .. NumberOfPortsPerNode] OF PortStatusRecord;

        NodeStatusRecord = RECORD

            Address    : INTEGER;
            Status     : StatusType;
            PortStatus : PortStatusArray;

        END;

        ChannelStatusRecord = RECORD

            GPCAddress : INTEGER;   (*GPCAddressType;*)
            ChannelID  : ChannelIDType;
            Status     : StatusType;

        END;

        NodeStatusArray = ARRAY [1 .. NumberOfNodes] OF NodeStatusRecord;
        ChannelStatusArray = ARRAY ChannelIDType OF ChannelStatusRecord;

END TypeConst.
```

CENTRALDB

```
DEFINITION DEVM CentralDB;

    FROM BusMessag IMPORT PortNameType, NumberOfPortsPerNode, NumberOfNodes;

    FROM TypeConst IMPORT NodeRecordType, NodeArrayType, NumberOfNetworks,
                          NumberOfIOSPerChannel;

    EXPORT IOSConnectionType, UpdateNodeData, ReadNodeInterConnections,
                          FindNodeNumber, FindIOSConnections;

    TYPE IOSConnectionRecord = RECORD

             GPCAddress       : INTEGER;
             NodeConnectedTo  : INTEGER;

         END;

         IOSConnectionType = ARRAY [1 .. NumberOfIOSPerChannel] OF IOSConnectionRecord;

    (***************************************************************************)
    (* This procedure will update on element of the data that represents
       the interelement connection of the I/O Network.  *)

    PROCEDURE UpdateNodeData(NodeNumber   : INTEGER;
                             NetworkID    : INTEGER;
                             Data         : NodeRecordType);

    (***************************************************************************)
    (* The procedure will return the internal data structure that each
       AIPS node has initialized with its own data.  *)

    PROCEDURE ReadNodeInterConnections(NetworkID                : INTEGER;
                                       VAR NodeConnections : NodeArrayType);

    (***************************************************************************)
    (* This procedure will search the Central Database for a node with
       "Address" as its node address and return its node number.
       BEWARE: No check is done to see if the address given coresponds
       to an address on your network.  If an address is given that
       is not on your network, then the wrong number will be retruned.  *)
    PROCEDURE FindNodeNumber(Address : INTEGER) : INTEGER;

    (***************************************************************************)
    (* This procedure will search the Central Database an return all the
       IOS connections for a the indicated network.  An IOS which is not
       installed will have a zero ID.  *)
    PROCEDURE FindIOSConnections(NetworkID          : INTEGER;
                                 VAR ConnectionArray : IOSConnectionType);

    (***************************************************************************)

    TYPE NetworkDescriptionType = ARRAY [1 .. NumberOfNetworks] OF NodeArrayType;

    VAR NetworkDescription : NetworkDescriptionType;

END CentralDB.
```

CENTRALDB

```
DEVM CentralDB;

    FROM BusMessag IMPORT PortNameType, NumberOfPortsPerNode, NumberOfNodes;

    FROM TypeConst IMPORT NodeRecordType, NetworkElementType, ChannelIDType,
                        NodeArrayType, NumberOfNetworks, NumberOfIOSPerChannel;

    EXPOSE

        (*********************************************************************)

        PROCEDURE UpdateNodeData(NodeNumber  : INTEGER;
                                 NetworkID   : INTEGER;
                                 Data        : NodeRecordType);

            VAR PortIndex        : PortNameType;
                NodeIndex        : INTEGER;
                CurrentNodeIndex : INTEGER;
                CurrentNode      : INTEGER;

        BEGIN

            WITH NetworkDescription[NetworkID][NodeNumber] DO

                NodeAddress := Data.NodeAddress;

                FOR PortIndex := 1 TO NumberOfPortsPerNode DO

                    CASE Data.PortArray[PortIndex].AdjacentElement OF

                        GPC:

                            PortArray[PortIndex].AdjacentElement := GPC;
                            PortArray[PortIndex].GPCAddress      := Data.PortArray[PortIndex].GPCAddress;
                            PortArray[PortIndex].Channel         := Data.PortArray[PortIndex].Channel;

                        | Node:

                            PortArray[PortIndex].AdjacentElement := Node;
                            PortArray[PortIndex].NodeAddress     := Data.PortArray[PortIndex].NodeAddress;
                            PortArray[PortIndex].Port            := Data.PortArray[PortIndex].Port;

                        | DIU:

                            PortArray[PortIndex].AdjacentElement := DIU;
                            PortArray[PortIndex].DIUAddress      := Data.PortArray[PortIndex].DIUAddress;

                        | None:

                            PortArray[PortIndex].AdjacentElement := None;

                    END;

                END;

            END;

            (* Due to the nature of this DENET solution, the AIPS node numbers
               of AIPS adjacent nodes cannot be set when this procedure
               is called. To solve this problem, the new information is entered,
               (see above for loop) the database will be searched and
               correct node numbers entered.  *)

            FOR CurrentNodeIndex := 1 TO NumberOfNodes DO

                CurrentNode    := NetworkDescription[NetworkID][CurrentNodeIndex].
                                      NodeAddress;
                FOR NodeIndex := 1 TO NumberOfNodes DO

                    WITH NetworkDescription[NetworkID][NodeIndex] DO

                        IF (NodeAddress <> CurrentNode) THEN
                    FOR PortIndex := 1 TO NumberOfPortsPerNode DO

                        IF (PortArray[PortIndex].AdjacentElement =
                            Node) AND (PortArray[PortIndex].NodeAddress
                            = CurrentNode) THEN

                            PortArray[PortIndex].NodeNumber :=
                                CurrentNodeIndex;

                        END;

                    END;

                END;

            END;

        END;

    END;

END UpdateNodeData;
(**********************************************************************)

PROCEDURE ReadNodeInterConnections(NetworkID           : INTEGER;
                                   VAR NodeConnections : NodeArrayType);

    VAR NodeIndex : INTEGER;
        PortIndex : PortNameType;

BEGIN

    FOR NodeIndex := 1 TO NumberOfNodes DO

        NodeConnections[NodeIndex].NodeAddress :=
            NetworkDescription[NetworkID][NodeIndex].NodeAddress;

        FOR PortIndex := 1 TO NumberOfPortsPerNode DO

            WITH NetworkDescription[NetworkID][NodeIndex].PortArray[PortIndex] DO

                CASE AdjacentElement OF

                    GPC:

                        NodeConnections[NodeIndex].PortArray[PortIndex].
                            AdjacentElement := GPC;
                        NodeConnections[NodeIndex].PortArray[PortIndex].
                            GPCAddress := GPCAddress;
                        NodeConnections[NodeIndex].PortArray[PortIndex].
                            Channel := Channel;

                    | Node:

                        NodeConnections[NodeIndex].PortArray[PortIndex].
                            AdjacentElement := Node;
                        NodeConnections[NodeIndex].PortArray[PortIndex].
                            NodeNumber := NodeNumber;
                        NodeConnections[NodeIndex].PortArray[PortIndex].
                            NodeAddress := NodeAddress;
                        NodeConnections[NodeIndex].PortArray[PortIndex].
                            Port := Port;

                    | DIU:

                        NodeConnections[NodeIndex].PortArray[PortIndex].
                            AdjacentElement := DIU;
                        NodeConnections[NodeIndex].PortArray[PortIndex].
                            DIUAddress := DIUAddress;
                    | None:

                        NodeConnections[NodeIndex].PortArray[PortIndex].
                            AdjacentElement := None;

                END;

            END;

        END;

    END;

END ReadNodeInterConnections;

(***********************************************************************)

PROCEDURE FindNodeNumber(Address : INTEGER) :INTEGER;

    VAR NetworkIndex : INTEGER;
        NodeIndex    : INTEGER;

BEGIN

    FOR NetworkIndex := 1 TO NumberOfNetworks DO

        FOR NodeIndex := 1 TO NumberOfNodes DO

            IF NetworkDescription[NetworkIndex][NodeIndex].NodeAddress
                = Address THEN

                RETURN (NodeIndex);

            END;

        END;

    END;

    RETURN(0);

END FindNodeNumber;

(***********************************************************************)

PROCEDURE FindIOSConnections(NetworkID           : INTEGER;
                             VAR ConnectionArray : IOSConnectionType);

    VAR NodeNumber : INTEGER;
        Counter    : INTEGER;
        Index      : INTEGER;
        PortIndex  : PortNameType;

BEGIN

    Counter    := 1;
    NodeNumber := 1;

    WHILE NodeNumber <= NumberOfNodes DO

        WITH NetworkDescription[NetworkID][NodeNumber] DO

            FOR PortIndex := 1 TO NumberOfPortsPerNode DO

                IF PortArray[PortIndex].AdjacentElement = GPC THEN

                    ConnectionArray[Counter].GPCAddress := PortArray[PortIndex].
                                                             GPCAddress;
                    ConnectionArray[Counter].NodeConnectedTo := NodeNumber;
                    Counter := Counter + 1;

                END;
```

```
                 END;

                 NodeNumber := NodeNumber + 1;

             END;

         END;

         (* Set elements in  connection array who do not have IOS's
            connected to them to 0.  *)
         IF Counter <= NumberOfIOSPerChannel THEN

             FOR Index := Counter TO NumberOfIOSPerChannel DO

                 ConnectionArray[Index].GPCAddress := 0;

             END;

         END;

      END FindIOSConnections;

      (*************************************************************************)

  END;

      EVENT Dummy : INTEGER;


      VAR PortIndex        : PortNameType;
          NodeIndex        : INTEGER;
          NetworkIndex     : INTEGER;

  (****************************************************************************)
  BEGIN

       (* Initialize the Network Description so that all nodes are
          connected to no elements as an initial state.  *)
       FOR NetworkIndex := 1 TO NumberOfNetworks DO

           FOR NodeIndex := 1 TO NumberOfNodes DO

               NetworkDescription[NetworkIndex][NodeIndex].NodeAddress := NodeIndex;

               FOR PortIndex := 1 TO NumberOfPortsPerNode DO

                   NetworkDescription[NetworkIndex][NodeIndex].PortArray[PortIndex].AdjacentElement
                       := None;


               END;

           END;

       END;

   LOOP

       WAITUNTIL EVENT

            Dummy:;

       END;


    END;
END CentralDB.
```

AIPSNODE

```
DEVM AIPSNode;

    FROM BusMessag REACH BusMessageType*;

    FROM BusMessag IMPORT PortEnableRegisterType, PortNameType,
              NumberOfPortsPerNode, PortStateType, MessageType,
              NodeCommandType;

    FROM CentralDB IMPORT UpdateNodeData;

    FROM TypeConst IMPORT PortArrayType, PortRecord, NetworkElementType,
                          NodeRecordType, ChannelIDType;

    (* This will allow the initialization of the database that will be
       used by the network manager.  The array imported is from the
       DENET Node Manager module, and it describes the interconnections
       of each instance of the DENET nodes.  *)

    FROM NodeM IMPORT FindConnector;

    INPUTS
        EVENT NodeCommandFrame : BusMessageType;
              Reset            : BOOLEAN;

        PARA NetworkID                : INTEGER;
             NodeNumber               : INTEGER;
             SequencerTimeLower       : REAL;
             SequencerTimeUpper       : REAL;
             InitialConfiguration     : ARRAY [1 .. NumberOfPortsPerNode] OF BOOLEAN;

    END;

    OUTPUTS
        VAR NodeResponseFrame : BusMessageType;

    END;

    VAR MessageFromNetwork    : BusMessageType;
        PortEnableRegister    : PortEnableRegisterType;
        Port                  : PortNameType;
        NodeData              : NodeRecordType;
        PortIndex             : PortNameType;
        AdjacentNodeID        : INTEGER;
        AdjacentNodeType      : INTEGER;
        InPort                : INTEGER;
        OutPort               : INTEGER;

    (*************************************************************************)
    (* This procedure is responsible for transmitting traffic to the
       network from the sequencer.
       It consults the port activity register to determine if a port is
       enabled, if it is, the message is transmitted to the network
       from that port. *)

    PROCEDURE Transmit(Message : BusMessageType);

        VAR Stream     : INTEGER;
        SequencerTime  : REAL;
        PortIndex      : PortNameType;

    BEGIN

        Stream := 1;

        (* Compute the Sequencer Response Time for this message.  *)
        SequencerTime := Random(Stream, SequencerTimeLower, SequencerTimeUpper);

        FOR PortIndex := 1 TO OutArcs DO

            (* Check to see if this port is enabled.  *)
            IF PortEnableRegister[PortIndex] = Enabled THEN
            AFTER SequencerTime outport[PortIndex]^.NodeResponseFrame <- Message;

        END;

    END;

END Transmit;

(****************************************************************************)
(* This procedure is responsible for processing network traffic received
   on a port of this node and formatting a response for transmission
   if one is required. *)

PROCEDURE Sequencer (Command       : BusMessageType);

    VAR CommandResponseFrame : BusMessageType;
        PortIndex            : PortNameType;

BEGIN

    (* Begin processing message.  *)
    (* Check to see if message is addressed to this node.  *)
    IF Command^.Address = MyNodeID THEN

        (* Execute the command.  *)
        CASE Command^.Message OF

            NodeInput :  (* This is a command from the
                            Network Manager.  *)

                (* Determine what the command is.  *)
                CASE Command^.Input.Command OF

                    ChangePortEnable :

                        FOR PortIndex := 1 TO NumberOfPortsPerNode DO

                            PortEnableRegister[PortIndex] :=
                                Command^.Input.PortEnableRegister[PortIndex];

                        END; |

                    Null: (* This is a do nothing command.  *)

                END;

                NEW(CommandResponseFrame);

                (* Format the response message.  *)
                WITH CommandResponseFrame^ DO

                    Address := MyNodeID;
                    Message := NodeOutput;

                    FOR PortIndex := 1 TO NumberOfPortsPerNode DO

                        Output.PortEnableRegister[PortIndex] :=
                                PortEnableRegister[PortIndex];

                    END;

                END;

                (* Transmit the Node Resposne to the Network.  *)
                Transmit(CommandResponseFrame);

            ELSE  (* No node output message will ever be addressed
                     to a node.  *)

        END;

    ELSE
```

```
        END;

    END Sequencer;
    (***************************************************************************)

BEGIN

    (* Initialize Port Array variable to all ports having no connection.
       The InitializeDB EVENT will fill with the proper data.  *)
    FOR Port := 1 TO NumberOfPortsPerNode DO

        NodeData.PortArray[Port].AdjacentElement := None;

    END;

   (* Initialize the Database for the Network Manager that
      describes the network interconnections for this node. *)

    NodeData.NodeAddress := MyNodeID;

    FOR Port := 1 TO OutArcs DO

       AdjacentNodeID   := GetOutNode(Port);

       IF CheckNodeType(AdjacentNodeID, "IOS", TRUE) THEN

          NodeData.PortArray[Port].AdjacentElement := GPC;
          NodeData.PortArray[Port].GPCAddress      := AdjacentNodeID;
          NodeData.PortArray[Port].Channel          := A;

       ELSIF CheckNodeType(AdjacentNodeID, "AIPSNode", TRUE) THEN

          NodeData.PortArray[Port].AdjacentElement := Node;
          NodeData.PortArray[Port].NodeAddress     := AdjacentNodeID;
          GetInOutPort(AdjacentNodeID, MyNodeID, OutPort, InPort);
          NodeData.PortArray[Port].Port := OutPort;

       ELSIF CheckNodeType(AdjacentNodeID, "DIU", TRUE) THEN

          NodeData.PortArray[Port].AdjacentElement := DIU;
          NodeData.PortArray[Port].DIUAddress      := AdjacentNodeID;

       ELSE

          WriteString(ParamOut, "invalid node");
          WriteLn(ParamOut);

       END;

    END;

    (* Send the data to the database.  *)
    UpdateNodeData(NodeNumber, NetworkID, NodeData);

    (* Load port enable register with intial configuration.  *)
    FOR PortIndex := 1 TO NumberOfPortsPerNode DO

        IF InitialConfiguration[PortIndex] THEN

            PortEnableRegister[PortIndex] := Enabled;

        ELSE

            PortEnableRegister[PortIndex] := Disabled;

        END;

    END;
    (***************************************************************************)
```

```
      LOOP

          WAITUNTIL EVENT

              NodeCommandFrame:

                  MessageFromNetwork := ActivePort^.NodeCommandFrame;
                  (* First check to see if this port is enabled, if so
                     then send traffic to other ports on this node.
                     A digression from the AIPS Network Node has been
                     taken at this point in the implementation.
                     The traffic is only sent to the ports on this node
                     that are enabled.  This shifts the responsibility of
                     the port to check to see if it is enabled before
                     transmitting, to the sender of the traffic.  *)

                  (* Check to see if the port the traffic is received on
                     is enabled.   *)
                  IF PortEnableRegister[ActivePort^.index] = Enabled THEN

                      (* Send traffice to all other ports on this node.  *)
                      FOR Port := 1 TO OutArcs DO

                          (* Do not retransmit on port which traffic
                             was recieved  *)
                          IF ActivePort^.index <> INTEGER(Port) THEN

                              (* Transmit message back to the network
                                 only if the port is enabled. *)

                              IF PortEnableRegister[Port] = Enabled THEN

                                  NOW outport[Port]^.NodeResponseFrame <-
                                      MessageFromNetwork;

                              END;

                          END;

                      END;

                  END;

                  (* Send message to Sequencer for Processing.  *)
                  Sequencer(MessageFromNetwork);

              | Reset:

                  (* Reload port enable register with intial configuration.  *)
                  FOR PortIndex := 1 TO NumberOfPortsPerNode DO

                      IF InitialConfiguration[PortIndex] THEN

                          PortEnableRegister[PortIndex] := Enabled;

                      ELSE

                          PortEnableRegister[PortIndex] := Disabled;

                      END;

                  END;

        END;

    END;

END AIPSNode.
```

DIU

```
DEVM DIU;

    FROM BusMessag REACH BusMessageType*;

    FROM BusMessag IMPORT MessageType, DIUCommandType, IOActivityChoice;

    INPUTS
        EVENT
             DIUCommandFrame : BusMessageType;

        PARA OverheadTime      : REAL;
             CommandTimes      : ARRAY[1 .. 3] OF REAL;

    END;

    OUTPUTS
        VAR DIUResponseFrame : BusMessageType;

    END;

    VAR Command                : BusMessageType;
        Response               : BusMessageType;
        DIUComputeTime         : REAL;
        DIUEndTransmissionTime : REAL;

BEGIN

    LOOP

        WAITUNTIL EVENT

            DIUCommandFrame:

                Command := ActivePort^.DIUCommandFrame;
                IF Command^.Address = MyNodeID THEN

                    WITH Command^.DIUCommand DO

                        IF Activity = Input THEN

                           DIUComputeTime := CommandTimes[CommandNumber];

                        ELSIF Activity = Output THEN

                           DIUComputeTime := 0.0;

                        ELSE (* Actvity = Grouped *)

                           DIUComputeTime := CommandTimes[CommandNumber];

                        END;

                    END;

                    IF Command^.DIUCommand.Activity <> Output THEN (* no response for a write *)

                       NEW(Response);
                       WITH Response^ DO

                           Address              := MyNodeID;
                           Message              := DIUOutput;

                       END;

                       DIUEndTransmissionTime := DIUComputeTime + OverheadTime
                                + Random(1, 0.0, 0.000010);
                       AFTER DIUEndTransmissionTime outport[1]^.DIUResponseFrame <- Response;

                    END;

                END;
                END;

            END;

        END DIU.
```

IOS

```
DEFINITION DEVM IOS;

    FROM BusMessag REACH BusMessageType*;

    FROM TypeConst IMPORT NumberOfIOSPerChannel, NumberOfNetworks;

    EXPORT IOSRecordType, IOSRecordTypePointer, ChainType*,
               TimeOutIndicatorType, TransactionQueueType,
               TransactionType*, InputFrameQueueType,
               InputFrameType*, ChainStatusData*, TransactionTurnAroundTime;

    TYPE IOSRecordType = RECORD

             NetworkID : INTEGER;
             IOSID     : INTEGER;
             Number    : INTEGER;

         END;

         IOSRecordTypePointer = POINTER TO IOSRecordType;

         TimeOutIndicatorType = (NormalCompletion, TimedOut);

         TransactionType = ENTITY

            Identifier      : INTEGER;
            TimeOutValue    : REAL;
            OutputFrame     : BusMessageType;

         END;

         TransactionQueueType = QUEUE OF TransactionType;

         ChainType = ENTITY

            ChainIdentifier        : INTEGER;
            NumberOfTransactions   : INTEGER;
            NetworkToBeExecutedOn  : INTEGER;
            TransactionQueue       : TransactionQueueType;
            FrameCount             : INTEGER;

         END;

         InputFrameType = ENTITY

            TransactionIdentifier : INTEGER;
            MessageAddress        : INTEGER;

            CASE TransactionTimeOutIndicator : TimeOutIndicatorType OF

                NormalCompletion :

                    InputFrame     : BusMessageType;

                | TimedOut :

            END;

         END;

         InputFrameQueueType = QUEUE OF InputFrameType;

         ChainStatusData = ENTITY

            ChainTimeOutIndicator : TimeOutIndicatorType;
            AllFailed             : BOOLEAN;
            AnyFailed             : BOOLEAN;
            InputFrameQueue       : InputFrameQueueType;

         END;

      CONST TransactionTurnAroundTime = 0.000010;
```

```
END IOS.
```

IOS

```
DEVM IOS;

    FROM BusMessag REACH BusMessageType*;

    FROM BusMessag IMPORT PortNameType, MessageType, IOActivityChoice,
                        NodeCommandType, NumberOfPortsPerNode;

    FROM Controls REACH SystemProbe, NumberOfProbes;

    FROM Senddata IMPORT WriteDataElementType, DataElementType,
                        FrequencyType, CyclicDataType, CyclicVariationType,
                        NonCyclicVariationType, ChainStateType;

    INPUTS
        EVENT InputTransaction  : BusMessageType;
              ChainToProcess    : ChainType;
              StopChain         : BOOLEAN;
              Reset             : BOOLEAN;
              ProbeReset        : BOOLEAN;

        PARA NetworkID                   : INTEGER;
             RootNodeID                  : INTEGER;
             IOServiceID                 : INTEGER;
             NodeCommandBitsOnBus        : REAL;
             NodeResponseBitsOnBus       : REAL;
             DIUIBitsOnBus               : REAL;
             ApplicationTransmitBits     : ARRAY[1 .. 3], [0 .. 9] OF REAL;
             ApplicationResponseBits     : ARRAY[1 .. 3], [0 .. 9] OF REAL;
             ProbeNumber                 : INTEGER;

   END;

   OUTPUTS

       VAR OutputTransaction : BusMessageType;
           IOChainResponse   : ChainStatusData;
           ChainFinished     : BOOLEAN;

   END;

   EVENT
         EndIOActivity       : BOOLEAN;
         TransactionTimeOut  : BOOLEAN;
         DIUWritten          : BOOLEAN;

   CONST DataExchangeQuantity      = 2.0;  (* bytes per exchange *)
         TransactionTurnAroundTime = 0.000010;
         Max_TransactionsInChain   = 10;

   VAR ChainUnderExecution          : ChainType;
       TransactionUnderExecution    : TransactionType;
       ResponseChain                : ChainStatusData;
       NetworkInput                 : InputFrameType;
       BitTimeForResponse           : REAL;
       NumberOfTransactions         : INTEGER;
       TransactionTimeOutNotice     : EventPointer;
       NetworkPort                  : INTEGER;
       IOServicePort                : INTEGER;
       TempTime                     : REAL;
       StartIODataElement           : DataElementType;
       EndIODataElement             : DataElementType;
       ChainStatus                  : ChainStateType;

   (************************************************************************)

   PROCEDURE ComputeTransmissionBitTime(Transaction  : TransactionType;
                                        ChainID      : INTEGER) : REAL;

       VAR   BitTime : REAL;

   BEGIN
```

```
    WITH Transaction^ DO

        CASE OutputFrame^.Message OF

            NodeInput:

                BitTime := NodeCommandBitsOnBus;

            | DIUInput:

                IF OutputFrame^.DIUCommand.Activity <> Input THEN

                    BitTime := ApplicationTransmitBits[ChainID, (Identifier MOD Max_TransactionsInChain)];

                ELSE

                    BitTime := DIUIBitsOnBus;

                END;

        END;

    END;

    RETURN(BitTime);

END ComputeTransmissionBitTime;

(************************************************************************)

PROCEDURE ComputeResponseBitTime(Transaction : TransactionType;
                                 ChainID     : INTEGER) : REAL;

    VAR   BitTime : REAL;

BEGIN

    WITH Transaction^ DO

        CASE OutputFrame^.Message OF

            NodeInput:

                BitTime := NodeResponseBitsOnBus;

            | DIUInput:

                BitTime := ApplicationResponseBits[ChainID, (Identifier MOD Max_TransactionsInChain)];

        END;

    END;

    RETURN(BitTime);

END ComputeResponseBitTime;

(************************************************************************)
(* This procedure takes the next transaction from the Chain Currently
   Executing and sends it to the network. It also decrements the
   transactions left to execute for this chain counter, and schedules
   the timeout event for this transaction.  *)

PROCEDURE ProcessTransaction(PreviousTransaction           : BOOLEAN;
                             PreviousInputFrame            : InputFrameType;
                             Chain                         : ChainType;
                             VAR TransactionExecuting      : TransactionType;
                             VAR TransactionsLeftToExecute : INTEGER;
                             VAR ResponseBitTime           : REAL);

    VAR NextMessageBitTime    : REAL;
        TransactionOutputTime : REAL;
```

```
      BEGIN

          (* Calculate the bits on the bus time for the previous transaction.  *)
          IF PreviousTransaction THEN

              (* Compute the next transaction, send it to the network and
                 schedule its timeout event.  *)
              TransactionExecuting  := QSucc(TransactionExecuting, Chain^.TransactionQueue);
              (* Bit Time to transmit the next message.  *)
              NextMessageBitTime    := ComputeTransmissionBitTime(TransactionExecuting,
                                       (Chain^.ChainIdentifier MOD 4));
              TransactionOutputTime := ResponseBitTime + TransactionTurnAroundTime +
                                          NextMessageBitTime;

         ELSE

              (* Compute the next transaction, send it to the network and
                 schedule its timeout event.  *)
              TransactionExecuting  := FirstQ(Chain^.TransactionQueue);
              NextMessageBitTime    := ComputeTransmissionBitTime(TransactionExecuting,
                                       (Chain^.ChainIdentifier MOD 4));
              TransactionOutputTime := NextMessageBitTime;

         END;

         (* Decrement transactions left to process counter.  *)
         TransactionsLeftToExecute := TransactionsLeftToExecute - 1;

         WITH TransactionExecuting^ DO

             REPORT "%d"  Identifier TAGGED "Transaction started Execution.";

             AFTER TransactionOutputTime outport[NetworkPort]^.OutputTransaction <- OutputFrame;

             IF (OutputFrame^.Message = DIUInput) AND
                (OutputFrame^.DIUCommand.Activity = Output) THEN

                AFTER NextMessageBitTime DIUWritten <- TRUE;

             ELSE

                 (* Schedule Transaction timeout.  *)
                 ResponseBitTime := ComputeResponseBitTime(TransactionExecuting,
                                       (Chain^.ChainIdentifier MOD 4));

                 AFTER (TimeOutValue + TransactionOutputTime) TransactionTimeOut <- TRUE;
                 TransactionTimeOutNotice := CurrentNotice;

             END;

        END;

    END ProcessTransaction;

    (****************************************************************************)
BEGIN

    NetworkPort    := GetOutPort(RootNodeID);
    IOServicePort  := GetOutPort(IOServiceID);

    LOOP

        WAITUNTIL (ChainToProcess, ProbeReset, Reset)

            (*********************************************************************)
            (* This event handles the Chain Loading mechanism from
               the I/O Request Processing.  First it sets Chain Finished
               to FALSE and the pointer to the first input frame to nil,
               then it accumalates some time
               to represent the loading of the chain into the IOS,
    schedules the chain timeout event, initailzes the
    Network Chain Data so that the Input Frames can be
    processed as they are received, sends the first
    transaction to the network, and schedules the
    transaction timeout for the first transaction.  *)
ChainToProcess:

     IF NetworkID = 1 THEN
        SAMPLE 1.0 WITH SystemProbe[ProbeNumber];
     END;

     (* Create memory for local copy of the responses
        for this chain.  *)
     NEW(ResponseChain);
     ResponseChain^.AllFailed := TRUE;
     ResponseChain^.AnyFailed := FALSE;
     ResponseChain^.InputFrameQueue := InitQ("InputFrameQueue", FALSE, 0);

     (* Make the response available to the I/O System Service.
        This makes the chain response data immediately available
        by keeping the data on the port.
        This will allow the I/O System Service to start reading
        the IOS when a chain has timed out.  *)
     NOW outport[IOServicePort]^.IOChainResponse <- ResponseChain;

     ChainUnderExecution := ActivePort^.ChainToProcess;

     (* Record start I/O activity data collection event.
        Increment the proper frame counter.  *)
     StartIODataElement.SimulationTime          := clock;

     IF ChainUnderExecution^.ChainIdentifier < 20 THEN

          StartIODataElement.Frequency              := Cyclic;
          StartIODataElement.CyclicData.FrameCount  := ChainUnderExecution^.FrameCount;

          CASE ChainUnderExecution^.ChainIdentifier OF

               1:

                    StartIODataElement.CyclicData.C_Variation := StartNW1IOActivity;
                    StartIODataElement.EventID                := 7;

             | 2:

                    StartIODataElement.CyclicData.C_Variation := StartNW1IOActivity;
                    StartIODataElement.EventID                := 8;

             | 3:

                    StartIODataElement.CyclicData.C_Variation := StartNW1IOActivity;
                    StartIODataElement.EventID                := 9;

             | 5:

                    StartIODataElement.CyclicData.C_Variation := StartNW1IActivity;
                    StartIODataElement.EventID                := 16;

             | 6:

                    StartIODataElement.CyclicData.C_Variation := StartNW1IActivity;
                    StartIODataElement.EventID                := 17;

             | 7:

                    StartIODataElement.CyclicData.C_Variation := StartNW1IActivity;
                    StartIODataElement.EventID                := 18;

             | 9:

                    StartIODataElement.CyclicData.C_Variation := StartNW1OActivity;
                    StartIODataElement.EventID                := 22;
```

[redacted]

```
        | 10:

            StartIODataElement.CyclicData.C_Variation := StartNW1OActivity;
            StartIODataElement.EventID                := 23;

        | 11:

            StartIODataElement.CyclicData.C_Variation := StartNW1OActivity;
            StartIODataElement.EventID                := 24;

    END;

ELSIF (ChainUnderExecution^.ChainIdentifier >= 300) AND
    (ChainUnderExecution^.ChainIdentifier <= 310) THEN

    StartIODataElement.Frequency                  := NonCyclic;
    StartIODataElement.NonCyclicData.N_Variation  := ReconfigIOActivity;
    StartIODataElement.EventID                    := 30;

END;

IF ((NetworkID = 1) AND
    (ChainUnderExecution^.ChainIdentifier <= 20)) OR
    ((NetworkID = 2) AND
    (ChainUnderExecution^.ChainIdentifier >= 300)) THEN

    WriteDataElementType(StartIODataElement);

END;

REPORT "%d"  ChainUnderExecution^.ChainIdentifier
    TAGGED "Chain started Execution.";

TransactionUnderExecution := FirstQ(ChainUnderExecution^.
        TransactionQueue);
NumberOfTransactions := QSize(ChainUnderExecution^.
                                TransactionQueue);

(* Initialize IOS to I/O Request processing so that
   I/O Request does confuse previous chain completion
   with this chain completion on its first 2ms poll,
   it this chain has not completed.  *)
NOW outport[IOServicePort]^.ChainFinished <- FALSE;

(* Send out the first transaction.  *)
ProcessTransaction(FALSE, NIL,ChainUnderExecution,
        TransactionUnderExecution, NumberOfTransactions,
        BitTimeForResponse);

LOOP

    WAITUNTIL (InputTransaction, TransactionTimeOut,
        StopChain, DIUWritten, Reset, EndIOActivity, ProbeReset)

    (*********************************************************)
    (* This event handles the Input Frames from the I/O Network.
       It places the Input Data in the Input Frame, cancels the
       the transaction timeout for this transaction, checks for
       more frames to be processed for this chain, if so the
       next transaction is sent to the network and the and the
       transaction timeout is scheduled for the next transaction.
       If no more transaction are to be processed for this chain,
       the Chain Timeout event is canceled for this chain,
       and then Chain Finished is set and the Network Chain Data
       is sent to I/O Request Processing.  *)
        InputTransaction:

            ResponseChain^.AllFailed := FALSE;
            NEW(NetworkInput);

            (* Put Network Data in Input Frame.  *)
```

```
                           WITH NetworkInput^ DO

                               TransactionIdentifier   := TransactionUnderExecution^.
                                                          Identifier;
                               MessageAddress          := TransactionUnderExecution^.
                                                          OutputFrame^.Address;
                               TransactionTimeOutIndicator := NormalCompletion;
                               InputFrame                      := ActivePort^.InputTransaction;

                           END;

                           REPORT "%d"  NetworkInput^.TransactionIdentifier
                               TAGGED "Transaction completed normally.";

(*
                               INSERT NetworkInput LAST IN ResponseChain^InputFrameQueue;

*)
                               QInsert(NetworkInput, ResponseChain^.InputFrameQueue, FALSE);

                            (* Cancel Transaction Timeout for the just completed
                               transaction.  *)
                            CANCEL TransactionTimeOutNotice;

                            (* Check for more transactions to process.  *)
                            IF NumberOfTransactions = 0 THEN

                                ChainStatus := CompletedNormally;

                                (* Set completion flag of the I/O System
                                   service.. *)
                                AFTER BitTimeForResponse + TransactionTurnAroundTime
                                    EndIOActivity <- TRUE;

                            ELSE

                                ProcessTransaction(TRUE, NetworkInput,
                                        ChainUnderExecution,
                                        TransactionUnderExecution, NumberOfTransactions,
                                        BitTimeForResponse);

                            END;

                      (*************************************************************)
                      (* This event will handle the transaction timeout event.
                         It assigns transaction timeout in the Network Chain
                         data corresponding to this transaction. A check will
                         then be made for more transactions to be processed for
                         this chain, if so the transaction timeout is scheduled
                         for the next transaction.  If no more transactions are to
                         be processed for this chain, the Chain Timeout event is
                         canceled for this chain, and then Chain Finished is set and
                         the Network Chain Data is sent to the I/O Request Processing. *)
                          | TransactionTimeOut:

                              BitTimeForResponse        := 0.0;
                              ResponseChain^.AnyFailed := TRUE;
                              NEW(NetworkInput);

                              (* Put data in Input Frame for this transaction indicating
                                 that the current transaction timed out.  *)
                              WITH NetworkInput^ DO

                                  TransactionIdentifier := TransactionUnderExecution^.
                                                  Identifier;
                                  MessageAddress    := TransactionUnderExecution^.
                                                  OutputFrame^.Address;
                                  TransactionTimeOutIndicator := TimedOut;

                              END;

                              REPORT "%d"  NetworkInput^.TransactionIdentifier
                                  TAGGED "Transaction timed out.";
```

```
(*                  (* Put Input Frame on response queue.  *)

*)                  INSERT NetworkInput LAST IN ResponseChain^.InputFrameQueue;

                    QInsert(NetworkInput, ResponseChain^.InputFrameQueue, FALSE);

                    (* Check for more transactions to process.  *)
                    IF NumberOfTransactions = 0 THEN

                        (* Set completion flag of the I/O System
                           service.. *)
                        ChainStatus := CompletedNormally;
                        AFTER BitTimeForResponse + TransactionTurnAroundTime
                            EndIOActivity <- TRUE;

                   ELSE

                        ProcessTransaction(TRUE, NetworkInput,
                                ChainUnderExecution,
                                TransactionUnderExecution, NumberOfTransactions,
                                BitTimeForResponse);

                   END;

               (*****************************************************)
               (* This event handles a stop chain from the I/O System
                  Service when a chain has timed out. This process
                  marks the current transaction as timed out and
                  prepares to receive the next chain.  *)
               | StopChain:

                   NEW(NetworkInput);

                   (* Put data in Input Frame for this transaction indicating
                      that the current transaction timed out.  *)
                   WITH NetworkInput^ DO

                       TransactionIdentifier := TransactionUnderExecution^.
                                       Identifier;
                       MessageAddress     := TransactionUnderExecution^.
                                       OutputFrame^.Address;
                       TransactionTimeOutIndicator := TimedOut;

                   END;

(*                 (* Put Input Frame on response queue.  *)

*)                 INSERT NetworkInput LAST IN ResponseChain^.InputFrameQueue;

                   QInsert(NetworkInput, ResponseChain^.InputFrameQueue, FALSE);

                   ChainStatus := timedout;

                   EXIT;

               | EndIOActivity:

                   IF NetworkID = 1 THEN
                      SAMPLE 0.0 WITH SystemProbe[ProbeNumber];
                   END;

                   NOW outport[IOServicePort]^.ChainFinished <- TRUE;

                   REPORT "%d,"  ChainUnderExecution^.ChainIdentifier
                       TAGGED "End I/O Activity.";
                   (* Record end I/O activity data collection event.  *)
                   EndIODataElement.SimulationTime          := clock;

                   IF ChainUnderExecution^.ChainIdentifier < 20 THEN

                       EndIODataElement.Frequency               := Cyclic;
```

```
  EndIODataElement.CyclicData.FrameCount     := ChainUnderExecution^.FrameCount;

  CASE ChainUnderExecution^.ChainIdentifier OF

      1:

          EndIODataElement.CyclicData.C_Variation    := EndNW1IOActivity;
          EndIODataElement.EventID                        := 10;
          EndIODataElement.CyclicData.IOChainStatus  := ChainStatus;

      | 2:

          EndIODataElement.CyclicData.C_Variation    := EndNW1IOActivity;
          EndIODataElement.EventID                        := 11;
          EndIODataElement.CyclicData.IOChainStatus  := ChainStatus;

      | 3:

          EndIODataElement.CyclicData.C_Variation    := EndNW1IOActivity;
          EndIODataElement.EventID                        := 12;
          EndIODataElement.CyclicData.IOChainStatus  := ChainStatus;


      | 5:

          EndIODataElement.CyclicData.C_Variation    := EndNW1IActivity;
          EndIODataElement.EventID                        := 19;
          EndIODataElement.CyclicData.IChainStatus  := ChainStatus;

      | 6:

          EndIODataElement.CyclicData.C_Variation    := EndNW1IActivity;
          EndIODataElement.EventID                        := 20;
          EndIODataElement.CyclicData.IChainStatus  := ChainStatus;

      | 7:

          EndIODataElement.CyclicData.C_Variation    := EndNW1IActivity;
          EndIODataElement.EventID                        := 21;
          EndIODataElement.CyclicData.IChainStatus  := ChainStatus;


      | 9:

          EndIODataElement.CyclicData.C_Variation    := EndNW1OActivity;
          EndIODataElement.EventID                        := 25;

      | 10:

          EndIODataElement.CyclicData.C_Variation    := EndNW1OActivity;
          EndIODataElement.EventID                        := 26;

      | 11:

          EndIODataElement.CyclicData.C_Variation    := EndNW1OActivity;
          EndIODataElement.EventID                        := 27;

    END;

ELSIF (ChainUnderExecution^.ChainIdentifier >= 300) AND
    (ChainUnderExecution^.ChainIdentifier <= 310) THEN

        EndIODataElement.Frequency                  := NonCyclic;
        EndIODataElement.NonCyclicData.N_Variation := ReconfigIOActivity;
        EndIODataElement.EventID                    := 31;

END;

IF ((NetworkID = 1) AND
    (ChainUnderExecution^.ChainIdentifier <= 20)) OR
    ((NetworkID = 2) AND
    (ChainUnderExecution^.ChainIdentifier >= 300)) THEN
```

```
                        WriteDataElementType(EndIODataElement);

                    END;

                    EXIT;

                | DIUWritten:

                    REPORT "%d"  TransactionUnderExecution^.Identifier
                        TAGGED "DIU Output Transaction completed normally.";

                    (* Check for more transactions to process.  *)
                    IF NumberOfTransactions = 0 THEN

                        REPORT "%d"  ChainUnderExecution^.ChainIdentifier
                           TAGGED "Output Chain has completed execution.";

                        ChainStatus := CompletedNormally;

                        IF NetworkID = 1 THEN
                           SAMPLE 0.0 WITH SystemProbe[ProbeNumber];
                        END;

                        (* Set completion flag of the I/O System service.. *)
                        AFTER TransactionTurnAroundTime EndIOActivity <- TRUE;

                   ELSE

                       TransactionUnderExecution := QSucc(TransactionUnderExecution,
                                                           ChainUnderExecution^.TransactionQueue);

                       DEC(NumberOfTransactions);

                       WITH TransactionUnderExecution^ DO

                          (* Bit Time to transmit the next message.  *)
                          TempTime := ApplicationTransmitBits[ChainUnderExecution^.ChainIdentifier,
                                      (Identifier MOD Max_TransactionsInChain)];

                          REPORT "%d"  Identifier TAGGED "Transaction started Execution.";

                          AFTER TempTime outport[NetworkPort]^.OutputTransaction <- OutputFrame;
                          AFTER TempTime DIUWritten <- TRUE;

                       END;

                  END;

              | Reset:

                  EXIT;

              | ProbeReset:

                  IF NetworkID = 1 THEN
                     ClearProbe(SystemProbe[ProbeNumber]);
                     SAMPLE 1.0 WITH SystemProbe[ProbeNumber];
                  END;

          END;

      END;

| ProbeReset:

     IF NetworkID = 1 THEN
        ClearProbe(SystemProbe[ProbeNumber]);
        SAMPLE 0.0 WITH SystemProbe[ProbeNumber];
     END;

| Reset:
        END;
    END;
END IOS.
```

PROCESSOR

```
DEFINITION DEVM Processor;

(* 1/27/88 PRB added import of LONGREAL *)
FROM Util IMPORT LONGREAL;

EXPORT ProcessingUnit*;

(*
 * a processing unit is an entity that contains information about processes
 * which compete for the Processor.
 *)
TYPE
   ProcessingUnit = ENTITY
                         Priority           : INTEGER;
                         ProcessingRequired : REAL;
                         ProcessID          : ARRAY [0 .. 31] OF CHAR;
                         Frame              : INTEGER;
                         WriteData          : BOOLEAN;
                         ProcssngAfterBlock : REAL;
                         Data               : ADDRESS;
                         ProcessingLeftToDo : REAL;    (* don't touch *)
(* 1/27/88 PRB changed Started to type LONGREAL from REAL *)
                         Started            : LONGREAL;(* don't touch *)
                         SendBackOutThisPort: INTEGER; (* don't touch *)
                         DataWritten        : BOOLEAN; (* don't touch *)
                     END;

END Processor.
```

PROCESSOR

```
DEVM Processor;

(* 1/27/88 PRB imported DFLoatReal *)
FROM Conversions IMPORT DFloatToReal;

FROM Controls REACH SystemProbe, NumberOfProbes;    (* NOTICE that this is a REACH and not and IMPORT - this *)
                                                    (*      forces the denet compiler to read controls.ddef  *)
FROM Senddata IMPORT WriteDataElementType, CyclicDataType, DataElementType,
                     CyclicVariationType, FrequencyType;

INPUTS

   EVENT
      SubmitProcess : ProcessingUnit; (* submit a process to the processor *)
      StartSystem   : BOOLEAN;        (* start the system process running *)
      Reset         : BOOLEAN;
      ProbeReset    : BOOLEAN;

   PARA
      SystemPriority          : INTEGER;    (* priority of system process *)
      SystemProcessingNeeded  : REAL;       (* amount of processing needed to
                                             * execute the system process *)
      SystemFrequency         : REAL;       (* how often the system needs
                                             * to run *)
      ContextSwitchTime       : REAL;       (* context switching time *)
      ProcessorReportLevel    : INTEGER;    (* report parameter *)
      ProbeNumber             : INTEGER;    (* which system probe to use *)

END;

OUTPUTS

   VAR
      Completed      : ProcessingUnit;      (* When a process has completed, it
                                             * is returned to the caller *)

END;

EVENT
   ProcessCompleted : BOOLEAN;  (* check for completed processes *)
   RunSystem        : BOOLEAN;  (* system needs to run *)

TYPE

   ProcessingQueue = QUEUE OF DESCENDING ProcessingUnit;

CONST

   ContextSwitchPriority = 12345678;  (* big number *)

VAR

   ReadyQueue        : ProcessingQueue;  (* processes waiting for processing  *)

   CurrentProcess    : ProcessingUnit;   (* process that comes in on a port *)

   ExecutingProcess  : ProcessingUnit;   (* process that has the processor *)

   SystemProcess     : ProcessingUnit; (* the system process *)

   ContextSwitchProcess : ProcessingUnit; (* process which represents the context switch time *)

   CompletionNotice  : EventPointer;

   ProcessSwitchingTo : ProcessingUnit;  (* process that the cpu is switching to *)

   OutputDataElement : DataElementType;


PROCEDURE BeginExecutionOfProcess(DesiredProcess : ProcessingUnit);
BEGIN
   ExecutingProcess := DesiredProcess;
```

```
 (* 1/27/88 PRB changed clock to longclock *)
    ExecutingProcess^.Started := LongClock;
    AFTER ExecutingProcess^.ProcessingLeftToDo ProcessCompleted <- TRUE;
    CompletionNotice := CurrentNotice;
    REPORT[ProcessorReportLevel] "%s" ExecutingProcess^.ProcessID TAGGED "Started";
END BeginExecutionOfProcess;


BEGIN

   NEW(SystemProcess);
   WITH SystemProcess^ DO
       Priority := SystemPriority;
       ProcessingLeftToDo := SystemProcessingNeeded;
       ProcessID := 'System';
       DataWritten := TRUE;
   END;

   NEW(ContextSwitchProcess);
   WITH ContextSwitchProcess^ DO
       Priority := ContextSwitchPriority;
       ProcessingLeftToDo := ContextSwitchTime;
       ProcessID := 'Context Switch';
       DataWritten := TRUE;
   END;

   ExecutingProcess := NIL;

   LOOP

      WAITUNTIL EVENT

         SubmitProcess :  (* process is ready to use the processor *)

            CurrentProcess := ActivePort^.SubmitProcess;
           (*
            * Figure out who sent the process to you, so that you can
            * send it back to them once the process has completed.
            *)
            CurrentProcess^.SendBackOutThisPort := GetOutPort(GetInNode(
                                     ActivePort^.index));
            CurrentProcess^.ProcessingLeftToDo := CurrentProcess^.ProcessingRequired;
            CurrentProcess^.DataWritten := NOT CurrentProcess^.WriteData;

            REPORT[ProcessorReportLevel] "%s" CurrentProcess^.ProcessID TAGGED "Submitted";

            IF ExecutingProcess = NIL THEN
               INSERT CurrentProcess IN ReadyQueue;
               BeginExecutionOfProcess(ContextSwitchProcess);
               ProcessSwitchingTo := CurrentProcess;
               SAMPLE 1.0 WITH SystemProbe[ProbeNumber];
            ELSIF CurrentProcess^.Priority > ExecutingProcess^.Priority THEN
               CANCEL CompletionNotice;
(* 1/27/88 PRB Changed calculation to double precision *)
               ExecutingProcess^.ProcessingLeftToDo := ExecutingProcess^.ProcessingLeftToDo -
                                          DFloatToReal(LongClock - ExecutingProcess^.Started);

               REPORT[ProcessorReportLevel] "%s,%10.6f"  ExecutingProcess^.ProcessID,
               ExecutingProcess^.ProcessingLeftToDo TAGGED "Preempted";
               INSERT ExecutingProcess IN ReadyQueue;
               INSERT CurrentProcess IN ReadyQueue;
               BeginExecutionOfProcess(ContextSwitchProcess);
               ProcessSwitchingTo := CurrentProcess;
            ELSIF ExecutingProcess^.Priority = ContextSwitchPriority THEN
               INSERT CurrentProcess IN ReadyQueue;
(* 1/27/88 PRB Changed calculation to double precision *)
               IF ExecutingProcess^.Started = LongClock THEN
                  ProcessSwitchingTo := FirstQ(ReadyQueue);
               END;
            ELSE
               INSERT CurrentProcess IN ReadyQueue;
            END;
```

```
| ProcessCompleted :   (* check for completed processes *)

    REPORT[ProcessorReportLevel] "%s"  ExecutingProcess^.ProcessID TAGGED "Finished";
    IF NOT ((ExecutingProcess^.Priority = SystemPriority) OR
            (ExecutingProcess^.Priority = ContextSwitchPriority)) THEN
       NOW outport[ExecutingProcess^.SendBackOutThisPort]^.Completed <- ExecutingProcess;
    END;
    IF ExecutingProcess^.Priority <> ContextSwitchPriority THEN
       BeginExecutionOfProcess(ContextSwitchProcess);
       IF QSize(ReadyQueue) <> 0 THEN
          ProcessSwitchingTo := FirstQ(ReadyQueue);
       ELSE
          ProcessSwitchingTo := NIL;
       END;
    ELSIF QSize(ReadyQueue) <> 0 THEN
       IF ProcessSwitchingTo = FirstQ(ReadyQueue) THEN
          REMOVE FIRST CurrentProcess FROM ReadyQueue;
          IF (MyNodeID = 45) AND (NOT CurrentProcess^.DataWritten) THEN
             CurrentProcess^.DataWritten := TRUE;
             OutputDataElement.SimulationTime := clock;
             OutputDataElement.Frequency := Cyclic;
             OutputDataElement.CyclicData.FrameCount := CurrentProcess^.Frame;
             OutputDataElement.CyclicData.C_Variation := StartComputing;
             OutputDataElement.CyclicData.ProcessingNeededThisFrame :=
                  CurrentProcess^.ProcessingRequired + CurrentProcess^.ProcsngAfterBlock;
             CASE CurrentProcess^.Priority OF
                10:  OutputDataElement.EventID := 1;
              |  9:  OutputDataElement.EventID := 2;
              |  8:  OutputDataElement.EventID := 3;
             END;
             WriteDataElementType(OutputDataElement);
          ELSIF (MyNodeID = 46) AND (NOT CurrentProcess^.DataWritten) THEN
             CurrentProcess^.DataWritten := TRUE;
             OutputDataElement.SimulationTime := clock;
             OutputDataElement.Frequency := Cyclic;
             OutputDataElement.CyclicData.FrameCount := CurrentProcess^.Frame;
             OutputDataElement.CyclicData.C_Variation := StartChain;
             CASE CurrentProcess^.Priority OF
                10:  OutputDataElement.EventID := 13;
              |  9:  OutputDataElement.EventID := 14;
              |  8:  OutputDataElement.EventID := 15;
             END;
             WriteDataElementType(OutputDataElement);
          END;
          BeginExecutionOfProcess(CurrentProcess);
       ELSE   (* process of higher priority arrived during context switch *)
          BeginExecutionOfProcess(ContextSwitchProcess);
          ProcessSwitchingTo := FirstQ(ReadyQueue);
       END;
    ELSE
       ExecutingProcess := NIL;
       SAMPLE 0.0 WITH SystemProbe[ProbeNumber];
    END;

| RunSystem :  (* system process needs to run *)

    INSERT SystemProcess IN ReadyQueue;
    REPORT[ProcessorReportLevel] "%s"  SystemProcess^.ProcessID TAGGED "Submitted";
    IF (ExecutingProcess <> NIL) AND ((ExecutingProcess^.Priority = ContextSwitchPriority) AND
                                      (QSize(ReadyQueue) = 1)) THEN
         (*
          * Trying to submit the process during a context switch to the background process.  This requires
          * a second context switch to occur before the system process can run.
          *)
          ProcessSwitchingTo := NIL;    (* has effect of causing another context switch *)
    ELSIF ExecutingProcess = NIL THEN
       BeginExecutionOfProcess(ContextSwitchProcess);
       ProcessSwitchingTo := SystemProcess;
       SAMPLE 1.0 WITH SystemProbe[ProbeNumber];
    ELSIF ExecutingProcess^.Priority <> ContextSwitchPriority THEN
       CANCEL CompletionNotice;
```

```
 (* 1/27/88 PRB Changed calculation to double precision *)
               ExecutingProcess^.ProcessingLeftToDo := ExecutingProcess^.ProcessingLeftToDo -
                                             DFloatToReal(LongClock - ExecutingProcess^.Started);
               INSERT ExecutingProcess IN ReadyQueue;
               REPORT[ProcessorReportLevel] "%s"  ExecutingProcess^.ProcessID TAGGED "Preempted";
               BeginExecutionOfProcess(ContextSwitchProcess);
               ProcessSwitchingTo := SystemProcess;
            END;
            AFTER SystemFrequency RunSystem <- TRUE;

        | StartSystem :  (* Start the system process running periodically *)

            NOW RunSystem <- TRUE;

        | Reset :

            IF (ExecutingProcess <> NIL) AND ((ExecutingProcess^.Priority <> SystemPriority) AND
                                              (ExecutingProcess^.Priority <> ContextSwitchPriority)) THEN
               DISPOSE(ExecutingProcess);
            END;

            (*
             * Clear the queue
             *)
            FOREACH ExecutingProcess IN ReadyQueue DO
               REMOVE FIRST ExecutingProcess FROM ReadyQueue;
               IF (ExecutingProcess^.Priority <> SystemPriority) AND
                  (ExecutingProcess^.Priority <> ContextSwitchPriority) THEN
                  DISPOSE(ExecutingProcess);
               END;
            END;

            ExecutingProcess := NIL;

        | ProbeReset :

            ClearProbe(SystemProbe[ProbeNumber]);
            IF ExecutingProcess <> NIL THEN
               SAMPLE 1.0 WITH SystemProbe[ProbeNumber];
            ELSE
               SAMPLE 0.0 WITH SystemProbe[ProbeNumber];
            END;

      END;
   END;
END Processor.
```

MATH

```
DEFINITION MODULE Math;

    EXPORT QUALIFIED RealMod;

    PROCEDURE RealMod(X : REAL;
                      Y : REAL) : REAL;

END Math.
```

MATH

```
IMPLEMENTATION MODULE Math;

    FROM MathLib0 IMPORT mod;

    PROCEDURE RealMod(X : REAL;
                      Y : REAL) : REAL;

    BEGIN

        RETURN(mod(X,Y));

    END RealMod;

END Math.
```

IOSERVICE

```
DEFINITION DEVM IOService;

    FROM IOS REACH ChainType*, ChainStatusData*;

    FROM TypeConst IMPORT NumberOfNetworks;

    EXPORT IORequestType*, IOResponseType*,
                IOActivityType, RequestActivityType, ResponseActivityType,
                NetworkManagerActivityType,
                NetworkManagerServiceRequest*, NetworkHealthType,
                ChainStatusType, NewNetworkStateType*,
                ChainExecutedWithoutError, ReleaseChainResponseMemory;

    TYPE NetworkHealthType = (InService, OutOfService);
         IOActivityType = (ApplicationRequest, NetworkManagerRequest,
                MonitorForFault, ApplicationResponse,
                NetworkManagerResponse, FaultMonitorResponse);

         RequestActivityType        = [ApplicationRequest .. MonitorForFault];
         ResponseActivityType       = [ApplicationResponse .. FaultMonitorResponse];
         NetworkManagerActivityType = (GrowNetwork, RepairFault, SwitchRootLink);
         ChainStatusType            = (NoFaults, CommunicationFault,
                                       NotExecuted);

         NewNetworkStateType = ENTITY

           NetworkID    : INTEGER;
           State        : NetworkHealthType;
           MonitorChain : ChainType;

         END;

         IORequestType = ENTITY

            Priority            : INTEGER;
            Identifier          : INTEGER;
            RequestTimeoutValue : REAL;
            OnDemand            : BOOLEAN;
            Frame               : INTEGER;
            ResponseExpected    : BOOLEAN;

            CASE RequestType : RequestActivityType OF

               ApplicationRequest:

                   ChainArray : ARRAY [1 .. NumberOfNetworks] OF ChainType;

               | NetworkManagerRequest, MonitorForFault:

                   Chain            : ChainType;

           END;

        END;

        IOResponseType = ENTITY

            Identifier    : INTEGER;

            CASE ResponseType  : ResponseActivityType OF

               ApplicationResponse:

                   Frame         : INTEGER;
                   ChainStatus   : ARRAY [1 .. NumberOfNetworks] OF ChainStatusType;
                   ResponseArray : ARRAY [1 .. NumberOfNetworks] OF ChainStatusData;

               | NetworkManagerResponse, FaultMonitorResponse:

                   Response : ChainStatusData;

           END;
```

```
       END;

       NetworkManagerServiceRequest = ENTITY

           CASE ServiceRequest : NetworkManagerActivityType OF

               GrowNetwork:

                   ActiveRootLink : INTEGER;

                | RepairFault:

                   MonitorChainResponse : IOResponseType;

                | SwitchRootLink:

                   FailedRootNode : INTEGER;
                   NewRootNode    : INTEGER;

            END;

        END;
   (*************************************************************************)

   PROCEDURE ChainExecutedWithoutError(Chain : ChainStatusData) :BOOLEAN;

   (*************************************************************************)

   PROCEDURE ReleaseChainResponseMemory(VAR Response : ChainStatusData);

   (*************************************************************************)

END IOService.
```

IOSERVICE

```
(******************************************************************************)
(* 1/27/88  This change supports implements:

        a) Waiting until all higher priority processing is completed
           before submitting a pending I/O Request.

        b) When a pending I/O request is made, the semaphore is
           marked busy when the pending I/O processor request is made.   *)

(******************************************************************************)

DEVM IOService;

    FROM IOS REACH ChainType*, ChainStatusData*,
            InputFrameType*, TransactionType*;

    FROM Processor REACH ProcessingUnit*;

    FROM Controls REACH SystemProbe, NumberOfProbes;

    FROM CentralDB IMPORT IOSConnectionType, FindIOSConnections,
            ReadNodeInterConnections;

    FROM IOS IMPORT TimeOutIndicatorType;

    FROM GrowNet IMPORT MakeMonitorRequest;

    FROM Utilities IMPORT ManagerChainUnloadTime, ComputeChainTimeout;

    FROM TypeConst IMPORT StatusType, NumberOfNetworks, NumberOfIOSPerChannel,
            NodeArrayType;

    FROM BusMessag IMPORT MessageType, IOActivityChoice;

    FROM Senddata IMPORT DataElementType, FrequencyType, NonCyclicVariationType,
            NonCyclicDataType, WriteDataElementType;

    FROM Math IMPORT RealMod;

    IMPORT SYSTEM;

    IMPORT QueueM;

    EXPOSE

        (***********************************************************************)

        PROCEDURE ChainExecutedWithoutError(Chain : ChainStatusData) :BOOLEAN;

        BEGIN

            RETURN NOT((Chain^.AnyFailed) OR (Chain^.AllFailed) OR
                (Chain^.ChainTimeOutIndicator = TimedOut));

        END ChainExecutedWithoutError;

        (***********************************************************************)

        PROCEDURE ReleaseChainResponseMemory(VAR Response : ChainStatusData);

            VAR Frame            : InputFrameType;
                ElementCounter   : INTEGER;
                NumberOfElements : INTEGER;

        BEGIN

            IF Response <> NIL THEN

                NumberOfElements := QSize(Response^.InputFrameQueue);
                FOR ElementCounter := 1 TO NumberOfElements DO

                    Frame := QRemove(Response^.InputFrameQueue, TRUE);
                    IF Frame^.TransactionTimeOutIndicator = NormalCompletion THEN

                        DISPOSE(Frame^.InputFrame);

                    END;

                    DISPOSE(Frame);

                END;

                QDispose(Response^.InputFrameQueue, SIZE(Frame));
                DISPOSE(Response);

            ELSE

                WriteLn(ParamOut);
                WriteString(ParamOut, "Tried to DISPOSE a NIL Chain Response.");
                WriteLn(ParamOut);

            END;

        END ReleaseChainResponseMemory;

        (**********************************************************************)

    END;

    INPUTS
        EVENT IOServiceRequest        : IORequestType;
              RtnNetworkToService     : NewNetworkStateType;
              ProcessorResponse       : ProcessingUnit;
              MissedDeadLine          : INTEGER;
              Reset                   : BOOLEAN;
              ProbeReset              : BOOLEAN;

        VAR   DataFromIOS             : ChainStatusData;
              ChainCompleted          : BOOLEAN;

        PARA  ManagerIDNetwork2             : INTEGER;
              ApplicationID                 : ARRAY [1 .. 3] OF INTEGER;
              IOPIdentifier                 : INTEGER;
              ControlsIdentifier            : INTEGER;
              StrategyForReconfiguration    : INTEGER;
              ChainProcessing100Hz          : REAL;
              ChainProcessing50Hz           : REAL;
              ChainProcessing25Hz           : REAL;
              EndOfChainProcessing100Hz     : REAL;
              EndOfChainProcessing50Hz      : REAL;
              EndOfChainProcessing25Hz      : REAL;
              EndOfChainProcessingMonitor   : REAL;
              ProbeNumber                   : INTEGER;


    END;

    OUTPUTS
        VAR   ChainToIOS              : ChainType;
              ApplicationResponse     : IOResponseType;
              IOManager2Response      : ChainStatusData;
              ManagerServiceRqst      : NetworkManagerServiceRequest;
              ProcessorRequest        : ProcessingUnit;
              ServiceAvailable        : BOOLEAN;
              StopIOS                 : BOOLEAN;

    END;

    TYPE IOServiceStateType = RECORD

            Executing100Hz : BOOLEAN;
            Executing50Hz  : BOOLEAN;
            Executing25Hz  : BOOLEAN;
    NetworkHealth   : ARRAY[1 .. NumberOfNetworks] OF NetworkHealthType;
    ActiveRootLink : ARRAY[1 .. NumberOfNetworks] OF INTEGER;

END;

NetworkConnectionType = RECORD

    NetworkConnections : IOSConnectionType;
    ConnectionStatus   : ARRAY [1 .. NumberOfIOSPerChannel] OF StatusType;

END;

IOServiceNetworkConnectionsType = ARRAY [1 .. NumberOfNetworks] OF
       NetworkConnectionType;

RequestExecutionStatus = ENTITY

    Request     : IORequestType;

    CASE RequestType : IOActivityType OF

       ApplicationRequest:

          ExecutingOn : ARRAY [1 .. NumberOfNetworks] OF BOOLEAN;
          PortArray   : ARRAY [1 .. NumberOfNetworks] OF INTEGER;

       | NetworkManagerRequest, MonitorForFault:

          Port  : INTEGER;

   END;

END;

SemaphoreType = ENTITY

    Priority         : INTEGER;
    ExecutionStatus : RequestExecutionStatus;

END;

SemaphoreQueueType = QUEUE OF DESCENDING SemaphoreType;

AlgorithmType = (PowerUp, IORequestSchedule, IOResponseSchedule,
                 OutputResponseSchedule, StartPendingIO, NetworkRepair,
                 StartFaultMonitor);

UnloadRequestType = RECORD

    Response  : IOResponseType;

    CASE ResponseType : IOActivityType OF

         NetworkManagerResponse, FaultMonitorResponse:

            ManagerID : INTEGER;

    END;

END;

DataType = RECORD

     CASE Function : AlgorithmType OF

          PowerUp:

          | IORequestSchedule:

             IORequest : IORequestType;

          | IOResponseSchedule:
```

```
                     IOResponse : IOResponseType;

                 | OutputResponseSchedule:

                     Rate : INTEGER;

                 | StartPendingIO:

(* 1/27/88 *)        RequestStatus : RequestExecutionStatus;

                 | StartFaultMonitor:

                 | NetworkRepair:

                     RepairData : NetworkManagerServiceRequest;

             END;

         END;

         DataPointer = POINTER TO DataType;

    EVENT IOCompletionPoll    : RequestExecutionStatus;
          InitializeService   : BOOLEAN;

    CONST IOSystemServicePriority          = 1;
          NetworkID2                       = 2;

          PowerupIntialize                 = 0.000100;
          RequestProcessing                = 0.000025;
          ResponseCompletionProcessing     = 0.000025;
          ChainProcessingOverhead          = 0.000050;
          EndOfChainProcessingOverhead     = 0.000050;
          ChainProcessing1Transaction      = 0.000018;
          ChainProcessing2Transactions     = 0.000036;
          EndOfChainProcessing1Transaction = 0.000052;
          EndOfChainProcessing2Transactions = 0.000104;
          SwitchRootLinkProcessing         = 0.000025;

          SemaphoreGranted    = -1;
          SemaphoreNotGranted = -2;
          NoPendingRequest    = -3;

    VAR FaultMonitor2             : IORequestType;
        ServiceRequest            : IORequestType;
        RequestToService          : RequestExecutionStatus;
        ResponseData              : UnloadRequestType;
        NewNetworkState           : NewNetworkStateType;
        IOServiceState            : IOServiceStateType;
        PendingRequest            : SemaphoreType;
        PendingStatus             : INTEGER;
        IOSystemStatus            : BOOLEAN;
        ServiceNetworkRequest     : NetworkManagerServiceRequest;
        IONetworkConnections      : IOServiceNetworkConnectionsType;
        InterfaceID               : INTEGER;
        NetworkOutOfServiceTime   : ARRAY [1 .. NumberOfNetworks] OF REAL;
        TimeNetworkOutOfService   : REAL;
        RequestTime               : REAL;
        OnDemandRequestDelay      : REAL;
        TimeToUnloadRequest       : REAL;
        FailedRootNumber          : INTEGER;
        NetworkCounter            : INTEGER;
        InterfaceCounter          : INTEGER;

        ProcessingInformation     : DataPointer;
        PowerUpProcessing         : DataPointer;
        NetworkData               : DataPointer;
        ProcessRequest            : DataPointer;
        PendingIOProcessing       : DataPointer;
        ResponseDataProcessing    : DataPointer;
        RootLinkSwitchProcessing  : DataPointer;
```

```
   RepairNetworkProcessing     : DataPointer;
   PowerUpRequest              : ProcessingUnit;
   ProcessResponse             : ProcessingUnit;
   ProcessingOutputResponse    : ProcessingUnit;
   ProcessingStartPendingIO    : ProcessingUnit;
   ProcessingNetworkRepair     : ProcessingUnit;

   DataCollectionRecord        : DataElementType;

   NetworkManager2Port         : INTEGER;
   ApplicationPort             : ARRAY [1 .. 3] OF INTEGER;
   IOPPort                     : INTEGER;
   ControlsPort                : INTEGER;

   Network2Connections         : NodeArrayType;
(************************************************************************)

MODULE Semaphore;

    FROM SYSTEM IMPORT ADDRESS;

    FROM QueueM IMPORT InitQ, PrQInsert, QSize, QRemove, FirstQ,
            QSucc, carrier;

    IMPORT SemaphoreType, SemaphoreQueueType, SemaphoreGranted,
            SemaphoreNotGranted, NoPendingRequest;

    IMPORT WriteString, WriteLn, ParamOut;

    IMPORT dmeasure, SystemProbe, ProbeNumber;
    EXPORT Request, Release, NextRequest, SemaphoreIdle, SetSemaphoreIdle;

    VAR Idle           : BOOLEAN;
        SemaphoreQueue : SemaphoreQueueType;

    (*********************************************************************)

    PROCEDURE Request(Semaphore : SemaphoreType) : INTEGER;

        VAR Status        : INTEGER;

    BEGIN

        IF Idle THEN

            Status :- SemaphoreGranted;
            Idle   :- FALSE;

        ELSE

            Status :- SemaphoreNotGranted;
            INSERT Semaphore IN SemaphoreQueue;

        END;

        RETURN(Status);

    END Request;

    (*********************************************************************)
    (* This function returns the priority of the highest priority pending
        request.  If no request is waiting the semaphore is set to idle
        and No pending request is returned.  *)
    PROCEDURE Release (VAR Status : INTEGER);

        VAR PendingRequest : SemaphoreType;
    BEGIN

        Idle :- TRUE;
        IF QSize(SemaphoreQueue) <> 0 THEN
```

```
                PendingRequest := FirstQ(SemaphoreQueue);
                Status := PendingRequest^.Priority;

            ELSE

                Status := NoPendingRequest;

            END;

        END Release;

        (**********************************************************************)

        PROCEDURE NextRequest(VAR Request : SemaphoreType);

        BEGIN

            IF QSize(SemaphoreQueue) > 0 THEN

                REMOVE FIRST Request FROM SemaphoreQueue;
                Idle := FALSE;
                SAMPLE 1.0 WITH SystemProbe[ProbeNumber];

            ELSE

                WriteString(ParamOut, "Problem with semphore logic.");
                WriteLn(ParamOut);
                WriteString(ParamOut, "Trying to remove SEMAPHORE FROM EMPTY QUEUE.");
                WriteLn(ParamOut);

            END;

        END NextRequest;

        (**********************************************************************)
        PROCEDURE SemaphoreIdle(VAR Status : BOOLEAN);

        BEGIN

            Status := Idle;

        END SemaphoreIdle;

        (**********************************************************************)

        PROCEDURE SetSemaphoreIdle;

        BEGIN

            Idle := TRUE;

        END SetSemaphoreIdle;

        (**********************************************************************)

        BEGIN

            Idle := TRUE;

    END Semaphore;

    (**********************************************************************)

    PROCEDURE SubmitRequestProcessing(Request         : DataPointer;
                                      WriteTheData    : BOOLEAN;
                                      FrameCount      : INTEGER;
                                      Processing      : REAL;
                                      ProcessPriority : INTEGER);

        VAR ProcessRequest  : ProcessingUnit;

    BEGIN
    NEW(ProcessRequest);
    WITH ProcessRequest^ DO

        Priority           := ProcessPriority;
        ProcessingRequired := Processing;
        WriteData          := WriteTheData;
        Frame              := FrameCount;
        ProcessID          := 'RequestProcessing';
        Data               := Request;

    END;

    NOW outport[IOPPort]^.ProcessorRequest <- ProcessRequest;

END SubmitRequestProcessing;

(**************************************************************************)

PROCEDURE SubmitUnloadIORequest(ServiceResponse : DataPointer;
                                Processing      : REAL;
                                ProcessPriority : INTEGER);

    VAR ProcessRequest  : ProcessingUnit;

BEGIN

    NEW(ProcessRequest);
    WITH ProcessRequest^ DO

        Priority           := ProcessPriority;
        ProcessingRequired := Processing;
        WriteData          := FALSE;
        ProcessID          := 'UnloadIORequest';
        Data               := ServiceResponse;

    END;

    NOW outport[IOPPort]^.ProcessorRequest <- ProcessRequest;

END SubmitUnloadIORequest;

(**************************************************************************)

PROCEDURE ResponseCompletion(Response : IOResponseType);

    VAR ChainCounter : INTEGER;

BEGIN

    WITH Response^ DO

        IF ResponseType = ApplicationResponse THEN

            FOR ChainCounter := 1 TO NumberOfNetworks DO

                IF ChainStatus[ChainCounter] <> NotExecuted THEN

                    IF (ResponseArray[ChainCounter]^.AnyFailed) OR
                        (ResponseArray[ChainCounter]^.AllFailed) THEN

                        ChainStatus[ChainCounter] := CommunicationFault;

                    ELSE

                        ChainStatus[ChainCounter] := NoFaults;

                    END;

                ELSE

                    (* Chain not executed so nothing to process.  *)
                END;

            END;

        ELSIF (ResponseType = NetworkManagerResponse) OR
            (ResponseType = FaultMonitorResponse) THEN

            (* Network Manager is responsible for processing
               these I/O responses.  *)

        END;

    END;

END ResponseCompletion;

(**************************************************************************)
(* This procedure marks a root link on Network 2 failed.  *)
PROCEDURE FailRootLink(VAR RootLinkStatus : NetworkConnectionType;
                       RootLink           : INTEGER) :INTEGER;

    VAR RootLinkCounter : INTEGER;
        FailedNumber    : INTEGER;

BEGIN

    FOR RootLinkCounter := 1 TO NumberOfIOSPerChannel DO

        IF RootLinkStatus.NetworkConnections[RootLinkCounter].GPCAddress
            = RootLink THEN

            RootLinkStatus.ConnectionStatus[RootLinkCounter] := Failed;
            FailedNumber := RootLinkCounter;

        END;

    END;

    RETURN(FailedNumber);

END FailRootLink;

(**************************************************************************)

PROCEDURE SwitchRootLinks(VAR RootLinkStatus  : NetworkConnectionType;
                          NetworkID           : INTEGER;
                          FailedNumber        : INTEGER) : INTEGER;

    VAR RootLinkCounter : INTEGER;
        NewRootLink     : INTEGER;

BEGIN

    WITH RootLinkStatus DO

        RootLinkCounter := 1;

        LOOP

            IF ConnectionStatus[RootLinkCounter] = Idle THEN

                (* A new root link has been found.  *)
                ConnectionStatus[RootLinkCounter] := Active;
                NewRootLink := NetworkConnections[RootLinkCounter].GPCAddress;

                EXIT;

            ELSIF RootLinkCounter < NumberOfIOSPerChannel THEN

                RootLinkCounter := RootLinkCounter + 1;
            ELSE

                (* A good root link cannot be found.  *)
                NewRootLink := 0;
                EXIT;

            END;

        END;

    END;

    RETURN (NewRootLink);

END SwitchRootLinks;

(**************************************************************************)
PROCEDURE ComputeIOLoadTime(IORequest    : IORequestType;
                            ServiceState : IOServiceStateType) :REAL;

    VAR LoadTime : REAL;

    (*********************************************************************)
    (* This function looks at the first transaction in chain 1 of
       application IORequest to determine if the request is an separated-I
       reqeust. If so, TRUE is returned, otherwise, FALSE is returned.  *)
    PROCEDURE SeparatedIRequest(Chain : ChainType) : BOOLEAN;

        VAR Transaction : TransactionType;

    BEGIN

        Transaction := FirstQ(Chain^.TransactionQueue);
        IF (Transaction^.OutputFrame^.Message = DIUInput) AND
            (Transaction^.OutputFrame^.DIUCommand.Activity = Input) THEN

            RETURN(TRUE);

        ELSE

            RETURN(FALSE);

        END;

    END SeparatedIRequest;

    (*********************************************************************)

BEGIN

    IF IORequest^.RequestType = ApplicationRequest THEN

        (* Need to check for type of request, Grouped, Separated-I,
           Separated-O.  Separated I has different loading time.  *)
        IF SeparatedIRequest(IORequest^.ChainArray[1]) THEN

            LoadTime := 2.0 * ChainProcessingOverhead;

        ELSE

            CASE IORequest^.Identifier OF

                100:

                    LoadTime := (2.0 * ChainProcessingOverhead) + ChainProcessing100Hz;

                | 50:

                    LoadTime := (2.0 * ChainProcessingOverhead) + ChainProcessing50Hz;

                | 25:
                    LoadTime := (2.0 * ChainProcessingOverhead) + ChainProcessing25Hz;

            END;

        END;

        WITH ServiceState DO

            IF NOT ((NetworkHealth[1] = InService) AND (NetworkHealth[2]
                = InService)) THEN

                (* Cut load time in half since only one network
                   is in service.  *)
                LoadTime := 0.5 * LoadTime;

            END;

        END;

    ELSIF IORequest^.RequestType = NetworkManagerRequest THEN

        CASE IORequest^.Chain^.NumberOfTransactions OF

            1:

                LoadTime := RequestProcessing + ChainProcessingOverhead
                                + ChainProcessing1Transaction;

            | 2:

                LoadTime := RequestProcessing + ChainProcessingOverhead
                                    + ChainProcessing2Transactions;

            | 4, 18:

                (* This handle talker out of turn request during
                   network growth.  *)
                LoadTime := RequestProcessing + ChainProcessingOverhead;

        END;

    ELSIF IORequest^.RequestType = MonitorForFault THEN

        LoadTime := 0.0;

    END;

    RETURN(LoadTime);

END ComputeIOLoadTime;

(***************************************************************************)
PROCEDURE ComputeIOUnloadTime(IOResponse : IOResponseType) :REAL;

    VAR UnloadTime  : REAL;

BEGIN

    IF IOResponse^.ResponseType = ApplicationResponse THEN

        CASE IOResponse^.Identifier OF

            100:

                UnloadTime := (2.0 * EndOfChainProcessingOverhead) + EndOfChainProcessing100Hz;

            | 50:

                UnloadTime := (2.0 * EndOfChainProcessingOverhead) + EndOfChainProcessing50Hz;

            | 25:
              UnloadTime := (2.0 * EndOfChainProcessingOverhead) + EndOfChainProcessing25Hz;

        END;

        WITH IOResponse^ DO

            IF NOT ((ResponseArray[1] <> NIL) AND (ResponseArray[2] <> NIL)) THEN

                (* Cut unload time in half since only one network
                   is in service.  *)
                UnloadTime := 0.5 * UnloadTime;

            END;

        END;

    ELSIF IOResponse^.ResponseType = NetworkManagerResponse THEN

        CASE QSize(IOResponse^.Response^.InputFrameQueue) OF

            1:

                UnloadTime := EndOfChainProcessingOverhead + EndOfChainProcessing1Transaction;

            | 2:

                UnloadTime := EndOfChainProcessingOverhead + EndOfChainProcessing2Transactions;

            | 4, 18:

                (* This handle talker out of turn request during
                   network growth.  *)
                UnloadTime := EndOfChainProcessingOverhead + EndOfChainProcessingMonitor;

        END;

    ELSIF IOResponse^.ResponseType = FaultMonitorResponse THEN

        UnloadTime := EndOfChainProcessingOverhead + EndOfChainProcessingMonitor;

    END;

    RETURN(UnloadTime + ResponseCompletionProcessing);

END ComputeIOUnloadTime;

(***************************************************************************)
(* Check to make sure chain on network 2 executed without error.  If not,
   take the network 2 out of service and schedule a monitor for fault chain
   if the reconfiguration strategy is one shot repair, otherwise have
   the network manager regrow network 2.  *)
PROCEDURE CheckNetworksForFault(Response         : IOResponseType;
                                VAR ServiceState : IOServiceStateType);

    VAR FaultCheckRequest        : DataPointer;
        DataCollectionRecord     : DataElementType;
        RegrowNetworkRequest     : NetworkManagerServiceRequest;
        FaultMonitorProcessing   : DataPointer;
        ProcessingFaultMonitor   : ProcessingUnit;
        RegrowNetworkProcessing  : DataPointer;
        ProcessingRegrowNetwork  : ProcessingUnit;

BEGIN

    IF (ServiceState.NetworkHealth[2] = InService)
        AND (Response^.ChainStatus[2] =
        CommunicationFault) THEN

        ServiceState.NetworkHealth[2] := OutOfService;
        NetworkOutOfServiceTime[2]    := clock;
        REPORT "%12.8f" clock TAGGED "Network Out of Service at ";
```

```
     REPORT "%d" NetworkCounter TAGGED "The network with the failure is ";

     DataCollectionRecord.EventID                    := 28;
     DataCollectionRecord.SimulationTime             := clock;
     DataCollectionRecord.Frequency                  := NonCyclic;
     DataCollectionRecord.NonCyclicData.N_Variation  := NetServiceChange;
     DataCollectionRecord.NonCyclicData.NetworkID    := 2;

     WriteDataElementType(DataCollectionRecord);

     IF StrategyForReconfiguration = 0 THEN

         REPORT "%12.8f" clock TAGGED "Application Chain Fault";

         NEW(FaultMonitorProcessing);
         FaultMonitorProcessing^.Function := StartFaultMonitor;

         NEW(ProcessingFaultMonitor);
         WITH ProcessingFaultMonitor^ DO

             Priority           := IOSystemServicePriority;
             ProcessingRequired := RequestProcessing + ChainProcessingOverhead;
             WriteData          := FALSE;
             ProcessID          := 'Fault MonitorProcessing';
             Data               := FaultMonitorProcessing;

         END;

         NOW outport[IOPPort]^.ProcessorRequest <- ProcessingFaultMonitor;

     ELSE

         IOServiceState.ActiveRootLink[2] := IONetworkConnections[2].
                                                NetworkConnections[2].GPCAddress;
         NEW(RegrowNetworkRequest);
         NEW(RegrowNetworkRequest);
         WITH RegrowNetworkRequest^ DO

             ServiceRequest := GrowNetwork;
             ActiveRootLink := IOServiceState.ActiveRootLink[2];

         END;

         IONetworkConnections[2].ConnectionStatus[2] := Active;

         NEW(RegrowNetworkProcessing);
         RegrowNetworkProcessing^.Function     := NetworkRepair;
         RegrowNetworkProcessing^.RepairData := RegrowNetworkRequest;

         NEW(ProcessingRegrowNetwork);
         WITH ProcessingRegrowNetwork^ DO

             Priority           := IOSystemServicePriority;
             ProcessingRequired := 0.0;
             WriteData          := FALSE;
             ProcessID          := 'Regrow Network Processing';
             Data               := RegrowNetworkProcessing;

         END;

         NOW outport[IOPPort]^.ProcessorRequest <- ProcessingRegrowNetwork;

     END;

ELSIF (ServiceState.NetworkHealth[1] = InService)
    AND (Response^.ChainStatus[1] = CommunicationFault) THEN

    WriteString(ParamOut, "Unexpected fault found in network 1.");
    WriteLn(ParamOut);

ELSE
```

```
        (* Either the network is already out of service, or no
           communication faults occured. In either case there is
           nothing to do.  *)
    END;

END CheckNetworksForFault;

(*************************************************************************)
(* This procedure computes the time to load the I/O request on the
   network(s) that are in service, starts the chains, and schedules
   the I/O completion poll.  *)
PROCEDURE ExecuteApplicationRequest(IORequest    : IORequestType;
                                    ServiceState : IOServiceStateType);

    VAR Semaphore       : SemaphoreType;
        SemaphoreStatus : INTEGER;
        InterfaceID     : INTEGER;
        NetworkCounter  : INTEGER;
        LoadRequest     : RequestExecutionStatus;
BEGIN

    NEW(LoadRequest);
    LoadRequest^.Request     := IORequest;
    LoadRequest^.RequestType := ApplicationRequest;

    FOR NetworkCounter := 1 TO NumberOfNetworks DO

        IF (ServiceState.NetworkHealth[NetworkCounter] = InService) THEN

            (* Determine the active interface for the network that this
               request will execute on.  *)
            InterfaceID := ServiceState.ActiveRootLink[NetworkCounter];

            LoadRequest^.PortArray[NetworkCounter] := GetOutPort(InterfaceID);

            (* Mark network loaded with a chain.  *)
            LoadRequest^.ExecutingOn[NetworkCounter] := TRUE;

        ELSE

            (* Mark network not loaded with a chain.  *)
            LoadRequest^.ExecutingOn[NetworkCounter] := FALSE;

        END;

    END;

    (* Make a request for the I/O system semaphore, if the semaphore
       is idle, the request will be executed. Otherwise, if no other
       request of the same priority is waiting, the current request
       will be pended for later execution. If a request of the same
       priority is waiting, the current request will be ignored.  *)

    NEW(Semaphore);
    Semaphore^.Priority        := IORequest^.Priority;
    Semaphore^.ExecutionStatus := LoadRequest;

    SemaphoreStatus := Request(Semaphore);

    IF (SemaphoreStatus = SemaphoreGranted) THEN

        StartIOSs(LoadRequest, ServiceState);
        DISPOSE(Semaphore);

    END;

END ExecuteApplicationRequest;

(*************************************************************************)

PROCEDURE ExecuteManager2Request(IORequest    : IORequestType;
                                 ServiceState : IOServiceStateType);
```

```
     CONST Network2ID        = 2;

     VAR   TimeToLoadRequest : REAL;
           InterfaceID       : INTEGER;
           ExecutionRequest  : RequestExecutionStatus;
           LoadIOProcessing  : DataPointer;

 BEGIN

     (* This must is a request for a network that is out of service,
        Network Manager chain or Monitor for fault chain.  The status
        of the network is assumed out of service and no other chain
        is pending on this network.   *)

     (* Determine the active interface for the network that this
        request will execute on.  *)
     InterfaceID := ServiceState.ActiveRootLink[Network2ID];

     (* Compute the time to load this request in the DPM through
        the Data Exchange.  *)
     TimeToLoadRequest := ComputeIOLoadTime(IORequest, ServiceState);

     NEW(ExecutionRequest);
     ExecutionRequest^.Request      := IORequest;
     ExecutionRequest^.RequestType  := IORequest^.RequestType;
     ExecutionRequest^.Port         := GetOutPort(InterfaceID);

     StartIOSs(ExecutionRequest, ServiceState);

END ExecuteManager2Request;

(**************************************************************************)

PROCEDURE UnloadResponse(RequestToService : RequestExecutionStatus;
                         ServiceState     : IOServiceStateType;
                         VAR Data         : UnloadRequestType);

    VAR NetworkCounter : INTEGER;
        InterfaceID    : INTEGER;

BEGIN

    NEW(Data.Response);

    IF RequestToService^.Request^.RequestType = ApplicationRequest THEN

        Data.ResponseType             := ApplicationResponse;
        Data.Response^.ResponseType   := ApplicationResponse;
        Data.Response^.Identifier     := RequestToService^.Request^.Identifier;
        Data.Response^.Frame          := RequestToService^.Request^.Frame;
        (* Only unload those network(s) executing a request.  *)
        FOR NetworkCounter := 1 TO NumberOfNetworks DO

            IF RequestToService^.ExecutingOn[NetworkCounter] THEN

                (* This is set to force Request Completion processing
                   to process this chain.  Request completion
                   processing will set this to the proper value.  *)
                Data.Response^.ChainStatus[NetworkCounter] := NoFaults;

                (* Determine the active interface for the network
                   that this request was executing on.  *)
                InterfaceID := ServiceState.ActiveRootLink[NetworkCounter];

                WITH Data.Response^ DO

                    ResponseArray[NetworkCounter] := inport[GetInPort(
                                InterfaceID)]^.DataFromIOS;

                    IF inport[GetInPort(InterfaceID)]^.ChainCompleted THEN
```

```
                       (* Chain Completed normally.   *)
                       ResponseArray[NetworkCounter]^.ChainTimeOutIndicator
                               := NormalCompletion;

                   ELSE

                       (* Chain has timed out. Command IOS to stop
                          execution and unload chain.  *)
                       ResponseArray[NetworkCounter]^.ChainTimeOutIndicator
                               := TimedOut;

                       NOW outport[GetOutPort(InterfaceID)]^.StopIOS <- TRUE;

                   END;

               END;

           ELSE

               Data.Response^.ChainStatus[NetworkCounter]    := NotExecuted;
               Data.Response^.ResponseArray[NetworkCounter]  := NIL;

           END;

       END;

   ELSE

       IF RequestToService^.RequestType = NetworkManagerRequest THEN

           Data.ResponseType            := NetworkManagerResponse;
           Data.Response^.ResponseType  := NetworkManagerResponse;

       ELSE

           Data.ResponseType            := FaultMonitorResponse;
           Data.Response^.ResponseType  := FaultMonitorResponse;

       END;

       (* Unload the request *)
       InterfaceID               := ServiceState.ActiveRootLink[NetworkID2];
       Data.Response^.Response  := inport[GetInPort(InterfaceID)]^.DataFromIOS;

       IF inport[GetInPort(InterfaceID)]^.ChainCompleted THEN

           Data.Response^.Response^.ChainTimeOutIndicator := NormalCompletion;

       ELSE

           Data.Response^.Response^.ChainTimeOutIndicator := TimedOut;
           NOW outport[GetOutPort(InterfaceID)]^.StopIOS <- TRUE;

       END;

   END;

END UnloadResponse;
(**************************************************************************)

PROCEDURE StartIOSs(RequestToLoad  : RequestExecutionStatus;
                    ServiceState   : IOServiceStateType);

    CONST OneMilliSecond = 0.001;

    VAR   NetworkCounter            : INTEGER;
          BitTimeToNextMilliSecond  : REAL;

BEGIN

    WITH RequestToLoad^ DO
```

```
            IF RequestType = ApplicationRequest THEN

                REPORT "%d" Request^.Identifier TAGGED "Start Network Activity";

                FOR NetworkCounter := 1 TO NumberOfNetworks DO

                    IF ExecutingOn[NetworkCounter] THEN

                        IF ServiceState.NetworkHealth[NetworkCounter] = InService THEN

                            NOW outport[PortArray[NetworkCounter]]^.ChainToIOS
                                <- Request^.ChainArray[NetworkCounter];

                        ELSE

                            (* This chain was loaded when network was
                               in service, but has gone out of service
                               before the chain could begin execution.  *)
                            ExecutingOn[NetworkCounter] := FALSE;

                        END;

                    END;

                END;

            ELSE

                NOW outport[Port]^.ChainToIOS <- Request^.Chain;

            END;

            BitTimeToNextMilliSecond := OneMilliSecond - RealMod(clock, OneMilliSecond);
            AFTER (BitTimeToNextMilliSecond + Request^.RequestTimeoutValue)
                IOCompletionPoll <- RequestToLoad;

        END;

    END StartIOSs;

    (*************************************************************************)

BEGIN

    NetworkManager2Port := GetOutPort(ManagerIDNetwork2);
    ApplicationPort[1]  := GetOutPort(ApplicationID[1]);
    ApplicationPort[2]  := GetOutPort(ApplicationID[2]);
    ApplicationPort[3]  := GetOutPort(ApplicationID[3]);
    IOPPort             := GetOutPort(IOPIdentifier);
    ControlsPort        := GetOutPort(ControlsIdentifier);

    (* Request IOP to complete Power Up Initialization.  *)
    NEW(PowerUpProcessing);
    WITH PowerUpProcessing^ DO

        Function := PowerUp;

    END;

    NEW(PowerUpRequest);
    WITH PowerUpRequest^ DO

        Priority           := IOSystemServicePriority;
        ProcessingRequired := PowerupIntialize;
        WriteData          := FALSE;
        ProcessID          := 'IOServicePowerUp';
        Data               := PowerUpProcessing;

        END;

        NOW outport[IOPPort]^.ProcessorRequest <- PowerUpRequest;
```

```
LOOP

    WAITUNTIL (ProcessorResponse)

        ProcessorResponse:

            ProcessResponse := ActivePort^.ProcessorResponse;
            NetworkData     := ProcessResponse^.Data;

            WITH NetworkData^ DO

               IF Function = PowerUp THEN

                    NOW InitializeService <- TRUE;
                    EXIT;

                ELSE

                    WriteString(ParamOut, "Problem during power up processing. ");
                    WriteLn(ParamOut);

                END;

            END;

    END;

    DISPOSE(NetworkData);
    DISPOSE(ProcessResponse);

END;

(* This loop waits for the initialize service request.  This request
   would happen at power up in a flight system.  *)

LOOP

    WAITUNTIL (InitializeService)

        InitializeService:

            (* Find out how many IOS's each network has.  *)
            FindIOSConnections(1, IONetworkConnections[1].NetworkConnections);
            FindIOSConnections(2, IONetworkConnections[2].NetworkConnections);

            (* Initialize all network connections to idle.  *)
            FOR NetworkCounter := 1 TO NumberOfNetworks DO

                FOR InterfaceCounter := 1 TO NumberOfIOSPerChannel DO

                    IF IONetworkConnections[NetworkCounter].
                        NetworkConnections[InterfaceCounter].GPCAddress
                        <> 0 THEN

                        IONetworkConnections[NetworkCounter].
                            ConnectionStatus[InterfaceCounter] := Idle;

                    ELSE

                        (* This simulation run does not have an interface
                           with this ID, set its status to failed.  *)

                        IONetworkConnections[NetworkCounter].
                            ConnectionStatus[InterfaceCounter] := Failed;

                    END;

                END;

            END;

            (* Initialize Active interfaces. Since all hardware should
               be "good", the first interface will be used.  *)

            IONetworkConnections[1].ConnectionStatus[1] := Active;
            IONetworkConnections[2].ConnectionStatus[1] := Active;
            IOServiceState.NetworkHealth[1]             := InService;
            IOServiceState.NetworkHealth[2]             := InService;
            IOServiceState.ActiveRootLink[1]            := IONetworkConnections[1].NetworkConnections[1].GPCAddress;
            IOServiceState.ActiveRootLink[2]            := IONetworkConnections[2].NetworkConnections[1].GPCAddress;
            IOServiceState.Executing100Hz               := FALSE;
            IOServiceState.Executing50Hz                := FALSE;
            IOServiceState.Executing25Hz                := FALSE;

            (* Create chain to monitor network 2 for faults.  *)
            ReadNodeInterConnections(2, Network2Connections);
            FaultMonitor2 := MakeMonitorRequest(Network2Connections);

            WITH FaultMonitor2^ DO

                Identifier                 := 302;
                Priority                   := 1;
                ResponseExpected           := TRUE;
                RequestType                := MonitorForFault;
                Chain^.NetworkToBeExecutedOn := 2;
                Chain^.ChainIdentifier     := 302;

            END;

            (* Notify controller that the networks are available to use.  *)
            NOW outport[ControlsPort]^.ServiceAvailable <- TRUE;
            EXIT;

    END;

END;

LOOP

    WAITUNTIL EVENT

        IOServiceRequest:

            ServiceRequest := ActivePort^.IOServiceRequest;
            IF ServiceRequest^.OnDemand THEN

                OnDemandRequestDelay := Random(1, 0.000020, 0.000035);

            ELSE

                OnDemandRequestDelay := 0.0;

            END;

            IF (ServiceRequest^.RequestType = ApplicationRequest) AND
                (ServiceRequest^.Priority = 10) AND
                NOT IOServiceState.Executing100Hz THEN

                SAMPLE 1.0 WITH SystemProbe[ProbeNumber];

                IOServiceState.Executing100Hz := TRUE;
                RequestTime := ComputeIOLoadTime(ServiceRequest,IOServiceState)
                            + OnDemandRequestDelay + RequestProcessing;

                NEW(ProcessRequest);
                WITH ProcessRequest^ DO

                    Function  := IORequestSchedule;
                    IORequest := ServiceRequest;

                END;

                SubmitRequestProcessing(ProcessRequest,
                    NOT ServiceRequest^.OnDemand, ServiceRequest^.Frame,
            RequestTime, ServiceRequest^.Priority);

    ELSIF (ServiceRequest^.RequestType = ApplicationRequest) AND
        (ServiceRequest^.Priority = 9) AND
        NOT IOServiceState.Executing50Hz THEN

        SAMPLE 1.0 WITH SystemProbe[ProbeNumber];

        IOServiceState.Executing50Hz := TRUE;
        RequestTime  := ComputeIOLoadTime(ServiceRequest,IOServiceState)
                    + OnDemandRequestDelay + RequestProcessing;

        NEW(ProcessRequest);
        WITH ProcessRequest^ DO

            Function   := IORequestSchedule;
            IORequest  := ServiceRequest;

        END;

        SubmitRequestProcessing(ProcessRequest,
            NOT ServiceRequest^.OnDemand, ServiceRequest^.Frame,
            RequestTime, ServiceRequest^.Priority);

    ELSIF (ServiceRequest^.RequestType = ApplicationRequest) AND
        (ServiceRequest^.Priority = 8) AND
        NOT IOServiceState.Executing25Hz THEN

        SAMPLE 1.0 WITH SystemProbe[ProbeNumber];

        IOServiceState.Executing25Hz := TRUE;
        RequestTime  := ComputeIOLoadTime(ServiceRequest,IOServiceState)
                    + OnDemandRequestDelay + RequestProcessing;

        NEW(ProcessRequest);
        WITH ProcessRequest^ DO

            Function   := IORequestSchedule;
            IORequest  := ServiceRequest;

        END;

        SubmitRequestProcessing(ProcessRequest,
            NOT ServiceRequest^.OnDemand, ServiceRequest^.Frame,
            RequestTime, ServiceRequest^.Priority);

    ELSIF ServiceRequest^.RequestType = NetworkManagerRequest THEN

        RequestTime  := ComputeIOLoadTime(ServiceRequest,IOServiceState);

        NEW(ProcessRequest);
        WITH ProcessRequest^ DO

            Function   := IORequestSchedule;
            IORequest  := ServiceRequest;

        END;

        SubmitRequestProcessing(ProcessRequest, FALSE, 0, RequestTime,
            ServiceRequest^.Priority);

    END;

| IOCompletionPoll:

    RequestToService := IOCompletionPoll;
    IF RequestToService^.Request^.ResponseExpected THEN

        UnloadResponse(RequestToService, IOServiceState, ResponseData);
        TimeToUnloadRequest := ComputeIOUnloadTime(ResponseData.Response);

        IF RequestToService^.RequestType = ApplicationRequest THEN
```

```
                        ResponseCompletion(ResponseData.Response);
                        CheckNetworksForFault(ResponseData.Response,
                            IOServiceState);

                        (* Release semaphore and check for a higher priority
                           request that is waiting for start IOS.  *)
                        Release(PendingStatus);
                        IF PendingStatus > RequestToService^.Request^.Priority THEN

                            NEW(PendingIOProcessing);
                            PendingIOProcessing^.Function := StartPendingIO;
(* 1/27/88 *)               NextRequest(PendingRequest);
(* 1/27/88 *)               PendingIOProcessing^.RequestStatus := PendingRequest^.ExecutionStatus;
(* 1/27/88 *)               DISPOSE(PendingRequest);
                            NEW(ProcessingStartPendingIO);
                            WITH ProcessingStartPendingIO^ DO

                                Priority           := PendingStatus;
                                ProcessingRequired := 0.0;
                                WriteData          := FALSE;
                                ProcessID          := 'Start Pending I/O';
                                Data               := PendingIOProcessing;

                            END;

                            NOW outport[IOPPort]^.ProcessorRequest <- ProcessingStartPendingIO;

                        END;

                        NEW(ResponseDataProcessing);
                        WITH ResponseDataProcessing^ DO

                            Function   := IOResponseSchedule;
                            IOResponse := ResponseData.Response;

                        END;

                        SubmitUnloadIORequest(ResponseDataProcessing,
                            TimeToUnloadRequest, RequestToService^.Request^.Priority);

                    ELSIF RequestToService^.RequestType = NetworkManagerRequest THEN

                        NEW(ResponseDataProcessing);
                        WITH ResponseDataProcessing^ DO

                            Function   := IOResponseSchedule;
                            IOResponse := ResponseData.Response;

                        END;

                        SubmitUnloadIORequest(ResponseDataProcessing,
                            TimeToUnloadRequest, RequestToService^.Request^.Priority);

                    ELSIF RequestToService^.RequestType = MonitorForFault THEN

                        IF ResponseData.Response^.Response^.AllFailed THEN

                            NEW(ServiceNetworkRequest);
                            WITH ServiceNetworkRequest^ DO

                                ServiceRequest := SwitchRootLink;

                                (* This section of code uses internal workings
                                   of DENET to determine the FailedRootNode.
                                   IT IS CONFIGURATION DEPENDANT.  *)
                                CurrentNodeID  := IOServiceState.ActiveRootLink[2];
                                FailedRootNode := GetOutNode(2);
                                CurrentNodeID  := MyNodeID;

                                FailedRootNumber := FailRootLink(IONetworkConnections[NetworkID2],
                                                IOServiceState.ActiveRootLink[NetworkID2]);
```

```
        IOServiceState.ActiveRootLink[NetworkID2]
            := SwitchRootLinks(IONetworkConnections[NetworkID2],
                NetworkID2, FailedRootNumber);

        IF IOServiceState.ActiveRootLink[NetworkID2] = 0 THEN

            (* A good root link cannot be found.  This
               network will not be returned to service.  *)

            WriteString(ParamOut, "Network 2 has no more root links.");
            WriteLn(ParamOut);
            WriteString(ParamOut, "Continue operation with Network 2 out of service. ");
            WriteLn(ParamOut);

        END;

        (* This section of code uses internal workings
           of DENET to determine the FailedRootNode.
           IT IS CONFIGURATION DEPENDANT.  *)
        CurrentNodeID := IOServiceState.ActiveRootLink[2];
        NewRootNode   := GetOutNode(2);
        CurrentNodeID := MyNodeID;

    END;

    (* Make a request to the IOP for execution time
       to switch root links.  *)
    NEW(RootLinkSwitchProcessing);
    RootLinkSwitchProcessing^.Function    := NetworkRepair;
    RootLinkSwitchProcessing^.RepairData  := ServiceNetworkRequest;

    NEW(ProcessingNetworkRepair);
    WITH ProcessingNetworkRepair^ DO

        Priority           := IOSystemServicePriority;
        ProcessingRequired := TimeToUnloadRequest + SwitchRootLinkProcessing;
        WriteData          := FALSE;
        ProcessID          := 'Switch Root Link Processing';
        Data               := RootLinkSwitchProcessing;

    END;

    ReleaseChainResponseMemory(ResponseData.Response^.Response);
    DISPOSE(ResponseData.Response); (* PRB *)

ELSE

    NEW(ServiceNetworkRequest);
    WITH ServiceNetworkRequest^ DO

        ServiceRequest        := RepairFault;
        MonitorChainResponse  := ResponseData.Response;

    END;

    NEW(RepairNetworkProcessing);
    RepairNetworkProcessing^.Function    := NetworkRepair;
    RepairNetworkProcessing^.RepairData  := ServiceNetworkRequest;

    NEW(ProcessingNetworkRepair);
    WITH ProcessingNetworkRepair^ DO

        Priority           := IOSystemServicePriority;
        ProcessingRequired := TimeToUnloadRequest;
        WriteData          := FALSE;
        ProcessID          := 'Repair Network Processing';
        Data               := RepairNetworkProcessing;

    END;

END;
```

```
                        NOW outport[IOPPort]^.ProcessorRequest <- ProcessingNetworkRepair;

                    END;

                ELSE

                    (* The request for this poll just contains output
                       transactions with no input transactions, check
                       for any pending request that is higher priority.  *)
(* 1/27/88 *)       (* Deleted Lines *)
                    (* Release semaphore and check for a higher priority
                       request that is waiting for start IOS.  *)
                    Release(PendingStatus);
                    IF PendingStatus > RequestToService^.Request^.Priority THEN

                        NEW(PendingIOProcessing);
                        PendingIOProcessing^.Function := StartPendingIO;
(* 1/27/88 *)           NextRequest(PendingRequest);
(* 1/27/88 *)           PendingIOProcessing^.RequestStatus := PendingRequest^.ExecutionStatus;
(* 1/27/88 *)           DISPOSE(PendingRequest);

                        NEW(ProcessingStartPendingIO);
                        WITH ProcessingStartPendingIO^ DO

                            Priority           := PendingStatus;
                            ProcessingRequired := 0.0;
                            WriteData          := FALSE;
                            ProcessID          := 'Start Pending I/O';
                            Data               := PendingIOProcessing;

                        END;

                        NOW outport[IOPPort]^.ProcessorRequest <- ProcessingStartPendingIO;

                    END;

                    NEW(ResponseDataProcessing);
                    WITH ResponseDataProcessing^ DO

                        Function := OutputResponseSchedule;
                        Rate     := RequestToService^.Request^.Identifier;

                    END;

                    NEW(ProcessingOutputResponse);
                    WITH ProcessingOutputResponse^ DO

                        Priority            := RequestToService^.Request^.Priority;

                        IF (IOServiceState.NetworkHealth[1] = InService) AND
                            (IOServiceState.NetworkHealth[2] = InService) THEN

                            ProcessingRequired := ResponseCompletionProcessing
                                                + (2.0 * EndOfChainProcessingOverhead);

                        ELSE

                            ProcessingRequired := ResponseCompletionProcessing
                                                + EndOfChainProcessingOverhead;

                        END;

                        WriteData           := FALSE;
                        ProcessID           := 'Output Repsonse Processing';
                        Data                := ResponseDataProcessing;

                    END;

                    NOW outport[IOPPort]^.ProcessorRequest <- ProcessingOutputResponse;

                END;
```

```
      DISPOSE(RequestToService);  (* PRB *)

  | RtnNetworkToService:

      (* This event is used to return network 2 to service
         after the NetworkManager has repaired it.  *)

      NewNetworkState := ActivePort^.RtnNetworkToService;

      DataCollectionRecord.EventID                      := 29;
      DataCollectionRecord.SimulationTime               := clock;
      DataCollectionRecord.Frequency                    := NonCyclic;
      DataCollectionRecord.NonCyclicData.N_Variation    := NetServiceChange;
      DataCollectionRecord.NonCyclicData.NetworkID      := NewNetworkState^.NetworkID;

      WriteDataElementType(DataCollectionRecord);

      WITH NewNetworkState^ DO

          FaultMonitor2^.Chain := MonitorChain;
          FaultMonitor2^.RequestTimeoutValue := ComputeChainTimeout(0,
                        QSize(FaultMonitor2^.Chain^.TransactionQueue));

          TimeNetworkOutOfService := clock - NetworkOutOfServiceTime[NetworkID];
          WriteLn(ParamOut);
          WriteString(ParamOut, " Network 2 was out of service for ");
          WriteReal(ParamOut, TimeNetworkOutOfService, 0);
          WriteString(ParamOut, " seconds.");
          WriteLn(ParamOut);
          WriteLn(ParamOut);

          IOServiceState.NetworkHealth[NetworkID]   := InService;
          NOW outport[ControlsPort]^.ServiceAvailable <- TRUE;

      END;

      DISPOSE(NewNetworkState);   (* PRB *)

  | ProcessorResponse:

      ProcessResponse := ActivePort^.ProcessorResponse;
      NetworkData     := ProcessResponse^.Data;

      WITH NetworkData^ DO

          CASE Function OF

              IORequestSchedule:

                  IF IORequest^.RequestType = ApplicationRequest THEN

                      ExecuteApplicationRequest(IORequest, IOServiceState);

                  ELSE

                      ExecuteManager2Request(IORequest, IOServiceState);

                  END;

            | IOResponseSchedule:

                  IF IOResponse^.ResponseType = ApplicationResponse THEN

                      REPORT "%d" IOResponse^.Identifier TAGGED "Finish IOP Activity";

                      CASE IOResponse^.Identifier OF

                          100: NOW outport[ApplicationPort[1]]^.
                                   ApplicationResponse <- IOResponse;
                               IOServiceState.Executing100Hz := FALSE;
```

```
                                     | 50: NOW outport[ApplicationPort[2]]^.
                                             ApplicationResponse <- IOResponse;
                                          IOServiceState.Executing50Hz := FALSE;

                                     | 25: NOW outport[ApplicationPort[3]]^.
                                             ApplicationResponse <- IOResponse;
                                          IOServiceState.Executing25Hz := FALSE;

                                 END;

                                 IF (NOT IOServiceState.Executing100Hz) AND
                                     (NOT IOServiceState.Executing50Hz) AND
                                     (NOT IOServiceState.Executing25Hz) THEN

                                        SAMPLE 0.0 WITH SystemProbe[ProbeNumber];

                                END;

                                (* Check for any pending requests.  *)
                                SemaphoreIdle(IOSystemStatus);
                                IF IOSystemStatus THEN

                                    Release(PendingStatus);
(* 1/27/88 *)                       IF (PendingStatus = 10) OR
(* 1/27/88 *)                       ((PendingStatus = 9) AND
(* 1/27/88 *)                       (NOT IOServiceState.Executing100Hz)) OR
(* 1/27/88 *)                       ((PendingStatus = 8) AND
(* 1/27/88 *)                       (NOT IOServiceState.Executing100Hz) AND
(* 1/27/88 *)                       (NOT IOServiceState.Executing50Hz)) THEN

                                        NEW(PendingIOProcessing);
                                        PendingIOProcessing^.Function := StartPendingIO;
(* 1/27/88 *)                           NextRequest(PendingRequest);
(* 1/27/88 *)                           PendingIOProcessing^.RequestStatus := PendingRequest^.ExecutionStatus;
(* 1/27/88 *)                           DISPOSE(PendingRequest);

                                        NEW(ProcessingStartPendingIO);
                                        WITH ProcessingStartPendingIO^ DO

                                            Priority           := PendingStatus;
                                            ProcessingRequired := 0.0;
                                            WriteData          := FALSE;
                                            ProcessID          := 'Start Pending I/O';
                                            Data               := PendingIOProcessing;

                                       END;

                                       NOW outport[IOPPort]^.ProcessorRequest <- ProcessingStartPendingIO;

                                   END;

                               END;

                           ELSE

                               NOW outport[NetworkManager2Port]^.
                                   IOManager2Response <- IOResponse^.Response;

                               IOResponse^.Response := NIL; (* MJS *)
                               DISPOSE(IOResponse);    (* PRB *)

                          END;

                      | OutputResponseSchedule:

                          REPORT "%d" Rate TAGGED "Finish IOP Activity";

                          CASE Rate OF

                              100:
                                   IOServiceState.Executing100Hz := FALSE;
```

```
                                  | 50:

                                       IOServiceState.Executing50Hz := FALSE;

                                  | 25:

                                       IOServiceState.Executing25Hz := FALSE;

                              END;

                              IF (NOT IOServiceState.Executing100Hz) AND
                                  (NOT IOServiceState.Executing50Hz) AND
                                  (NOT IOServiceState.Executing25Hz) THEN

                                  SAMPLE 0.0 WITH SystemProbe[ProbeNumber];

                              END;

                              (* Check for any pending requests.  *)
                              SemaphoreIdle(IOSystemStatus);
                              IF IOSystemStatus THEN

                                  Release(PendingStatus);
(* 1/27/88 *)                     IF (PendingStatus = 10) OR
(* 1/27/88 *)                          ((PendingStatus = 9) AND
(* 1/27/88 *)                          (NOT IOServiceState.Executing100Hz)) OR
(* 1/27/88 *)                          ((PendingStatus = 8) AND
(* 1/27/88 *)                          (NOT IOServiceState.Executing100Hz) AND
(* 1/27/88 *)                          (NOT IOServiceState.Executing50Hz)) THEN

                                       NEW(PendingIOProcessing);
                                       PendingIOProcessing^.Function := StartPendingIO;
(* 1/27/88 *)                          NextRequest(PendingRequest);
(* 1/27/88 *)                          PendingIOProcessing^.RequestStatus := PendingRequest^.ExecutionStatus;
(* 1/27/88 *)                          DISPOSE(PendingRequest);

                                       NEW(ProcessingStartPendingIO);
                                       WITH ProcessingStartPendingIO^ DO

                                           Priority           := PendingStatus;
                                           ProcessingRequired := 0.0;
                                           WriteData          := FALSE;
                                           ProcessID          := 'Start Pending I/O';
                                           Data               := PendingIOProcessing;

                                       END;

                                       NOW outport[IOPPort]^.ProcessorRequest <- ProcessingStartPendingIO;

                                   END;

                               END;

                            | StartPendingIO:

(* 1/27/88 *)                   StartIOSs(RequestStatus, IOServiceState);

                            | StartFaultMonitor:

                                ExecuteManager2Request(FaultMonitor2, IOServiceState);

                            | NetworkRepair:

                                IF NetworkData^.RepairData^.ServiceRequest =
                                    SwitchRootLink THEN

                                    WITH DataCollectionRecord DO

                                        EventID                   := 29;
                                        SimulationTime            := clock;
                                        Frequency                 := NonCyclic;
                                        NonCyclicData.N_Variation := NetServiceChange;
```

```
                        NonCyclicData.NetworkID   := 2;

                    END;

                    WriteDataElementType(DataCollectionRecord);

                    TimeNetworkOutOfService := clock - NetworkOutOfServiceTime[2];
                    WriteLn(ParamOut);
                    WriteString(ParamOut, " Network 2 was out of service for ");
                    WriteReal(ParamOut, TimeNetworkOutOfService, 0);
                    WriteString(ParamOut, " seconds.");
                    WriteLn(ParamOut);
                    WriteLn(ParamOut);

                    IOServiceState.NetworkHealth[2]  := InService;
                    NOW outport[ControlsPort]^.ServiceAvailable <- TRUE;

                END;

                NOW outport[NetworkManager2Port]^.
                        ManagerServiceRqst <- NetworkData^.RepairData;

            ELSE

                WriteString(ParamOut, "Unexpected Response from the IOP in the I/O Service.");
                WriteLn(ParamOut);

        END;

    END;

    DISPOSE(NetworkData);
    DISPOSE(ProcessResponse);

| Reset:

    SetSemaphoreIdle;

    (* Reinitialize all network connections to idle.  *)
    FOR NetworkCounter := 1 TO NumberOfNetworks DO

        FOR InterfaceCounter := 1 TO NumberOfIOSPerChannel DO

            IF IONetworkConnections[NetworkCounter].
                NetworkConnections[InterfaceCounter].GPCAddress
                <> 0 THEN

                IONetworkConnections[NetworkCounter].
                    ConnectionStatus[InterfaceCounter] := Idle;

            ELSE

                (* This simulation run does not have an interface
                   with this ID, set its status to failed.  *)

                IONetworkConnections[NetworkCounter].
                    ConnectionStatus[InterfaceCounter] := Failed;

            END;

        END;

    END;

    (* Reinitialize Active interfaces. Since all hardware should
       be "good", the first interface will be used.  *)

    IONetworkConnections[1].ConnectionStatus[1]  := Active;
    IONetworkConnections[2].ConnectionStatus[1]  := Active;
    IOServiceState.NetworkHealth[1]              := InService;
    IOServiceState.NetworkHealth[2]              := InService;
    IOServiceState.ActiveRootLink[1]             := IONetworkConnections[1].NetworkConnections[1].GPCAddress;
```

```
                IOServiceState.ActiveRootLink[2]          := IONetworkConnections[2].NetworkConnections[1].GPCAddress;
                IOServiceState.Executing100Hz             := FALSE;
                IOServiceState.Executing50Hz              := FALSE;
                IOServiceState.Executing25Hz              := FALSE;

            | ProbeReset:

                (*
                 *   check if the io system is busy by looking at the last sample taken with the probe
                 *)
                IF SystemProbe[ProbeNumber]^.ProbeValue = 1.0 THEN

                   ClearProbe(SystemProbe[ProbeNumber]);
                   SAMPLE 1.0 WITH SystemProbe[ProbeNumber];

                ELSE

                   ClearProbe(SystemProbe[ProbeNumber]);
                   SAMPLE 0.0 WITH SystemProbe[ProbeNumber];

                END;

        END;

    END;

END IOService.
```

UTILITIES

```
DEFINITION MODULE Utilities;

    FROM IOS IMPORT ChainType, ChainStatusData;

    FROM BusMessag IMPORT PortNameType, PortEnableRegisterType;

    FROM TypeConst IMPORT ChannelIDType, StatusType, NodeArrayType,
                NodeStatusArray, PortStatusArray, ChannelStatusRecord;

    EXPORT QUALIFIED ClearPortStatusArray, UpdateLinkStatus, SetNodeStatusFailed,
                ConvertPortStatusToEnable, InitializeStatusVariables,
                NodesInThisSimulation,  ComputeChainTimeout,
                ManagerChainUnloadTime;


    (**************************************************************************)
    (* This procedure computes the time element to perform the unloading
       of a network manager chain from the DPM in the IOS to the IOP.  *)

    PROCEDURE ManagerChainUnloadTime(Response : ChainStatusData) :REAL;

    (**************************************************************************)
    (* This procedure will return an array of type PortStatusArray
       with all the elements set to IdlePort.  *)
    PROCEDURE ClearPortStatusArray(VAR PortStatus : PortStatusArray);

    (**************************************************************************)

    PROCEDURE UpdateLinkStatus(VAR StatusArray : NodeStatusArray;
                                   SpawningNode    : INTEGER;
                                   SpawningPort    : PortNameType;
                                   TargetNode      : INTEGER;
                                   TargetPort      : PortNameType;
                                   LinkStatus      : StatusType);

    (**************************************************************************)

    PROCEDURE SetNodeStatusFailed(VAR StatusArray    : NodeStatusArray;
                                      NetworkConnections : NodeArrayType;
                                      FailedNode         : INTEGER);

    (**************************************************************************)
    (* This procedure takes an array of type PortStatusArray and
       converts it to a PortEnableRegister.  If an element in the
       PortStatusArray is Active then the corresponding element in
       the PortEnableRegister is set to Enabled, otherwise the
       element is set to Disabled.  *)
    PROCEDURE ConvertPortStatusToEnable(VAR PortStatus     : PortStatusArray;
                                        VAR EnableRegister : PortEnableRegisterType);

    (**************************************************************************)
    (* This procedure will initialize the variables that the network manager
       needs to maintain the network.  *)

    PROCEDURE InitializeStatusVariables(NodeConnections   : NodeArrayType;
                                        VAR NodeStatus    : NodeStatusArray;
                                        VAR ChannelStatus : ChannelStatusRecord);


    (**************************************************************************)
    (* This procedure will read the node connections array and determine
       how many nodes are in the current simulation. This number will be
       0 < NodesInSimulation <= NumberOfNodes.  *)

    PROCEDURE NodesInThisSimulation(NodeConnections : NodeArrayType)
                                     : INTEGER;

    (**************************************************************************)
    (* This procedure computes the time to execute a chain based on the
       number of normally completing transactions and transactions actions
       that time out.  A time is also included that represents the turn
       around time between transactions at the IOS.  *)
    PROCEDURE ComputeChainTimeout(NormalCompletions  : INTEGER;
                                  TimeoutCompletions : INTEGER) :REAL;

    (****************************************************************************)
END Utilities.
```

UTILITIES

```
IMPLEMENTATION MODULE Utilities;

   FROM BusMessag IMPORT BusMessageType, PortNameType,
             PortEnableRegisterType, PortStateType, NumberOfPortsPerNode,
             NumberOfNodes, MakeNodeConfigurationCommand;

   FROM TypeConst IMPORT ChannelIDType, StatusType, PortConfigurationType,
             NodeArrayType, NodeStatusArray, PortStatusArray,
             ChannelStatusRecord, NetworkElementType;

   FROM CentralDB IMPORT FindNodeNumber;

   FROM IOS IMPORT ChainType, ChainStatusData, TransactionType,
             InputFrameType, TimeOutIndicatorType;

   FROM InOut IMPORT WriteLn, WriteString, WriteReal, WriteInt;

   FROM Storage IMPORT ALLOCATE;

   FROM QueueM IMPORT InitQ, QSize, QInsert, QSucc, FirstQ;

   FROM MathLib0 IMPORT mod;

   CONST TransactionTimeOut       = 0.000500;        (* sec *)
         FixedChainUnloadTime     = 0.000050;        (* sec *)
         DataExchangeUnloadTime   = 0.000004;        (* sec/byte *)
         OneMilliSecond           = 0.001;
         ManagerDXResponseLength  = 13;              (* bytes *)
   (************************************************************************)

   PROCEDURE ManagerChainUnloadTime(Response : ChainStatusData) :REAL;

      VAR UnloadTime             :-REAL;
          TransactionUnloadTime  : REAL;
          NumberOfInputFrames    : INTEGER;

   BEGIN

      NumberOfInputFrames := QSize(Response^.InputFrameQueue);
      TransactionUnloadTime := FLOAT(NumberOfInputFrames *
                                     ManagerDXResponseLength) *
                                     DataExchangeUnloadTime;

      UnloadTime := FixedChainUnloadTime + TransactionUnloadTime;

      RETURN(UnloadTime);

   END ManagerChainUnloadTime;

   (************************************************************************)
   (* This procedure will return an array of type PortStatusArray
      with all the elements set to IdlePort.  *)
   PROCEDURE ClearPortStatusArray(VAR PortStatus : PortStatusArray);

      VAR PortIndex : PortNameType;

   BEGIN

      FOR PortIndex := 1 TO NumberOfPortsPerNode DO

          PortStatus[PortIndex].Status    := Idle;
          PortStatus[PortIndex].Direction := Inboard;

      END;

   END ClearPortStatusArray;

   (************************************************************************)

   PROCEDURE UpdateLinkStatus(VAR StatusArray : NodeStatusArray;
                              SpawningNode    : INTEGER;
```

```
                           SpawningPort     : PortNameType;
                           TargetNode       : INTEGER;
                           TargetPort       : PortNameType;
                           LinkStatus       : StatusType);

    VAR SpawningNodeNumber : INTEGER;
        TargetNodeNumber   : INTEGER;

BEGIN

    SpawningNodeNumber := FindNodeNumber(SpawningNode);
    TargetNodeNumber   := FindNodeNumber(TargetNode);

    WITH StatusArray[SpawningNodeNumber].PortStatus[SpawningPort] DO

        IF LinkStatus = Active THEN

            Status    := Active;
            Direction := Outboard;

        ELSE

            Status  := LinkStatus;

        END;

    END;

    WITH StatusArray[TargetNodeNumber].PortStatus[TargetPort] DO

        IF LinkStatus = Active THEN

            Status    := Active;
            Direction := Inboard;

        ELSE

            Status  := LinkStatus;

        END;

    END;

 END UpdateLinkStatus;

 (**************************************************************************)
 PROCEDURE SetNodeStatusFailed(VAR StatusArray      : NodeStatusArray;
                                NetworkConnections : NodeArrayType;
                                FailedNode         : INTEGER);

     VAR NodeNumber          : INTEGER;
         AdjacentNodeNumber  : INTEGER;
         PortIndex           : PortNameType;
         AdjacentPort        : PortNameType;

  BEGIN

     NodeNumber := FindNodeNumber(FailedNode);

     (* Set status of node to failed.  *)
     StatusArray[NodeNumber].Status  := Failed;

     (* Set status of ports on this node to failed and the
        ports on any adjacent node to failed.  *)
     FOR PortIndex := 1 TO NumberOfPortsPerNode DO

         StatusArray[NodeNumber].PortStatus[PortIndex].Status  := Failed;

         WITH NetworkConnections[NodeNumber].PortArray[PortIndex] DO

             CASE AdjacentElement OF
```

```
                GPC, DIU, None:

                | Node:

                    AdjacentNodeNumber := FindNodeNumber(NodeAddress);
                    AdjacentPort       := Port;
                    StatusArray[AdjacentNodeNumber].PortStatus
                            [AdjacentPort].Status := Failed;

            END;

        END;

    END;

END SetNodeStatusFailed;

(**********************************************************************)
(* This procedure takes an array of type PortStatusArray and
   converts it to a PortEnableRegister.  If an element in the
   PortStatusArray is Active then the corresponding element in
   the PortEnableRegister is set to Enabled, otherwise the
   element is set to Disabled.  *)
PROCEDURE ConvertPortStatusToEnable(VAR PortStatus     : PortStatusArray;
                                    VAR EnableRegister : PortEnableRegisterType);

    VAR PortIndex : PortNameType;

BEGIN

    FOR PortIndex := 1 TO NumberOfPortsPerNode DO

        IF PortStatus[PortIndex].Status = Active THEN

            EnableRegister[PortIndex] := Enabled;

        ELSE

            EnableRegister[PortIndex] := Disabled;

        END;

    END;

END ConvertPortStatusToEnable;

(**********************************************************************)
(* This procedure will initialize the variables that the network manager
   needs to maintain the network.  *)

P).OCEDURE InitializeStatusVariables(NodeConnections   : NodeArrayType;
                                    VAR NodeStatus    : NodeStatusArray;
                                    VAR ChannelStatus : ChannelStatusRecord);

    VAR NodeIndex    : INTEGER;
        ChannelIndex : ChannelIDType;
        PortIndex    : PortNameType;

BEGIN

    FOR NodeIndex := 1 TO NumberOfNodes DO

        WITH NodeStatus[NodeIndex] DO

            Address := NodeConnections[NodeIndex].NodeAddress;
            Status  := Idle;
            ClearPortStatusArray(PortStatus);

        END;

    END;
    ChannelStatus.GPCAddress  := 100;
    ChannelStatus.ChannelID   := A;
    ChannelStatus.Status      := Idle;

END InitializeStatusVariables;

(*********************************************************************)
(* This procedure computes how many nodes are in the current simulation
   based on the NodeAddress for each node being different than the
   NodeNumber.  The Central Database initializes its network description
   so that each node number is the same as the node address.  If
   this strategy is not continued in the future, this routine may
   need to be changed.  *)
PROCEDURE NodesInThisSimulation(NodeConnections : NodeArrayType)
                                : INTEGER;

    VAR NodeCount : INTEGER;
        NodeIndex : INTEGER;

BEGIN

    NodeCount := 0;

    FOR NodeIndex := 1 TO NumberOfNodes DO

        IF NodeIndex <> INTEGER(NodeConnections[NodeIndex].NodeAddress) THEN

            NodeCount := NodeCount + 1;

        END;

    END;

    RETURN(NodeCount);

END NodesInThisSimulation;

(*************************************************************************)
(* Since the network manager is the only one to use this routine,
   the returned time will be based on all transactions timing out
   and then adding one millisecond to account for transaction turn
   around and transaction transmission time.  *)
PROCEDURE ComputeChainTimeout(NormalCompletions  : INTEGER;
                              TimeoutCompletions : INTEGER) :REAL;

    VAR ExecutionTime       : REAL;

BEGIN

    IF ((NormalCompletions + TimeoutCompletions) MOD 2) = 0 THEN

        RETURN (FLOAT(NormalCompletions + TimeoutCompletions)
            * TransactionTimeOut) + OneMilliSecond;

    ELSE

        RETURN (FLOAT(TRUNC(((FLOAT(NormalCompletions + TimeoutCompletions)
            * TransactionTimeOut) + OneMilliSecond) / OneMilliSecond))
            * OneMilliSecond);

    END;

END ComputeChainTimeout;

(*************************************************************************)
END Utilities.
```

GROWNET

```
DEFINITION MODULE GrowNet;

    FROM IOService IMPORT IORequestType;

    FROM TypeConst IMPORT PortStatusArray, NodeStatusArray, NodeArrayType;

    FROM BusMessag IMPORT PortNameType;

    EXPORT QUALIFIED GROWTOROOTNODE, TransactionTimeOut, NetworkManagerPriority,
                EnableLink, DeleteNodeFromNetwork, AddDIUToNetwork,
                AddGPCToNetwork, DisabledTransmitTest, DisabledRetransmitTest,
                ResetConfigurationCommand, MakeMonitorRequest;

    CONST TransactionTimeOut      = 0.000500;
          NetworkManagerPriority = 1;

    (***************************************************************************)

    PROCEDURE GROWTOROOTNODE(RootNodeAddress      : INTEGER;
                             RootNodeInboardPort : PortNameType;
                             NodeStatus          : NodeStatusArray;
                             VAR IORequest       : IORequestType);

    (***************************************************************************)

    PROCEDURE EnableLink(SpawningNode         : INTEGER;
                         TargetNode           : INTEGER;
                         NodeStatus           : NodeStatusArray;
                         VAR AddNodeIORequest : IORequestType);

    (***************************************************************************)

    PROCEDURE DeleteNodeFromNetwork (SpawningNode             : INTEGER;
                                     SpawningNodeOutboardPort : PortNameType;
                                     TargetNode               : INTEGER;
                                     TargetNodeInboardPort    : PortNameType;
                                     NodeStatus               : NodeStatusArray;
                                     VAR DeleteNodeIORequest  : IORequestType);

    (***************************************************************************)

    PROCEDURE AddDIUToNetwork(Node          : INTEGER;
                              Port          : PortNameType;
                              NodeStatus    : NodeStatusArray;
                              VAR IORequest : IORequestType);

    (***************************************************************************)

    PROCEDURE AddGPCToNetwork(Node          : INTEGER;
                              Port          : PortNameType;
                              NodeStatus    : NodeStatusArray;
                              VAR IORequest : IORequestType);

    (***************************************************************************)

    PROCEDURE DisabledTransmitTest(TargetNode    : INTEGER;
                                   NodeStatus    : NodeStatusArray;
                                   VAR IORequest : IORequestType);

    (***************************************************************************)

    PROCEDURE DisabledRetransmitTest(TestNode      : INTEGER;
                                     TargetNode    : INTEGER;
                                     NodeStatus    : NodeStatusArray;
                                     VAR IORequest : IORequestType);

    (***************************************************************************)

    PROCEDURE ResetConfigurationCommand(TestNode      : INTEGER;
                                        NodeStatus    : NodeStatusArray;
                                        VAR IORequest : IORequestType);
    (**************************************************************************)

    PROCEDURE MakeMonitorRequest(NodeConnection : NodeArrayType) : IORequestType;

    (**************************************************************************)
END GrowNet.
```

GROWNET

```
IMPLEMENTATION MODULE GrowNet;

    FROM IOService IMPORT IORequestType, RequestActivityType;

    FROM IOS IMPORT ChainType, TransactionType;

    FROM CentralDB IMPORT FindNodeNumber;

    FROM TypeConst IMPORT NodeStatusArray, NodeArrayType;

    FROM BusMessag IMPORT PortNameType, PortEnableRegisterType,
                BusMessageType, DIUCommandType, NumberOfNodes,
                MakeNodeConfigurationCommand, MakeMonitorCommand;

    FROM Utilities IMPORT ConvertPortStatusToEnable, ComputeChainTimeout;

    FROM QueueM IMPORT InitQ, QInsert, QSize, dNEW;

    FROM Storage IMPORT ALLOCATE;

    FROM SYSTEM IMPORT SIZE;

    FROM InOut IMPORT WriteString, WriteInt, WriteReal, WriteLn;

    (********************************************************************)

    PROCEDURE GROWTOROOTNODE(RootNodeAddress      : INTEGER;
                             RootNodeInboardPort : PortNameType;
                             NodeStatus           : NodeStatusArray;
                             VAR IORequest        : IORequestType);

        CONST GrowToRootNodeIdentifier = 305;

        VAR Transaction          : TransactionType;
            Command              : BusMessageType;
            PortEnableRegister   : PortEnableRegisterType;
            RootNumber           : INTEGER;

    BEGIN

        dNEW(IORequest,SIZE(IORequest^));
        IORequest^.refcnt := 0;
        IORequest^.copycnt := 1;
        IORequest^.nextq := NIL;
        dNEW(IORequest^.Chain,SIZE(IORequest^.Chain^));
        IORequest^.Chain^.refcnt := 0;
        IORequest^.Chain^.copycnt := 1;
        IORequest^.Chain^.nextq := NIL;
        dNEW(Transaction,SIZE(Transaction^));
        Transaction^.refcnt := 0;
        Transaction^.copycnt := 1;
        Transaction^.nextq := NIL;

        IORequest^.Chain^.TransactionQueue := InitQ("TransactionQueue", FALSE, 0);

        RootNumber := FindNodeNumber(RootNodeAddress);

        (* Now convert Configuration into a Port Enable Register command. *)
        ConvertPortStatusToEnable(NodeStatus[RootNumber].PortStatus, PortEnableRegister);

        (* Generate the command to grow to the Root Node.  *)
        Command := MakeNodeConfigurationCommand(RootNodeAddress, PortEnableRegister);

        (* Generate the transaction.  *)
        WITH Transaction^ DO

            Identifier   := RootNodeAddress;
            TimeOutValue := TransactionTimeOut;
            OutputFrame  := Command;

        END;
        (* Enter the transaction on the Transaction Queue.  *)
        QInsert(Transaction, IORequest^.Chain^.TransactionQueue, FALSE);
(*
        INSERT Transaction LAST IN IORequest^.Chain^.TransactionQueue;
*)

        WITH IORequest^.Chain^ DO

            ChainIdentifier      := GrowToRootNodeIdentifier;
            NumberOfTransactions := QSize(IORequest^.Chain^.TransactionQueue);

        END;

        WITH IORequest^ DO

            Priority            := NetworkManagerPriority;
            OnDemand            := FALSE;
            RequestTimeoutValue := ComputeChainTimeout(0, Chain^.NumberOfTransactions);
            RequestType         := NetworkManagerRequest;

        END;

    END GROWTOROOTNODE;

    (****************************************************************************)

    PROCEDURE EnableLink(SpawningNode         : INTEGER;
                         TargetNode           : INTEGER;
                         NodeStatus           : NodeStatusArray;
                         VAR AddNodeIORequest : IORequestType);

        CONST AddNodeToNetworkChainIdentifier = 306;

        VAR Transaction        : TransactionType;
            Command            : BusMessageType;
            PortEnableRegister : PortEnableRegisterType;
            SpawningNumber     : INTEGER;
            TargetNumber       : INTEGER;

    BEGIN

        SpawningNumber := FindNodeNumber(SpawningNode);
        TargetNumber   := FindNodeNumber(TargetNode);

        dNEW(AddNodeIORequest,SIZE(AddNodeIORequest^));
        AddNodeIORequest^.refcnt := 0;
        AddNodeIORequest^.copycnt := 1;
        AddNodeIORequest^.nextq := NIL;
        dNEW(AddNodeIORequest^.Chain,SIZE(AddNodeIORequest^.Chain^));
        AddNodeIORequest^.Chain^.refcnt := 0;
        AddNodeIORequest^.Chain^.copycnt := 1;
        AddNodeIORequest^.Chain^.nextq := NIL;

        AddNodeIORequest^.Chain^.TransactionQueue := InitQ("TransactionQueue", FALSE, 0);

        (* Convert the spawning node's port status to a Port Enable Register. *)
        ConvertPortStatusToEnable(NodeStatus[SpawningNumber].PortStatus, PortEnableRegister);

        (* Generate command for Spawning node to turn on outboard port.  *)
        Command := MakeNodeConfigurationCommand(SpawningNode, PortEnableRegister);

        dNEW(Transaction,SIZE(Transaction^));
        Transaction^.refcnt := 0;
        Transaction^.copycnt := 1;
        Transaction^.nextq := NIL;
        WITH Transaction^ DO

            Identifier   := SpawningNode;
            TimeOutValue := TransactionTimeOut;
            OutputFrame  := Command;

        END;
        (* Enter the transaction on the Transaction Queue.  *)
        QInsert(Transaction, AddNodeIORequest^.Chain^.TransactionQueue, FALSE);
(*
        INSERT Transaction LAST IN AddNodeIORequest^.Chain^.TransactionQueue;
*)

        (* Generate a transaction for target node to turn on its inboard port.*)

        (* Convert the target node's port status to a Port Enable Register.  *)
        ConvertPortStatusToEnable(NodeStatus[TargetNumber].PortStatus, PortEnableRegister);

        (* Generate command for Spawning node to turn on outboard port.  *)
        Command := MakeNodeConfigurationCommand(TargetNode, PortEnableRegister);

        dNEW(Transaction,SIZE(Transaction^));
        Transaction^.refcnt := 0;
        Transaction^.copycnt := 1;
        Transaction^.nextq := NIL;
        WITH Transaction^ DO

            Identifier   := TargetNode;
            TimeOutValue := TransactionTimeOut;
            OutputFrame  := Command;

        END;

        (* Enter the transaction on the Transaction Queue.  *)
        QInsert(Transaction, AddNodeIORequest^.Chain^.TransactionQueue, FALSE);
(*
        INSERT Transaction LAST IN AddNodeIORequest^.Chain^.TransactionQueue;
*)

        WITH AddNodeIORequest^.Chain^ DO

            ChainIdentifier       := AddNodeToNetworkChainIdentifier;
            NumberOfTransactions  := QSize(AddNodeIORequest^.Chain^.TransactionQueue);

        END;

        WITH AddNodeIORequest^ DO

            Priority            := NetworkManagerPriority;
            OnDemand            := FALSE;
            RequestTimeoutValue := ComputeChainTimeout(0, Chain^.NumberOfTransactions);
            RequestType         := NetworkManagerRequest;

        END;

    END EnableLink;

    (***************************************************************************)

    PROCEDURE DeleteNodeFromNetwork(SpawningNode               : INTEGER;
                                    SpawningNodeOutboardPort   : PortNameType;
                                    TargetNode                 : INTEGER;
                                    TargetNodeInboardPort      : PortNameType;
                                    NodeStatus                 : NodeStatusArray;
                                    VAR DeleteNodeIORequest    : IORequestType);

        CONST AddNodeToNetworkChainIdentifier = 307;

        VAR Transaction         : TransactionType;
            Command             : BusMessageType;
            PortEnableRegister  : PortEnableRegisterType;
            SpawningNumber      : INTEGER;
            TargetNumber        : INTEGER;

    BEGIN

        dNEW(DeleteNodeIORequest,SIZE(DeleteNodeIORequest^));
        DeleteNodeIORequest^.refcnt := 0;
```

```
         DeleteNodeIORequest^.copycnt := 1;
         DeleteNodeIORequest^.nextq := NIL;
         dNEW(DeleteNodeIORequest^.Chain,SIZE(DeleteNodeIORequest^.Chain^));
         DeleteNodeIORequest^.Chain^.refcnt := 0;
         DeleteNodeIORequest^.Chain^.copycnt := 1;
         DeleteNodeIORequest^.Chain^.nextq := NIL;

         DeleteNodeIORequest^.Chain^.TransactionQueue := InitQ("TransactionQueue", FALSE, 0);

         (* Generate a transaction for target node to turn off its inboard port.*)
         TargetNumber := FindNodeNumber(TargetNode);

         (* Convert the target node's port status to a Port Enable Register.  *)
         ConvertPortStatusToEnable(NodeStatus[TargetNumber].PortStatus, PortEnableRegister);

         (* Generate command for Spawning node to turn on outboard port.  *)
         Command := MakeNodeConfigurationCommand(TargetNode, PortEnableRegister);

         dNEW(Transaction,SIZE(Transaction^));
         Transaction^.refcnt := 0;
         Transaction^.copycnt := 1;
         Transaction^.nextq := NIL;
         WITH Transaction^ DO

             Identifier   := TargetNode;
             TimeOutValue := TransactionTimeOut;
             OutputFrame  := Command;

         END;

         (* Enter the transaction on the Transaction Queue.  *)
         QInsert(Transaction, DeleteNodeIORequest^.Chain^.TransactionQueue, FALSE);
(*
         INSERT Transaction LAST IN DeleteNodeIORequest^.Chain^.TransactionQueue;
*)
         SpawningNumber := FindNodeNumber(SpawningNode);

         (* Convert the spawning node's port status to a Port Enable Register. *)
         ConvertPortStatusToEnable(NodeStatus[SpawningNumber].PortStatus, PortEnableRegister);

         (* Generate command for Spawning node to turn on outboard port.  *)
         Command := MakeNodeConfigurationCommand(SpawningNode, PortEnableRegister);

         dNEW(Transaction,SIZE(Transaction^));
         Transaction^.refcnt := 0;
         Transaction^.copycnt := 1;
         Transaction^.nextq := NIL;
         WITH Transaction^ DO

             Identifier   := SpawningNode;
             TimeOutValue := TransactionTimeOut;
             OutputFrame  := Command;

         END;

         (* Enter the transaction on the Transaction Queue.  *)
         QInsert(Transaction, DeleteNodeIORequest^.Chain^.TransactionQueue, FALSE);
(*
         INSERT Transaction LAST IN DeleteNodeIORequest^.Chain^.TransactionQueue;
*)

         WITH DeleteNodeIORequest^.Chain^ DO

             ChainIdentifier       := AddNodeToNetworkChainIdentifier;
             NumberOfTransactions  := QSize(DeleteNodeIORequest^.Chain^.TransactionQueue);

         END;

         WITH DeleteNodeIORequest^ DO

             Priority            := NetworkManagerPriority;
             OnDemand            := FALSE;
```

```
            RequestTimeoutValue := ComputeChainTimeout(0, Chain^.NumberOfTransactions);
            RequestType         := NetworkManagerRequest;

        END;

    END DeleteNodeFromNetwork;

    (*************************************************************************)

    PROCEDURE AddDIUToNetwork(Node          : INTEGER;
                              Port          : PortNameType;
                              NodeStatus    : NodeStatusArray;
                              VAR IORequest : IORequestType);


        CONST AddDIUIdentifier  = 308;

        VAR Transaction          : TransactionType;
            Command              : BusMessageType;
            PortEnableRegister   : PortEnableRegisterType;
            NodeNumber           : INTEGER;

    BEGIN

        dNEW(IORequest,SIZE(IORequest^));
        IORequest^.refcnt := 0;
        IORequest^.copycnt := 1;
        IORequest^.nextq := NIL;
        dNEW(IORequest^.Chain,SIZE(IORequest^.Chain^));
        IORequest^.Chain^.refcnt := 0;
        IORequest^.Chain^.copycnt := 1;
        IORequest^.Chain^.nextq := NIL;
        dNEW(Transaction,SIZE(Transaction^));
        Transaction^.refcnt := 0;
        Transaction^.copycnt := 1;
        Transaction^.nextq := NIL;

        IORequest^.Chain^.TransactionQueue := InitQ("TransactionQueue", FALSE, 0);

        NodeNumber := FindNodeNumber(Node);

        (* Now convert Configuration into a Port Enable Register command. *)
        ConvertPortStatusToEnable(NodeStatus[NodeNumber].PortStatus, PortEnableRegister);

        (* Generate the command to grow to the Root Node.  *)
        Command := MakeNodeConfigurationCommand(Node, PortEnableRegister);

        (* Generate the transaction.  *)
        WITH Transaction^ DO

            Identifier   := Node;
            TimeOutValue := TransactionTimeOut;
            OutputFrame  := Command;

        END;

        (* Enter the transaction on the Transaction Queue.  *)
        QInsert(Transaction, IORequest^.Chain^.TransactionQueue, FALSE);
(*
        INSERT Transaction LAST IN IORequest^.Chain^.TransactionQueue;
*)

        WITH IORequest^.Chain^ DO

            ChainIdentifier      := AddDIUIdentifier;
            NumberOfTransactions := QSize(IORequest^.Chain^.TransactionQueue);

        END;

        WITH IORequest^ DO

            Priority            := NetworkManagerPriority;
```

```
          OnDemand            := FALSE;
          RequestTimeoutValue := ComputeChainTimeout(0, Chain^.NumberOfTransactions);
          RequestType         := NetworkManagerRequest;

      END;

  END AddDIUToNetwork;

  (***************************************************************************)

  PROCEDURE AddGPCToNetwork(Node           : INTEGER;
                            Port           : PortNameType;
                            NodeStatus     : NodeStatusArray;
                            VAR IORequest  : IORequestType);


      CONST AddGPCIdentifier  = 309;

      VAR Transaction          : TransactionType;
          Command              : BusMessageType;
          PortEnableRegister   : PortEnableRegisterType;
          NodeNumber           : INTEGER;

  BEGIN

      dNEW(IORequest,SIZE(IORequest^));
      IORequest^.refcnt := 0;
      IORequest^.copycnt := 1;
      IORequest^.nextq := NIL;
      dNEW(IORequest^.Chain,SIZE(IORequest^.Chain^));
      IORequest^.Chain^.refcnt := 0;
      IORequest^.Chain^.copycnt := 1;
      IORequest^.Chain^.nextq := NIL;
      dNEW(Transaction,SIZE(Transaction^));
      Transaction^.refcnt := 0;
      Transaction^.copycnt := 1;
      Transaction^.nextq := NIL;

      IORequest^.Chain^.TransactionQueue := InitQ("TransactionQueue", FALSE, 0);

      NodeNumber := FindNodeNumber(Node);

      (* Now convert Configuration into a Port Enable Register command. *)
      ConvertPortStatusToEnable(NodeStatus[NodeNumber].PortStatus, PortEnableRegister);

      (* Generate the command to grow to the Root Node.  *)
      Command := MakeNodeConfigurationCommand(Node, PortEnableRegister);

      (* Generate the transaction.  *)
      WITH Transaction^ DO

          Identifier   := Node;
          TimeOutValue := TransactionTimeOut;
          OutputFrame  := Command;

      END;

      (* Enter the transaction on the Transaction Queue.  *)
      QInsert(Transaction, IORequest^.Chain^.TransactionQueue, FALSE);
(*
      INSERT Transaction LAST IN IORequest^.Chain^.TransactionQueue;
*)

      WITH IORequest^.Chain^ DO

          ChainIdentifier       := AddGPCIdentifier;
          NumberOfTransactions  := QSize(IORequest^.Chain^.TransactionQueue);

      END;

      WITH IORequest^ DO
```

```
             Priority           := NetworkManagerPriority;
             OnDemand           := FALSE;
             RequestTimeoutValue := ComputeChainTimeout(0, Chain^.NumberOfTransactions);
             RequestType        := NetworkManagerRequest;

         END;

     END AddGPCToNetwork;

     (***************************************************************************)

     PROCEDURE DisabledTransmitTest(TargetNode     : INTEGER;
                                    NodeStatus     : NodeStatusArray;
                                    VAR IORequest  : IORequestType);

         CONST DisabledTransmitIdentifier = 250;

         VAR Transaction          : TransactionType;
             Command              : BusMessageType;
             PortEnableRegister   : PortEnableRegisterType;
             TargetNumber         : INTEGER;

     BEGIN

         dNEW(IORequest,SIZE(IORequest^));
         IORequest^.refcnt := 0;
         IORequest^.copycnt := 1;
         IORequest^.nextq := NIL;
         dNEW(IORequest^.Chain,SIZE(IORequest^.Chain^));
         IORequest^.Chain^.refcnt := 0;
         IORequest^.Chain^.copycnt := 1;
         IORequest^.Chain^.nextq := NIL;
         dNEW(Transaction,SIZE(Transaction^));
         Transaction^.refcnt := 0;
         Transaction^.copycnt := 1;
         Transaction^.nextq := NIL;

         IORequest^.Chain^.TransactionQueue := InitQ("TransactionQueue", FALSE, 0);

         TargetNumber := FindNodeNumber(TargetNode);

         (* Now convert Configuration into a Port Enable Register command. *)
         ConvertPortStatusToEnable(NodeStatus[TargetNumber].PortStatus, PortEnableRegister);

         (* Generate the command to disable all ports on target node*)
         Command := MakeNodeConfigurationCommand(TargetNode, PortEnableRegister);

         (* Generate the transaction.  *)
         WITH Transaction^ DO

             Identifier   := TargetNode;
             TimeOutValue := TransactionTimeOut;
             OutputFrame  := Command;

         END;

         (* Enter the transaction on the Transaction Queue.  *)
         QInsert(Transaction, IORequest^.Chain^.TransactionQueue, FALSE);
(*
         INSERT Transaction LAST IN IORequest^.Chain^.TransactionQueue;
*)

         WITH IORequest^.Chain^ DO

             ChainIdentifier      := DisabledTransmitIdentifier;
             NumberOfTransactions := QSize(IORequest^.Chain^.TransactionQueue);

         END;

         WITH IORequest^ DO

             Priority           := NetworkManagerPriority;
```

```
          OnDemand            := FALSE;
          RequestTimeoutValue := ComputeChainTimeout(0, Chain^.NumberOfTransactions);
          RequestType         := NetworkManagerRequest;

      END;

  END DisabledTransmitTest;

  (************************************************************************)

  PROCEDURE DisabledRetransmitTest(TestNode      : INTEGER;
                                   TargetNode    : INTEGER;
                                   NodeStatus    : NodeStatusArray;
                                   VAR IORequest : IORequestType);

      CONST DisabledRetransmitChainIdentifier = 251;

      VAR Transaction        : TransactionType;
          Command            : BusMessageType;
          PortEnableRegister : PortEnableRegisterType;
          SpawningNumber     : INTEGER;
          TargetNumber       : INTEGER;

  BEGIN

      SpawningNumber := FindNodeNumber(TestNode);
      TargetNumber   := FindNodeNumber(TargetNode);

      dNEW(IORequest,SIZE(IORequest^));
      IORequest^.refcnt := 0;
      IORequest^.copycnt := 1;
      IORequest^.nextq := NIL;
      dNEW(IORequest^.Chain,SIZE(IORequest^.Chain^));
      IORequest^.Chain^.refcnt := 0;
      IORequest^.Chain^.copycnt := 1;
      IORequest^.Chain^.nextq := NIL;

      IORequest^.Chain^.TransactionQueue := InitQ("TransactionQueue", FALSE, 0);

      (* Convert the spawning node's port status to a Port Enable Register. *)
      ConvertPortStatusToEnable(NodeStatus[SpawningNumber].PortStatus, PortEnableRegister);

      (* Generate command for Spawning node to turn off the port connected
         to the target node.  *)
      Command := MakeNodeConfigurationCommand(TestNode, PortEnableRegister);

      dNEW(Transaction,SIZE(Transaction^));
      Transaction^.refcnt := 0;
      Transaction^.copycnt := 1;
      Transaction^.nextq := NIL;
      WITH Transaction^ DO

          Identifier   := TestNode;
          TimeOutValue := TransactionTimeOut;
          OutputFrame  := Command;

      END;

      (* Enter the transaction on the Transaction Queue.  *)
      QInsert(Transaction, IORequest^.Chain^.TransactionQueue, FALSE);
(*
      INSERT Transaction LAST IN IORequest^.Chain^.TransactionQueue;
*)

      (* Generate a transaction for target node to return its status.  *)

      (* Convert the target node's port status to a Port Enable Register.  *)
      ConvertPortStatusToEnable(NodeStatus[TargetNumber].PortStatus, PortEnableRegister);

      (* Generate command for Spawning node to return its status.  *)
      Command := MakeMonitorCommand(TargetNode);
```

```
        dNEW(Transaction,SIZE(Transaction^));
        Transaction^.refcnt := 0;
        Transaction^.copycnt := 1;
        Transaction^.nextq := NIL;
        WITH Transaction^ DO

            Identifier   := TargetNode;
            TimeOutValue := TransactionTimeOut;
            OutputFrame  := Command;

        END;

        (* Enter the transaction on the Transaction Queue.  *)
        QInsert(Transaction, IORequest^.Chain^.TransactionQueue, FALSE);
(*
        INSERT Transaction LAST IN IORequest^.Chain^.TransactionQueue;
*)

        WITH IORequest^.Chain^ DO

            ChainIdentifier       := DisabledRetransmitChainIdentifier;
            NumberOfTransactions  := QSize(IORequest^.Chain^.TransactionQueue);

        END;

        WITH IORequest^ DO

            Priority            := NetworkManagerPriority;
            OnDemand            := FALSE;
            RequestTimeoutValue := ComputeChainTimeout(0, Chain^.NumberOfTransactions);
            RequestType         := NetworkManagerRequest;

        END;

    END DisabledRetransmitTest;

    (**************************************************************************)

    PROCEDURE ResetConfigurationCommand(TestNode      : INTEGER;
                                        NodeStatus    : NodeStatusArray;
                                        VAR IORequest : IORequestType);

        CONST ResetConfigurationIdentifier = 252;

        VAR Transaction         : TransactionType;
            Command             : BusMessageType;
            PortEnableRegister  : PortEnableRegisterType;
            TestNumber          : INTEGER;

    BEGIN

        dNEW(IORequest,SIZE(IORequest^));
        IORequest^.refcnt := 0;
        IORequest^.copycnt := 1;
        IORequest^.nextq := NIL;
        dNEW(IORequest^.Chain,SIZE(IORequest^.Chain^));
        IORequest^.Chain^.refcnt := 0;
        IORequest^.Chain^.copycnt := 1;
        IORequest^.Chain^.nextq := NIL;
        dNEW(Transaction,SIZE(Transaction^));
        Transaction^.refcnt := 0;
        Transaction^.copycnt := 1;
        Transaction^.nextq := NIL;

        IORequest^.Chain^.TransactionQueue := InitQ("TransactionQueue", FALSE, 0);

        TestNumber := FindNodeNumber(TestNode);

        (* Now convert Configuration into a Port Enable Register command. *)
        ConvertPortStatusToEnable(NodeStatus[TestNumber].PortStatus, PortEnableRegister);

        (* Generate the command to grow to the Root Node.  *)
```

```
       Command := MakeNodeConfigurationCommand(TestNode, PortEnableRegister);

       (* Generate the transaction.  *)
       WITH Transaction^ DO

           Identifier   := TestNode;
           TimeOutValue := TransactionTimeOut;
           OutputFrame  := Command;

       END;

       (* Enter the transaction on the Transaction Queue.  *)
       QInsert(Transaction, IORequest^.Chain^.TransactionQueue, FALSE);
(*
       INSERT Transaction LAST IN IORequest^.Chain^.TransactionQueue;
*)

       WITH IORequest^.Chain^ DO

           ChainIdentifier      := ResetConfigurationIdentifier;
           NumberOfTransactions := QSize(IORequest^.Chain^.TransactionQueue);

       END;

       WITH IORequest^ DO

           Priority            := NetworkManagerPriority;
           OnDemand            := FALSE;
           RequestTimeoutValue := ComputeChainTimeout(0, Chain^.NumberOfTransactions);
           RequestType         := NetworkManagerRequest;

       END;

    END ResetConfigurationCommand;

    (**********************************************************************)

    PROCEDURE MakeMonitorRequest(NodeConnection : NodeArrayType) : IORequestType;

        CONST TalkerOutOfTurnIdentifier = 253;

        VAR MonitorRequest  : IORequestType;
            Transaction     : TransactionType;
            Command         : BusMessageType;
            NodeIndex       : INTEGER;

    BEGIN

        dNEW(MonitorRequest,SIZE(MonitorRequest^));
        MonitorRequest^.refcnt := 0;
        MonitorRequest^.copycnt := 1;
        MonitorRequest^.nextq := NIL;
        dNEW(MonitorRequest^.Chain,SIZE(MonitorRequest^.Chain^));
        MonitorRequest^.Chain^.refcnt := 0;
        MonitorRequest^.Chain^.copycnt := 1;
        MonitorRequest^.Chain^.nextq := NIL;
        MonitorRequest^.Chain^.TransactionQueue := InitQ("TransactionQueue", FALSE, 0);

        FOR NodeIndex := 1 TO NumberOfNodes DO

            WITH NodeConnection[NodeIndex] DO

                (* A node addresses are initialize to their index in the
                   array, if the address is not the index then a node
                   must be present in this index.  *)
                IF NodeAddress <> NodeIndex THEN

                    Command := MakeMonitorCommand(NodeAddress);

                    dNEW(Transaction,SIZE(Transaction^));
                    Transaction^.refcnt := 0;
                    Transaction^.copycnt := 1;
                    Transaction^.nextq := NIL;
                    WITH Transaction^ DO

                        Identifier := NodeAddress;
                        TimeOutValue := TransactionTimeOut;
                        OutputFrame := Command;

                    END;

                    QInsert(Transaction, MonitorRequest^.Chain^.TransactionQueue, FALSE);

                END;

            END;

        END;

        WITH MonitorRequest^.Chain^ DO

            ChainIdentifier      := TalkerOutOfTurnIdentifier;
            NumberOfTransactions := QSize(MonitorRequest^.Chain^.TransactionQueue)

        END;

        WITH MonitorRequest^ DO

            Priority            := NetworkManagerPriority;
            RequestTimeoutValue := ComputeChainTimeout(0, Chain^.NumberOfTransactions);
            OnDemand            := FALSE;
            RequestType         := NetworkManagerRequest;

        END;

        RETURN(MonitorRequest);

    END MakeMonitorRequest;

    (*************************************************************************)

END GrowNet.
```

REPAIR

```
DEFINITION MODULE Repair;

    FROM IOService IMPORT IORequestType;

    FROM IOS IMPORT ChainStatusData;

    FROM TypeConst IMPORT NodeArrayType, NodeStatusArray, ChannelIDType;

    FROM BusMessag IMPORT PortNameType, NumberOfNodes;

    EXPORT QUALIFIED NodeSetRange, NodeSet, AnalysisStatusType, FaultType,
                FaultAnalysisRecordType, ErrorRecordType, BranchArrayType,
                ReconnectRoot,  BranchReconnect, TalkToRootOfTree,
                ErrorAnalysis;

    TYPE NodeSetRange = [1 .. NumberOfNodes];
         NodeSet = SET OF NodeSetRange;

         BranchArrayType = ARRAY [1 .. 3] OF NodeSet;

         AnalysisStatusType = (AnalysisSucessful, AnalysisUnsucessful);
         FaultType = (RootLinkFailure, LinkFailure);

         FaultAnalysisRecordType = RECORD

             CASE Fault : FaultType OF

                RootLinkFailure:

                    FailedChannel : ChannelIDType;

                | LinkFailure:

                    FailedRoot        : INTEGER;
                    FailedInboardPort : PortNameType;
                    FailedNodeSet     : NodeSet;
                    FailedNodeCount   : INTEGER;

             END;

         END;

         ErrorRecordType = RECORD

             CASE Status : AnalysisStatusType OF

                AnalysisSucessful:

                    FaultAnalysisRecord : FaultAnalysisRecordType;

                | AnalysisUnsucessful:

             END;

         END;

   (***************************************************************************)

   PROCEDURE ReconnectRoot(NodeConnections         : NodeArrayType;
                           VAR NetworkStatus       : NodeStatusArray;
                           TargetNode              : INTEGER;
                           FailedNodeSet           : NodeSet;
                           VAR ReconnectIOReqeust  : IORequestType;
                           VAR PortCount           : INTEGER)
                           :BOOLEAN;

   (***************************************************************************)

   PROCEDURE BranchReconnect(ErrorReport                : ErrorRecordType;
                             NodeConnections            : NodeArrayType;
                             VAR ReconnectNode          : INTEGER;
                             VAR ReconnectPort             : PortNameType;
                             VAR NodeStatus                : NodeStatusArray;
                             VAR Branches                  : BranchArrayType;
                             VAR BranchReconnectIORequest  : IORequestType;
                             VAR PortCount                 : INTEGER);

    (**************************************************************************)

    PROCEDURE ErrorAnalysis(MonitorResponse      : ChainStatusData;
                            NetworkConfiguration : NodeStatusArray;
                            NodeConnections      : NodeArrayType;
                            VAR ErrorReport      : ErrorRecordType;
                            VAR PortCount        : INTEGER);

    (**************************************************************************)

    PROCEDURE TalkToRootOfTree(Root              : INTEGER;
                               VAR TalkIORequest : IORequestType);

    (**************************************************************************)

END Repair.
```

REPAIR

```
IMPLEMENTATION MODULE Repair;

   FROM TypeConst IMPORT NodeArrayType, NodeStatusArray, PortStatusArray,
             StatusType, PortConfigurationType, NetworkElementType,
             ChannelIDType;

   FROM BusMessag IMPORT PortNameType, BusMessageType,
             PortEnableRegisterType, NumberOfPortsPerNode, NumberOfNodes,
             MakeNodeConfigurationCommand, MakeMonitorCommand;

   FROM IOService IMPORT IORequestType, RequestActivityType;

   FROM IOS IMPORT ChainType, TransactionType, ChainStatusData,
             InputFrameType, InputFrameQueueType, TimeOutIndicatorType;

   FROM CentralDB IMPORT FindNodeNumber;

   FROM GrowNet IMPORT TransactionTimeOut, NetworkManagerPriority;

   FROM Utilities IMPORT ConvertPortStatusToEnable, ComputeChainTimeout;

   FROM QueueM IMPORT QInsert, InitQ, FirstQ, QSucc, QSize, dNEW;

   FROM Storage IMPORT ALLOCATE;

   FROM SYSTEM IMPORT SIZE;

   FROM InOut IMPORT WriteString, WriteLn;


   CONST ReconnectChainIdentifier = 221;
         TalkChainIdentifier      = 222;

   (**************************************************************************)

   PROCEDURE GetNodeResponse(Responses          : InputFrameQueueType;
                             Node               : INTEGER;
                             VAR NodeResponse   : InputFrameType);

      VAR InputFrame : InputFrameType;

   BEGIN

      NodeResponse := NIL;
      InputFrame := FirstQ(Responses);

      LOOP

         IF FindNodeNumber(InputFrame^.MessageAddress) = Node THEN

            NodeResponse := InputFrame;
            EXIT;

         END;

         InputFrame := QSucc(InputFrame, Responses);
         IF InputFrame = NIL THEN

            EXIT;

         END;

      END;

   END GetNodeResponse;

   (**************************************************************************)
   (* This procedure searchs the network status for an idle port on this
      node that is connected to another node.
      If so, the port and a true value will be returned, otherwise false
      will be returned.  *)
```

```
PROCEDURE IdlePortOnNode(TestNode           : INTEGER;
                         NetworkStatus      : NodeStatusArray;
                         NetworkConnections : NodeArrayType;
                         VAR IdlePort       : PortNameType;
                         VAR Count          : INTEGER) :BOOLEAN;

    VAR IdlePortFound : BOOLEAN;
        TestNumber  : INTEGER;

BEGIN

    TestNumber := FindNodeNumber(TestNode);

    LOOP

        INC(Count);
        IF (NetworkStatus[TestNumber].PortStatus[IdlePort].Status = Idle)
            AND (NetworkConnections[TestNumber].PortArray[IdlePort].
            AdjacentElement = Node) THEN

            IdlePortFound := TRUE;
            EXIT;

        ELSIF (IdlePort = NumberOfPortsPerNode) THEN

            IdlePortFound := FALSE;
            EXIT;

        ELSE

            IdlePort := IdlePort + 1;

        END;

    END;

    RETURN(IdlePortFound);

END IdlePortOnNode;

(*************************************************************************)
(* This procedure checks the Spawning node to see if it is present in
   the failed node set.  If it is not, the true will be returned with
   the spawning port, otherwise false will be returned.  *)

PROCEDURE ValidSpawningNode(SpawningNode       : INTEGER;
                            TargetNode         : INTEGER;
                            NetworkStatus      : NodeStatusArray;
                            NetworkConnections : NodeArrayType;
                            FailedNodeSet      : NodeSet;
                            VAR SpawningPort   : PortNameType;
                            VAR Count          : INTEGER) :BOOLEAN;

    VAR ValidSpawningNodeFound : BOOLEAN;
        SpawningNumber         : INTEGER;

BEGIN

    SpawningNumber := FindNodeNumber(SpawningNode);

    (* Check to see if spawning node is in failed node list.  *)
    IF NodeSetRange(SpawningNumber) IN FailedNodeSet THEN

        (* Spawning Node was in failed node set.  A valid spawning node
           has NOT been found.  *)
        ValidSpawningNodeFound := FALSE;

    ELSE

        (* Determine outboard port of spawning node *)
        SpawningPort := 1;
        LOOP
```

```
            INC(Count);
            IF SpawningPort = NumberOfPortsPerNode THEN

                ValidSpawningNodeFound := FALSE;
                EXIT;

            ELSIF NetworkStatus[SpawningNumber].PortStatus[SpawningPort].
                Status <> Idle THEN

                SpawningPort := SpawningPort + 1;

            ELSIF (NetworkStatus[SpawningNumber].PortStatus
                [SpawningPort].Status = Idle) AND
                (NetworkConnections[SpawningNumber].PortArray
                [SpawningPort].NodeAddress <> TargetNode) THEN

                SpawningPort := SpawningPort + 1;

            ELSE

                ValidSpawningNodeFound := TRUE;
                EXIT;

            END;

        END;

    END;

    RETURN(ValidSpawningNodeFound);

END ValidSpawningNode;

(*****************************************************************************)
(* This procedure formats an I/O chain for the Network manager to
   reconnect a spawing node to a target node.  *)
PROCEDURE MakeReconnectChain(SpawningNode          : INTEGER;
                             SpawningPort          : PortNameType;
                             TargetNode            : INTEGER;
                             TargetPort            : PortNameType;
                             NetworkStatus         : NodeStatusArray;
                             VAR ReconnectRequest  : IORequestType);

    VAR SpawningNumber     : INTEGER;
        TargetNumber       : INTEGER;
        Transaction        : TransactionType;
        Command            : BusMessageType;
        PortEnableRegister : PortEnableRegisterType;

BEGIN

    SpawningNumber := FindNodeNumber(SpawningNode);
    TargetNumber   := FindNodeNumber(TargetNode);
    dNEW(ReconnectRequest,SIZE(ReconnectRequest^));
    ReconnectRequest^.refcnt := 0;
    ReconnectRequest^.copycnt := 1;
    ReconnectRequest^.nextq := NIL;
    dNEW(ReconnectRequest^.Chain,SIZE(ReconnectRequest^.Chain^));
    ReconnectRequest^.Chain^.refcnt := 0;
    ReconnectRequest^.Chain^.copycnt := 1;
    ReconnectRequest^.Chain^.nextq := NIL;

    ReconnectRequest^.Chain^.TransactionQueue := InitQ("TransactionQueue", FALSE, 0);

    (* Convert the spawning node's port status to a Port Enable Register. *)
    ConvertPortStatusToEnable(NetworkStatus[SpawningNumber].PortStatus,
            PortEnableRegister);

    (* Generate command for Spawning node to turn on outboard port.  *)
    Command := MakeNodeConfigurationCommand(SpawningNode, PortEnableRegister);
```

```
        dNEW(Transaction,SIZE(Transaction^));
        Transaction^.refcnt := 0;
        Transaction^.copycnt := 1;
        Transaction^.nextq := NIL;
        WITH Transaction^ DO

            Identifier   := SpawningNode;
            TimeOutValue := TransactionTimeOut;
            OutputFrame  := Command;

        END;

        (* Enter the transaction on the Transaction Queue.  *)
        QInsert(Transaction, ReconnectRequest^.Chain^.TransactionQueue, FALSE);
(*
        INSERT Transaction LAST IN ReconnectRequest^.Chain^.TransactionQueue;
*)

        (* Generate a transaction for target node to turn on  its inboard port.*)
        (* Convert the target node's port status to a Port Enable Register.  *)
        ConvertPortStatusToEnable(NetworkStatus[TargetNumber].PortStatus,
                PortEnableRegister);

        (* Generate command for Spawning node to turn on outboard port.  *)
        Command := MakeNodeConfigurationCommand(TargetNode, PortEnableRegister);

        dNEW(Transaction,SIZE(Transaction^));
        Transaction^.refcnt := 0;
        Transaction^.copycnt := 1;
        Transaction^.nextq := NIL;
        WITH Transaction^ DO

            Identifier   := TargetNode;
            TimeOutValue := TransactionTimeOut;
            OutputFrame  := Command;

        END;

        (* Enter the transaction on the Transaction Queue.  *)
        QInsert(Transaction, ReconnectRequest^.Chain^.TransactionQueue, FALSE);
(*
        INSERT Transaction LAST IN ReconnectReqeust^.Chain^.TransactionQueue;
*)

        WITH ReconnectRequest^.Chain^ DO

            ChainIdentifier      := ReconnectChainIdentifier;
            NumberOfTransactions := QSize(ReconnectRequest^.Chain^.TransactionQueue);

        END;

        WITH ReconnectRequest^ DO

            Priority            := NetworkManagerPriority;
            OnDemand            := FALSE;
            RequestTimeoutValue := ComputeChainTimeout(0, Chain^.NumberOfTransactions);
            RequestType         := NetworkManagerRequest;

        END;

    END MakeReconnectChain;

    (*************************************************************************)

    PROCEDURE FindBranches (FailedNodeSet         : NodeSet;
                            FailedRoot            : INTEGER;
                            NodeConnections       : NodeArrayType;
                            NodeStatus            : NodeStatusArray;
                            VAR BranchesFromRoot  : BranchArrayType);

    VAR NodeIndex        : INTEGER;
        PortIndex        : PortNameType;
```

```
  BranchIndex      : INTEGER;
  FailedRootNumber : INTEGER;
  FailedSet        : NodeSet;

  (**********************************************************************)

  PROCEDURE TraverseBranch(VAR BranchSet    : NodeSet;
                           VAR FailedSet    : NodeSet;
                           FailedRoot       : INTEGER;
                           NodeConnections  : NodeArrayType;
                           NodeStatus       : NodeStatusArray);

  VAR PortIndex        : PortNameType;
      FailedRootNumber : INTEGER;

  BEGIN

      FailedRootNumber := FindNodeNumber(FailedRoot);

      FOR PortIndex := 1 TO NumberOfPortsPerNode DO

          (*Determine the Number of Branches.  Find the Root of each
            branch and put in Branch from Root Array. *)

          WITH NodeConnections[FailedRootNumber].PortArray[PortIndex] DO

              IF (NodeStatus[FailedRootNumber].PortStatus[PortIndex].Status
                  = Active) AND (AdjacentElement = Node) AND
                  (NodeSetRange(NodeNumber) IN FailedSet) THEN

                  (* A root of a branch has been found.  *)
                  INCL (BranchSet, NodeSetRange(NodeNumber));
                  EXCL (FailedSet, NodeSetRange(NodeNumber));

                  TraverseBranch(BranchSet, FailedSet, NodeAddress,
                      NodeConnections, NodeStatus);

              END;

          END;

      END;

   END TraverseBranch;

   (**********************************************************************)

BEGIN

    (* Copy Failed Node Set in Failed Set for local manipulation.  *)
    FailedSet := NodeSet { };
    FailedSet := FailedSet + FailedNodeSet;

    FailedRootNumber := FindNodeNumber(FailedRoot);

    (* Remove Failed Root from failed set.  The failed root
       is from where the branches start, so it should not be
       in the failed node set.  *)
    EXCL (FailedSet, NodeSetRange(FailedRootNumber));

    (* Initialize Branches from Root Of Failed Tree to Nil.  *)
    FOR BranchIndex := 1 TO 3 DO

        BranchesFromRoot[BranchIndex] := NodeSet { };

    END;

    BranchIndex := 1;

    (*Determine the Number of Branches.  Find the Root of each
      branch and put in Branch from Root Array. *)
    FOR PortIndex := 1 TO NumberOfPortsPerNode DO
```

```
        WITH NodeConnections[FailedRootNumber].PortArray[PortIndex] DO

            IF (NodeStatus[FailedRootNumber].PortStatus[PortIndex].Status
                = Active) AND (AdjacentElement = Node) AND
                (NodeSetRange(NodeNumber) IN FailedNodeSet) THEN

                (* A root of a branch has been found.  *)
                INCL (BranchesFromRoot[BranchIndex], NodeSetRange(NodeNumber));
                EXCL (FailedSet, NodeSetRange(NodeNumber));


                TraverseBranch(BranchesFromRoot[BranchIndex], FailedSet,
                    NodeAddress, NodeConnections, NodeStatus);

                BranchIndex := BranchIndex + 1;

            END;

        END;

    END;

END FindBranches;

(***************************************************************************)

PROCEDURE ErrorAnalysis(MonitorResponse      : ChainStatusData;
                        NetworkConfiguration : NodeStatusArray;
                        NodeConnections      : NodeArrayType;
                        VAR ErrorReport      : ErrorRecordType;
                        VAR PortCount        : INTEGER);

    VAR NodeResponse     : InputFrameType;
        FailedRootNumber : INTEGER;
        NodeIndex        : INTEGER;
        PortIndex        : PortNameType;

BEGIN

    PortCount := 0;
    ErrorReport.Status := AnalysisSucessful;
    IF MonitorResponse^.AllFailed THEN

        WITH ErrorReport.FaultAnalysisRecord DO

            Fault         := RootLinkFailure;
            FailedChannel := C;

        END;

    ELSE

        WITH ErrorReport.FaultAnalysisRecord DO

            Fault           := LinkFailure;
            FailedNodeCount := 0;
            FailedNodeSet   := NodeSet {};

        END;

(*      FOREACH NodeResponse IN MonitorResponse^.InputFrameQueue DO
*)
        NodeResponse := FirstQ(MonitorResponse^.InputFrameQueue);
        LOOP

            IF NodeResponse^.TransactionTimeOutIndicator = TimedOut THEN

                WITH ErrorReport.FaultAnalysisRecord DO

                    FailedNodeCount := FailedNodeCount + 1;
                    INCL(FailedNodeSet, FindNodeNumber
                           (NodeResponse^.MessageAddress));

              END;

          END;

          NodeResponse := QSucc(NodeResponse,
                                  MonitorResponse^.InputFrameQueue);

          IF NodeResponse = NIL THEN

              EXIT;

          END;

      END;

      (* Find the inboard port of the failed tree.  *)
      WITH ErrorReport.FaultAnalysisRecord DO

          IF FailedNodeSet = NodeSet {} THEN

              (* Must have been a DIU link failure. Set analysis to
                 unsuccessful.  *)
              ErrorReport.Status := AnalysisUnsucessful;

          ELSIF RootOfFailedTree(FailedNodeSet, NetworkConfiguration,
              NodeConnections, MonitorResponse, FailedRoot, PortCount) THEN

              FailedRootNumber := FindNodeNumber(FailedRoot);

              PortIndex := 1;
              WHILE (NetworkConfiguration[FailedRootNumber].PortStatus
                  [PortIndex].Status <> Active) OR
                  (NetworkConfiguration[FailedRootNumber].PortStatus
                  [PortIndex].Direction <> Inboard) OR
                  (NodeConnections[FailedRootNumber].PortArray[PortIndex].
                  AdjacentElement <> Node) DO

                  INC(PortCount);
                  PortIndex := PortIndex + 1;

              END;

              FailedInboardPort := PortIndex;

          ELSE

              WriteString("Unable to find root of failed tree.");
              WriteLn();

          END;

      END;

    END;

END ErrorAnalysis;
(***************************************************************************)

PROCEDURE RootOfFailedTree(FailedSet            : NodeSet;
                           NetworkConfiguration : NodeStatusArray;
                           NetworkConnections   : NodeArrayType;
                           MonitorResponses     : ChainStatusData;
                           VAR Root             : INTEGER;
                           VAR Count            : INTEGER)
                           :BOOLEAN;

    VAR FoundRoot          : BOOLEAN;
        FailedNodes        : NodeSet;
        RootNumber         : INTEGER;
        AdjacentNode         : INTEGER;
        SetIndex             : NodeSetRange;
        PortIndex            : PortNameType;
        InboardPort          : PortNameType;
        AdjacentNodeResponse : InputFrameType;

BEGIN

    FoundRoot := FALSE;
    FailedNodes := FailedSet;

    FOR SetIndex := 1 TO NumberOfNodes DO

        IF SetIndex IN FailedNodes THEN

            EXCL(FailedNodes, SetIndex);
            (* Find the inboard port of this node.  *)
            PortIndex := 1;

            LOOP

                INC(Count);
                WITH NetworkConfiguration[SetIndex].PortStatus[PortIndex] DO

                    IF (Status = Active) AND (Direction = Inboard) AND
                        (NetworkConnections[SetIndex].PortArray[PortIndex].
                        AdjacentElement = Node) THEN

                        InboardPort := PortIndex;
                        EXIT;

                    ELSE

                        (* No check is make for last port on node because
                           we expect to find an inboard port.  *)
                        PortIndex := PortIndex + 1;

                    END;

                END;

            END;

            CASE NetworkConnections[SetIndex].PortArray[InboardPort].
                AdjacentElement OF

                Node:

                    AdjacentNode := NetworkConnections[SetIndex].
                                    PortArray[InboardPort].NodeNumber;

                    GetNodeResponse(MonitorResponses^.InputFrameQueue,
                            AdjacentNode, AdjacentNodeResponse);

                    (* This loop will need to modified to support
                       a failed node set that does not have a
                       root.  *)
                    IF AdjacentNodeResponse^.TransactionTimeOutIndicator
                        = NormalCompletion THEN

                        FoundRoot := TRUE;
                        Root      := NetworkConnections[SetIndex].NodeAddress;

                    END;

            END;

        END;

    END;

    RETURN(FoundRoot);
```

```
END RootOfFailedTree;

(***************************************************************************)

PROCEDURE ReconnectRoot(NodeConnections        : NodeArrayType;
                        VAR NetworkStatus      : NodeStatusArray;
                        TargetNode             : INTEGER;
                        FailedNodeSet          : NodeSet;
                        VAR ReconnectIORequest : IORequestType;
                        VAR PortCount          : INTEGER)
                        :BOOLEAN;

    VAR ReconnectSuccessful  : BOOLEAN;
        SpawningNode         : INTEGER;
        SpawningNumber       : INTEGER;
        SpawningPort         : PortNameType;
        TargetPort           : PortNameType;
        TargetNumber         : INTEGER;
BEGIN

    PortCount           := 0;
    TargetNumber        := FindNodeNumber(TargetNode);
    ReconnectSuccessful := FALSE;

    (* Set Target Port to start searching with port 1.  *)
    TargetPort := 1;
    LOOP

        INC(PortCount);
        IF IdlePortOnNode(TargetNode, NetworkStatus, NodeConnections,
            TargetPort, PortCount) THEN

            (* A potential spawing node has been identified.  *)
            SpawningNode   := NodeConnections[TargetNumber].PortArray
                                     [TargetPort].NodeAddress;
            SpawningNumber := FindNodeNumber(SpawningNode);

            (* Check for a valid spawning node.  *)
            IF ValidSpawningNode(SpawningNode, TargetNode, NetworkStatus,
                    NodeConnections, FailedNodeSet, SpawningPort, PortCount) THEN

                (* Upadate status of the new link used to reconnect
                   nodes.  *)
                WITH NetworkStatus[SpawningNumber].PortStatus[SpawningPort] DO

                    Status    := Active;
                    Direction := Outboard;

                END;

                WITH NetworkStatus[TargetNumber].PortStatus[TargetPort] DO

                    Status    := Active;
                    Direction := Inboard;

                END;

                (* Make chain to reconnect spawning node to target node.  *)
                MakeReconnectChain(SpawningNode, SpawningPort, TargetNode,
                            TargetPort, NetworkStatus, ReconnectIORequest);

                ReconnectSuccessful := TRUE;
                EXIT;

            END;

        ELSIF TargetPort < NumberOfPortsPerNode THEN

            TargetPort := TargetPort + 1;

        ELSE
```

```
              ReconnectSuccessful := FALSE;
              EXIT;

          END;

      END;

      RETURN(ReconnectSuccessful);

  END ReconnectRoot;

  (**************************************************************************)

  PROCEDURE BranchReconnect(ErrorReport                    : ErrorRecordType;
                            NodeConnections                : NodeArrayType;
                            VAR ReconnectNode              : INTEGER;
                            VAR ReconnectPort              : PortNameType;
                            VAR NodeStatus                 : NodeStatusArray;
                            VAR Branches                   : BranchArrayType;
                            VAR BranchReconnectIORequest : IORequestType;
                            VAR PortCount                  : INTEGER);

      VAR BranchNode        : NodeSetRange;
          BranchNumber      : INTEGER;
          SpawningNode      : INTEGER;
          SpawningPort      : PortNameType;
          FailedRootNumber  : INTEGER;
          SpawningNumber    : INTEGER;
          ReconnectNumber   : INTEGER;

    (**********************************************************************)
    (* This function will return a node from the branch.  If assumes
       that a branch with no nodes will be passed to it.  *)

    PROCEDURE GetNodeFromBranch(VAR Branch : NodeSet) :NodeSetRange;

        VAR Node : NodeSetRange;

    BEGIN

        Node:= 1;

        LOOP

            IF Node IN Branch THEN

                EXCL(Branch, Node);
                EXIT;

            ELSE

                Node := Node + 1;

            END;

        END;

        RETURN(Node);

    END GetNodeFromBranch;

    (**********************************************************************)

BEGIN

    WITH ErrorReport.FaultAnalysisRecord DO

        FindBranches (FailedNodeSet, FailedRoot, NodeConnections,
            NodeStatus, Branches);

        FailedRootNumber := FindNodeNumber(FailedRoot);
```

```
END;

BranchNumber := 1;

LOOP

    (* Check current branch to see if it has any nodes left to check.  *)
    IF Branches[BranchNumber] <> NodeSet { } THEN

        (* Get a node from the branch.  *)
        BranchNode      := GetNodeFromBranch(Branches[BranchNumber]);
        ReconnectNode   := NodeConnections[BranchNode].NodeAddress;
        ReconnectNumber := FindNodeNumber(ReconnectNode);

        (* Check this node to see if it contains any idle ports
           which are not connected to the failed node set.  *)
        ReconnectPort := 1;
        IF IdlePortOnNode(ReconnectNode, NodeStatus, NodeConnections,
               ReconnectPort, PortCount) THEN

            (* A potential spawing node has been identified.  *)
            SpawningNode    := NodeConnections[BranchNode].PortArray
                                 [ReconnectPort].NodeAddress;
            SpawningNumber := FindNodeNumber(SpawningNode);

            (* Check for a valid spawning node.  *)
            WITH ErrorReport.FaultAnalysisRecord DO


                IF ValidSpawningNode(SpawningNode, ReconnectNode,
                    NodeStatus, NodeConnections, FailedNodeSet,
                    SpawningPort, PortCount) THEN

                    (* Upadate status of the new link used to reconnect
                       nodes.  *)
                    WITH NodeStatus[SpawningNumber].PortStatus[
                        SpawningPort] DO

                        Status    := Active;
                        Direction := Outboard;

                    END;

                    WITH NodeStatus[ReconnectNumber].PortStatus[
                        ReconnectPort] DO

                        Status    := Active;
                        Direction := Inboard;

                    END;

                    (* Make chain to reconnect spawning node to target
                       node.  *)
                    MakeReconnectChain(SpawningNode, SpawningPort,
                        ReconnectNode, ReconnectPort, NodeStatus,
                        BranchReconnectIORequest);

                    (* A Node on a branch has been identified for
                       attempted reconnection to the network.  *)
                    EXIT;

                END;

            END;

        ELSE

            (* Loop again and check for next node in branch.  *)

        END;
```

```
            ELSE

                BranchNumber := BranchNumber + 1;

            END;

        END;

    END BranchReconnect;
    (*************************************************************************)
    PROCEDURE TalkToRootOfTree(Root                 : INTEGER;
                               VAR TalkIORequest : IORequestType);

        VAR Transaction : TransactionType;
            Command     : BusMessageType;

    BEGIN

        dNEW(TalkIORequest,SIZE(TalkIORequest^));
        TalkIORequest^.refcnt := 0;
        TalkIORequest^.copycnt := 1;
        TalkIORequest^.nextq := NIL;
        dNEW(TalkIORequest^.Chain,SIZE(TalkIORequest^.Chain^));
        TalkIORequest^.Chain^.refcnt := 0;
        TalkIORequest^.Chain^.copycnt := 1;
        TalkIORequest^.Chain^.nextq := NIL;

        TalkIORequest^.Chain^.TransactionQueue := InitQ("TransactionQueue", FALSE, 0);

        Command := MakeMonitorCommand(Root);

        dNEW(Transaction,SIZE(Transaction^));
        Transaction^.refcnt := 0;
        Transaction^.copycnt := 1;
        Transaction^.nextq := NIL;
        WITH Transaction^ DO

            Identifier   := Root;
            TimeOutValue := TransactionTimeOut;
            OutputFrame  := Command;

        END;

        (* Enter the transaction on the Transaction Queue. *)
        QInsert(Transaction, TalkIORequest^.Chain^.TransactionQueue, FALSE);
 (*
        INSERT Transaction LAST IN TalkIORequest^.Chain^.TransactionQueue;
 *)
        WITH TalkIORequest^.Chain^ DO

            ChainIdentifier       := TalkChainIdentifier;
            NumberOfTransactions  := QSize(TalkIORequest^.Chain^.TransactionQueue);

        END;

        WITH TalkIORequest^ DO

            Priority            := NetworkManagerPriority;
            OnDemand            := FALSE;
            RequestTimeoutValue := ComputeChainTimeout(0, Chain^.NumberOfTransactions);
            RequestType         := NetworkManagerRequest;

        END;

    END TalkToRootOfTree;
    (*************************************************************************)
END Repair.
```

NETMANGER

```
DEFINITION DEVM NetManger;
    EXPORT SpawningNodeType*, SpawningQueueType;
    TYPE SpawningNodeType = ENTITY
             Node : INTEGER;
         END;
         SpawningQueueType = QUEUE OF SpawningNodeType;
END NetManger.
```

NETMANGER

```
DEVM NetManger;

    FROM IOService REACH IORequestType*,
               IOResponseType*,
               NetworkManagerServiceRequest*,
               NewNetworkStateType*;

    FROM IOS REACH ChainStatusData*, ChainType*,
               InputFrameType*, TransactionType*;

    FROM Processor REACH ProcessingUnit*;

    FROM BusMessag REACH BusMessageType*;

    FROM IOService IMPORT NetworkManagerActivityType, NetworkHealthType,
               ChainExecutedWithoutError, ReleaseChainResponseMemory;

    FROM IOS IMPORT TimeOutIndicatorType;

    FROM CentralDB IMPORT ReadNodeInterConnections, FindNodeNumber;

    FROM TypeConst IMPORT NodeArrayType, NodeStatusArray, ChannelStatusRecord,
               NetworkElementType, PortStatusArray, StatusType,
               PortConfigurationType;

    FROM BusMessag IMPORT PortNameType, NumberOfPortsPerNode,
               NumberOfNodes, MakeMonitorCommand;

    FROM Utilities IMPORT InitializeStatusVariables, NodesInThisSimulation,
               UpdateLinkStatus, SetNodeStatusFailed;

    FROM GrowNet IMPORT GROWTOROOTNODE, EnableLink, DeleteNodeFromNetwork,
               AddDIUToNetwork, AddGPCToNetwork, DisabledTransmitTest,
               DisabledRetransmitTest, ResetConfigurationCommand,
               MakeMonitorRequest;

    FROM Repair IMPORT NodeSetRange, NodeSet, FaultType,
               FaultAnalysisRecordType, ErrorRecordType, BranchArrayType,
               ErrorAnalysis,  AnalysisStatusType, ReconnectRoot,
               BranchReconnect, TalkToRootOfTree;

    FROM Senddata IMPORT DataElementType, FrequencyType, NonCyclicDataType,
               NonCyclicVariationType, WriteDataElementType;

    INPUTS
        EVENT ServiceRequest     : NetworkManagerServiceRequest;
              IONetworkResponse  : ChainStatusData;
              ProcessorResponse  : ProcessingUnit;
              MissedDeadLine     : INTEGER;      (* ignored, since turned off for the IOP *)
              Reset              : BOOLEAN;

        PARA  NetworkIDToManage        : INTEGER;
              ProcessingPriority       : INTEGER;
              IOPIdentifier            : INTEGER;
              InitialNodeConfiguration : ARRAY [1 .. NumberOfNodes], [1 .. NumberOfPortsPerNode] OF BOOLEAN;
              InitialOrientation       : ARRAY [1 .. NumberOfNodes], [1 .. NumberOfPortsPerNode] OF BOOLEAN;
    END;

    OUTPUTS
        VAR IONetworkRequest  : IORequestType;
            NewNetworkState   : NewNetworkStateType;
            ProcessorRequest  : ProcessingUnit;
    END;

    TYPE NetworkManagerJobType = (GROWNetwork, RepairNetwork);
         GrowNetworkModeType = (GrowToRootNode, AddRemainingNodes, AddDIUS,
                                AddGPCS, DiagnosticCheck);
         RepairNetworkModeType = (DisconnectLink, RepairLink, ReconnectLink,
                                  ReconnectBranch, TalkToRootFailedTree);
         DiagnosticModeType = (LinkEnable, DisabledTransmit, DisabledRetransmit,
                       ResetConfiguration, TalkerOutOfTurn,
                       DiagnosticsComplete);

DIUSpawningRecord = ENTITY

    SpawningNode : INTEGER;
    OutboardPort : PortNameType;

END;

GPCSpawningRecord = ENTITY

    SpawningNode : INTEGER;
    InboardPort  : PortNameType;

END;

DIUListQueueType = QUEUE OF DIUSpawningRecord;
GPCListQueueType = QUEUE OF GPCSpawningRecord;

DiagnosticRecordType = RECORD

    Diagnostic           : DiagnosticModeType;
    TestNode             : INTEGER;
    TestPort             : PortNameType;
    TargetNode           : INTEGER;
    TargetPort           : PortNameType;
    LinkEnableSucessful  : BOOLEAN;
    PortDiagnosticsRun   : BOOLEAN;
    TalkerRequest        : IORequestType;

END;

NetworkGrowthProgressType = RECORD

    CASE Mode : GrowNetworkModeType OF

        GrowToRootNode:

            RootNodeAddress  : INTEGER;
            InboardPort      : PortNameType;

        | AddRemainingNodes:

            SpawningNode  : INTEGER;
            SpawningPort  : PortNameType;
            TargetNode    : INTEGER;
            TargetPort    : PortNameType;

        | AddDIUS:

            DIUAddress : INTEGER;

        | AddGPCS:

            SpareRootNodeAddress : INTEGER;

        | DiagnosticCheck:

    END;

END;

ProcessingType = (Powerup, GrowRequest, NetworkRequest,
       NewState, FaultAnalysis);

ProcessType = RECORD

    CASE Message : ProcessingType OF

        Powerup:
             | GrowRequest, NetworkRequest, FaultAnalysis :

                 IORequest : IORequestType;

             | NewState :

                 StateData : NewNetworkStateType;

         END;

     END;

     DataPointer = POINTER TO ProcessType;

CONST NetworkManagerPriority      = 1;
      NodePowerupInitilizeTime    = 0.000025;
      GrowInitializeNodeTime      = 0.000025;
      NetworkResponseComputation  = 0.000050;
      ComputeOneTransactionChain  = 0.000075;
      ComputeTwoTransactionChain  = 0.000150;
      ChangeNetworkStatus         = 0.000025;
      FixedErrorReportAnalysis    = 0.000075;
      PortErrorAnalysis           = 0.000005;

VAR NetworkState                  : NewNetworkStateType;
    RequestForService             : NetworkManagerServiceRequest;
    NodeConnections               : NodeArrayType;
    NodeStatus                    : NodeStatusArray;
    ChannelStatus                 : ChannelStatusRecord;
    SpawningQueue                 : SpawningQueueType;
    DIUList                       : DIUListQueueType;
    GPCList                       : GPCListQueueType;
    NetworkManagerStatus          : NetworkManagerJobType;
    NetworkGrowthProgress         : NetworkGrowthProgressType;
    DiagnosticStatus              : DiagnosticRecordType;

    NodesInNetwork                : INTEGER;
    NumberOfActiveNodes           : INTEGER;

    NetworkResponse               : ChainStatusData;
    RepairNetworkMode             : RepairNetworkModeType;
    ErrorReport                   : ErrorRecordType;
    BranchNode                    : INTEGER;
    BranchPort                    : PortNameType;
    BranchesOfRoot                : BranchArrayType;
    ReconfigurationStrategy       : RepairNetworkModeType;
    AnalysisPortDecisions         : INTEGER;
    DataCollectionRecord          : DataElementType;
    NodeIndex                     : INTEGER;
    PortIndex                     : PortNameType;
    ErrorIndex                    : INTEGER;
    Transaction                   : TransactionType;
    SourceNode                    : INTEGER;
    TargetNode                    : INTEGER;

    PreviousChainFailed           : BOOLEAN;
    GoodRootNodeFound             : BOOLEAN;

    PowerupRequest                : DataPointer;
    NetworkResponseRequest        : DataPointer;
    ProcessResponse               : DataPointer;
    PowerupProcessing             : ProcessingUnit;
    NetworkResponseProcessing     : ProcessingUnit;
    ProcessingResponse            : ProcessingUnit;
    LastNetworkRequest            : IORequestType;

    RootNode                      : INTEGER;
    FailedNode                    : INTEGER;
    CurrentPort                   : PortNameType;

    IOPPort                       : INTEGER;
```

```
(***************************************************************************)
(* This procedure uses a Node Status Array and a Network Connection
   Array to determine if a Spawning Node has any ports to which
   an idle node is connected.  When an idle node is found, the
   search is stop, the Idle Node Variable is set, and the BOOLEAN
   return variable is set to TRUE, the First Port to Check variable
   is set to the next port to check on the next call to this routine.
   Any ports connected to DIU's found during this search are entered
   on a DIU list for later use.   If no ports connected to idle
   ports are found, then the BOOLEAN is set to FALSE.   *)
PROCEDURE FoundAdjacentIdleNode(SpawningNode                 : INTEGER;
                                VAR SpawningNodeOutboardPort : PortNameType;
                                VAR IdleNode                 : INTEGER;
                                VAR TargetNodeInboardPort    : PortNameType;
                                NodeConnections              : NodeArrayType;
                                NodeStatus                   : NodeStatusArray;
                                DIUList                      : DIUListQueueType;
                                GPCList                      : GPCListQueueType)

                                :BOOLEAN;

    VAR PortIndex          : PortNameType;
        FoundIdleNode      : BOOLEAN;
        DIURecord          : DIUSpawningRecord;
        GPCRecord          : GPCSpawningRecord;
        SpawningNodeNumber : INTEGER;
        IdleNodeNumber     : INTEGER;
BEGIN

    FoundIdleNode      := FALSE;
    SpawningNodeNumber := FindNodeNumber(SpawningNode);

    LOOP

        CASE NodeConnections[SpawningNodeNumber].
            PortArray[SpawningNodeOutboardPort].AdjacentElement OF

            Node:

                IF (NodeStatus[NodeConnections[SpawningNodeNumber].
                    PortArray[SpawningNodeOutboardPort].NodeNumber].
                    Status = Idle) AND (NodeStatus[SpawningNodeNumber].
                    PortStatus[SpawningNodeOutboardPort].Status = Idle) THEN

                    IdleNode       := NodeConnections[SpawningNodeNumber].PortArray
                                          [SpawningNodeOutboardPort].NodeAddress;
                    IdleNodeNumber := NodeConnections[SpawningNodeNumber].PortArray
                                          [SpawningNodeOutboardPort].NodeNumber;
                    FoundIdleNode  := TRUE;
                    PortIndex      := 1;

                    LOOP

                        WITH NodeConnections[IdleNodeNumber].PortArray[PortIndex] DO

                            CASE AdjacentElement OF

                                Node:

                                    IF NodeAddress = SpawningNode THEN

                                        TargetNodeInboardPort := PortIndex;
                                        EXIT;

                                    ELSE

                                        PortIndex := PortIndex + 1;

                                    END;

                                ELSE
```

```
                                     PortIndex := PortIndex + 1;

                             END;

                         END;

                     END;

                 END;

             | DIU:

                 NEW(DIURecord);

                 DIURecord^.SpawningNode := SpawningNode;
                 DIURecord^.OutboardPort := SpawningNodeOutboardPort;

                 INSERT DIURecord LAST IN DIUList;

             | GPC:

                 (* This check keeps the root link currently used
                    for growth from being put on the GPCList.  *)
                 IF NodeStatus[SpawningNodeNumber].PortStatus[SpawningNodeOutboardPort].
                     Status = Idle THEN

                     NEW(GPCRecord);

                     GPCRecord^.SpawningNode := SpawningNode;
                     GPCRecord^.InboardPort  := SpawningNodeOutboardPort;

                     INSERT GPCRecord LAST IN GPCList;

                 END;

            ELSE

        END;

        IF FoundIdleNode THEN

            EXIT;

        ELSIF SpawningNodeOutboardPort < NumberOfPortsPerNode THEN

            SpawningNodeOutboardPort := SpawningNodeOutboardPort + 1;

        ELSE

            EXIT;

        END;

    END;

    RETURN (FoundIdleNode);

END FoundAdjacentIdleNode;

(**************************************************************************)

PROCEDURE NextNodeToAdd(VAR GrowthStatus : NetworkGrowthProgressType;
                        NodeConnections    : NodeArrayType;
                        VAR NodeStatus     : NodeStatusArray;
                        DIUList            : DIUListQueueType);

    VAR NextSpawningNode           : SpawningNodeType;
        NodeToAddToNetwork         : SpawningNodeType;
        DIUElement                 : DIUSpawningRecord;
        AddRemainingNodesIORequest : IORequestType;
        AddDIUIORequest            : IORequestType;
        RequestToAddNode           : DataPointer;
```

```
         ProcessingToAddNode         : ProcessingUnit;
         RequestToAddDIU             : DataPointer;
         ProcessingToAddDIU          : ProcessingUnit;
         NodeIndex                   : INTEGER;

BEGIN

    (* Find out if all the ports on the spawning node have been checked,
       if not continue from the current spawning node,
       otherwise get a new spawning node.  *)
    LOOP

        WITH GrowthStatus DO

            IF FoundAdjacentIdleNode(SpawningNode, SpawningPort, TargetNode,
                TargetPort, NodeConnections, NodeStatus, DIUList, GPCList) THEN

                (*Add the idle node to spawning queue.  *)
                NEW(NodeToAddToNetwork);
                NodeToAddToNetwork^.Node := TargetNode;
                INSERT NodeToAddToNetwork LAST IN SpawningQueue;

                (* Update Status of the two nodes that are being used to
                   add the next node.  *)
                UpdateStatusLinkEnable(SpawningNode, SpawningPort, TargetNode,
                    TargetPort, NodeStatus);

                EnableLink(SpawningNode, TargetNode, NodeStatus, AddRemainingNodesIORequest);

                AddRemainingNodesIORequest^.Chain^.NetworkToBeExecutedOn := NetworkIDToManage;
                AddRemainingNodesIORequest^.Identifier        := MyNodeID;
                AddRemainingNodesIORequest^.ResponseExpected := TRUE;

                NEW(RequestToAddNode);
                RequestToAddNode^.Message    := NetworkRequest;
                RequestToAddNode^.IORequest := AddRemainingNodesIORequest;

                NEW(ProcessingToAddNode);
                ProcessingToAddNode^.Priority            := NetworkManagerPriority;
                ProcessingToAddNode^.ProcessingRequired := ComputeTwoTransactionChain
                                                            + NetworkResponseComputation;
                ProcessingToAddNode^.WriteData           := FALSE;
                ProcessingToAddNode^.ProcessID           := 'NW Add Node Processing';
                ProcessingToAddNode^.Data                := RequestToAddNode;

                NOW outport[IOPPort]^.ProcessorRequest <- ProcessingToAddNode;
                EXIT;

            ELSIF QSize(SpawningQueue) = 0 THEN

                FOR NodeIndex := 1 TO NumberOfNodes DO

                    IF NodeStatus[NodeIndex].Status = Idle THEN

                        NodeStatus[NodeIndex].Status := Failed;

                    END;

                END;

                (* Set mode to add DIUs. *);
                Mode := AddDIUS;

                (* All good nodes are a part of the network, now start
                   adding dius.  *)
                DIUElement := FirstQ(DIUList);
                REMOVE DIUElement FROM DIUList;

                (* Update spawning node and spawning port status so
                   network manager will know what status to change.  *)
                WITH DIUElement^ DO
```

```
                    NodeStatus[FindNodeNumber(SpawningNode)].PortStatus
                        [OutboardPort].Status := Active;
                    NodeStatus[FindNodeNumber(SpawningNode)].PortStatus
                        [OutboardPort].Direction := Outboard;

                    AddDIUToNetwork(SpawningNode, OutboardPort, NodeStatus,
                        AddDIUIORequest);

                    (* Data Collection.  *)
                    DIUAddress := NodeConnections[FindNodeNumber(SpawningNode)].
                                    PortArray[OutboardPort].DIUAddress;

                END;

                DISPOSE(DIUElement);
                AddDIUIORequest^.Chain^.NetworkToBeExecutedOn := NetworkIDToManage;
                AddDIUIORequest^.Identifier         := MyNodeID;
                AddDIUIORequest^.ResponseExpected := TRUE;

                NEW(RequestToAddDIU);
                RequestToAddDIU^.Message   := NetworkRequest;
                RequestToAddDIU^.IORequest := AddDIUIORequest;

                NEW(ProcessingToAddDIU);
                ProcessingToAddDIU^.Priority           := NetworkManagerPriority;
                ProcessingToAddDIU^.ProcessingRequired := ComputeOneTransactionChain
                                                           + NetworkResponseComputation;
                ProcessingToAddDIU^.ProcessID          := 'NW Add DIU Processing';
                ProcessingToAddDIU^.WriteData          := FALSE;
                ProcessingToAddDIU^.Data               := RequestToAddDIU;

                NOW outport[IOPPort]^.ProcessorRequest <- ProcessingToAddDIU;
                EXIT;

            END;

            REMOVE FIRST NextSpawningNode FROM SpawningQueue;
            SpawningNode := NextSpawningNode^.Node;
            SpawningPort := 1;
            DISPOSE(NextSpawningNode);

        END;

    END;

END NextNodeToAdd;

(*************************************************************************)
(* This procdure performs the diagnostic checks durning network growth.
   Since the current model does not include the failure modes that
   this diagnostic sequence checks, not attempt is made to process the
   responses.  It is assumed that the chains produced by this routine
   will complete normally.  *)
PROCEDURE RunDiagnostic(VAR DiagnosticStatus : DiagnosticRecordType;
                       VAR NodeStatus       : NodeStatusArray;
                       NodeConfiguration    : NodeArrayType);

    VAR TestNodeNumber              : INTEGER;
        TargetNodeNumber            : INTEGER;
        ProcessingTime              : REAL;
        DisconnectIORequest         : IORequestType;
        DiagnosticRequest           : DataPointer;
        DiagnosticProcessing        : ProcessingUnit;

BEGIN

    WITH DiagnosticStatus DO

        TestNodeNumber := FindNodeNumber(TestNode);

        (* Determine if the current test port should run tests.
           If so, start tests, otherwise, determine if all ports
   have been tested. If so, run reset configuration and
   talker out of turn test.  *)
IF Diagnostic = LinkEnable THEN

    (* Determine the next port under test.  *)
    LOOP

        WITH NodeConfiguration[TestNodeNumber].PortArray[TestPort] DO

            IF (AdjacentElement = Node) AND
                (NodeStatus[NodeNumber].Status = Idle) AND
                (NodeStatus[TestNodeNumber].PortStatus
                [TestPort].Status = Idle) THEN

                (* A Port with an adjacent idle node has been found.  *)
                TargetNode  := NodeConfiguration[TestNodeNumber].
                                   PortArray[TestPort].NodeAddress;
                TargetPort  := NodeConfiguration[TestNodeNumber].
                                   PortArray[TestPort].Port;

                EXIT;

            ELSIF TestPort < NumberOfPortsPerNode THEN

                TestPort := TestPort + 1;

            ELSIF PortDiagnosticsRun THEN

                (* No more ports to test and some idle ports have
                   been tested.  *)
                Diagnostic := ResetConfiguration;
                EXIT;

            ELSE

                (* This node has no idle ports to run diagnostics on,
                   no reason to reset its configuration.  *)
                Diagnostic := TalkerOutOfTurn;
                EXIT;

            END;

        END;

    END;

END;

TargetNodeNumber  := NodeConfiguration[TestNodeNumber].
                     PortArray[TestPort].NodeNumber;

NEW(DiagnosticRequest);
DiagnosticRequest^.Message  := NetworkRequest;

CASE Diagnostic OF

    LinkEnable:

        (* Update port status so that Enable Link will generate
           the proper command. Only set status since it will be
           changed back later in diagnostics.  *)
        PortDiagnosticsRun := TRUE;

        NodeStatus[TestNodeNumber].PortStatus[TestPort].Status  := Active;
        NodeStatus[TargetNodeNumber].PortStatus[TargetPort].Status  := Active;

        EnableLink(TestNode, TargetNode, NodeStatus, DiagnosticRequest^.IORequest);

        DiagnosticRequest^.IORequest^.Chain^.ChainIdentifier := 200;
        DiagnosticRequest^.IORequest^.Chain^.NetworkToBeExecutedOn := NetworkIDToManage;
        DiagnosticRequest^.IORequest^.Identifier        := MyNodeID;
        DiagnosticRequest^.IORequest^.ResponseExpected := TRUE;
```

```
      ProcessingTime                                  := ComputeTwoTransactionChain;
      Diagnostic                                      := DisabledTransmit;

   | DisabledTransmit:

      IF NOT(LinkEnableSucessful) THEN

         (* Mark the links failed, generate request to
            disconnect failed links, and set diagnostic
            mode to link enable.  *)
         NodeStatus[FindNodeNumber(TestNode)].PortStatus
             [TestPort].Status := Failed;
         NodeStatus[FindNodeNumber(TargetNode)].
             PortStatus[TargetPort].Status := Failed;

         DeleteNodeFromNetwork(TestNode,TestPort, TargetNode,
             TargetPort, NodeStatus, DiagnosticRequest^.IORequest);

         DiagnosticRequest^.IORequest^.Chain^.NetworkToBeExecutedOn := NetworkIDToManage;
         DiagnosticRequest^.IORequest^.Identifier         := MyNodeID;
         DiagnosticRequest^.IORequest^.ResponseExpected   := TRUE;
         ProcessingTime                                   := ComputeTwoTransactionChain;
         Diagnostic                                       := LinkEnable;

      ELSE

         NodeStatus[TargetNodeNumber].PortStatus[TargetPort].Status   := Idle;
         DisabledTransmitTest(TargetNode, NodeStatus, DiagnosticRequest^.IORequest);

         DiagnosticRequest^.IORequest^.Chain^.NetworkToBeExecutedOn := NetworkIDToManage;
         DiagnosticRequest^.IORequest^.Identifier         := MyNodeID;
         DiagnosticRequest^.IORequest^.ResponseExpected := TRUE;
         ProcessingTime                             := ComputeOneTransactionChain;
         Diagnostic                                       := DisabledRetransmit;

      END;

   | DisabledRetransmit:

      NodeStatus[TestNodeNumber].PortStatus[TestPort].Status   := Idle;

      DisabledRetransmitTest(TestNode, TargetNode, NodeStatus,
          DiagnosticRequest^.IORequest);

      DiagnosticRequest^.IORequest^.Chain^.NetworkToBeExecutedOn := NetworkIDToManage;
      DiagnosticRequest^.IORequest^.Identifier := MyNodeID;
      DiagnosticRequest^.IORequest^.ResponseExpected := TRUE;
      ProcessingTime                            := ComputeTwoTransactionChain;
      Diagnostic                                := LinkEnable;
      TestPort                                  := TestPort + 1;

   | ResetConfiguration:

      ResetConfigurationCommand(TestNode, NodeStatus,
          DiagnosticRequest^.IORequest);

      DiagnosticRequest^.IORequest^.Chain^.NetworkToBeExecutedOn := NetworkIDToManage;
      DiagnosticRequest^.IORequest^.Identifier := MyNodeID;
      DiagnosticRequest^.IORequest^.ResponseExpected := TRUE;
      ProcessingTime                            := ComputeOneTransactionChain;
      Diagnostic                                := TalkerOutOfTurn;

   | TalkerOutOfTurn:

      DiagnosticRequest^.IORequest := MakeMonitorRequest(NodeConnections);   (* PRB *)
      WITH DiagnosticRequest^.IORequest^ DO

         Chain^.NetworkToBeExecutedOn := NetworkIDToManage;
         Identifier                   := MyNodeID;
         ResponseExpected             := TRUE;

      END;
```

```
                ProcessingTime               := 0.0;
                Diagnostic                   := DiagnosticsComplete;

        END;

    END;

    NEW(DiagnosticProcessing);
    WITH DiagnosticProcessing^ DO

        Priority              := NetworkManagerPriority;
        ProcessingRequired    := ProcessingTime + NetworkResponseComputation;
        ProcessID             := 'NM Diagnostics';
        WriteData             := FALSE;
        Data                  := DiagnosticRequest;

    END;

    NOW outport[IOPPort]^.ProcessorRequest <- DiagnosticProcessing;

END RunDiagnostic;

(****************************************************************************)
(* This procedure updates the network status variables after a successful
   branch reconnect.  It sets the status of all the active ports
   of the failed node set, except the node through which the reconnect
   was made, to outboard.  *)
PROCEDURE UpdateNodeStatusBranchReconnect(ReconnectNode      : INTEGER;
                                          ReconnectPort      : PortNameType;
                                          FailedRoot         : INTEGER;
                                          VAR NetworkStatus  : NodeStatusArray);

    VAR ReconnectNumber  : INTEGER;
        FailedRootNumber : INTEGER;
        SpawningNumber   : INTEGER;
        TargetNumber     : INTEGER;
        InboardPort      : PortNameType;
        NewInboardPort   : PortNameType;
        OutboardPort     : PortNameType;

 BEGIN

     ReconnectNumber  := FindNodeNumber(ReconnectNode);
     FailedRootNumber := FindNodeNumber(FailedRoot);

     (* Since the repair was sucessful, there will be two ports on the
        Reconnect node that have the status "inboard".
        One is due to the reconnection of the branch is
        connected to the Reconnect port, the other is the inboard
        port prior to the failure, this one should now be outboard.  *)
     SpawningNumber := ReconnectNumber;

     (* Find the port that must be changed to outboard.  *)
     OutboardPort := 1;
     LOOP

        WITH NetworkStatus[SpawningNumber].PortStatus[OutboardPort] DO

        IF (Status = Active) AND (Direction = Inboard) AND
              (OutboardPort <> ReconnectPort) THEN

              (* The old inboard port has been found.  *)
              EXIT;

          ELSE

              OutboardPort := OutboardPort + 1;

          END;

          END;
```

```
      END;

      LOOP

          (* Change Status of old inboard port to outboard.  *)
          NetworkStatus[SpawningNumber].PortStatus[OutboardPort].Status        := Active;
          NetworkStatus[SpawningNumber].PortStatus[OutboardPort].Direction := Outboard;

          (* Change the status of the port adjacent to the new outboard port
             to inboard.  *)
          WITH NodeConnections[SpawningNumber].PortArray[OutboardPort] DO

              NetworkStatus[NodeNumber].PortStatus[Port].Status    := Active;
              NetworkStatus[NodeNumber].PortStatus[Port].Direction := Inboard;

              IF NodeNumber = FailedRootNumber THEN

                  (* No more port statuses need to be changed.  *)
                  EXIT;

              ELSE

                  SpawningNumber := NodeNumber;
                  OutboardPort   := 1;
                  NewInboardPort := Port;

                  LOOP

                      WITH NetworkStatus[SpawningNumber].PortStatus[
                          OutboardPort] DO

                          IF (Status = Active) AND (Direction = Inboard) AND
                              (OutboardPort <> NewInboardPort) THEN

                              (* The old inboard port has been found.  *)
                              EXIT;

                          ELSE

                              OutboardPort := OutboardPort + 1;

                          END;

                      END;

                  END;

              END;

          END;

      END;
    END;
END UpdateNodeStatusBranchReconnect;

(***************************************************************************)

PROCEDURE UpdateStatusLinkEnable(SpawningNode    : INTEGER;
                                 SpawningPort    : PortNameType;
                                 TargetNode      : INTEGER;
                                 TargetPort      : PortNameType;
                                 VAR NodeStatus  : NodeStatusArray);

    VAR SpawningNumber  : INTEGER;
        TargetNumber    : INTEGER;

BEGIN

    SpawningNumber := FindNodeNumber(SpawningNode);
    TargetNumber   := FindNodeNumber(TargetNode);

    NodeStatus[SpawningNumber].PortStatus[SpawningPort].Status    := Active;
    NodeStatus[SpawningNumber].PortStatus[SpawningPort].Direction := Outboard;
```

```
        NodeStatus[TargetNumber].PortStatus[TargetPort].Status          := Active;
        NodeStatus[TargetNumber].PortStatus[TargetPort].Direction       := Inboard;

    END UpdateStatusLinkEnable;

    (***************************************************************************)

    PROCEDURE StartNetworkGrowth(RootLinkToGrowFrom : INTEGER;
                                 VAR GrowProgress   : NetworkGrowthProgressType;
                                 NodeConnections    : NodeArrayType;
                                 VAR NodeStatus     : NodeStatusArray;
                                 NodesInThisNetwork : INTEGER);

        VAR RootNodeNumber          : INTEGER;
            NodeIndex               : INTEGER;
            PortIndex               : PortNameType;
            GrowIORequest           : IORequestType;
            ProcessingTime          : REAL;
            RequestToGrow           : DataPointer;
            GrowRequestProcessing   : ProcessingUnit;

    BEGIN
        (* Now find the Root Node connected to the active IOS.  *)
        FOR NodeIndex := 1 TO NodesInNetwork DO

            FOR PortIndex := 1 TO NumberOfPortsPerNode DO

                IF (NodeConnections[NodeIndex].PortArray[PortIndex].
                    AdjacentElement = GPC) AND
                    (NodeConnections[NodeIndex].PortArray[PortIndex].
                    GPCAddress = RootLinkToGrowFrom) THEN

                    GrowProgress.RootNodeAddress := NodeConnections[NodeIndex].NodeAddress;
                    RootNodeNumber               := FindNodeNumber(GrowProgress.RootNodeAddress);
                    GrowProgress.InboardPort     := PortIndex;

                END;

            END;

        END;

        WITH GrowProgress DO

            Mode := GrowToRootNode;
            (* Set status of Root Node that is being used for network growth.  *)
            NodeStatus[RootNodeNumber].PortStatus[InboardPort].Status    := Active;
            NodeStatus[RootNodeNumber].PortStatus[InboardPort].Direction := Inboard;

            GROWTOROOTNODE(RootNodeAddress, InboardPort, NodeStatus, GrowIORequest);

        END;

        GrowIORequest^.Chain^.NetworkToBeExecutedOn := NetworkIDToManage;
        GrowIORequest^.Identifier                   := MyNodeID;
        GrowIORequest^.ResponseExpected             := TRUE;

        (* This is the processing time required to execute this procedure.  *)
        ProcessingTime := FLOAT(NodesInThisNetwork) * GrowInitializeNodeTime;

        NEW(RequestToGrow);
        RequestToGrow^.Message   := GrowRequest;
        RequestToGrow^.IORequest := GrowIORequest;

        NEW(GrowRequestProcessing);
        GrowRequestProcessing^.Priority             := NetworkManagerPriority;
        GrowRequestProcessing^.ProcessingRequired   := ProcessingTime;
        GrowRequestProcessing^.WriteData            := FALSE;
        GrowRequestProcessing^.ProcessID            := 'NW Grow Processing';
        GrowRequestProcessing^.Data                 := RequestToGrow;
    NOW outport[IOPPort]^.ProcessorRequest <- GrowRequestProcessing;

END StartNetworkGrowth;

(***************************************************************************)

PROCEDURE NetworkRepair(FaultDetectionData : ChainStatusData;
                        VAR NodeStatus     : NodeStatusArray;
                        NodeConnections    : NodeArrayType;
                        VAR ErrorReport    : ErrorRecordType;
                        VAR RepairMode     : RepairNetworkModeType);

    VAR SpawningNode                : INTEGER;
        SpawningPort                : PortNameType;
        SpawningNodeNumber          : INTEGER;
        DisconnectIORequest         : IORequestType;
        RequestToDisconnectLink     : DataPointer;
        ProcessingToDisconnectLink  : ProcessingUnit;
        RequestNewState             : DataPointer;
        ProcessingNewState          : ProcessingUnit;

BEGIN

    (* Analysis monitor response to determine the fault type.  *)
    ErrorAnalysis(FaultDetectionData, NodeStatus, NodeConnections,
            ErrorReport, AnalysisPortDecisions);

    IF (ErrorReport.Status = AnalysisSucessful) AND
        (ErrorReport.FaultAnalysisRecord.Fault = LinkFailure) THEN

        REPORT "%12.8f" clock TAGGED "Link Failure";
        RepairMode := DisconnectLink;

        (* Disconnect the failed tree from the rest of the
           network through the broken node.  *)
        WITH ErrorReport.FaultAnalysisRecord DO

            SpawningNode       := NodeConnections[FindNodeNumber(
                                          FailedRoot)].PortArray[
                                          FailedInboardPort].NodeAddress;
            SpawningNodeNumber := NodeConnections[FindNodeNumber(
                                          FailedRoot)].PortArray[
                                          FailedInboardPort].NodeNumber;
            SpawningPort       := NodeConnections[FindNodeNumber(
                                          FailedRoot)].PortArray[
                                          FailedInboardPort].Port;

            (* Update Status of the failed Links.  *)
            NodeStatus[SpawningNodeNumber].PortStatus[SpawningPort].Status := Failed;
            NodeStatus[FindNodeNumber(FailedRoot)].PortStatus[
                            FailedInboardPort].Status := Failed;

            DeleteNodeFromNetwork(SpawningNode,SpawningPort, FailedRoot,
                        FailedInboardPort, NodeStatus, DisconnectIORequest);

        END;

        DisconnectIORequest^.Chain^.NetworkToBeExecutedOn := NetworkIDToManage;
        DisconnectIORequest^.Chain^.ChainIdentifier := 300;
        DisconnectIORequest^.Identifier             := MyNodeID;
        DisconnectIORequest^.ResponseExpected       := TRUE;

        NEW(RequestToDisconnectLink);
        RequestToDisconnectLink^.Message   := NetworkRequest;
        RequestToDisconnectLink^.IORequest := DisconnectIORequest;

        NEW(ProcessingToDisconnectLink);
        ProcessingToDisconnectLink^.Priority           := NetworkManagerPriority;
        ProcessingToDisconnectLink^.ProcessingRequired := ComputeTwoTransactionChain
                                                             + NetworkResponseComputation;
        ProcessingToDisconnectLink^.WriteData          := FALSE;
        ProcessingToDisconnectLink^.ProcessID          := 'Disconnect/Error Processing';
        ProcessingToDisconnectLink^.Data              := RequestToDisconnectLink;

        NOW outport[IOPPort]^.ProcessorRequest <- ProcessingToDisconnectLink;

    ELSE

        WriteString(ParamOut, "Don't know how to repair this failure.");
        WriteLn(ParamOut);
        WriteString(ParamOut, "Returning network to service.");
        WriteLn(ParamOut);
        REPORT "%12.8f" clock TAGGED "Fault Analysis Unsucessful";
        REPORT "%d" NetworkIDToManage TAGGED "ASSUME DIU LINK FAILURE, RETURN TO SERVICE";

        WITH NetworkState^ DO

            NetworkID    := NetworkIDToManage;
            State        := InService;
            MonitorChain := DiagnosticStatus.TalkerRequest^.Chain;

        END;

        NEW(RequestNewState);
        RequestNewState^.Message   := NewState;
        RequestNewState^.StateData := NetworkState;

        NEW(ProcessingNewState);
        ProcessingNewState^.Priority           := NetworkManagerPriority;
        ProcessingNewState^.ProcessingRequired := ChangeNetworkStatus
                                                    + NetworkResponseComputation;
        ProcessingNewState^.WriteData          := FALSE;
        ProcessingNewState^.ProcessID          := 'New State Processing';
        ProcessingNewState^.Data               := RequestNewState;

        NOW outport[IOPPort]^.ProcessorRequest <- ProcessingNewState;

    END;

END NetworkRepair;

(****************************************************************************)

PROCEDURE DisconnectLinkProcess(Error_Report       : ErrorRecordType;
                                DisconnectResponse : ChainStatusData;
                                VAR NodeStatus     : NodeStatusArray;
                                NodeConnections    : NodeArrayType);

    VAR ReconnectIORequest         : IORequestType;
        RequestForErrorAnalysis    : DataPointer;
        ProcessingForErrorAnalysis : ProcessingUnit;
        ReconnectRootDecisions     : INTEGER;
        BranchReconnectDecisions   : INTEGER;
        IsolationPortDecisions     : INTEGER;
        ErrorAnalysisTime          : REAL;

BEGIN

    ReconnectRootDecisions   := 0;
    BranchReconnectDecisions := 0;

    IF ChainExecutedWithoutError(DisconnectResponse) THEN

        WriteString(ParamOut, "Disconnect Link chain executed without error.");
        WriteLn(ParamOut);
        WriteString(ParamOut, "Something is wrong in network. ");
        WriteLn(ParamOut);
        WriteLn(ParamOut);

    ELSE

        WITH ErrorReport.FaultAnalysisRecord DO

            IF ReconnectRoot(NodeConnections, NodeStatus, FailedRoot,
                             FailedNodeSet, ReconnectIORequest,
                             ReconnectRootDecisions) THEN

                RepairNetworkMode       := ReconnectLink;
                ReconfigurationStrategy := ReconnectLink;

            ELSIF FailedNodeCount = 1 THEN

                (* This situation should not happen for the IAPSA II
                   experiments.  *)
                WriteString(ParamOut, "Found a node which is unreachable.");
                WriteLn(ParamOut);
                WriteString(ParamOut, "Logic not implace to repair this fault.");
                WriteLn(ParamOut);
                WriteLn(ParamOut);

            ELSE

                (* Reconnect through the branches of the failed tree.  *)
                BranchReconnect(ErrorReport, NodeConnections, BranchNode,
                                BranchPort, NodeStatus, BranchesOfRoot,
                                ReconnectIORequest,
                                BranchReconnectDecisions);

                RepairNetworkMode       := ReconnectBranch;
                ReconfigurationStrategy := ReconnectBranch;

            END;

            IsolationPortDecisions := ReconnectRootDecisions +
                                           BranchReconnectDecisions;

            ErrorAnalysisTime := FixedErrorReportAnalysis +
                                 (FLOAT(AnalysisPortDecisions +
                                  IsolationPortDecisions) *
                                  PortErrorAnalysis)
                                  + ComputeTwoTransactionChain;

            ReconnectIORequest^.Chain^.NetworkToBeExecutedOn := NetworkIDToManage;
            ReconnectIORequest^.Chain^.ChainIdentifier       := 301;
            ReconnectIORequest^.Identifier                   := MyNodeID;
            ReconnectIORequest^.ResponseExpected             := TRUE;

            NEW(RequestForErrorAnalysis);
            RequestForErrorAnalysis^.Message   := FaultAnalysis;
            RequestForErrorAnalysis^.IORequest := ReconnectIORequest;

            NEW(ProcessingForErrorAnalysis);
            ProcessingForErrorAnalysis^.Priority           := NetworkManagerPriority;
            ProcessingForErrorAnalysis^.ProcessingRequired := ErrorAnalysisTime
                                                                 + NetworkResponseComputation;
            ProcessingForErrorAnalysis^.WriteData          := FALSE;
            ProcessingForErrorAnalysis^.ProcessID          := 'Link/Branch Connect Processing';
            ProcessingForErrorAnalysis^.Data               := RequestForErrorAnalysis;

            NOW outport[IOPPort]^.ProcessorRequest <- ProcessingForErrorAnalysis;

        END;

    END;

END DisconnectLinkProcess;

(*************************************************************************)
PROCEDURE ReconnectLinkProcess(ErrorReport       : ErrorRecordType;
                               ReconnectResponse : ChainStatusData);

    VAR TalkToRootOfFailedTreeIORequest : IORequestType;
        RequestToTalkToRoot             : DataPointer;
        ProcessingToTalkToRoot          : ProcessingUnit;

BEGIN
```

```
   IF ChainExecutedWithoutError(ReconnectResponse) THEN

       (* Reconnection was sucessful. Attempt to talk to root of
          failed tree to see if failed link assumption was true.  *)

       RepairNetworkMode := TalkToRootFailedTree;
       WITH ErrorReport.FaultAnalysisRecord DO

           TalkToRootOfTree(FailedRoot, TalkToRootOfFailedTreeIORequest);

       END;

       TalkToRootOfFailedTreeIORequest^.Chain^.NetworkToBeExecutedOn := NetworkIDToManage;
       TalkToRootOfFailedTreeIORequest^.Chain^.ChainIdentifier := 302;
       TalkToRootOfFailedTreeIORequest^.Identifier             := MyNodeID;
       TalkToRootOfFailedTreeIORequest^.ResponseExpected       := TRUE;

       NEW(RequestToTalkToRoot);
       RequestToTalkToRoot^.Message   := NetworkRequest;
       RequestToTalkToRoot^.IORequest := TalkToRootOfFailedTreeIORequest;

       NEW(ProcessingToTalkToRoot);
       ProcessingToTalkToRoot^.Priority           := NetworkManagerPriority;
       ProcessingToTalkToRoot^.ProcessingRequired := ComputeOneTransactionChain
                                                       + NetworkResponseComputation;

       ProcessingToTalkToRoot^.WriteData          := FALSE;
       ProcessingToTalkToRoot^.ProcessID          := 'NW Talk To Root Processing';
       ProcessingToTalkToRoot^.Data               := RequestToTalkToRoot;

       NOW outport[IOPPort]^.ProcessorRequest <- ProcessingToTalkToRoot;

    ELSE

        WriteString(ParamOut, "Reconnect Link encoutered network problems.");
        WriteLn(ParamOut);
        WriteString(ParamOut, "Either NM logic error or network problem. ");
        WriteLn(ParamOut);
        WriteLn(ParamOut);

    END;

END ReconnectLinkProcess;

(*************************************************************************)

PROCEDURE ReconnectBranchProcess(ErrorReport       : ErrorRecordType;
                                 ReconnectResponse : ChainStatusData);

    VAR TalkToRootOfFailedTreeIORequest : IORequestType;
        RequestToTalkToRoot             : DataPointer;
        ProcessingToTalkToRoot          : ProcessingUnit;

BEGIN

    IF ChainExecutedWithoutError(ReconnectResponse) THEN

        (* Branch reconnect was sucessful. Now talk to the root
           of the failed tree.  *)
        RepairNetworkMode := TalkToRootFailedTree;
        WITH ErrorReport.FaultAnalysisRecord DO

            TalkToRootOfTree(FailedRoot, TalkToRootOfFailedTreeIORequest);

        END;

        TalkToRootOfFailedTreeIORequest^.Chain^.NetworkToBeExecutedOn :=
                                                NetworkIDToManage;
        TalkToRootOfFailedTreeIORequest^.Chain^.ChainIdentifier := 302;
        TalkToRootOfFailedTreeIORequest^.Identifier             := MyNodeID;
        TalkToRootOfFailedTreeIORequest^.ResponseExpected       := TRUE;
        NEW(RequestToTalkToRoot);
        RequestToTalkToRoot^.Message   := NetworkRequest;
        RequestToTalkToRoot^.IORequest := TalkToRootOfFailedTreeIORequest;

        NEW(ProcessingToTalkToRoot);
        ProcessingToTalkToRoot^.Priority           := NetworkManagerPriority;
        ProcessingToTalkToRoot^.ProcessingRequired := ComputeOneTransactionChain
                                                        + NetworkResponseComputation;
        ProcessingToTalkToRoot^.WriteData          := FALSE;
        ProcessingToTalkToRoot^.ProcessID          := 'NW Talk To Root Processing';
        ProcessingToTalkToRoot^.Data               := RequestToTalkToRoot;

        NOW outport[IOPPort]^.ProcessorRequest <- ProcessingToTalkToRoot;

    ELSE

        (* Failure here indicates something unexpected happened.
           Either model is not working right, a latent failure
           is present, a second failure has occured, or there is a
           problem in the network manager logic.  *)

        WriteString(ParamOut, "Repair action for branch reconnect unsucessful. ");
        WriteLn(ParamOut);
        WriteString(ParamOut, "SIMULATION ERROR.");
        WriteLn(ParamOut);

    END;

END ReconnectBranchProcess;

(***************************************************************************)

PROCEDURE TalkToRootProcess(Response      : ChainStatusData;
                            ReconnectNode : INTEGER;
                            ReconnectPort : PortNameType);

    VAR RequestNewState    : DataPointer;
        ProcessingNewState : ProcessingUnit;


BEGIN

    IF ChainExecutedWithoutError(Response) THEN

        (* Successfully talked to root of failed tree so link has been
           repaired.  If repair strategy was branch reconnect, then
           update Node Status to reflect the new orientation of the nodes
           in the failed node set.  *)

        IF ReconfigurationStrategy = ReconnectBranch THEN

            UpdateNodeStatusBranchReconnect(ReconnectNode, ReconnectPort,
                    ErrorReport.FaultAnalysisRecord.FailedRoot, NodeStatus);

        END;

        WITH DiagnosticStatus.TalkerRequest^.Chain^ DO

            NumberOfTransactions := QSize(DiagnosticStatus.TalkerRequest^.Chain^.
                                        TransactionQueue);
            NetworkToBeExecutedOn := NetworkIDToManage;

        END;

        NEW(NetworkState);
        WITH NetworkState^ DO

            NetworkID    := NetworkIDToManage;
            State        := InService;
            MonitorChain := DiagnosticStatus.TalkerRequest^.Chain;

        END;
```

```
       NEW(RequestNewState);
       RequestNewState^.Message   := NewState;
       RequestNewState^.StateData := NetworkState;

       NEW(ProcessingNewState);
       ProcessingNewState^.Priority           := NetworkManagerPriority;
       ProcessingNewState^.ProcessingRequired := ChangeNetworkStatus
                                                   + NetworkResponseComputation;
       ProcessingNewState^.WriteData          := FALSE;
       ProcessingNewState^.ProcessID          := 'New State Processing';
       ProcessingNewState^.Data               := RequestNewState;

       NOW outport[IOPPort]^.ProcessorRequest <- ProcessingNewState;

    ELSE

        (* Could not talk to root of failed tree.  Link failure
           assumption wrong. Failure is a node failure. *)
        WriteLn(ParamOut);
        WriteString(ParamOut, "TIME TO IMPLEMENT NODE FAILURE ROUTINE");
        WriteLn(ParamOut);  .
        WriteLn(ParamOut);

    END;

END TalkToRootProcess;

(**************************************************************************)

PROCEDURE RootNodeProcess(RootNode                : INTEGER;
                          VAR GoodRootNode        : BOOLEAN;
                          SpawningQueue           : SpawningQueueType;
                          VAR GrowMode            : GrowNetworkModeType;
                          VAR DiagnosticStatus    : DiagnosticRecordType;
                          GrowToRootNodeResponse  : ChainStatusData;
                          VAR NodeStatus          : NodeStatusArray);

    VAR SpawningElement : SpawningNodeType;

BEGIN

    IF NOT GoodRootNode THEN

        IF ChainExecutedWithoutError(GrowToRootNodeResponse) THEN

            GoodRootNode := TRUE;

            (* Put Root Node on the Spawning Queue. *)
            NEW(SpawningElement);
            SpawningElement^.Node := RootNode;

            INSERT SpawningElement LAST IN SpawningQueue;

            NodeStatus[FindNodeNumber(RootNode)].Status := Active;

            (* A good root node has been found, change modes.  *)
            GrowMode                            := DiagnosticCheck;
            DiagnosticStatus.Diagnostic         := LinkEnable;
            DiagnosticStatus.TestNode           := RootNode;
            DiagnosticStatus.TestPort           := 1;
            DiagnosticStatus.PortDiagnosticsRun := FALSE;

            RunDiagnostic(DiagnosticStatus, NodeStatus, NodeConnections );

        ELSE

            WriteString(ParamOut, "First Root Link tried ");
            WriteString(ParamOut, "was bad.");
            WriteLn(ParamOut);

        END;
    END;

END RootNodeProcess;

(**************************************************************************)

PROCEDURE AddNodeProcess(VAR GrowStatus           : NetworkGrowthProgressType;
                         Response                 : ChainStatusData;
                         VAR StatusOfDiagnostics  : DiagnosticRecordType;
                         VAR PreviousAddFailed    : BOOLEAN;
                         VAR NodeStatus           : NodeStatusArray;
                         NodeConnections          : NodeArrayType);

    VAR DisconnectIORequest       : IORequestType;
        RequestDisconnectNode     : DataPointer;
        ProcessingDisconnectNode  : ProcessingUnit;

BEGIN

    IF ChainExecutedWithoutError(Response) THEN

        (* Update the status of the spawning node and the target
           node as a result of adding the new node.  *)

        NodeStatus[FindNodeNumber(GrowStatus.TargetNode)].Status := Active;

        GrowStatus.Mode                         := DiagnosticCheck;
        StatusOfDiagnostics.Diagnostic          := LinkEnable;
        StatusOfDiagnostics.TestNode            := GrowStatus.TargetNode;
        StatusOfDiagnostics.TestPort            := 1;
        StatusOfDiagnostics.PortDiagnosticsRun  := FALSE;

        RunDiagnostic(StatusOfDiagnostics, NodeStatus, NodeConnections);


    ELSIF PreviousAddFailed THEN

        PreviousAddFailed := FALSE;
        NextNodeToAdd(GrowStatus, NodeConnections, NodeStatus, DIUList);

    ELSE

        (* The previous try to connect to a node was unsucessful.
           Assume that the link is broken. Update link statuses to reflect.  *)
        PreviousAddFailed := TRUE;

        (* Update Status of the two nodes that were unsucceussfully
           added to the network.  *)
        NodeStatus[FindNodeNumber(GrowStatus.SpawningNode)].PortStatus[
            GrowStatus.SpawningPort].Status := Failed;

        NodeStatus[FindNodeNumber(GrowStatus.TargetNode)].PortStatus[
            GrowStatus.TargetPort].Status := Failed;

        DeleteNodeFromNetwork(GrowStatus.SpawningNode, GrowStatus.SpawningPort,
            GrowStatus.TargetNode, GrowStatus.TargetPort, NodeStatus, DisconnectIORequest);

        DisconnectIORequest^.Chain^.NetworkToBeExecutedOn := NetworkIDToManage;
        DisconnectIORequest^.Identifier       := MyNodeID;
        DisconnectIORequest^.ResponseExpected := TRUE;

        (* Disconnect failed link.  *)
        NEW(RequestDisconnectNode);
        WITH RequestDisconnectNode^ DO

            Message   := NetworkRequest;
            IORequest := DisconnectIORequest;

        END;

        NEW(ProcessingDisconnectNode);
        WITH ProcessingDisconnectNode^ DO

            Priority           := NetworkManagerPriority;
            ProcessingRequired := ComputeTwoTransactionChain
                                     + NetworkResponseComputation;
            WriteData          := FALSE;
            ProcessID          := 'NW Disconnect Node Processing';
            Data               := RequestDisconnectNode;

        END;

        NOW outport[IOPPort]^.ProcessorRequest <- ProcessingDisconnectNode;

    END;

END AddNodeProcess;

(***************************************************************************)
PROCEDURE AddDIUProcess(VAR GrowStatus : NetworkGrowthProgressType;
                        Response       : ChainStatusData;
                        VAR NodeStatus : NodeStatusArray);

    VAR DIUElement      : DIUSpawningRecord;
        AddDIUIORequest : IORequestType;
        GPCElement      : GPCSpawningRecord;
        AddGPCIORequest : IORequestType;
        AddDIURequest   : DataPointer;
        ProcessAddDIU   : ProcessingUnit;
        AddGPCRequest   : DataPointer;
        ProcessAddGPC   : ProcessingUnit;

BEGIN

    IF ChainExecutedWithoutError(Response) THEN

        DIUElement := FirstQ(DIUList);

        IF DIUElement <> NIL THEN

            REMOVE DIUElement FROM DIUList;

            (* Update Status of node that is conected to the DIU that
               is being added to the network.  *)
            WITH DIUElement^ DO

                NodeStatus[FindNodeNumber(SpawningNode)].
                    PortStatus[OutboardPort].Status := Active;
                NodeStatus[FindNodeNumber(SpawningNode)].
                    PortStatus[OutboardPort].Direction := Outboard;

                AddDIUToNetwork(SpawningNode, OutboardPort, NodeStatus, AddDIUIORequest);

                (* Data Collection.  *)
                GrowStatus.DIUAddress := NodeConnections[FindNodeNumber(SpawningNode)].
                                 PortArray[OutboardPort].DIUAddress;

            END;

            DISPOSE(DIUElement);
            AddDIUIORequest^.Chain^.NetworkToBeExecutedOn := NetworkIDToManage;
            AddDIUIORequest^.Identifier := MyNodeID;
            AddDIUIORequest^.ResponseExpected := TRUE;

            NEW(AddDIURequest);
            AddDIURequest^.Message   := NetworkRequest;
            AddDIURequest^.IORequest := AddDIUIORequest;

            NEW(ProcessAddDIU);
            ProcessAddDIU^.Priority           := NetworkManagerPriority;
            ProcessAddDIU^.ProcessingRequired := ComputeOneTransactionChain
                                                    + NetworkResponseComputation;
            ProcessAddDIU^.WriteData          := FALSE;
            ProcessAddDIU^.ProcessID          := 'NW Add DIU Processing';
            ProcessAddDIU^.Data               := AddDIURequest;

            NOW outport[IOPPort]^.ProcessorRequest <- ProcessAddDIU;

        ELSE

            (* All DIUs have been added to the network.  Now add the GPC
               connections.  Set mode to add GPCs. *)
            GrowStatus.Mode := AddGPCS;

            GPCElement := FirstQ(GPCList);
            REMOVE GPCElement FROM GPCList;

            (* Update status of the node connected to the spare root
               link connection to be added to the network.  *)
            WITH GPCElement^ DO

                NodeStatus[FindNodeNumber(SpawningNode)].
                    PortStatus[InboardPort].Status := Active;
                NodeStatus[FindNodeNumber(SpawningNode)].
                    PortStatus[InboardPort].Direction := Inboard;

                AddGPCToNetwork(SpawningNode, InboardPort, NodeStatus, AddGPCIORequest);

                GrowStatus.SpareRootNodeAddress := SpawningNode;

            END;

            DISPOSE(GPCElement);
            AddGPCIORequest^.Chain^.NetworkToBeExecutedOn := NetworkIDToManage;
            AddGPCIORequest^.Identifier := MyNodeID;
            AddGPCIORequest^.ResponseExpected := TRUE;

            NEW(AddGPCRequest);
            AddGPCRequest^.Message   := NetworkRequest;
            AddGPCRequest^.IORequest := AddGPCIORequest;

            NEW(ProcessAddGPC);
            ProcessAddGPC^.Priority           := NetworkManagerPriority;
            ProcessAddGPC^.ProcessingRequired := ComputeOneTransactionChain
                                                    + NetworkResponseComputation;
            ProcessAddGPC^.WriteData          := FALSE;
            ProcessAddGPC^.ProcessID          := 'NW Add GPC Processing';
            ProcessAddGPC^.Data               := AddGPCRequest;

            NOW outport[IOPPort]^.ProcessorRequest <- ProcessAddGPC;

        END;

    ELSE

        WriteString(ParamOut, "Problems encountered enabling a DIU.");
        WriteLn(ParamOut);

    END;

END AddDIUProcess;

(*************************************************************************)
PROCEDURE AddGPCProcess(VAR GrowStatus : NetworkGrowthProgressType;
                        Response       : ChainStatusData;
                        VAR NodeStatus : NodeStatusArray;
                        MonitorChain   : ChainType);

    VAR GPCElement      : GPCSpawningRecord;
        AddGPCIORequest : IORequestType;
        AddGPCRequest   : DataPointer;
        ProcessAddGPC   : ProcessingUnit;
        NewStateRequest : DataPointer;
        ProcessNewState : ProcessingUnit;
```

```
BEGIN

    IF ChainExecutedWithoutError(Response) THEN

        GPCElement := FirstQ(GPCList);

        IF GPCElement <> NIL THEN

            REMOVE GPCElement FROM GPCList;

            (* Update status of the node connected to the spare root
               link connection to be added to the network.  *)
            WITH GPCElement^ DO

                NodeStatus[FindNodeNumber(SpawningNode)].
                    PortStatus[InboardPort].Status := Active;
                NodeStatus[FindNodeNumber(SpawningNode)].
                    PortStatus[InboardPort].Direction := Inboard;

                AddGPCToNetwork(SpawningNode, InboardPort, NodeStatus, AddGPCIORequest);

                GrowStatus.SpareRootNodeAddress := SpawningNode;

            END;

            DISPOSE(GPCElement);
            AddGPCIORequest^.Chain^.NetworkToBeExecutedOn := NetworkIDToManage;
            AddGPCIORequest^.Identifier := MyNodeID;
            AddGPCIORequest^.ResponseExpected := TRUE;

            NEW(AddGPCRequest);
            AddGPCRequest^.Message   := NetworkRequest;
            AddGPCRequest^.IORequest := AddGPCIORequest;

            NEW(ProcessAddGPC);
            ProcessAddGPC^.Priority           := NetworkManagerPriority;
            ProcessAddGPC^.ProcessingRequired := ComputeOneTransactionChain
                                                   + NetworkResponseComputation;

            ProcessAddGPC^.WriteData          := FALSE;
            ProcessAddGPC^.ProcessID          := 'NW Add GPC Processing';
            ProcessAddGPC^.Data               := AddGPCRequest;

            NOW outport[IOPPort]^.ProcessorRequest <- ProcessAddGPC;

        ELSE

            (* All GPCs have been added to the network.  Notify the I/O Service
               that my network is now operational.  *)
            NEW(NetworkState);
            NetworkState^.NetworkID    := NetworkIDToManage;
            NetworkState^.State        := InService;
            NetworkState^.MonitorChain := MonitorChain;

            NEW(NewStateRequest);
            NewStateRequest^.Message   := NewState;
            NewStateRequest^.StateData := NetworkState;

            NEW(ProcessNewState);
            ProcessNewState^.Priority           := NetworkManagerPriority;
            ProcessNewState^.ProcessingRequired := ChangeNetworkStatus
                                                     + NetworkResponseComputation;

            ProcessNewState^.WriteData          := FALSE;
            ProcessNewState^.ProcessID          := 'Change Network State';
            ProcessNewState^.Data               := NewStateRequest;

            NOW outport[IOPPort]^.ProcessorRequest <- ProcessNewState;

        END;

    ELSE

        WriteString(ParamOut, "Tried to add a GPC and failed.");
        WriteLn(ParamOut);

    END;

END AddGPCProcess;

(*************************************************************************)
(* This procedure processes a network response and determines what the
   next network manager action.  *)
PROCEDURE ProcessNetworkResponse(Response                : ChainStatusData;
                                 Mode                    : NetworkManagerJobType;
                                 VAR StatusOfGrowth      : NetworkGrowthProgressType;
                                 VAR StatusOfDiagnostics : DiagnosticRecordType);

    VAR SpawningNode : SpawningNodeType;

BEGIN

    IF Mode = GROWNetwork THEN

        CASE StatusOfGrowth.Mode OF

            GrowToRootNode:

                RootNodeProcess(StatusOfGrowth.RootNodeAddress, GoodRootNodeFound,
                    SpawningQueue, StatusOfGrowth.Mode, StatusOfDiagnostics,
                    Response, NodeStatus);

            | AddRemainingNodes:

                AddNodeProcess(StatusOfGrowth, Response, StatusOfDiagnostics,
                    PreviousChainFailed, NodeStatus, NodeConnections);

            | AddDIUS:

                AddDIUProcess(StatusOfGrowth, Response, NodeStatus);

            | AddGPCS:

                AddGPCProcess(StatusOfGrowth, Response, NodeStatus,
                        StatusOfDiagnostics.TalkerRequest^.Chain);

            | DiagnosticCheck:

                IF StatusOfDiagnostics.Diagnostic <> DiagnosticsComplete THEN

                    IF StatusOfDiagnostics.Diagnostic = DisabledTransmit THEN

                        StatusOfDiagnostics.LinkEnableSucessful :=
                                    ChainExecutedWithoutError(Response);

                    END;

                    RunDiagnostic(StatusOfDiagnostics, NodeStatus,
                            NodeConnections );

                ELSIF QSize(SpawningQueue) <> 0 THEN

                    REPORT "%d"  StatusOfDiagnostics.TestNode TAGGED "Node added.";

                    IF NumberOfActiveNodes = 0 THEN

                        (* This indicates that the root node is the only
                           node in the network and the other nodes have
                           yet to be added.  *)
                        REMOVE FIRST SpawningNode FROM SpawningQueue;
                        StatusOfGrowth.SpawningNode := SpawningNode^.Node;
                        StatusOfGrowth.SpawningPort := 1;
                        DISPOSE(SpawningNode);

                    END;
                    NumberOfActiveNodes := NumberOfActiveNodes + 1;
                    StatusOfGrowth.Mode := AddRemainingNodes;

                    NextNodeToAdd(StatusOfGrowth, NodeConnections, NodeStatus,
                        DIUList);

                END;

        END;

    ELSE

        CASE RepairNetworkMode OF

            DisconnectLink:

                DisconnectLinkProcess(ErrorReport, Response, NodeStatus,
                    NodeConnections);

            | ReconnectLink:

                ReconnectLinkProcess(ErrorReport, Response);

            | ReconnectBranch:

                ReconnectBranchProcess(ErrorReport, Response);

            | TalkToRootFailedTree:

                TalkToRootProcess(Response, BranchNode, BranchPort);

            ELSE

                WriteString(ParamOut, "DISCONNECT LINK IS THE ONLY MODE SUPPORTED");
                WriteLn(ParamOut);

        END;

    END;

    ReleaseChainResponseMemory(Response);

END ProcessNetworkResponse;

(*************************************************************************)

PROCEDURE ReleaseManagerRequest(VAR Request : IORequestType);

    VAR Transaction      : TransactionType;
        ElementCounter   : INTEGER;
        NumberOfElements : INTEGER;

BEGIN

    IF Request <> NIL THEN

        WITH Request^.Chain^ DO

            NumberOfElements := QSize(TransactionQueue);
            FOR ElementCounter := 1 TO NumberOfElements DO

                Transaction := QRemove(TransactionQueue, TRUE);
                DISPOSE(Transaction^.OutputFrame);
                DISPOSE(Transaction);

            END;

            QDispose(TransactionQueue, SIZE(TransactionQueue));

        END;

        DISPOSE(Request^.Chain);
```

```
            DISPOSE(Request);

        END;

    END ReleaseManagerRequest;

    (**************************************************************************)
BEGIN

    IOPPort := GetOutPort(IOPIdentifier);

    ReadNodeInterConnections(NetworkIDToManage, NodeConnections);
    InitializeStatusVariables(NodeConnections, NodeStatus, ChannelStatus);
    NodesInNetwork := NodesInThisSimulation(NodeConnections);
    WriteLn(ParamOut);
    WriteLn(ParamOut);

    DiagnosticStatus.TalkerRequest := MakeMonitorRequest(NodeConnections);

    WITH DiagnosticStatus.TalkerRequest^ DO

        Chain^.NetworkToBeExecutedOn := NetworkIDToManage;
        Identifier                   := MyNodeID;
        ResponseExpected             := TRUE;

    END;

    NEW(PowerupRequest);
    PowerupRequest^.Message := Powerup;

    NEW(PowerupProcessing);
    PowerupProcessing^.Priority           := 2;
    PowerupProcessing^.ProcessingRequired := FLOAT(NodesInNetwork) * NodePowerupInitilizeTime;
    PowerupProcessing^.WriteData          := FALSE;
    PowerupProcessing^.ProcessID          := 'NM Power up';
    PowerupProcessing^.Data               := PowerupRequest;

    NOW outport[IOPPort]^.ProcessorRequest <- PowerupProcessing;

    LOOP

        WAITUNTIL (ProcessorResponse)

            ProcessorResponse:

                ProcessingResponse := ActivePort^.ProcessorResponse;
                ProcessResponse    := ProcessingResponse^.Data;

                IF ProcessResponse^.Message = Powerup THEN

                    EXIT;

                ELSE

                    WriteString(ParamOut, "Problem during NM power up processing");
                    WriteLn(ParamOut);

                END;


        END;

    END;

    DISPOSE(ProcessingResponse);
    DISPOSE(ProcessResponse);

    PreviousChainFailed := FALSE;
    NumberOfActiveNodes := 0;

    (* Initailize the Node Status variable to reflect the networks
   intial state.  *)
FOR NodeIndex := 1 TO NodesInNetwork DO

    WITH NodeStatus[NodeIndex] DO

        Address  := NodeConnections[NodeIndex].NodeAddress;
        Status   := Active;

        FOR PortIndex := 1 TO NumberOfPortsPerNode DO

            IF InitialNodeConfiguration[NodeIndex][PortIndex] THEN

                PortStatus[PortIndex].Status := Active;

                IF InitialOrientation[NodeIndex][PortIndex] THEN

                    PortStatus[PortIndex].Direction := Inboard;

                ELSE

                    PortStatus[PortIndex].Direction := Outboard;

                END;

            ELSE

                PortStatus[PortIndex].Status := Idle;

            END;

        END;

    END;

END;

LOOP

    WAITUNTIL EVENT

        ServiceRequest:

            RequestForService := ActivePort^.ServiceRequest;

            CASE RequestForService^.ServiceRequest OF

                GrowNetwork:

                    InitializeStatusVariables(NodeConnections, NodeStatus,
                            ChannelStatus);
                    GoodRootNodeFound          := FALSE;
                    NetworkManagerStatus       := GROWNetwork;
                    NetworkGrowthProgress.Mode := GrowToRootNode;

                    StartNetworkGrowth(RequestForService^.ActiveRootLink,
                        NetworkGrowthProgress, NodeConnections,NodeStatus,
                        NodesInNetwork);

                | RepairFault:

                    (* This reconfiguration strategy attempts
                       one shot repair of the network.  *)
                    NetworkManagerStatus   := RepairNetwork;
                    AnalysisPortDecisions  := 0;

                    NetworkRepair(RequestForService^.MonitorChainResponse^.Response,
                        NodeStatus, NodeConnections, ErrorReport,
                        RepairNetworkMode);

                | SwitchRootLink:

                    WITH RequestForService^ DO
                (* Get the node which is connected to the new
                   root link and the node connected to the failed
                   root link.  Deternine which port ot RootNode
                   which is connected to the NewRoot.  *)
                RootNode    := FindNodeNumber(NewRootNode);
                FailedNode  := FindNodeNumber(FailedRootNode);

                CurrentPort := 1;
                WHILE NodeConnections[RootNode].PortArray[
                    CurrentPort].AdjacentElement <> GPC DO

                    CurrentPort := CurrentPort + 1;

                END;

            END;

            (* Update node status to reflect the root link switch.  *)
            UpdateNodeStatusBranchReconnect(NodeStatus[RootNode].Address,
                CurrentPort, NodeStatus[FailedNode].Address,
                NodeStatus);

    END;

    DISPOSE(RequestForService);

| IONetworkResponse:

    NetworkResponse := ActivePort^.IONetworkResponse;

    ProcessNetworkResponse(NetworkResponse, NetworkManagerStatus,
        NetworkGrowthProgress, DiagnosticStatus);

    (* Get rid of memory for I/O request corresponding to
       this response. *)
    ReleaseManagerRequest(LastNetworkRequest);

| ProcessorResponse:

    ProcessingResponse := ActivePort^.ProcessorResponse;
    ProcessResponse    := ProcessingResponse^.Data;

    CASE ProcessResponse^.Message OF

       GrowRequest, NetworkRequest, FaultAnalysis:

           WITH DataCollectionRecord DO

               SimulationTime := clock;
               Frequency      := NonCyclic;

               IF NetworkManagerStatus = GROWNetwork THEN

                   NonCyclicData.N_Variation := RegrowAction;

                   CASE NetworkGrowthProgress.Mode OF

                     GrowToRootNode:

                         EventID                 := 32;
                         NonCyclicData.Node_DIU_ID := NetworkGrowthProgress.RootNodeAddress;

                         WriteDataElementType(DataCollectionRecord);

                   | AddRemainingNodes:

                         EventID                 := 33;
                         NonCyclicData.Node_DIU_ID := NetworkGrowthProgress.TargetNode;

                         WriteDataElementType(DataCollectionRecord);
```

```
     | AddDIUS:

         EventID                   := 34;
         NonCyclicData.Node_DIU_ID := NetworkGrowthProgress.DIUAddress;

         WriteDataElementType(DataCollectionRecord);

     | AddGPCS:

         EventID                   := 35;
         NonCyclicData.Node_DIU_ID := NetworkGrowthProgress.SpareRootNodeAddress;

         WriteDataElementType(DataCollectionRecord);

     ELSE

     END;

ELSIF NetworkManagerStatus = RepairNetwork THEN

    CASE RepairNetworkMode OF

     DisconnectLink:

         EventID      := 36;
         NonCyclicData.N_Variation  := ErrorEval;
         NonCyclicData.NumberFailed := ErrorReport.FaultAnalysisRecord.FailedNodeCount;
         ErrorIndex   := 1;

         FOR NodeIndex := 1 TO NodesInNetwork DO

             IF NodeSetRange(NodeIndex) IN
                 ErrorReport.FaultAnalysisRecord.FailedNodeSet THEN

                 NonCyclicData.Node_ID[ErrorIndex] := NodeConnections[NodeIndex].NodeAddress;
                 INC(ErrorIndex);

             END;

         END;

         WriteDataElementType(DataCollectionRecord);

         EventID      := 37;
         NonCyclicData.N_Variation := OneShotAction;
         Transaction := FirstQ(ProcessResponse^.IORequest^.Chain^.TransactionQueue);
         TargetNode  := Transaction^.OutputFrame^.Address;
         Transaction := QSucc(Transaction, ProcessResponse^.IORequest^.Chain^.TransactionQueue
         SourceNode  := 100 *Transaction^.OutputFrame^.Address;
         NonCyclicData.Link_Node_ID := SourceNode + TargetNode;

     | ReconnectLink:

         EventID      := 38;
         NonCyclicData.N_Variation := OneShotAction;
         Transaction  := FirstQ(ProcessResponse^.IORequest^.Chain^.TransactionQueue);
         SourceNode   := 100 *Transaction^.OutputFrame^.Address;
         Transaction  := QSucc(Transaction, ProcessResponse^.IORequest^.Chain^.TransactionQueue
         TargetNode   := Transaction^.OutputFrame^.Address;
         NonCyclicData.Link_Node_ID := SourceNode + TargetNode;

     | ReconnectBranch:

         EventID      := 39;
         NonCyclicData.N_Variation := OneShotAction;
         Transaction  := FirstQ(ProcessResponse^.IORequest^.Chain^.TransactionQueue);
         SourceNode   := 100 *Transaction^.OutputFrame^.Address;
         Transaction  := QSucc(Transaction, ProcessResponse^.IORequest^.Chain^.TransactionQueue
         TargetNode   := Transaction^.OutputFrame^.Address;
         NonCyclicData.Link_Node_ID := SourceNode + TargetNode;

     | TalkToRootFailedTree:
```

```
                               EventID     := 40;
                               NonCyclicData.N_Variation := OneShotAction;
                               Transaction := FirstQ(ProcessResponse^.IORequest^.Chain^.TransactionQueue);
                               TargetNode  := Transaction^.OutputFrame^.Address;
                               NonCyclicData.Link_Node_ID := TargetNode;

                          END;

                          WriteDataElementType(DataCollectionRecord);

                   END;

               END;

               NOW outport[1]^.IONetworkRequest <- ProcessResponse^.IORequest;
               LastNetworkRequest := ProcessResponse^.IORequest;

           | NewState :

               NOW outport[1]^.NewNetworkState <- ProcessResponse^.StateData;

        END;

        DISPOSE(ProcessingResponse);
        DISPOSE(ProcessResponse);

      | Reset:

          (* Reset manager's view of the network.  *)
          (* Reinitailize the Node Status variable to reflect
             the network's intial state.  *)
          PreviousChainFailed := FALSE;
          NumberOfActiveNodes := 0;
          InitializeStatusVariables(NodeConnections, NodeStatus, ChannelStatus);

          FOR NodeIndex := 1 TO NodesInNetwork DO

              WITH NodeStatus[NodeIndex] DO

                  Status  := Active;

                  FOR PortIndex := 1 TO NumberOfPortsPerNode DO

                      IF InitialNodeConfiguration[NodeIndex][PortIndex] THEN

                          PortStatus[PortIndex].Status := Active;

                          IF InitialOrientation[NodeIndex][PortIndex] THEN

                              PortStatus[PortIndex].Direction := Inboard;

                          ELSE

                              PortStatus[PortIndex].Direction := Outboard;

                          END;

                      ELSE

                          PortStatus[PortIndex].Status := Idle;

                      END;

                  END;

              END;

        END;

END;
```

```
    END;

END NetManger.
```

BLDCHAINS

```
DEFINITION MODULE BldChains;

    FROM IOService IMPORT IORequestType;

    FROM BusMessag IMPORT IOActivityChoice;

    EXPORT QUALIFIED ApplicationType, BuildRequest;

    TYPE ApplicationType = (FlightControl, EngineControl);

    (*************************************************************************)

    PROCEDURE BuildRequest (Application : ApplicationType;
                            Rate        : INTEGER;
                            IOActivity  : IOActivityChoice) : IORequestType;

    (*************************************************************************)

END BldChains.
```

BLDCHAINS

```
IMPLEMENTATION MODULE BldChains;

    FROM IOService IMPORT IOActivityType, IORequestType;

    FROM IOS IMPORT ChainType, TransactionType;

    FROM BusMessag IMPORT BusMessageType, MessageType, IOActivityChoice;

    FROM QueueM IMPORT InitQ, QInsert, QSize;

    FROM Storage IMPORT ALLOCATE;

    CONST FlightControlRequestTimeout100          = 0.002;
          FlightControlRequestTimeout50           = 0.002;
          FlightControlRequestTimeout25           = 0.001;

          EngineControlRequestTimeout100          = 0.001;
          EngineControlRequestTimeout50           = 0.001;
          EngineControlRequestTimeout25           = 0.001;

          EngineControlIdentifier100              = 100;
          EngineControlIdentifier50               = 50;
          EngineControlIdentifier25               = 25;

          FlightControlIdentifier100              = 100;
          FlightControlIdentifier50               = 50;
          FlightControlIdentifier25               = 25;

          InletRequestPriority                    = 10;
          NozzleRequestPriority                   = 9;
          EngineRequestPriority                   = 8;

          FastRequestPriority                     = 10;
          MiddleRequestPriority                   = 9;
          SlowRequestPriority                     = 8;

          InletDIUAddress1                        = 81;
          NozzleDIUAddress1                       = 83;
          EngineDIUAddress1                       = 82;
          InletDIUAddress2                        = 111;
          NozzleDIUAddress2                       = 113;
          EngineDIUAddress2                       = 112;

          Sensor2DIUAddress1                      = 91;
          Cockpit2DIUAddress1                     = 92;
          CanardRightDIUAddress1                  = 93;
          LeadingEdgeRightDIUAddress1             = 94;
          OutboardFlaperonRightDIUAddress1        = 95;
          InboardFlaperonRightDIUAddress1         = 96;
          TrailingEdgeRightDIUAddress1            = 97;
          RudderRightDIUAddress1                  = 98;
          RudderLeftDIUAddress1                   = 99;
          TrailingEdgeLeftDIUAddress1             = 100;
          InboardFlaperonLeftDIUAddress1          = 101;
          OutboardFlaperonLeftDIUAddress1         = 102;
          NoseDIUAddress1                         = 103;
          CanardLeftDIUAddress1                   = 104;
          Cockpit1DIUAddress1                     = 105;
          Sensor1DIUAddress1                      = 106;

          Sensor2DIUAddress2                      = 111;
          Cockpit2DIUAddress2                     = 112;
          CanardRightDIUAddress2                  = 113;
          LeadingEdgeRightDIUAddress2             = 114;
          OutboardFlaperonRightDIUAddress2        = 115;
          InboardFlaperonRightDIUAddress2         = 116;
          TrailingEdgeRightDIUAddress2            = 117;
          RudderRightDIUAddress2                  = 118;
          RudderLeftDIUAddress2                   = 119;
          TrailingEdgeLeftDIUAddress2             = 120;
          InboardFlaperonLeftDIUAddress2          = 121;
          OutboardFlaperonLeftDIUAddress2         = 122;
```

```
        NoseDIUAddress2                          = 123;
        CanardLeftDIUAddress2                    = 124;
        Cockpit1DIUAddress2                      = 125;
        Sensor1DIUAddress2                       = 126;

        (* This section controls the transaction timeouts of for transactions
           in the flight control and engine application.  *)
        (* This section sets transaction timeouts for the engine application.
           Since there is only one transaction in each of the chains,
           timeouts will be set at maximum execution time plus
           time for response bits on the bus plus 10% of the previos sum.  *)
        InletTransactionTimeout                  = 0.000305; (* max 0.000277 *)
        NozzleTransactionTimeout                 = 0.000229; (* max 0.000208 *)
        EngineTransactionTimeout                 = 0.000637; (* max 0.000579 *)

        (* This sections sets the transaction timeouts for the flight
           control application.  The timeouts are set such that if a
           transaction times out, the succeding transaction in the chain
           for the network with the failed transacton, will be
           transmitted from the IOS approximately the same time as the
           corresponding transaction on the unfailed network.  *)
        Sensor2FastTransactionTimeout            = 0.000208;
        Sensor2MiddleTransactionTimeout          = 0.000116;
        Cockpit2TransactionTimeout               = 0.000139;
        CanardRightTransactionTimeout            = 0.000116;
        LeadingEdgeRightTransactionTimeout       = 0.000208;
        OutboardFlaperonRightTransactionTimeout  = 0.000139;
        InboardFlaperonRightTransactionTimeout   = 0.000162;
        TrailingEdgeRightTransactionTimeout      = 0.000162;
        RudderRightTransactionTimeout            = 0.000116;
        RudderLeftTransactionTimeout             = 0.000116;
        TrailingEdgeLeftTransactionTimeout       = 0.000139;
        InboardFlaperonLeftTransactionTimeout    = 0.000162;
        OutboardFlaperonLeftTransactionTimeout   = 0.000139;
        NoseTransactionTimeout                   = 0.000116;
        CanardLeftTransactionTimeout             = 0.000116;
        Cockpit1MiddleTransactionTimeout         = 0.000139;
        Cockpit1SlowTransactionTimeout           = 0.000093;
        Sensor1FastTransactionTimeout            = 0.000208;
        Sensor1MiddleTransactionTimeout          = 0.000116;
        Sensor1SlowTransactionTimeout            = 0.000093;

(*****************************************************************************)

PROCEDURE FltCon100Request(IOActivity : IOActivityChoice) :IORequestType;

    VAR FlightControl100Request   : IORequestType;
        Network1Chain             : ChainType;
        Network2Chain             : ChainType;
        Transaction               : TransactionType;
        Command                   : BusMessageType;

BEGIN

    (* This process controls the 100Hz I/O request. I/O Requests will
       be generated at a 100Hz rate, i.e every 10ms. After the I/O
       response is received, this process will delay for a time interval
       Intended to represent the time it takes to generate the next output.  *)

    NEW(FlightControl100Request);
    NEW(Network1Chain);
    NEW(Network2Chain);

    Network1Chain^.TransactionQueue := InitQ("TransactionQueue", FALSE, 0);
    Network2Chain^.TransactionQueue := InitQ("TransactionQueue", FALSE, 0);
    Network1Chain^.ChainIdentifier  := 1;
    Network2Chain^.ChainIdentifier  := 1;

    NEW(Transaction);
    NEW(Command);
    WITH Command^ DO
    Address := Sensor1DIUAddress1;
    Message := DIUInput;

    WITH DIUCommand DO

        Activity      := IOActivity;
        CommandNumber := 1;

    END;

END;

WITH Transaction^ DO

    Identifier   := 100;
    TimeOutValue := Sensor1FastTransactionTimeout;
    OutputFrame  := Command;

END;

QInsert(Transaction, Network1Chain^.TransactionQueue, FALSE);

NEW(Transaction);
NEW(Command);

WITH Command^ DO

    Address := Sensor2DIUAddress1;
    Message := DIUInput;

    WITH DIUCommand DO

        Activity      := IOActivity;
        CommandNumber := 1;

    END;

END;

WITH Transaction^ DO

    Identifier   := 101;
    TimeOutValue := Sensor2FastTransactionTimeout;
    OutputFrame  := Command;

END;

QInsert(Transaction, Network1Chain^.TransactionQueue, FALSE);

NEW(Transaction);
NEW(Command);

WITH Command^ DO

    Address := OutboardFlaperonLeftDIUAddress1;
    Message := DIUInput;

    WITH DIUCommand DO

        Activity      := IOActivity;
        CommandNumber := 1;

    END;

END;

WITH Transaction^ DO

    Identifier   := 102;
    TimeOutValue := OutboardFlaperonLeftTransactionTimeout;
    OutputFrame  := Command;
```

```
END;

QInsert(Transaction, Network1Chain^.TransactionQueue, FALSE);

NEW(Transaction);
NEW(Command);

WITH Command^ DO

    Address := OutboardFlaperonRightDIUAddress1;
    Message := DIUInput;

    WITH DIUCommand DO

        Activity      := IOActivity;
        CommandNumber := 1;

    END;

END;

WITH Transaction^ DO

    Identifier   := 103;
    TimeOutValue := OutboardFlaperonRightTransactionTimeout;
    OutputFrame  := Command;

END;

QInsert(Transaction, Network1Chain^.TransactionQueue, FALSE);

NEW(Transaction);
NEW(Command);

WITH Command^ DO

    Address := InboardFlaperonLeftDIUAddress1;
    Message := DIUInput;

    WITH DIUCommand DO

        Activity      := IOActivity;
        CommandNumber := 1;

    END;

END;

WITH Transaction^ DO

    Identifier   := 104;
    TimeOutValue := InboardFlaperonLeftTransactionTimeout;
    OutputFrame  := Command;

END;

QInsert(Transaction, Network1Chain^.TransactionQueue, FALSE);

NEW(Transaction);
NEW(Command);

WITH Command^ DO

    Address := InboardFlaperonRightDIUAddress1;
    Message := DIUInput;

    WITH DIUCommand DO

        Activity      := IOActivity;
        CommandNumber := 1;

    END;
```

```
      END;

      WITH Transaction^ DO

          Identifier   := 105;
          TimeOutValue := InboardFlaperonRightTransactionTimeout;
          OutputFrame  := Command;

      END;

      QInsert(Transaction, Network1Chain^.TransactionQueue, FALSE);

      NEW(Transaction);
      NEW(Command);

      WITH Command^ DO

          Address := TrailingEdgeLeftDIUAddress1;
          Message := DIUInput;

          WITH DIUCommand DO

              Activity      := IOActivity;
              CommandNumber := 1;

          END;

      END;

      WITH Transaction^ DO

          Identifier   := 106;
          TimeOutValue := TrailingEdgeLeftTransactionTimeout;
          OutputFrame  := Command;

      END;

      QInsert(Transaction, Network1Chain^.TransactionQueue, FALSE);

      NEW(Transaction);
      NEW(Command);

      WITH Command^ DO

          Address := TrailingEdgeRightDIUAddress1;
          Message := DIUInput;

          WITH DIUCommand DO

              Activity      := IOActivity;
              CommandNumber := 1;

          END;

      END;

      WITH Transaction^ DO

          Identifier   := 107;
          TimeOutValue := TrailingEdgeRightTransactionTimeout;
          OutputFrame  := Command;

      END;

      QInsert(Transaction, Network1Chain^.TransactionQueue, FALSE);

(*
      INSERT Transaction LAST IN Network1Chain^.TranactionQueue);
*)

      WITH Network1Chain^ DO
```

```
    NetworkToBeExecutedOn := 1;
    NumberOfTransactions  := QSize(Network1Chain^.TransactionQueue);

END;
NEW(Transaction);
NEW(Command);
WITH Command^ DO

    Address := Sensor1DIUAddress2;
    Message := DIUInput;

    WITH DIUCommand DO

        Activity      := IOActivity;
        CommandNumber := 1;

    END;

END;

WITH Transaction^ DO

    Identifier   := 200;
    TimeOutValue := Sensor1FastTransactionTimeout;
    OutputFrame  := Command;

END;

QInsert(Transaction, Network2Chain^.TransactionQueue, FALSE);

NEW(Transaction);
NEW(Command);

WITH Command^ DO

    Address := Sensor2DIUAddress2;
    Message := DIUInput;

    WITH DIUCommand DO

        Activity      := IOActivity;
        CommandNumber := 1;

    END;

END;

WITH Transaction^ DO

    Identifier   := 201;
    TimeOutValue := Sensor2FastTransactionTimeout;
    OutputFrame  := Command;

END;

QInsert(Transaction, Network2Chain^.TransactionQueue, FALSE);

NEW(Transaction);
NEW(Command);

WITH Command^ DO

    Address := OutboardFlaperonLeftDIUAddress2;
    Message := DIUInput;

    WITH DIUCommand DO

        Activity      := IOActivity;
        CommandNumber := 1;

    END;
```

```
END;

WITH Transaction^ DO

    Identifier   := 202;
    TimeOutValue := OutboardFlaperonLeftTransactionTimeout;
    OutputFrame  := Command;

END;

QInsert(Transaction, Network2Chain^.TransactionQueue, FALSE);

NEW(Transaction);
NEW(Command);

WITH Command^ DO

    Address := OutboardFlaperonRightDIUAddress2;
    Message := DIUInput;

    WITH DIUCommand DO

        Activity      := IOActivity;
        CommandNumber := 1;

    END;

END;

WITH Transaction^ DO

    Identifier   := 203;
    TimeOutValue := OutboardFlaperonRightTransactionTimeout;
    OutputFrame  := Command;

END;

QInsert(Transaction, Network2Chain^.TransactionQueue, FALSE);

NEW(Transaction);
NEW(Command);

WITH Command^ DO

    Address := InboardFlaperonLeftDIUAddress2;
    Message := DIUInput;

    WITH DIUCommand DO

        Activity      := IOActivity;
        CommandNumber := 1;

    END;

END;

WITH Transaction^ DO

    Identifier   := 204;
    TimeOutValue := InboardFlaperonLeftTransactionTimeout;
    OutputFrame  := Command;

END;

QInsert(Transaction, Network2Chain^.TransactionQueue, FALSE);

NEW(Transaction);
NEW(Command);

WITH Command^ DO
```

```
    Address := InboardFlaperonRightDIUAddress2;
    Message := DIUInput;

    WITH DIUCommand DO

        Activity      := IOActivity;
        CommandNumber := 1;

    END;

END;

WITH Transaction^ DO

    Identifier   := 205;
    TimeOutValue := InboardFlaperonRightTransactionTimeout;
    OutputFrame  := Command;

END;

QInsert(Transaction, Network2Chain^.TransactionQueue, FALSE);

NEW(Transaction);
NEW(Command);

WITH Command^ DO

    Address := TrailingEdgeLeftDIUAddress2;
    Message := DIUInput;

    WITH DIUCommand DO

        Activity      := IOActivity;
        CommandNumber := 1;

    END;

END;

WITH Transaction^ DO

    Identifier   := 206;
    TimeOutValue := TrailingEdgeLeftTransactionTimeout;
    OutputFrame  := Command;

END;

QInsert(Transaction, Network2Chain^.TransactionQueue, FALSE);

NEW(Transaction);
NEW(Command);

WITH Command^ DO

    Address := TrailingEdgeRightDIUAddress2;
    Message := DIUInput;

    WITH DIUCommand DO

        Activity      := IOActivity;
        CommandNumber := 1;

    END;

END;

WITH Transaction^ DO

    Identifier   := 207;
    TimeOutValue := TrailingEdgeRightTransactionTimeout;
    OutputFrame  := Command;
```

```
        END;

        QInsert(Transaction, Network2Chain^.TransactionQueue, FALSE);

(*
        INSERT Transaction LAST IN Network2Chain^.TranactionQueue);
*)

        WITH Network2Chain^ DO

            NetworkToBeExecutedOn := 2;
            NumberOfTransactions  := QSize(Network2Chain^.TransactionQueue);

        END;


        WITH FlightControl100Request^ DO

            Priority            := FastRequestPriority;
            Identifier          := FlightControlIdentifier100;;
            RequestTimeoutValue := FlightControlRequestTimeout100;
            RequestType         := ApplicationRequest;
            ChainArray[1]       := Network1Chain;
            ChainArray[2]       := Network2Chain;

        END;


        RETURN (FlightControl100Request);

     END FltCon100Request;

     (****************************************************************************)

     PROCEDURE FltCon50Request(IOActivity : IOActivityChoice) :IORequestType;

         VAR FlightControl50Request  : IORequestType;
             Network1Chain           : ChainType;
             Network2Chain           : ChainType;
             Transaction             : TransactionType;
             Command                 : BusMessageType;

     BEGIN

         (* This process controls the 50Hz I/O Request. I/O Requests will be
            generated at a 50Hz rate, i.e every 10ms. After the I/O response
            is received, this process will delay for a time interval intended
            to represent the time it takes to generate the next output.  *)

         NEW(FlightControl50Request);
         NEW(Network1Chain);
         NEW(Network2Chain);

         Network1Chain^.TransactionQueue := InitQ("TransactionQueue", FALSE, 0);
         Network2Chain^.TransactionQueue := InitQ("TransactionQueue", FALSE, 0);
         Network1Chain^.ChainIdentifier  := 2;
         Network2Chain^.ChainIdentifier  := 2;

         NEW(Transaction);
         NEW(Command);
         WITH Command^ DO

             Address := Sensor1DIUAddress1;
             Message := DIUInput;

             WITH DIUCommand DO

                 Activity      := IOActivity;
                 CommandNumber := 2;

             END;
```

```
END;

WITH Transaction^ DO

    Identifier   := 100;
    TimeOutValue := Sensor1MiddleTransactionTimeout;
    OutputFrame  := Command;

END;

QInsert(Transaction, Network1Chain^.TransactionQueue, FALSE);

NEW(Transaction);
NEW(Command);

WITH Command^ DO

    Address := Sensor2DIUAddress1;
    Message := DIUInput;

    WITH DIUCommand DO

        Activity      := IOActivity;
        CommandNumber := 2;

    END;

END;

WITH Transaction^ DO

    Identifier   := 101;
    TimeOutValue := Sensor2MiddleTransactionTimeout;
    OutputFrame  := Command;

END;

QInsert(Transaction, Network1Chain^.TransactionQueue, FALSE);

NEW(Transaction);
NEW(Command);

WITH Command^ DO

    Address := Cockpit1DIUAddress1;
    Message := DIUInput;

    WITH DIUCommand DO

        Activity      := IOActivity;
        CommandNumber := 2;

    END;

END;

WITH Transaction^ DO

    Identifier   := 102;
    TimeOutValue := Cockpit1MiddleTransactionTimeout;
    OutputFrame  := Command;

END;

QInsert(Transaction, Network1Chain^.TransactionQueue, FALSE);

NEW(Transaction);
NEW(Command);

WITH Command^ DO
```

```
    Address := Cockpit2DIUAddress1;
    Message := DIUInput;

    WITH DIUCommand DO

        Activity      := IOActivity;
        CommandNumber := 2;

    END;

END;

WITH Transaction^ DO

    Identifier   := 103;
    TimeOutValue := Cockpit2TransactionTimeout;
    OutputFrame  := Command;

END;

QInsert(Transaction, Network1Chain^.TransactionQueue, FALSE);

NEW(Transaction);
NEW(Command);

WITH Command^ DO

    Address := CanardLeftDIUAddress1;
    Message := DIUInput;

    WITH DIUCommand DO

        Activity      := IOActivity;
        CommandNumber := 2;

    END;

END;

WITH Transaction^ DO

    Identifier   := 104;
    TimeOutValue := CanardLeftTransactionTimeout;
    OutputFrame  := Command;

END;

QInsert(Transaction, Network1Chain^.TransactionQueue, FALSE);

NEW(Transaction);
NEW(Command);

WITH Command^ DO

    Address := CanardRightDIUAddress1;
    Message := DIUInput;

    WITH DIUCommand DO

        Activity      := IOActivity;
        CommandNumber := 2;

    END;

END;

WITH Transaction^ DO

    Identifier   := 105;
    TimeOutValue := CanardRightTransactionTimeout;
    OutputFrame  := Command;
```

```
END;

QInsert(Transaction, Network1Chain^.TransactionQueue, FALSE);

NEW(Transaction);
NEW(Command);

WITH Command^ DO

    Address := RudderLeftDIUAddress1;
    Message := DIUInput;

    WITH DIUCommand DO

        Activity      := IOActivity;
        CommandNumber := 2;

    END;

END;

WITH Transaction^ DO

    Identifier   := 106;
    TimeOutValue := RudderLeftTransactionTimeout;
    OutputFrame  := Command;

END;

QInsert(Transaction, Network1Chain^.TransactionQueue, FALSE);

NEW(Transaction);
NEW(Command);

WITH Command^ DO

    Address := RudderRightDIUAddress1;
    Message := DIUInput;

    WITH DIUCommand DO

        Activity      := IOActivity;
        CommandNumber := 2;

    END;

END;

WITH Transaction^ DO

    Identifier   := 107;
    TimeOutValue := RudderRightTransactionTimeout;
    OutputFrame  := Command;

END;

QInsert(Transaction, Network1Chain^.TransactionQueue, FALSE);

NEW(Transaction);
NEW(Command);

WITH Command^ DO

    Address := NoseDIUAddress1;
    Message := DIUInput;

    WITH DIUCommand DO

        Activity      := IOActivity;
        CommandNumber := 2;

    END;
```

```
END;

WITH Transaction^ DO

    Identifier   := 108;
    TimeOutValue := NoseTransactionTimeout;
    OutputFrame  := Command;

END;

QInsert(Transaction, Network1Chain^.TransactionQueue, FALSE);

NEW(Transaction);
NEW(Command);

WITH Command^ DO

    Address := LeadingEdgeRightDIUAddress1;
    Message := DIUInput;

    WITH DIUCommand DO

        Activity      := IOActivity;
        CommandNumber := 2;

    END;

END;

WITH Transaction^ DO

    Identifier   := 109;
    TimeOutValue := LeadingEdgeRightTransactionTimeout;
    OutputFrame  := Command;

END;

QInsert(Transaction, Network1Chain^.TransactionQueue, FALSE);

(*
        INSERT Transaction LAST IN Network1Chain^.TranactionQueue);
*)

WITH Network1Chain^ DO

    NetworkToBeExecutedOn := 1;
    NumberOfTransactions  := QSize(Network1Chain^.TransactionQueue);

END;

NEW(Transaction);
NEW(Command);
WITH Command^ DO

    Address := Sensor1DIUAddress2;
    Message := DIUInput;

    WITH DIUCommand DO

        Activity      := IOActivity;
        CommandNumber := 2;

    END;

END;

WITH Transaction^ DO

    Identifier   := 200;
    TimeOutValue := Sensor1MiddleTransactionTimeout;
    OutputFrame  := Command;
```

```
END;

QInsert(Transaction, Network2Chain^.TransactionQueue, FALSE);

NEW(Transaction);
NEW(Command);

WITH Command^ DO

    Address := Sensor2DIUAddress2;
    Message := DIUInput;

    WITH DIUCommand DO

        Activity      := IOActivity;
        CommandNumber := 2;

    END;

END;

WITH Transaction^ DO

    Identifier   := 201;
    TimeOutValue := Sensor2MiddleTransactionTimeout;
    OutputFrame  := Command;

END;

QInsert(Transaction, Network2Chain^.TransactionQueue, FALSE);

NEW(Transaction);
NEW(Command);

WITH Command^ DO

    Address := Cockpit1DIUAddress2;
    Message := DIUInput;

    WITH DIUCommand DO

        Activity      := IOActivity;
        CommandNumber := 2;

    END;

END;

WITH Transaction^ DO

    Identifier   := 202;
    TimeOutValue := Cockpit1MiddleTransactionTimeout;
    OutputFrame  := Command;

END;

QInsert(Transaction, Network2Chain^.TransactionQueue, FALSE);

NEW(Transaction);
NEW(Command);

WITH Command^ DO

    Address := Cockpit2DIUAddress2;
    Message := DIUInput;

    WITH DIUCommand DO

        Activity      := IOActivity;
        CommandNumber := 2;
    END;

END;

WITH Transaction^ DO

    Identifier   := 203;
    TimeOutValue := Cockpit2TransactionTimeout;
    OutputFrame  := Command;

END;

QInsert(Transaction, Network2Chain^.TransactionQueue, FALSE);

NEW(Transaction);
NEW(Command);

WITH Command^ DO

    Address := CanardLeftDIUAddress2;
    Message := DIUInput;

    WITH DIUCommand DO

        Activity      := IOActivity;
        CommandNumber := 2;

    END;

END;

WITH Transaction^ DO

    Identifier   := 204;
    TimeOutValue := CanardLeftTransactionTimeout;
    OutputFrame  := Command;

END;

QInsert(Transaction, Network2Chain^.TransactionQueue, FALSE);

NEW(Transaction);
NEW(Command);

WITH Command^ DO

    Address := CanardRightDIUAddress2;
    Message := DIUInput;

    WITH DIUCommand DO

        Activity      := IOActivity;
        CommandNumber := 2;

    END;

END;

WITH Transaction^ DO

    Identifier   := 205;
    TimeOutValue := CanardRightTransactionTimeout;
    OutputFrame  := Command;

END;

QInsert(Transaction, Network2Chain^.TransactionQueue, FALSE);

NEW(Transaction);
NEW(Command);

WITH Command^ DO
```

```
    Address := RudderLeftDIUAddress2;
    Message := DIUInput;

    WITH DIUCommand DO

        Activity      := IOActivity;
        CommandNumber := 2;

    END;

END;

WITH Transaction^ DO

    Identifier   := 206;
    TimeOutValue := RudderLeftTransactionTimeout;
    OutputFrame  := Command;

END;

QInsert(Transaction, Network2Chain^.TransactionQueue, FALSE);

NEW(Transaction);
NEW(Command);

WITH Command^ DO

    Address := RudderRightDIUAddress2;
    Message := DIUInput;

    WITH DIUCommand DO

        Activity      := IOActivity;
        CommandNumber := 2;

    END;

END;

WITH Transaction^ DO

    Identifier   := 207;
    TimeOutValue := RudderRightTransactionTimeout;
    OutputFrame  := Command;

END;

QInsert(Transaction, Network2Chain^.TransactionQueue, FALSE);

NEW(Transaction);
NEW(Command);

WITH Command^ DO

    Address := NoseDIUAddress2;
    Message := DIUInput;

    WITH DIUCommand DO

        Activity      := IOActivity;
        CommandNumber := 2;

    END;

END;

WITH Transaction^ DO

    Identifier   := 208;
    TimeOutValue := NoseTransactionTimeout;
    OutputFrame  := Command;
```

```
        END;

        QInsert(Transaction, Network2Chain^.TransactionQueue, FALSE);

        NEW(Transaction);
        NEW(Command);

        WITH Command^ DO

            Address := LeadingEdgeRightDIUAddress2;
            Message := DIUInput;

            WITH DIUCommand DO

                Activity      := IOActivity;
                CommandNumber := 2;

            END;

        END;

        WITH Transaction^ DO

            Identifier   := 209;
            TimeOutValue := LeadingEdgeRightTransactionTimeout;
            OutputFrame  := Command;

        END;

        QInsert(Transaction, Network2Chain^.TransactionQueue, FALSE);

(*
        INSERT Transaction LAST IN Network2Chain^.TranactionQueue);
*)

        WITH Network2Chain^ DO

            NetworkToBeExecutedOn := 2;
            NumberOfTransactions  := QSize(Network2Chain^.TransactionQueue);

        END;


        WITH FlightControl50Request^ DO

            Priority            := MiddleRequestPriority;
            Identifier          := FlightControlIdentifier50;
            RequestTimeoutValue := FlightControlRequestTimeout50;
            RequestType         := ApplicationRequest;
            ChainArray[1]       := Network1Chain;
            ChainArray[2]       := Network2Chain;

        END;

        RETURN (FlightControl50Request);

    END FltCon50Request;

    (*************************************************************************)

    PROCEDURE FltCon25Request(IOActivity : IOActivityChoice) :IORequestType;

        VAR FlightControl25Request  : IORequestType;
            Network1Chain           : ChainType;
            Network2Chain           : ChainType;
            Transaction             : TransactionType;
            Command                 : BusMessageType;

    BEGIN

        (* This process controls the 25Hz I/O Request.  I/O Requests will be
    generated at a 25Hz rate, i.e every 40ms. After the I/O response
    is received, this process will delay for a time interval intended
    to represent the time it takes to generate the next output.  *)

NEW(FlightControl25Request);
NEW(Network1Chain);
NEW(Network2Chain);

Network1Chain^.TransactionQueue := InitQ("TransactionQueue", FALSE, 0);
Network2Chain^.TransactionQueue := InitQ("TransactionQueue", FALSE, 0);
Network1Chain^.ChainIdentifier  := 3;
Network2Chain^.ChainIdentifier  := 3;

NEW(Transaction);
NEW(Command);
WITH Command^ DO

    Address := Sensor1DIUAddress1;
    Message := DIUInput;

    WITH DIUCommand DO

        Activity      := IOActivity;
        CommandNumber := 3;

    END;

END;

WITH Transaction^ DO

    Identifier   := 100;
    TimeOutValue := Sensor1SlowTransactionTimeout;
    OutputFrame  := Command;

END;

QInsert(Transaction, Network1Chain^.TransactionQueue, FALSE);

NEW(Transaction);
NEW(Command);

WITH Command^ DO

    Address := Cockpit1DIUAddress1;
    Message := DIUInput;

    WITH DIUCommand DO

        Activity      := IOActivity;
        CommandNumber := 3;

    END;

END;

WITH Transaction^ DO

    Identifier   := 101;
    TimeOutValue := Cockpit1SlowTransactionTimeout;
    OutputFrame  := Command;

END;

QInsert(Transaction, Network1Chain^.TransactionQueue, FALSE);

WITH Network1Chain^ DO

    NetworkToBeExecutedOn := 1;
    NumberOfTransactions   := QSize(Network1Chain^.TransactionQueue);

END;
```

```
NEW(Transaction);
NEW(Command);
WITH Command^ DO

    Address := Sensor1DIUAddress2;
    Message := DIUInput;

    WITH DIUCommand DO

        Activity      := IOActivity;
        CommandNumber := 3;

    END;

END;

WITH Transaction^ DO

    Identifier   := 200;
    TimeOutValue := Sensor1SlowTransactionTimeout;
    OutputFrame  := Command;

END;

QInsert(Transaction, Network2Chain^.TransactionQueue, FALSE);

NEW(Transaction);
NEW(Command);

WITH Command^ DO

    Address := Cockpit1DIUAddress2;
    Message := DIUInput;

    WITH DIUCommand DO

        Activity      := IOActivity;
        CommandNumber := 3;

    END;

END;

WITH Transaction^ DO

    Identifier   := 201;
    TimeOutValue := Cockpit1SlowTransactionTimeout;
    OutputFrame  := Command;

END;

QInsert(Transaction, Network2Chain^.TransactionQueue, FALSE);

WITH Network2Chain^ DO

    NetworkToBeExecutedOn := 2;
    NumberOfTransactions  := QSize(Network2Chain^.TransactionQueue);

END;

WITH FlightControl25Request^ DO

    Priority            := SlowRequestPriority;
    Identifier          := FlightControlIdentifier25;
    RequestTimeoutValue := FlightControlRequestTimeout25;
    RequestType         := ApplicationRequest;
    ChainArray[1]       := Network1Chain;
    ChainArray[2]       := Network2Chain;

END;
```

```
    RETURN (FlightControl25Request);

END FltCon25Request;

(***************************************************************************)

PROCEDURE Engine100Request(IOActivity : IOActivityChoice) :IORequestType;

    VAR EngineControl100Request  : IORequestType;
        Network1Chain            : ChainType;
        Network2Chain            : ChainType;
        Transaction              : TransactionType;
        Command                  : BusMessageType;

BEGIN

    (* This process controls the inlet, which is designated as DIU address
       81 on network 1 and DIU address 91 on network 2.  I/O Requests will
       be generated at a 100Hz rate, i.e every 10ms. After the I/O
       response is received, his process will delay for a time interval
       intended to represent the time it takes to generate the next output.  *)

    (* Initialize the Inlet Chain which has three actuator writes
       and six sensor reads.  *)
    NEW(EngineControl100Request);
    NEW(Network1Chain);
    NEW(Network2Chain);

    Network1Chain^.TransactionQueue := InitQ("TransactionQueue", FALSE, 0);
    Network2Chain^.TransactionQueue := InitQ("TransactionQueue", FALSE, 0);
    Network1Chain^.ChainIdentifier  := 1;
    Network2Chain^.ChainIdentifier  := 1;

    NEW(Transaction);
    NEW(Command);
    WITH Command^ DO

        Address := InletDIUAddress1;
        Message := DIUInput;

        WITH DIUCommand DO

            Activity      := IOActivity;
            CommandNumber := 1;

        END;

    END;

    WITH Transaction^ DO

        Identifier   := 100;
        TimeOutValue := InletTransactionTimeout;
        OutputFrame  := Command;

    END;

    QInsert(Transaction, Network1Chain^.TransactionQueue, FALSE);

    WITH Network1Chain^ DO

        NetworkToBeExecutedOn := 1;
        NumberOfTransactions  := QSize(Network1Chain^.TransactionQueue);

    END;

    NEW(Transaction);
    NEW(Command);
    WITH Command^ DO

        Address := InletDIUAddress2;
        Message := DIUInput;
```

```
      WITH DIUCommand DO

          Activity      := IOActivity;
          CommandNumber := 1;

      END;

   END;

   WITH Transaction^ DO

       Identifier   := 200;
       TimeOutValue := InletTransactionTimeout;
       OutputFrame  := Command;

   END;

   QInsert(Transaction, Network2Chain^.TransactionQueue, FALSE);

   WITH Network2Chain^ DO

       NetworkToBeExecutedOn := 2;
       NumberOfTransactions  := QSize(Network2Chain^.TransactionQueue);

   END;

   WITH EngineControl100Request^ DO

       Priority              := InletRequestPriority;
       Identifier            := EngineControlIdentifier100;;
       RequestType           := ApplicationRequest;
       RequestTimeoutValue   := EngineControlRequestTimeout100;
       RequestType           := ApplicationRequest;
       ChainArray[1]         := Network1Chain;
       ChainArray[2]         := Network2Chain;

   END;

   RETURN (EngineControl100Request);

END Engine100Request;

(***********************************************************************)

PROCEDURE Engine50Request(IOActivity : IOActivityChoice) :IORequestType;

    VAR EngineControl50Request : IORequestType;
        Network1Chain          : ChainType;
        Network2Chain          : ChainType;
        Transaction            : TransactionType;
        Command                : BusMessageType;

BEGIN

    (* This process controls the inlet, which is designated as DIU address
       82 on network 1 and DIU address 92 on network 2.  I/O Requests will
       be generated at a 50Hz rate, i.e every 10ms. After the I/O response
       is received, this process will delay for a time interval intended to
       represent the time it takes to generate the next output.  *)

    (* Initialize the Nozzle Chain which has three actuator writes
       and six sensor reads.  *)
    NEW(EngineControl50Request);
    NEW(Network1Chain);
    NEW(Network2Chain);

    Network1Chain^.TransactionQueue := InitQ("TransactionQueue", FALSE, 0);
    Network2Chain^.TransactionQueue := InitQ("TransactionQueue", FALSE, 0);
    Network1Chain^.ChainIdentifier  := 2;
    Network2Chain^.ChainIdentifier  := 2;
```

```
NEW(Transaction);
NEW(Command);
WITH Command^ DO

    Address := NozzleDIUAddress1;
    Message := DIUInput;

    WITH DIUCommand DO

        Activity      := IOActivity;
        CommandNumber := 2;

    END;

END;

WITH Transaction^ DO

    Identifier   := 100;
    TimeOutValue := NozzleTransactionTimeout;
    OutputFrame  := Command;

END;

QInsert(Transaction, Network1Chain^.TransactionQueue, FALSE);

WITH Network1Chain^ DO

    NetworkToBeExecutedOn := 1;
    NumberOfTransactions  := QSize(Network1Chain^.TransactionQueue);

END;

NEW(Transaction);
NEW(Command);
WITH Command^ DO

    Address := NozzleDIUAddress2;
    Message := DIUInput;

    WITH DIUCommand DO

        Activity      := IOActivity;
        CommandNumber := 2;

    END;

END;

WITH Transaction^ DO

    Identifier   := 200;
    TimeOutValue := NozzleTransactionTimeout;
    OutputFrame  := Command;

END;

QInsert(Transaction, Network2Chain^.TransactionQueue, FALSE);

WITH Network2Chain^ DO

    NetworkToBeExecutedOn := 2;
    NumberOfTransactions  := QSize(Network2Chain^.TransactionQueue);

END;

WITH EngineControl50Request^ DO

    Priority            := NozzleRequestPriority;
    Identifier          := EngineControlIdentifier50;
    RequestType         := ApplicationRequest;
    RequestTimeoutValue := EngineControlRequestTimeout50;
```

```
            ChainArray[1]       := Network1Chain;
            ChainArray[2]       := Network2Chain;

        END;

        RETURN (EngineControl50Request);

    END Engine50Request;

    (**************************************************************************)

    PROCEDURE Engine25Request(IOActivity : IOActivityChoice) :IORequestType;

        VAR EngineControl25Request  : IORequestType;
            Network1Chain           : ChainType;
            Network2Chain           : ChainType;
            Transaction             : TransactionType;
            Command                 : BusMessageType;

    BEGIN

        (* This process controls the inlet, which is designated as DIU address
           83 on network 1 and DIU address 93 on network 2.  I/O Requests will
           be generated at a 25Hz rate, i.e every 10ms. After the I/O response
           is received, this process will delay for a time interval intended to
           represent the time it takes to generate the next output.  *)

        (* Initialize the Engine Chain which has three actuator writes
           and six sensor reads.  *)
        NEW(EngineControl25Request);
        NEW(Network1Chain);
        NEW(Network2Chain);

        Network1Chain^.TransactionQueue := InitQ("TransactionQueue", FALSE, 0);
        Network2Chain^.TransactionQueue := InitQ("TransactionQueue", FALSE, 0);
        Network1Chain^.ChainIdentifier  := 3;
        Network2Chain^.ChainIdentifier  := 3;

        NEW(Transaction);
        NEW(Command);
        WITH Command^ DO

            Address := EngineDIUAddress1;
            Message := DIUInput;

            WITH DIUCommand DO

                Activity      := IOActivity;
                CommandNumber := 3;

            END;

        END;

        WITH Transaction^ DO

            Identifier   := 100;
            TimeOutValue := EngineTransactionTimeout;
            OutputFrame  := Command;

        END;

        QInsert(Transaction, Network1Chain^.TransactionQueue, FALSE);

        WITH Network1Chain^ DO

            NetworkToBeExecutedOn := 1;
            NumberOfTransactions  := QSize(Network1Chain^.TransactionQueue);

        END;

        NEW(Transaction);
    NEW(Command);
    WITH Command^ DO

        Address := EngineDIUAddress2;
        Message := DIUInput;

        WITH DIUCommand DO

            Activity      := IOActivity;
            CommandNumber := 3;

        END;

    END;

    WITH Transaction^ DO

        Identifier   := 200;
        TimeOutValue := EngineTransactionTimeout;
        OutputFrame  := Command;

    END;

    QInsert(Transaction, Network2Chain^.TransactionQueue, FALSE);

    WITH Network2Chain^ DO

        NetworkToBeExecutedOn := 2;
        NumberOfTransactions  := QSize(Network1Chain^.TransactionQueue);

    END;

    WITH EngineControl25Request^ DO

        Priority            := EngineRequestPriority;
        Identifier          := EngineControlIdentifier25;
        RequestType         := ApplicationRequest;
        RequestTimeoutValue := EngineControlRequestTimeout25;
        RequestType         := ApplicationRequest;
        ChainArray[1]       := Network1Chain;
        ChainArray[2]       := Network2Chain;

    END;

    RETURN (EngineControl25Request);

END Engine25Request;

(**************************************************************************)

PROCEDURE BuildRequest(Application  : ApplicationType;
                       Rate         : INTEGER;
                       IOActivity   : IOActivityChoice) :IORequestType;

    VAR Request : IORequestType;

BEGIN

    CASE Application OF


        FlightControl:

            CASE Rate OF

                 100:

                    Request := FltCon100Request(IOActivity);

                | 50:

                    Request := FltCon50Request(IOActivity);
                    | 25:
                        Request := FltCon25Request(IOActivity);
                END;
            | EngineControl:
                CASE Rate OF
                    100:
                        Request := Engine100Request(IOActivity);
                    | 50:
                        Request := Engine50Request(IOActivity);
                    | 25:
                        Request := Engine25Request(IOActivity);
                END;
        END;
        Request^.ResponseExpected := NOT (IOActivity = Output);
        RETURN (Request);
    END BuildRequest;
    (*****************************************************************************)
END BldChains.
```

APPLICATN

```
DEVM Applicatn;

    FROM IOService REACH IORequestType*, IOResponseType*, ChainStatusType,
                         ReleaseChainResponseMemory;

    FROM Processor REACH ProcessingUnit*;

    FROM BusMessag IMPORT IOActivityChoice;

    FROM BldChains IMPORT ApplicationType, BuildRequest;

    FROM Senddata IMPORT WriteDataElementType, CyclicDataType, DataElementType,
                         CyclicVariationType, FrequencyType;

    INPUTS
        EVENT
              ResponseApplication : IOResponseType;
              ProcessorResponse   : ProcessingUnit;
              Reset               : BOOLEAN;

        PARA
              CPID                    : INTEGER;
              IOServiceID             : INTEGER;
              ProcessingTimeMean      : REAL;
              ProcessingTimeSigma     : REAL;
              EngineApplication       : BOOLEAN;
              OnDemand                : BOOLEAN;
              GroupedIOActivity       : BOOLEAN;
              ApplicationPriority     : INTEGER;
              IORequestInterval       : REAL;
              InitialOffset           : REAL;
              ApplicationIdentifier   : INTEGER;

    END;

    OUTPUTS
        VAR
            RequestApplication : IORequestType;
            ProcessorRequest   : ProcessingUnit;

    END;

    EVENT
          CurrentFrame : INTEGER;

    CONST
          FrameStartUpTime = 0.000020;

    VAR
        ApplicationResponse          : IOResponseType;
        ApplicationRequest           : IORequestType;
        ApplicationInputRequest      : IORequestType;
        ApplicationOutputRequest     : IORequestType;
        PartialData                  : BOOLEAN;
        ProcessingRequest            : ProcessingUnit;
        CPResponseData               : ProcessingUnit;
        TempRequest                  : IORequestType;
        CPPort                       : INTEGER;
        IOServicePort                : INTEGER;
        OutputDataElement            : DataElementType;
        ThereIsAFrameExecuting       : BOOLEAN;

    (****************************************************************************)

    PROCEDURE CommunicationFaults(Response : IOResponseType) :BOOLEAN;

    BEGIN

        WITH Response^ DO

            RETURN (ChainStatus[1] <> NoFaults) OR (ChainStatus[2] <> NoFaults);
```

```
         END;

     END CommunicationFaults;

     (*************************************************************************)


     PROCEDURE VerifyCorrectResponse(Response : IOResponseType);

     BEGIN

         IF GroupedIOActivity THEN
             IF (Response^.Identifier <> ApplicationRequest^.Identifier) OR
                (QSize(Response^.ResponseArray[1]^.InputFrameQueue) <>
                       ApplicationRequest^.Chain^.NumberOfTransactions) OR
                (QSize(Response^.ResponseArray[2]^.InputFrameQueue) <>
                       ApplicationRequest^.Chain^.NumberOfTransactions) THEN

               WriteString(ParamOut, "Unexpected response received from the IO System");
               WriteLn(ParamOut);
               HALT;

             END;

        ELSIF (Response^.Identifier <> ApplicationInputRequest^.Identifier) OR
              (QSize(Response^.ResponseArray[1]^.InputFrameQueue) <>
                    ApplicationInputRequest^.Chain^.NumberOfTransactions) OR
              (QSize(Response^.ResponseArray[2]^.InputFrameQueue) <>
                    ApplicationInputRequest^.Chain^.NumberOfTransactions) THEN

               WriteString(ParamOut, "Unexpected response received from the IO System");
               WriteLn(ParamOut);
               HALT;

        END;

    END VerifyCorrectResponse;

    (*************************************************************************)

BEGIN

    CPPort := GetOutPort(CPID);
    IOServicePort := GetOutPort(IOServiceID);

    ThereIsAFrameExecuting := FALSE;

    IF GroupedIOActivity OR NOT OnDemand THEN    (* Scheduled IO must use GroupedIO *)

       IF EngineApplication THEN
          ApplicationRequest := BuildRequest(EngineControl, ApplicationIdentifier, Grouped);
       ELSE
          ApplicationRequest := BuildRequest(FlightControl, ApplicationIdentifier, Grouped);
       END;

       IF ApplicationIdentifier = 100 THEN

          ApplicationRequest^.ChainArray[1]^.ChainIdentifier := 1;
          ApplicationRequest^.ChainArray[2]^.ChainIdentifier := 1;

       ELSIF ApplicationIdentifier = 50 THEN

          ApplicationRequest^.ChainArray[1]^.ChainIdentifier := 2;
          ApplicationRequest^.ChainArray[2]^.ChainIdentifier := 2;

       ELSIF ApplicationIdentifier = 25 THEN

          ApplicationRequest^.ChainArray[1]^.ChainIdentifier := 3;
          ApplicationRequest^.ChainArray[2]^.ChainIdentifier := 3;

       END;
```

```
   ApplicationRequest^.OnDemand := OnDemand;

ELSE        (* separated IO Activity *)

   IF EngineApplication THEN
      ApplicationInputRequest   := BuildRequest(EngineControl, ApplicationIdentifier, Input);
      ApplicationOutputRequest := BuildRequest(EngineControl, ApplicationIdentifier, Output);
   ELSE
      ApplicationInputRequest   := BuildRequest(FlightControl, ApplicationIdentifier, Input);
      ApplicationOutputRequest := BuildRequest(FlightControl, ApplicationIdentifier, Output);
   END;

   IF ApplicationIdentifier = 100 THEN

      ApplicationInputRequest^.ChainArray[1]^.ChainIdentifier := 5;
      ApplicationInputRequest^.ChainArray[2]^.ChainIdentifier := 5;
      ApplicationOutputRequest^.ChainArray[1]^.ChainIdentifier := 9;
      ApplicationOutputRequest^.ChainArray[2]^.ChainIdentifier := 9;

   ELSIF ApplicationIdentifier = 50 THEN

      ApplicationInputRequest^.ChainArray[1]^.ChainIdentifier := 6;
      ApplicationInputRequest^.ChainArray[2]^.ChainIdentifier := 6;
      ApplicationOutputRequest^.ChainArray[1]^.ChainIdentifier := 10;
      ApplicationOutputRequest^.ChainArray[2]^.ChainIdentifier := 10;

   ELSIF ApplicationIdentifier = 25 THEN

      ApplicationInputRequest^.ChainArray[1]^.ChainIdentifier := 7;
      ApplicationInputRequest^.ChainArray[2]^.ChainIdentifier := 7;
      ApplicationOutputRequest^.ChainArray[1]^.ChainIdentifier := 11;
      ApplicationOutputRequest^.ChainArray[2]^.ChainIdentifier := 11;

   END;

   ApplicationInputRequest^.OnDemand  := OnDemand;
   ApplicationOutputRequest^.OnDemand := OnDemand;

END;

LOOP

   WAITUNTIL EVENT

      CurrentFrame:

         AFTER IORequestInterval CurrentFrame <- CurrentFrame + 1;

         IF OnDemand THEN

            IF ThereIsAFrameExecuting THEN

               REPORT "%d" CurrentFrame TAGGED "frame overrun at frame number";

            ELSE

               NEW(ProcessingRequest);
               ProcessingRequest^.Priority    := ApplicationPriority;
               ProcessingRequest^.ProcessID   := "FrameStartup";
               ProcessingRequest^.Frame       := CurrentFrame;
               ProcessingRequest^.ProcessingRequired := FrameStartUpTime;
               ProcessingRequest^.ProcssngAfterBlock := Normal(1, ProcessingTimeMean, ProcessingTimeSigma);
               ProcessingRequest^.WriteData := TRUE;

               IF GroupedIOActivity THEN

                  ProcessingRequest^.Data := ApplicationRequest;

               ELSE

                  ProcessingRequest^.Data := ApplicationInputRequest;
            END;

            NOW outport[CPPort]^.ProcessorRequest <- ProcessingRequest;

            ThereIsAFrameExecuting := TRUE;

        END;

    ELSE  (* scheduled IO *)

        ApplicationRequest^.Frame := CurrentFrame;
        ApplicationRequest^.ChainArray[1]^.FrameCount := CurrentFrame;
        ApplicationRequest^.ChainArray[2]^.FrameCount := CurrentFrame;

        NOW outport[IOServicePort]^.RequestApplication <- ApplicationRequest;

    END;

| ResponseApplication:

    ApplicationResponse := ActivePort^.ResponseApplication;

    (* Schedule use of the processor.  *)
    IF OnDemand THEN

        ProcessingRequest^.ProcessingRequired := ProcessingRequest^.ProcssngAfterBlock;
        ProcessingRequest^.WriteData := FALSE;
        ProcessingRequest^.ProcessID := "Response Processing";

        NOW outport[CPPort]^.ProcessorRequest <- ProcessingRequest;

    ELSE  (* scheduled IO *)

        IF ThereIsAFrameExecuting THEN

            REPORT "%d" CurrentFrame TAGGED "frame overrun at frame number";
            OutputDataElement.CyclicData.FrameOverrun := TRUE;

        ELSE

            NEW(ProcessingRequest);
            ProcessingRequest^.Priority           := ApplicationPriority;
            ProcessingRequest^.Frame              := ApplicationResponse^.Frame;
            ProcessingRequest^.ProcessingRequired := Normal(1, ProcessingTimeMean, ProcessingTimeSigma);
            ProcessingRequest^.ProcssngAfterBlock := 0.0;
            ProcessingRequest^.WriteData          := TRUE;
            ProcessingRequest^.ProcessID := "Response Processing";

            NOW outport[CPPort]^.ProcessorRequest <- ProcessingRequest;

            ThereIsAFrameExecuting := TRUE;
            OutputDataElement.CyclicData.FrameOverrun := FALSE;

        END;

    END;

    PartialData := CommunicationFaults(ApplicationResponse);

    IF NOT PartialData THEN

        VerifyCorrectResponse(ApplicationResponse);

    END;

    (* Dispose of the memory in the Application Response. *)
    IF ApplicationResponse^.ResponseArray[1] <> NIL THEN

        ReleaseChainResponseMemory(ApplicationResponse^.ResponseArray[1]);

    END;
    IF ApplicationResponse^.ResponseArray[2] <> NIL THEN

    ReleaseChainResponseMemory(ApplicationResponse^.ResponseArray[2]);

    END;

    DISPOSE(ApplicationResponse);

| ProcessorResponse:

    CPResponseData := ActivePort^.ProcessorResponse;

    IF OnDemand THEN

      IF CPResponseData^.WriteData THEN

         TempRequest := CPResponseData^.Data;
         TempRequest^.Frame := CPResponseData^.Frame;
         TempRequest^.ChainArray[1]^.FrameCount := CPResponseData^.Frame;
         TempRequest^.ChainArray[2]^.FrameCount := CPResponseData^.Frame;
         NOW outport[IOServicePort]^.RequestApplication <- TempRequest;

      ELSE

         CASE ApplicationIdentifier OF
           100: OutputDataElement.EventID := 4;
          | 50: OutputDataElement.EventID := 5;
          | 25: OutputDataElement.EventID := 6;
         END;
         OutputDataElement.SimulationTime := clock;
         OutputDataElement.Frequency := Cyclic;
         OutputDataElement.CyclicData.FrameCount := CPResponseData^.Frame;
         OutputDataElement.CyclicData.C_Variation := EndComputing;
         OutputDataElement.CyclicData.PartialDataUsedThisFrame := PartialData;
         OutputDataElement.CyclicData.FrameOverrun := CPResponseData^.Frame <> CurrentFrame;
         OutputDataElement.CyclicData.ProcessingNotCompleted := 0.0;
         WriteDataElementType(OutputDataElement);

         ThereIsAFrameExecuting := FALSE;

         IF (NOT GroupedIOActivity) AND
            (ApplicationOutputRequest^.Chain^.NumberOfTransactions <> 0) THEN

            ApplicationOutputRequest^.Frame := CPResponseData^.Frame;
            ApplicationOutputRequest^.ChainArray[1]^.FrameCount := CPResponseData^.Frame;
            ApplicationOutputRequest^.ChainArray[2]^.FrameCount := CPResponseData^.Frame;
            NOW outport[IOServicePort]^.RequestApplication <- ApplicationOutputRequest;

         END;

         DISPOSE(CPResponseData);

      END;

    ELSE   (* scheduled IO *)

      CASE ApplicationIdentifier OF
        100: OutputDataElement.EventID := 4;
       | 50: OutputDataElement.EventID := 5;
       | 25: OutputDataElement.EventID := 6;
      END;
      OutputDataElement.SimulationTime := clock;
      OutputDataElement.Frequency := Cyclic;
      OutputDataElement.CyclicData.FrameCount := CPResponseData^.Frame;
      OutputDataElement.CyclicData.C_Variation := EndComputing;
      OutputDataElement.CyclicData.PartialDataUsedThisFrame := PartialData;
      OutputDataElement.CyclicData.ProcessingNotCompleted := 0.0;
      WriteDataElementType(OutputDataElement);

      ThereIsAFrameExecuting := FALSE;
      DISPOSE(CPResponseData);
```

```
                END;
            | Reset :
                ThereIsAFrameExecuting := FALSE;
                REPORT "%12.8f"  clock TAGGED "Application started at ";
                AFTER InitialOffset CurrentFrame <- 1;
        END;
    END;
END Applicatn.
```

CONTROLS

```
DEFINITION DEVM Controls;

EXPORT SystemProbe, NumberOfProbes;

CONST

   NumberOfProbes = 4;

VAR

   SystemProbe : ARRAY [1 .. NumberOfProbes] OF AProbe;

END Controls.
```

CONTROLS

```
DEVM Controls;

FROM Senddata IMPORT StartCollectingData, FlushDataStructure, EndCollectingData,
                     WriteDataElementType, DataElementType, NonCyclicVariationType,
                     FrequencyType;

FROM BusMessag IMPORT NumberOfNodes;

FROM Math IMPORT RealMod;

FROM Util IMPORT GetSeedValue, SetSeedValue;

FROM NodeM IMPORT RemoveArc, RestoreArc;

FROM Conversions IMPORT RealToDFloat;


   INPUTS
      EVENT
         NetworkReady       : BOOLEAN;

      PARA
         ApplicationID      : ARRAY [1 .. 3] OF INTEGER;
         IOSID              : ARRAY [1 .. 6] OF INTEGER;
         Network1Nodes      : ARRAY [1 .. NumberOfNodes] OF INTEGER;
         Network2Nodes      : ARRAY [1 .. NumberOfNodes] OF INTEGER;
         IOServiceID        : INTEGER;
         NetworkManager2ID  : INTEGER;
         CPID               : INTEGER;
         IOPID              : INTEGER;
         ExperimentInfo     : ARRAY [1 .. 4] OF INTEGER;
         StartRunNumber     : INTEGER;
         EndRunNumber       : INTEGER;
         UseInitialSeed     : BOOLEAN;
         InitialSeed        : INTEGER;
         SimulationLength   : REAL;
         LinkFaults         : BOOLEAN;
         SourceFaultNode    : INTEGER;
         DestFaultNode      : INTEGER;
         StartCPFDIRTime    : REAL;
         StartIOPFDIRTime   : REAL;
         SystemReportLevel  : INTEGER;

  END;

  OUTPUTS
     VAR
        ResetCommand       : BOOLEAN;
        SubmitSystem       : BOOLEAN;
        ResetProbeCommand  : BOOLEAN;

  END;

  EVENT
     Restart        : INTEGER;
     InsertFault    : BOOLEAN;
     StopSimulation : BOOLEAN;
     StopProbe      : BOOLEAN;

  CONST
     MajorFrameLength = 0.040;
     SecondMajorFrame = 0.050;

  VAR
     ApplicationPort       : ARRAY [1 .. 3] OF INTEGER;
     IOSPort               : ARRAY [1 .. 6] OF INTEGER;
     IOServicePort         : INTEGER;
     NetworkManager2Port   : INTEGER;
     IOPPort               : INTEGER;
     CPPort                : INTEGER;
     NodeIndex             : INTEGER;
     IOSIndex              : INTEGER;
```

```
        NodeID               : INTEGER;
        FaultTime            : REAL;
        TimeToNextMajorFrame : REAL;
        NextMajorFrame       : REAL;
        StartDataElement     : DataElementType;
        TermDataElement      : DataElementType;
        Index                : INTEGER;

BEGIN

    IOServicePort        := GetOutPort(IOServiceID);
    NetworkManager2Port  := GetOutPort(NetworkManager2ID);
    IOPPort              := GetOutPort(IOPID);
    CPPort               := GetOutPort(CPID);

    FOR Index := 1 TO 3 DO

       ApplicationPort[Index] := GetOutPort(ApplicationID[Index]);

    END;

    FOR IOSIndex := 1 TO 6 DO

       IOSPort[IOSIndex] := GetOutPort(IOSID[IOSIndex]);

    END;


    FOR NodeID := 1 TO MaxNodeID DO

        ReportLevel[NodeID] := -1;    (* turn off the report control *)

    END;

    TermDataElement.EventID := 41;
    TermDataElement.Frequency := NonCyclic;
    TermDataElement.NonCyclicData.N_Variation := RunTerm;

    StartDataElement.EventID := 0;
    StartDataElement.Frequency := NonCyclic;
    StartDataElement.NonCyclicData.N_Variation := RunStart;
    StartDataElement.NonCyclicData.FaultTime := 0.0;
    StartDataElement.NonCyclicData.FirstFDIR := StartCPFDIRTime;

    LOOP

        WAITUNTIL EVENT

            NetworkReady:

               IF UseInitialSeed THEN

                  SetSeedValue(1, InitialSeed);

               END;

               NOW Restart <- StartRunNumber;

               FOR NodeID := 1 TO MaxNodeID DO

                  ReportLevel[NodeID] := SystemReportLevel;  (* turn report control back on if desired *)

               END;

           | Restart:

               WriteString(ParamOut, "Run number ");
               WriteInt(ParamOut, Restart, 0);
               WriteLn(ParamOut);

               IF Restart > StartRunNumber THEN
    TermDataElement.SimulationTime := clock;
    TermDataElement.NonCyclicData.FinalSeed := GetSeedValue(1);
    WriteDataElementType(TermDataElement);
    FlushDataStructure;
    EndCollectingData;

   (* Restore link removed during previous run.  *)
    IF LinkFaults THEN

       RestoreArc(SourceFaultNode, DestFaultNode);

    END;

 END;

 ResetTimer;
 StartCollectingData(ExperimentInfo[1],ExperimentInfo[2],ExperimentInfo[3],
                     Restart,ExperimentInfo[4]);

 StartDataElement.SimulationTime := clock;
 StartDataElement.NonCyclicData.InitialSeed := GetSeedValue(1);

(*
 *   Tell everybody to go back to initial state.
 *)
 IOSIndex := 1;

 WHILE (IOSIndex <= 6) AND (IOSID[IOSIndex] <> 0) DO

    NOW outport[IOSPort[IOSIndex]]^.ResetCommand <- TRUE;
    INC(IOSIndex);

 END;

 FOR NodeIndex := 1 TO NumberOfNodes DO

     IF Network1Nodes[NodeIndex] <> 0 THEN

         NOW outport[GetOutPort(Network1Nodes[NodeIndex])]^.ResetCommand <- TRUE;

     END;

     IF Network2Nodes[NodeIndex] <> 0 THEN

         NOW outport[GetOutPort(Network2Nodes[NodeIndex])]^.ResetCommand <- TRUE;

     END;

 END;

 NOW outport[IOServicePort]^.ResetCommand <- TRUE;
 NOW outport[NetworkManager2Port]^.ResetCommand <- TRUE;
 NOW outport[IOPPort]^.ResetCommand <- TRUE;
 NOW outport[CPPort]^.ResetCommand <- TRUE;

(*
 *   Schedule the first application to run.
 *)
 AFTER StartCPFDIRTime outport[CPPort]^.SubmitSystem <- TRUE;
 AFTER StartIOPFDIRTime outport[IOPPort]^.SubmitSystem <- TRUE;
 FOR Index := 1 TO 3 DO
    AFTER 0.010 outport[ApplicationPort[Index]]^.ResetCommand <- TRUE;
 END;
(*
 * Start sampling at the start of the 2nd major frame.  Note that
 * the command is only set to the first and fourth IOS.  This
 * is all that is needed to reset the network utilization probes.
 *)
 AFTER SecondMajorFrame outport[IOPPort]^.ResetProbeCommand <- TRUE;
 AFTER SecondMajorFrame outport[CPPort]^.ResetProbeCommand <- TRUE;
 AFTER SecondMajorFrame outport[IOServicePort]^.ResetProbeCommand <- TRUE;
```

```
    AFTER SecondMajorFrame outport[IOSPort[1]]^.ResetProbeCommand <- TRUE;
    AFTER SecondMajorFrame outport[IOSPort[4]]^.ResetProbeCommand <- TRUE;

    IF LinkFaults THEN

        FaultTime := Random(1, 0.0, 0.040);
        AFTER (SecondMajorFrame + FaultTime) InsertFault <- TRUE;
        StartDataElement.NonCyclicData.FaultTime := SecondMajorFrame + FaultTime;

    ELSE

        AT RealToDFloat(SimulationLength - MajorFrameLength) StopProbe <- TRUE;

        IF Restart = EndRunNumber THEN

            AFTER SimulationLength StopSimulation <- TRUE;

        ELSE

            AFTER SimulationLength Restart <- Restart + 1;

        END;

    END;

    WriteDataElementType(StartDataElement);

| StopSimulation:

    TermDataElement.SimulationTime := clock;
    TermDataElement.NonCyclicData.FinalSeed := GetSeedValue(1);
    WriteDataElementType(TermDataElement);
    FlushDataStructure;
    EndCollectingData;
    ResetTimer;   (* has effect of stopping the simulation *)

| InsertFault:

    RemoveArc(SourceFaultNode, DestFaultNode);
    REPORT "%d,%d" SourceFaultNode, DestFaultNode TAGGED "Removed link between nodes ";

    WAITUNTIL (NetworkReady)

        NetworkReady:

            TimeToNextMajorFrame := MajorFrameLength - RealMod(clock - 0.010, MajorFrameLength);
            NextMajorFrame := TimeToNextMajorFrame + MajorFrameLength;


            AFTER NextMajorFrame StopProbe <- TRUE;

            IF Restart = EndRunNumber THEN

                AFTER (NextMajorFrame ) StopSimulation <- TRUE;

            ELSE

                AFTER (NextMajorFrame ) Restart <- Restart + 1;

            END;

    END;

| StopProbe:

     IF StopProbe THEN

        FOR Index := 1 TO NumberOfProbes DO

           SAMPLE 0.0 WITH SystemProbe[Index];   (* this forces the last samples to be recorded *)

        END;
```

```
                TermDataElement.NonCyclicData.CPTimeUsed := SystemProbe[1]^.SumX;
                TermDataElement.NonCyclicData.CPTimeAvail := SystemProbe[1]^.TEnd - SystemProbe[1]^.TStart;
                TermDataElement.NonCyclicData.IOPTimeUsed := SystemProbe[2]^.SumX;
                TermDataElement.NonCyclicData.IOPTimeAvail := SystemProbe[2]^.TEnd - SystemProbe[2]^.TStart;
                TermDataElement.NonCyclicData.NWTimeUsed := SystemProbe[3]^.SumX;
                TermDataElement.NonCyclicData.NWTimeAvail := SystemProbe[3]^.TEnd - SystemProbe[3]^.TStart;
                TermDataElement.NonCyclicData.IOSysTimeUsed := SystemProbe[4]^.SumX;
                TermDataElement.NonCyclicData.IOSysTimeAvail := SystemProbe[4]^.TEnd - SystemProbe[4]^.TStart;

             END;

         END;

    END;

END Controls.
```

PRINCIPAL

```
DEVM principal;

DEVMS CentralDB, IOS, AIPSNode, DIU, IOService, NetManger, Applicatn,
         Processor, Controls;
DEVC
      RootLinkOutboard    = (OutputTransaction    NodeCommandFrame);
      RootLinkInboard     = (NodeResponseFrame    InputTransaction);
      NodeToNode          = (NodeResponseFrame    NodeCommandFrame);
      NodeToDIU           = (NodeResponseFrame    DIUCommandFrame);
      DIUToNode           = (DIUResponseFrame     NodeCommandFrame);
      ServiceManagerConn  = (ManagerServiceRqst   ServiceRequest,
                             IOManager2Response   IONetworkResponse);
      ManagerRequest      = (IONetworkRequest     IOServiceRequest,
                             NewNetworkState      RtnNetworkToService);
      AppResponse         = (ApplicationResponse  ResponseApplication);
      AppRequest          = (RequestApplication   IOServiceRequest);
      NIOutConnection     = (ChainToIOS           ChainToProcess,
                             StopIOS              StopChain);
      NIInConnection      = (IOChainResponse      DataFromIOS,
                             ChainFinished        ChainCompleted);
      Submit              = (ProcessorRequest     SubmitProcess);
      Receive             = (Completed            ProcessorResponse);
      SchAndResetSystem   = (SubmitSystem         StartSystem,
                             ResetCommand         Reset,
                             ResetProbeCommand    ProbeReset);
      SystemReady         = (ServiceAvailable     NetworkReady);
      ResetSystem         = (ResetCommand         Reset);
      ResetProbeAndSys    = (ResetCommand         Reset,
                             ResetProbeCommand    ProbeReset);


BEGIN

    CheckQueue := FALSE;
    CheckMemory := FALSE;
    WriteString(ParamOut,"******* MIRON IS DOING THE MEMORY MANAGEMENT *******");
    WriteLn(ParamOut);

    InitSimulator;
    Simulate(100.0);
    TerminateSimulator;

END principal.
```

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No.
NASA CR-182004

2. Government Accession No.

3. Recipient's Catalog No.

4. Title and Subtitle
Design of an Integrated Airframe/Propulsion Control System Architecture

5. Report Date
March 1990

6. Performing Organization Code

7. Author(s)

Gerald C. Cohen
C. William Lee
Michael J. Strickland

8. Performing Organization Report No.

9. Performing Organization Name and Address

Boeing Advanced Systems
P.O. Box 3707, MS 33-12
Seattle, WA 98124-2207

10. Work Unit No.
505-66-71-02

11. Contract or Grant No.
NAS1-18099

12. Sponsoring Agency Name and Address

NASA Langley Research Center
Hampton, VA 23665-5225

13. Type of Report and Period Covered
Contractor Report

14. Sponsoring Agency Code

15. Supplementary Notes

Langley Technical Monitor: Daniel L. Palumbo

16. Abstract

This report describes the design of an integrated Airframe/Propulsion Control System Architecture. The design is based on a prevalidation methodology that uses both reliability and performance tools. The report gives an account of the motivation for the final design and problems associated with both reliability and performance modelling. The appendices contain a listing of the code for both the reliability and performance model used in the design.

17. Key Words (Suggested by Author(s))

Flight critical architecture, redundancy techniques, integrated flight/propulsion control, ASSIST, SURE, DENET, reliability models, performance models

18. Distribution Statement

[redacted]

Subject Category 66

19. Security Classif. (of this report)
Unclassified

20. Security Classif. (of this page)
Unclassified

21. No. of pages
563

22. Price

[redacted]