NASA Contractor Report 181767

Advanced Information Processing System: Local System Services

Laura Burkhardt
Linda Alger
Roy Whittredge
Peter Stasiowski

THE CHARLES STARK DRAPER LABORATORY, INC.
CAMBRIDGE, MA 02139

Contract NAS1-18061
April 1989

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

1.0 INTRODUCTION

The purpose of this report is to document the functional requirements and detailed specifications for the Local System Services of the Advanced Information Processing System (AIPS). This introductory section is provided to outline the overall architecture and functional requirements of the AIPS system. Section 1.1 gives a brief overview of the AIPS architecture as well as a detailed description of the AIPS Fault Tolerant Processor (FTP) architecture, while Section 1.2 provides an introduction to the AIPS system software. Sections 2 through 6 describe the Local System Services functional requirements and design and detailed specifications. Each of these sections describes one of the Local System Services functions. Section 7 concludes with a summary of results and suggestions for future work in this area.

1.1 AIPS Architecture

The Advanced Information Processing System is designed to provide a fault- and damage-tolerant data processing architecture which can serve as the core avionics system for a broad range of aerospace vehicles being researched and developed by NASA. These vehicles include manned and unmanned space vehicles and platforms, deep space probes, commercial transports, and tactical military aircraft.

AIPS is a multicomputer architecture composed of hardware and software 'building blocks' that can be configured to meet a broad range of application requirements. The hardware building blocks are fault-tolerant, general purpose computers (GPCs), fault- and damage-tolerant inter-computer (IC) and input/output (I/O) networks, and interfaces between the networks and the general purpose computers. The software building blocks are the major software functions: local system services, input/output system services, inter-computer system services and the system manager. This software provides the services necessary in a traditional real-time computer such as task scheduling and dispatching, communication with sensors and actuators, etc. The software also supplies the redundancy management services necessary in a redundant computer and the services necessary in a distributed system such as inter-function communication across processing sites, management of distributed redundancy, management of networks, and migration of functions between processing sites.

The AIPS hardware consists of a number of computers which may be physically dispersed throughout a vehicle. These dispersed computers are linked together by a reliable, damage-tolerant data communication pathway called the IC network, or IC bus. (Since the hardware implementation is a circuit-switched network which appears to the communication software and the receiving and transmitting devices as a conventional bus, the terms 'network' and 'bus' are used interchangeably throughout this document.) A computer at any particular processing site may also have access to varying numbers and types of I/O buses, which are separate from the IC bus. The I/O buses may be global,

regional or local in nature. I/O devices on the global I/O bus are available to all, or at least a majority, of the AIPS computers. Regional buses connect I/O devices in a given region to the processing sites located in their vicinity. Local buses connect a computer to the I/O devices dedicated to that computer. Additionally, I/O devices may be connected directly to the internal bus of a processor and accessed as though the I/O devices reside in the computer memory (memory mapped I/O). Both the I/O buses and the IC bus are time-division multiple-access contention buses. Figure 1 shows the laboratory engineering model for a distributed AIPS configuration. This distributed AIPS configuration includes all the hardware and software building blocks mentioned earlier and was conceived to demonstrate the feasibility of the AIPS architecture.

The laboratory configuration of the distributed AIPS system shown in Figure 1 consists of four processing sites. Each processing site has a General Purpose Computer (GPC). GPCs may be simplex or they may be FTPs of varying redundancy levels. Of the four FTPs in the laboratory configuration, one is simplex, one is duplex, and two are triplex processors. An FTP may also be quadruply redundant but none was fabricated for the AIPS laboratory demonstration. The redundant FTPs are built such that they can be physically dispersed for damage tolerance; each of the redundant channels of a FTP can be as far as 5 meters from other channels of the same FTP. The FTP architecture is described in more detail in the following subsection.

The GPCs communicate with each other over the Inter-Computer Network, in which the circuit-switching nodes have been configured into redundant virtual buses. Each redundant bus is referred to as a layer; these layers are totally independent and are not cross-strapped to each other. Each layer contains a circuit-switched node for each processing site; thus every processing site is serviced by three nodes of the IC network. GPCs are designed to receive data on all three layers, but the capability of a GPC to transmit on the network depends on the GPC redundancy level. Triplex FTPs can transmit on all three layers, duplex FTPs on only two of the three layers, and simplex processors on only a single layer. In duplex and triplex FTPs, a given processor can transmit on only one network layer. Thus malicious behavior of a processor can disrupt only one layer.

The IC network and the GPC interfaces into the network are designed in strict accordance with fault-tolerant systems theory so that any arbitrary random hardware fault, even a Byzantine fault, can not disrupt communication between triplex FTPs. Thus the triplex IC network, in conjunction with the GPC interfaces into the network, provides error-masking capability for communication between two triplex computers.

The I/O network is demonstrated in the laboratory using a 15-node circuit-switched network that interfaces with each of the GPCs on 1 to 6 nodes, depending on the GPC redundancy level. The 15 I/O nodes can be configured in the laboratory as global, regional, and local I/O networks to demonstrate various dimensions of the AIPS I/O concept.

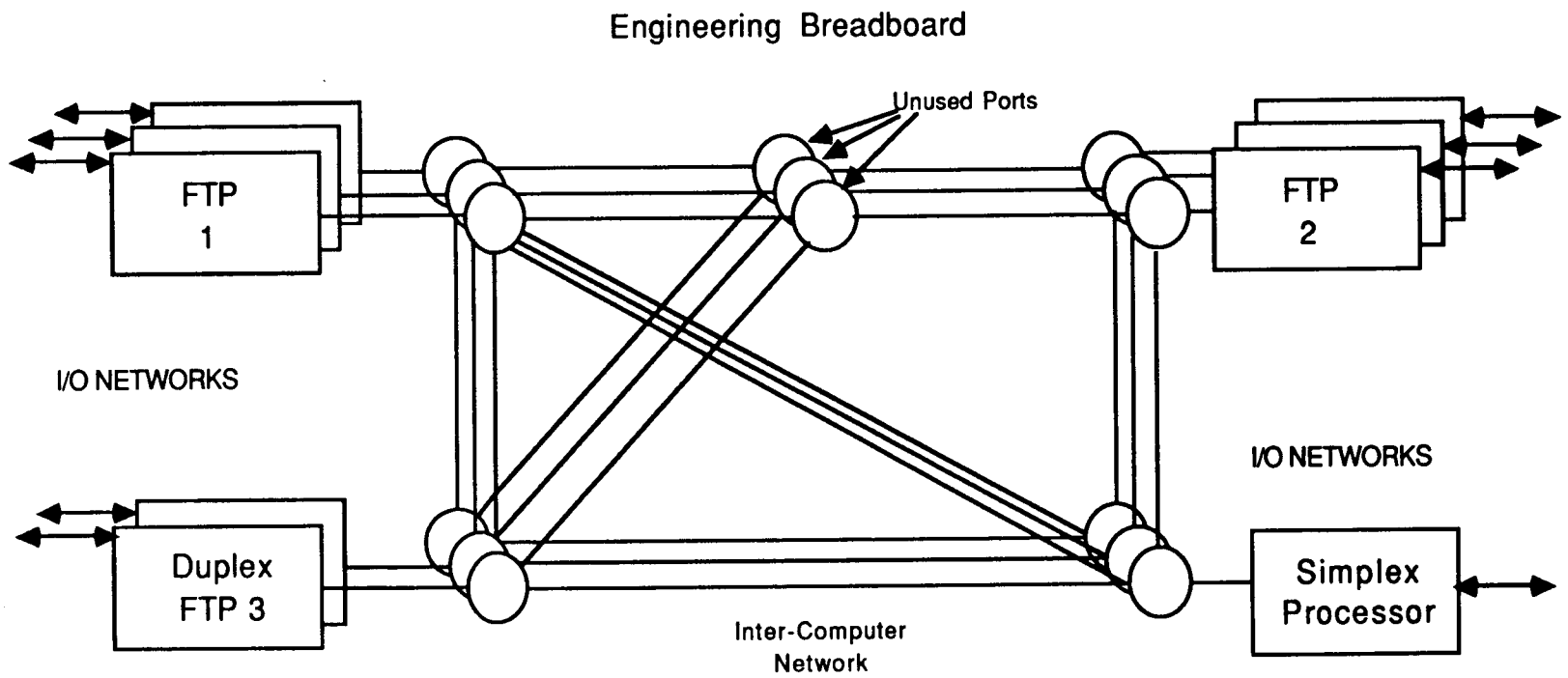


Figure 1. AIPS Distributed Configuration

1.1.1 AIPS Fault Tolerant Processors: Overview

The AIPS Fault-Tolerant Processor (FTP) is a significant enhancement of the CSDL FTP [1,2] developed and used in many centralized real-time applications. The basic FTP architecture has been enhanced to work efficiently and reliably in the distributed information processing environment of AIPS. Figure 2 shows a simplified schematic of a triplex AIPS FTP.

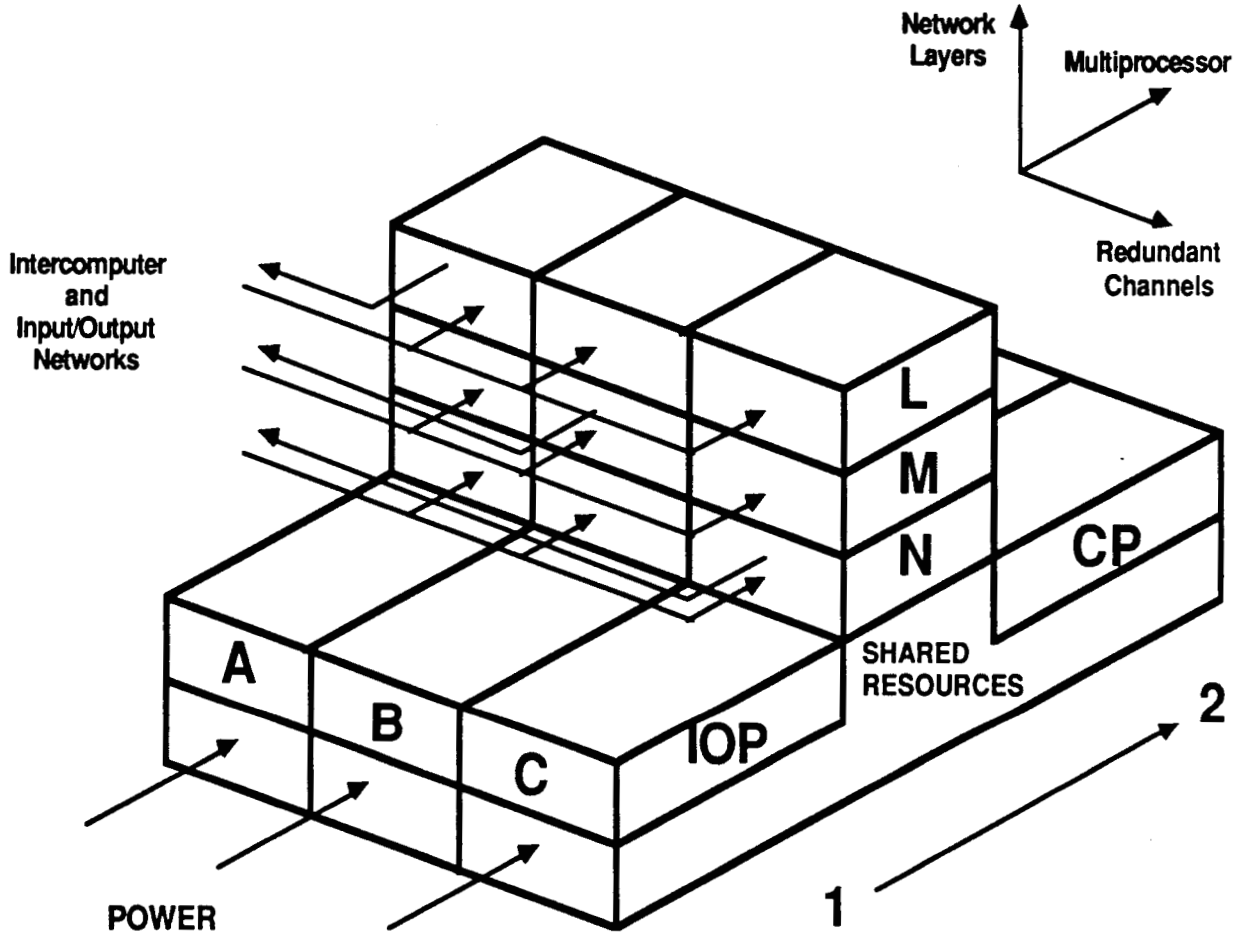


Figure 2. Simplified Schematic of AIPS Fault Tolerant Processor

The first enhancement was the addition of a second processor to each channel so that input/output operations can be performed in parallel with application function computations. The I/O processor (IOP) is used to perform the collection of sensor data and transmission of actuator commands that are typical in traditional real-time systems. It also

performs inter-computer communication, which is a significant burden in a distributed system. The IOP thereby leaves its counterpart, the computational processor (CP), free for calculation and decision-making tasks. This architecture has several important attributes.

First, the IOP and the CP share the data exchange and other hardware responsible for providing fault tolerance in the FTP. This results in increased throughput without additional hardware penalties for fault tolerance. Second, by performing all external communications the IOP can, with appropriate help from System Services Software, completely shield the CP and the applications programs running on it from the complexities of a distributed processing environment. Third, by making all the incoming data congruent before presenting it to the CP, the IOP also shields the applications programs from the redundant nature of the system. These last two attributes greatly simplify the process of developing applications software.

The second major enhancement to the basic FTP architecture was the addition of dedicated hardware interfaces between the FTP and the Input/Output and the Inter-Computer Networks. These are called the Input Output Sequencer (IOS) and the Inter-Computer Interface Sequencer (ICIS), respectively. These interfaces are shared by the CP and the IOP and are shown as blocks L, M, and N in Figure 2. L, M, and N refer to the three layers of the IC network. These interfaces take care of the low level bus protocol and formatting details (typically the physical layer and the data link layer), thus relieving the IOP from having to manage the I/O and IC networks at microsecond time intervals. The ICIS and IOS make it possible to interface the AIPS FTPs to very high speed buses. The possibility of Byzantine faults in the network nodes causing single point FTP failures is quite real, so the design of these interfaces adheres strictly to fault-tolerant systems theoretical principles.

The AIPS FTP architecture is both symmetric and modular. It is symmetric in that either processor can do the work of the other. Since both the CP and the IOP have access to all the external interfaces, the FTP can be operated with only one processor per channel, if desired. For AIPS applications that do not have intensive I/O and/or IC communication, one processor per channel may suffice. Or a combination of one-processor and two-processor GPCs may be used, where sites with little I/O and/or IC activity have only one processor per channel while other sites have two. The architecture is modular in that the number of I/O and IC interfaces per FTP can be varied to fit various processor and network redundancy levels and parallel and partitioned networks.

The AIPS FTP architecture, in combination with the networks, provides a system architecture that is extremely flexible and expandable. It has been designed from the outset to be a distributed architecture utilizing fault-tolerant computers. These qualities can only be appreciated fully if one has faced the task of mating various existing avionics computers,

such as flight control and engine control computers, in order to create an integrated fault-tolerant system.

1.1.2 Fault Tolerant Processor: Functional View

The Fault-Tolerant Processor (FTP) consists of a variable number of redundant processing channels depending on the reliability requirements of the application. The AIPS engineering breadboard FTP is intended to be operated primarily as a triplex, but it provides fail-safe capability when operated as a duplex. A single channel can also be used for non-critical operations as a simplex computer.

Each channel of an FTP consists of three sections: a computational section, an input/output section, and the resources shared between them. The first section contains a Computational Processor (CP), memory, timers and clocks. The second section contains an Input/Output Processor (IOP), memory, timers, and clocks. The shared resources include shared memory, data exchange hardware, timers, and external interface hardware. The redundant processors are tightly synchronized using a fault-tolerant clock. Data is exchanged among redundant channels on point-to-point links. The data exchange hardware also performs the bit-for-bit voting, fault detection and masking functions in a manner that satisfies all the requirements to protect the FTP from Byzantine failures, as described in Appendix A. Apart from redundancy, there are other features that provide hardware and software fault tolerance. These include watchdog timers, processor interlocks, a privileged operating mode, handlers for hardware and software exceptions, and self tests.

A functional view of one channel of an AIPS FTP is shown in Figure 3. The CP and IOP are identical, conventional processor architectures. Interval timers are used for scheduling tasks and maintaining time-out limits on applications tasks (task watchdog timers). A hardware watchdog timer is provided to increase fault coverage and to cause a processor to fail-safe in case of hardware or software malfunctions. This timer resets the processor and disables all of its outputs, if it is not reset periodically. The watchdog timer is implemented independently of the basic processor timing circuitry. A monitor and interlock circuit in each channel provides the capability to disable the outputs of faulty processors. Any two correctly operating processors in a triplex FTP can disable the outputs of the third failed processor through this interlock mechanism. A processor that is failed active is thus prevented from transmitting erroneous data or commands on I/O networks, IC networks, and local I/O devices.

The CP and IOP share resources through a bus that can be accessed by either processor. These shared resources include memory; a system timer; the interchannel data exchange and voting circuits; and interfaces to one or more I/O networks, memory mapped I/O devices, and the IC network.

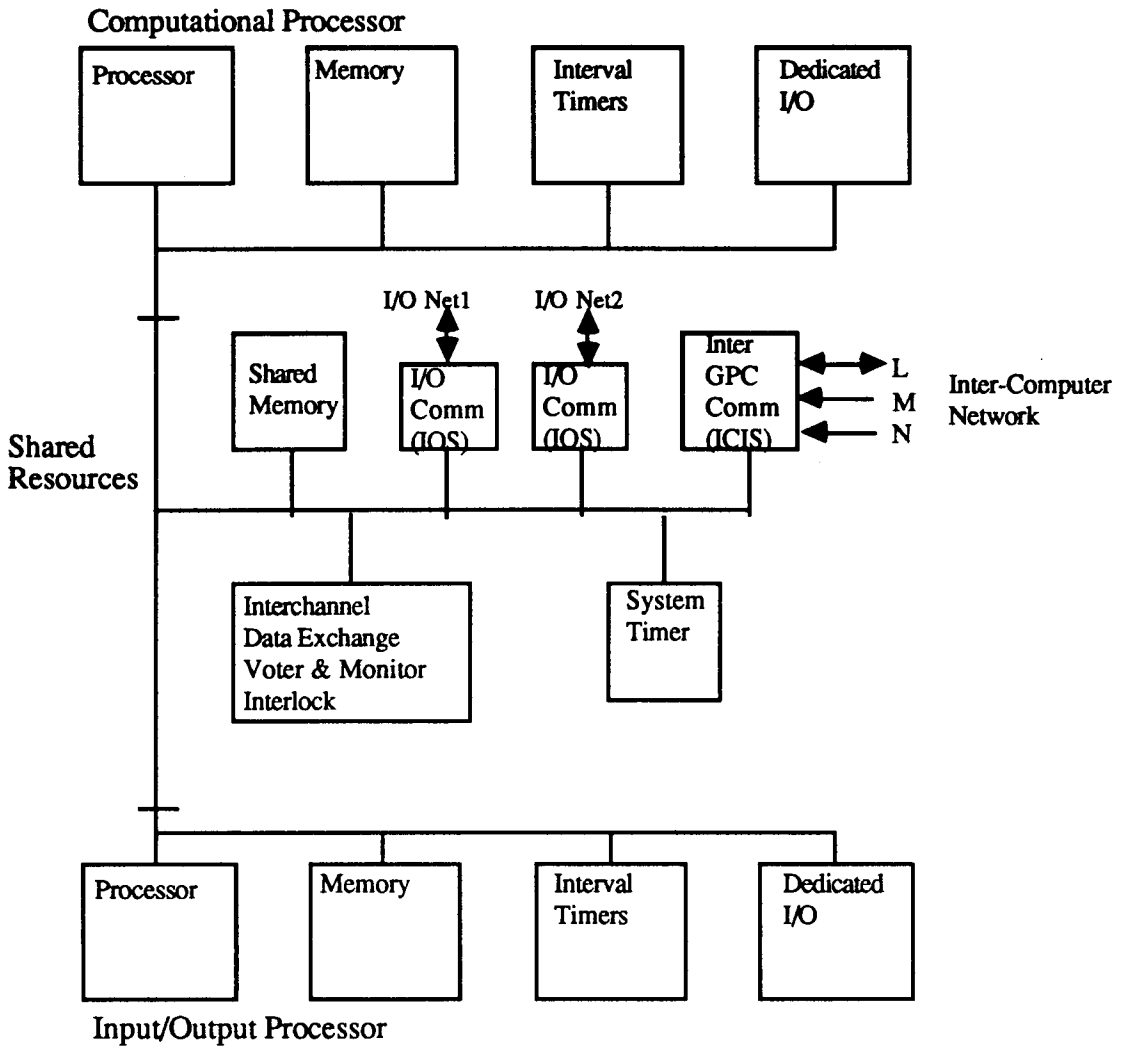


Figure 3. Fault Tolerant Processor Architecture: Functional View (One Channel)

One very important aspect of the FTP architecture is the interconnection hardware between redundant channels. The interchannel data exchange and voting hardware serves three purposes: it provides a path for distributing data in one channel to all other channels; it provides a mechanism for comparing results of the redundant channels; and it provides a path for distributing and comparing timing and control signals such as the fault tolerant clock and external interrupts.

Two types of data exchanges are possible: a simplex exchange or a voted exchange. The simplex exchange is used to distribute copies of data from one channel to all other channels, for example, the value of a sensor that is available in only one channel. A voted exchange, on the other hand, is used to compare and vote results of the redundant channels, for example, an actuator command produced by all three channels which must be voted before the command is actually issued.

The interchannel data exchange and voting circuits appear on the shared bus as a set of registers which include the transmit register, the receive register, and error latches. Data is exchanged between redundant channels one word at a time by writing the word to the transmit register and then reading the result from the receive register. When an exchange is initiated, the transmitter in each channel sends to all channels either its own data (in the case of a voted exchange) or the data available from another channel (in the case of a simplex exchange). Each channel thus receives three copies of the data, which are voted on a bit-by-bit basis. The majority result, which will be the same in all channels even in the presence of an error, is placed in each processor's receive register. The type of exchange (voted or simplex) which will be performed is determined by the particular transmit register that is referenced when the exchange is initiated.

Either type of exchange takes on the order of 5 microseconds in the engineering breadboard version of the AIPS FTP. The hardware is designed to lock out access to the receive register while the exchange is in progress; a processor which tries to read the receive register before the transaction has completed is suspended. As soon as the data becomes available, the processor is released and the register read cycle completes normally. The processor wait is thus transparent to the software.

The same software executes on a redundant FTP as on a simplex channel and application code is written as if it were to operate on a simplex computer. All redundant processors have identical software and execute identical instructions at exactly the same time. This feature of the architecture is carried out in the data exchange hardware and software as well. The data exchange hardware is designed such that all redundant processors execute identical instructions when exchanging data whether it is redundant data to be voted or simplex data being transmitted from one channel to others. Thus, for example, if a simplex exchange is to be made from channel A, all three channels write to their FROM_A register. While the contents of the FROM_A register are transmitted from A, voted, and deposited in the receive registers of all three processors, the contents of the FROM_A registers in channels B and C, which are meaningless, are ignored.

On a routine basis, the internally produced data that needs to be exchanged consists of error information and cross channel comparisons of results for fault detection. These operations can be easily confined to the program responsible for Fault Detection, Identification, and Reconfiguration (FDIR). Voting of the results of the redundant computational processors is performed in hardware by the input/output processors and the system software responsible for the I/O services. Therefore, the remaining pieces of the Operating System software and the applications programs need not be aware of the existence of the data exchange registers. The task scheduler and dispatcher, for example, can view the computational core as a single reliable processor.

Data from other processing sites is received by each IOP on the redundant IC buses,

hardware voted, and then deposited in their respective shared memories. Simplex source data such as that from I/O devices is received by the IOP in one channel to which the I/O device is physically connected. This data is then transmitted to the other two IOPs using the data exchange registers. The congruent data is then deposited in all three shared memories. Thus, the computational processor obtains all external data that has already been processed for errors and source congruency requirements by I/O System Services executing on the I/O processor.

The IOP and CP communicate through the shared memory. The IOP and CP have independent operating systems that cooperate to assure that the data from input devices is made available to the applications programs running in the CP in a timely and orderly fashion. Similarly, the two processors cooperate on the outgoing information so that the output devices receive commands at appropriate times. Hence the CP and IOP actions must be synchronized to some extent. To help achieve this synchronization in software, a hardware feature has been provided which enables one processor to interrupt the other. By writing to a reserved address in shared memory the CP can interrupt the IOP and by writing to another reserved location the IOP can interrupt the CP. Different meanings are assigned to this interrupt by leaving an event code in some other predefined part of the shared memory, before the inter-processor interrupt is asserted.

For routine flow of information in both directions, the shared memory is used without interrupts but with suitable locking semaphores to pass consistent data sets. The interrupts can be used to synchronize this activity as well as to pass time critical data that must meet tight response time requirements. In order to assure data consistency, it is necessary that while one side is updating a block of data the other side does not access that block of data. This has been implemented using software semaphores. Hardware support for semaphores is provided in the form of the test and set instruction.

The architectural approach described above provides several significant operational benefits. The most important of these is the decoupling of the computational and input/output streams of transactions. The computational processor is unburdened from having to do I/O transactions. To the CP, all I/O appears memory mapped including not only I/O devices but also all other computers in the system. That is, each sensor, actuator, switch, computer, etc., with which the FTP interfaces can be addressed simply by reading or writing words in the shared memory.

1.2 AIPS System Software

The AIPS system software, as well as the hardware, has been designed to provide a virtual machine architecture that hides hardware redundancy, hardware faults, multiplicity of resources, and distributed system characteristics from the applications programmer. Section 1.2.1 discusses the approach used for the AIPS system software design. Section 1.2.2 is a high level description of the system services that are provided for AIPS users.

1.2.1 AIPS Software Design Approach

The approach used to design the AIPS system software is part of the overall AIPS system design methodology. An abbreviated form of this system design methodology is shown in Figure 4. This methodology began with the application requirements and eventually led to a set of architectural specifications. The architecture was then partitioned into hardware and software functional requirements. This report documents the design approach used for Local System Services software, beginning with the functional requirements and proceeding through detailed specifications.

Hardware and software for the AIPS architecture is being designed and implemented in two phases. The first phase is the centralized AIPS configuration. The centralized AIPS architecture, as shown in Figure 5, is configured as one triplex Fault Tolerant Processor (FTP), an Input/Output network and the interfaces between the FTP and the network, referred to as input/output sequencers (IOS). The laboratory demonstration of the input/output network consists of 15 circuit-switched nodes which can be configured as multiple local I/O networks connected to the triplex GPC. For example, the I/O network may be configured as one 15-node network, as shown in Figure 5, or as three 5-node networks. The software building blocks that have been designed and implemented for the AIPS centralized architecture include local system services and I/O system services. The following subsection 1.2.2 gives an overview of all the AIPS software building blocks. The remainder of the document, Sections 2 through 6, focuses on the functional design and detailed specification of the Local System Services.

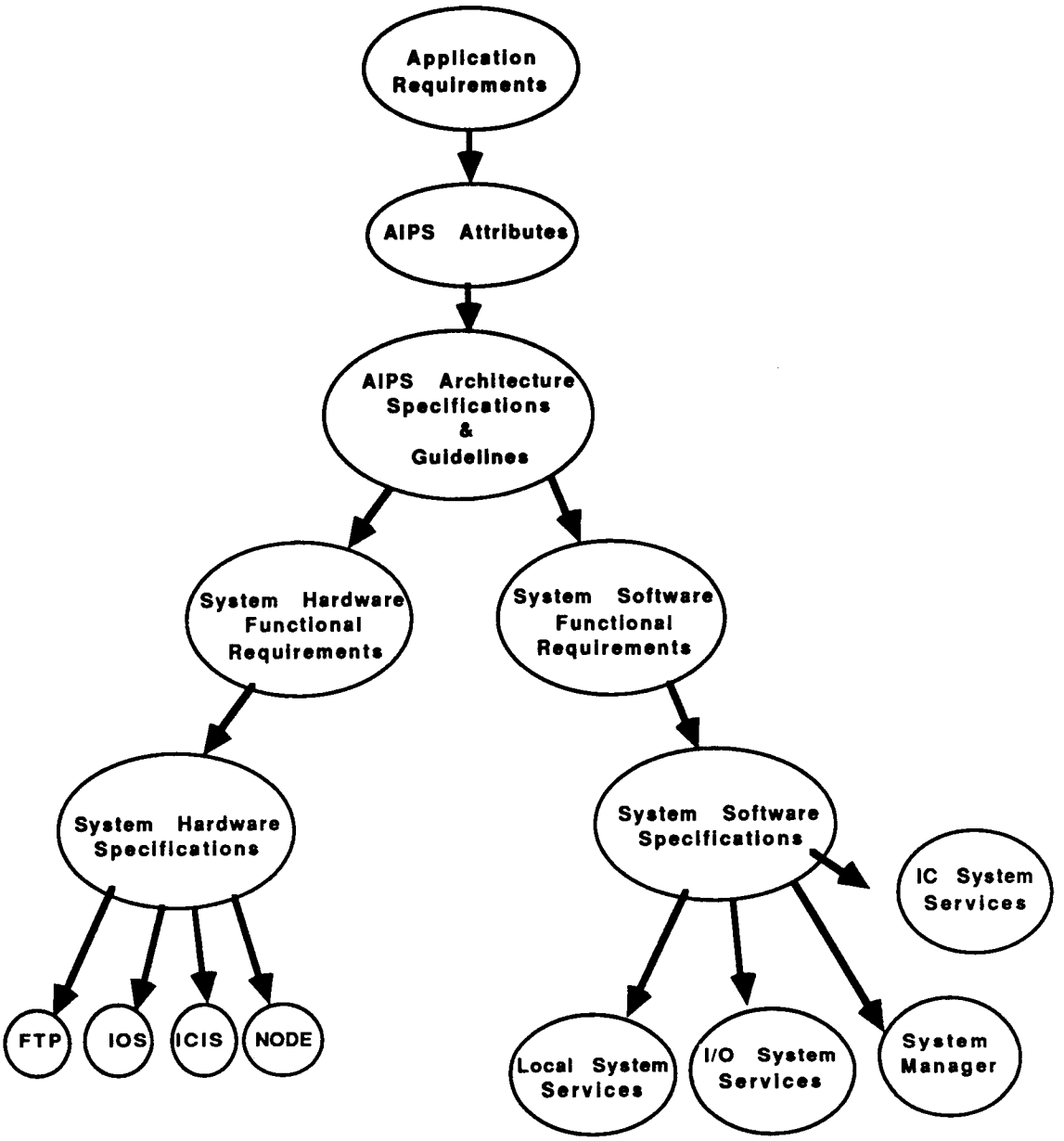


Figure 4. AIPS System Design Approach

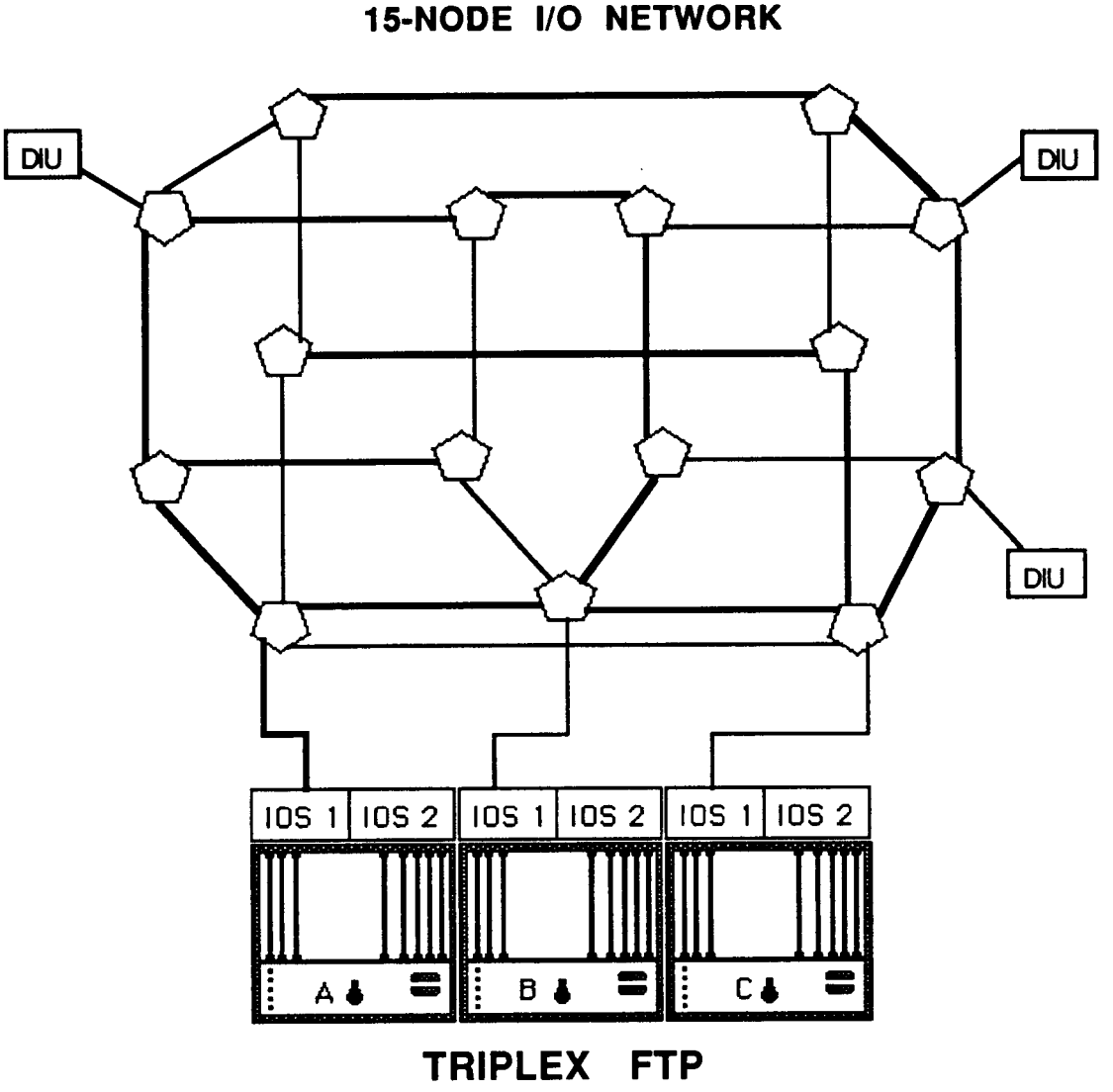


Figure 5. Centralized AIPS Configuration

1.2.2 AIPS System Software Overview

As shown in Figure 6, AIPS system software provides the following AIPS System Services: local system services, communication services, system management, and I/O system services. The system software is being developed in Ada. System services are modular and partitioned naturally according to hardware building blocks. The distributed AIPS configuration includes all the services. Versions of the system software for specific applications can be created by deleting unused services from this superset. The System Manager functions reside on only one GPC, but all functions of the System Manager are not necessarily on the same GPC. The other system services are replicated in each GPC. A brief description of each of the services follows.

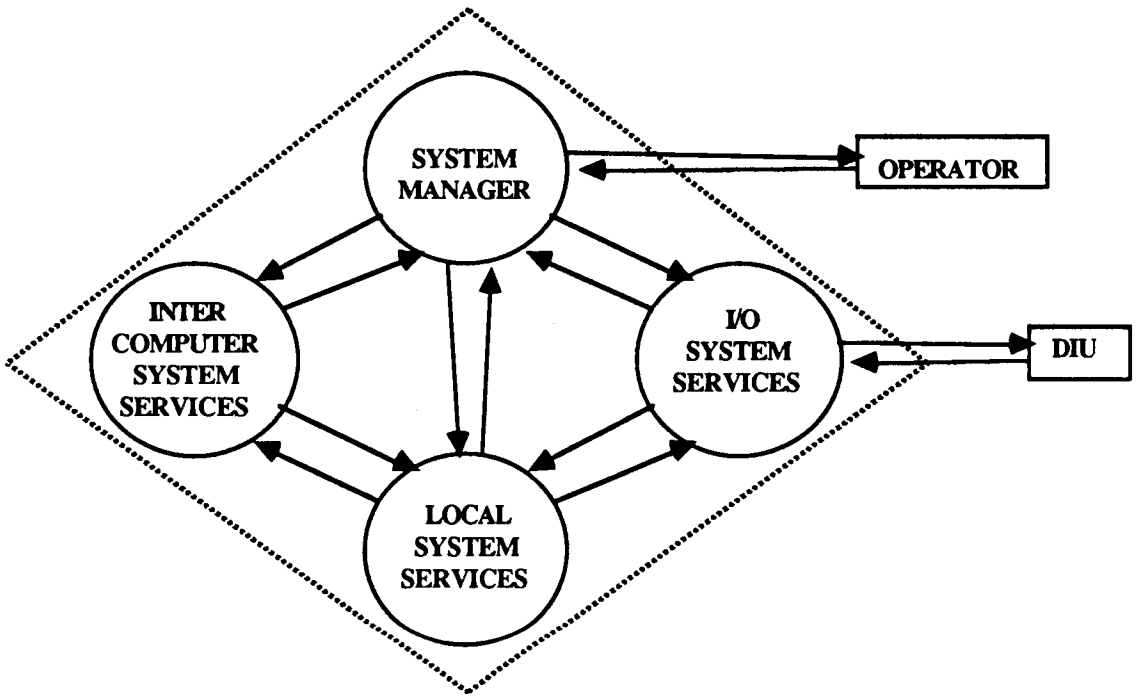


Figure 6. Top Level View Of System Services

1.2.2.1 Local System Services

The local system services provided in each GPC are: GPC initialization, real-time operating system, local resource allocation, local GPC Fault Detection, Isolation, and Reconfiguration (FDIR), GPC status reporting, and local time management (see Figure 7).

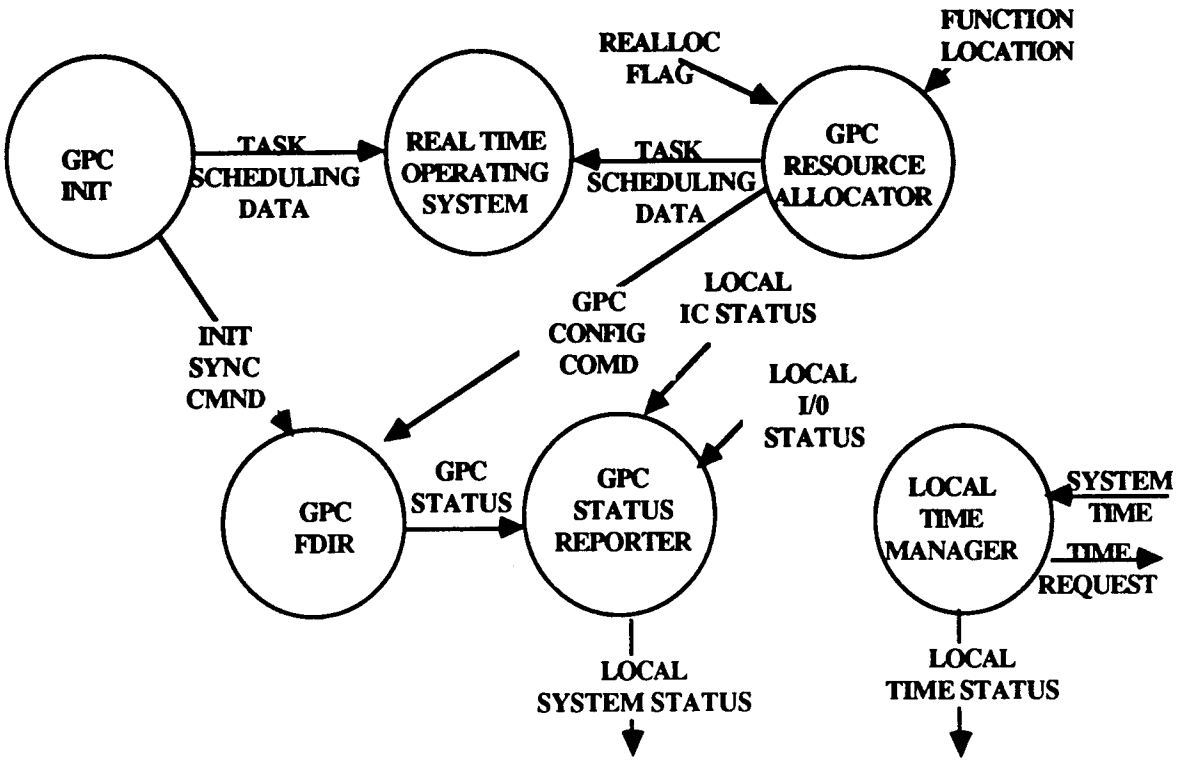


Figure 7. Local System Services

The function of GPC initialization is to bring the GPC to a known and operational state from an unknown condition (cold start). GPC initialization synchronizes the CPs, synchronizes the IOPs and resets or initializes the GPC hardware and interfaces (interval timers, real time clock, interface sequencers, DUART, etc.) It makes the hardware state of the redundant channels congruent by alignment of memory and control registers. It then activates the system baseline software that is common to every GPC.

The AIPS real-time operating system supports task execution management, including scheduling according to priority, time and event occurrence, and is responsible for task dispatching, suspension and termination. It also supports memory management, software exception handling, and intertask communication between companion processors (IOP and CP). The AIPS operating system resides on every CP and IOP in the system. It uses the vendor-supplied Ada Run Time System (RTS), and includes additional features required for the AIPS real-time distributed operating system.

The GPC resource allocator coordinates and determines responsibility for any global or migratable functions from the system resource manager. It also monitors commands from the system resource manager to start or stop any function.

The GPC status reporter collects status information from the local functions, the local GPC FDIR, the IC system services and the I/O system services. It updates its local data base and disseminates this status information to the system manager.

GPC FDIR has the responsibility for detecting and isolating hardware faults in the CPs, IOPs, and shared hardware. It is responsible for synchronizing both groups of processors in the redundant channels of the FTP and for disabling outputs of failed channel(s) through interlock hardware. After synchronization, all CPs will be executing the same machine language instruction within a bounded skew, and all IOPs will be executing the same machine language instruction within a bounded skew. GPC FDIR logs all faults and reports status to the GPC status reporter. It is responsible for the CPU hardware exception handling and downmoding/upmoding hardware in response to configuration commands from the system manager. It is also responsible for transient hardware fault detection and for running low priority self tests to detect latent faults. This redundancy management function is transparent to the application programmer.

The local time manager works in cooperation with the system time manager to keep the local real time initialized and consistent with the universal time. It is also responsible for providing time services to all users.

Sections 2 through 6 describe the Local System Services functional requirements and design and detailed specifications. Each of these sections describes one of the Local System Services functions. Section 7 concludes with a summary of results and suggestions for future work in this area.

1.2.2.2 Inter-Computer System Services

The inter-computer system services provide two functions: (1) inter-computer user communication services, that is, communication between functions not located in the same GPC, and (2) inter-computer network management (Figure 8).

The IC user communication service provides local and distributed inter-function communication which is transparent to the application user. It provides synchronous and asynchronous communication, performs error detection and source congruency on inputs, and records and reports IC communication errors to IC network managers. Inter-computer communication can be done in either point to point or broadcast mode and is implemented in each GPC.

The IC network manager is responsible for the fault detection, isolation and reconfiguration of the network. The AIPS distributed configuration consists of three identical, independent IC network layers which operate in parallel to dynamically mask faults in a single layer and provide reliable communication. There is one network manager for each network layer. However, the three network layer managers do not need to reside in the same GPC. They are responsible for detecting and isolating hardware faults in IC nodes and links and for

reconfiguring their respective network layer around any failed elements. The network manager function is transparent to all application users of the network.

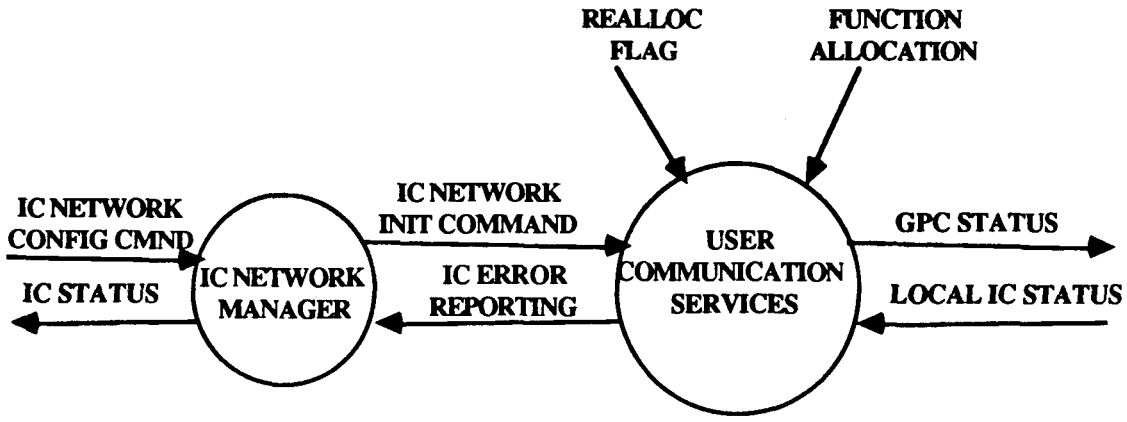


Figure 8. Inter-Computer System Services

1.2.2.3 System Manager

The system manager is a collection of system level services including the applications monitor, the system resource manager, the system fault detection, isolation and reconfiguration (FDIR), and the system time manager (Figure 9).

The applications monitor interfaces with the applications programs and the AIPS system operator. It accepts commands to migrate functions from one GPC to another, to display system status, to change the state of the system by requesting a hardware element state change, and to convey requests for desired hardware and software configurations to the system resource manager.

The system resource manager allocates migratable functions to GPCs. This involves the monitoring of the various triggers for function migration such as failure or repair of hardware components, mission phase or workload change, operator or crew requests and timed events. It reallocates functions in response to any of these events. It also designates managers for shared resources and sets up the task location data base in each GPC.

The system fault detection, isolation and reconfiguration (FDIR) is responsible for the collection of status from the inter-computer (IC) network managers, the I/O network managers, and the local GPC redundancy managers. It resolves conflicting local fault isolation decisions, isolates unresolved faults, correlates transient faults, and handles processing site failures.

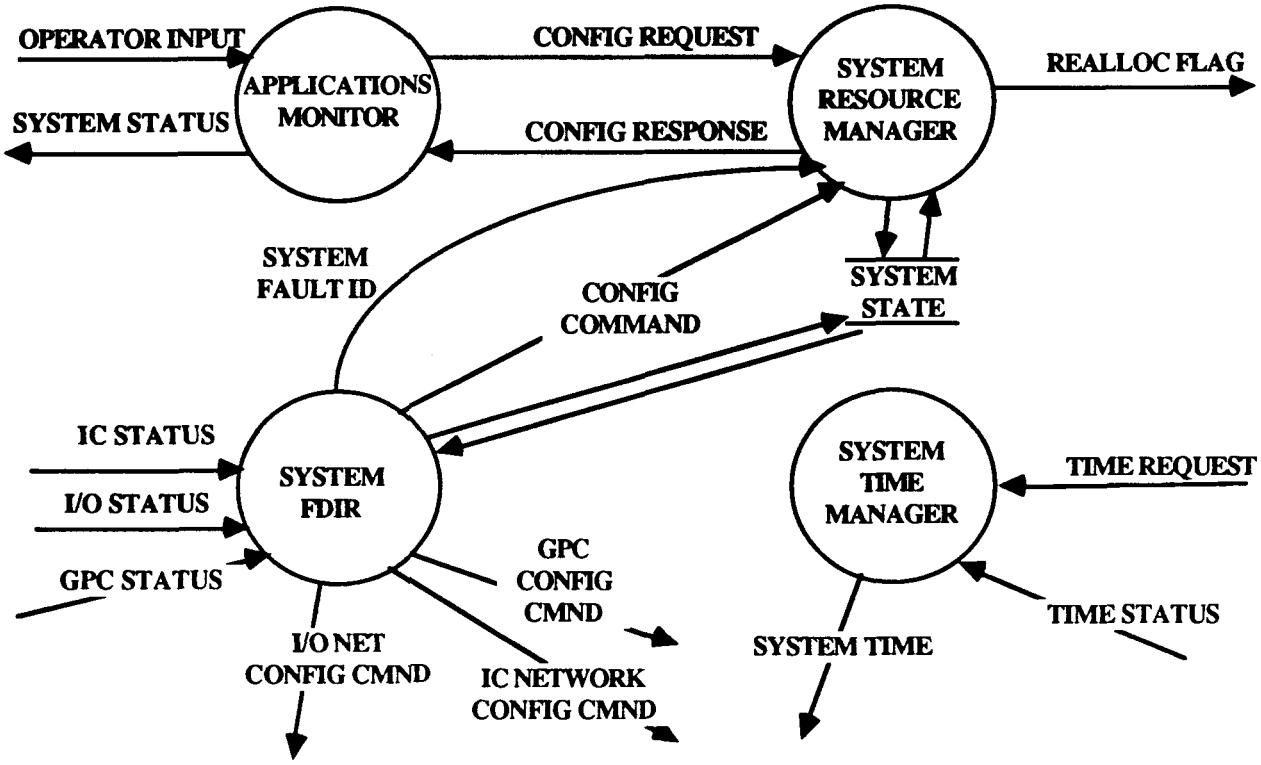


Figure 9. System Manager

The system time manager, in conjunction with the local time manager on each GPC, has the job of maintaining a consistent time across all GPCs. The system time manager indicates to the local time manager when to set its value of time. It also sends a periodic signal to enable the local time manager to adjust its time to maintain consistency with an external time source such as the GPS Satellites or an internal source such as the real time clock in the GPC which hosts the system time manager software.

1.2.2.4 I/O System Services

The I/O system services provide efficient and reliable communication between the user and external devices (sensors and actuators). The I/O system services software is also responsible for the fault detection, isolation and reconfiguration of the I/O network hardware and GPC/network interface hardware (input/output sequencers).

I/O system services is made up of three functional modules: I/O user interface, I/O communication management and the I/O network manager (Figure 10).

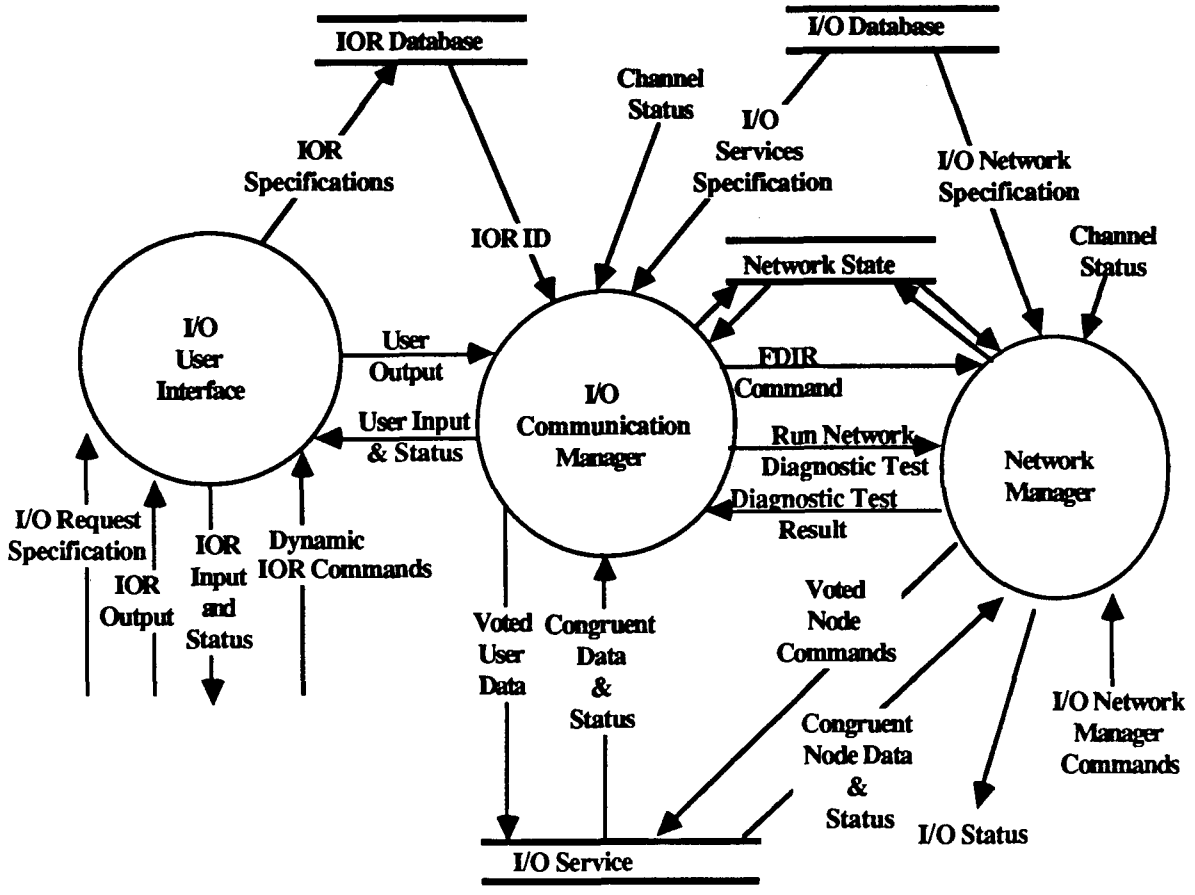


Figure 10. I/O System Services

The I/O user interface provides a user with read/write access to I/O devices or Device Interface Units (DIUs), such that the devices appear to be memory mapped. It also gives the user the ability to group I/O transactions into chains and I/O requests, and to schedule I/O requests either as periodic tasks or on demand tasks. A detailed description of the I/O user interface is provided in [3].

The I/O communication manager provides the functions necessary to control the flow of data between a GPC and the various I/O networks used by the GPC. It also performs source congruency and error detection on inputs, voting on all outputs, and reports communication errors to the I/O Network Manager. It is also responsible for the management of the I/O request queues. A detailed description of the I/O communication manager is provided in [3].

The I/O Network Manager is responsible for detecting and isolating hardware faults in I/O nodes, links, and interfaces and for reconfiguring the network around any failed elements. The network manager function is transparent to all application users of the network. A detailed description of the I/O Network Manager is provided in [4].

2.0 GPC INITIALIZATION

It is the responsibility of the GPC INIT module to bring the GPC to a known and operational state at startup and to initialize the system software tasks and the application procedures and/or tasks.

2.1 GPC Initialization Functional Requirements and Design

The function of GPC initialization is to bring the GPC to a known and operational state. If the GPC has a redundancy level greater than a simplex, the initialization software module is responsible for controlling the transition from an unknown, uninitialized state of the FTP hardware to a state such that all available channels are operating in an instruction-synchronous manner. Furthermore, the module will insure that the channels operating in such a synchronous manner will have "aligned", or "bit-for-bit identical", states of all redundant hardware elements (e.g., RAM and interval timers). Once this instruction-synchronous, hardware-aligned status is established for the FTP, it will continue to operate in a synchronous mode until a fault forces a channel out of synch. (This statement assumes all external data to the FTP is made congruent across channels by the appropriate I/O software.)

Upon reset, a processor will vector to the start-up entry point of the operating system. Standard initialization operations must be performed at this point:

- Initialization of standard hardware such as interval timers, real time clock, and DUARTs
- Exception vector initialization
- Interrupt handler initializations
- Software initialization (Ada "elaboration", including initialization and activation of tasks)
- Execution of power-on self tests

After the standard initialization functions are performed, the FTP specific initialization is performed. The redundant channels are synchronized to the instruction level. Since each channel has two processors, each processor is synchronized with the other processors of its type (i.e., CPs or IOPs). In addition, its companion must be synchronized with other processors of the companion's type, i.e., either both processors of a channel are good or they are both failed. Thus, processor activity within each channel must be coordinated before and during the synchronization process. The processors within each channel synchronize with one another to the extent that all of the intra-channel processors finish

their pre-synch initialization and signal their readiness to operate in synch with their corresponding inter-channel processors. After instruction-level synchronization, the hardware state of the redundant channels is made congruent by alignment of memory and control registers. The channel synchronization and alignment functions are performed within the SYNC software module which is described in Section 4.2.2.2.

After channel synchronization the power-on self tests are performed in order to check for any hardware faults in memory, error latches, voters, real-time clock and monitor interlock hardware. After the system is operational the self tests continue to run in background to check for any latent faults, but they are run in their entirety at system startup. (Execution of the self tests at power-on is optional in the laboratory engineering model.)

Next the operating system tasks are scheduled. These tasks include the redundancy management sequencer task, the GPC Status Reporter/CRT display tasks, and the I/O system services tasks. Finally, the application tasks are scheduled and the self test loop is entered. The self tests execute at the lowest priority when the system would otherwise be idle.

2.2 GPC Initialization Software Specifications

Process Name: Main Program

Inputs:
DUART, interval timers, real time clock
Unsynchronized Processors, Channels
Non Aligned Volatile memory, control registers
Uninitialized Task Control Blocks, Tasks Stacks
Uninitialized interrupt handlers, exception handlers
Uninitialized IOS, ICIS
Unscheduled Tasks

Outputs:
An operational FTP in a known synchronous state
All Tasks scheduled.

Notes: This process calls the SYNC process, Section 4.2.2.2

Description:

It is the responsibility of the GPC Init function to bring the FTP to an operational mode at startup and to initialize the application procedures and/or tasks. GPC Init resides on both the IOP and CP of each processing site. Since Ada requires that the process which initially executes be named 'Main Program', the GPC Init function is so named in the Ada source code. Figure 11 is a diagram of the control flow of the GPC Init function. First the

DUARTs, interval timers, real time clock and the interrupt and exception vectors are initialized. Then the elaboration of all system packages and tasks is completed. This includes initialization of every task control block and task stack area. Next the available channels of the FTP are synchronized and all volatile memory, control registers, interval timers, and real time clock are aligned in the synchronized channels. This is done by the SYNC process, Section 4.2.2.2, in both the CP and IOP processors.

Once the FTP is running synchronously, a complete iteration of the self tests is performed. These tests, which detect and identify faults in memory, error latches, voters, the real-time clock and the monitor interlock, are described in Section 4.2.1.3. The difference in the power-on self tests and the background self tests is that the power-on tests are not interrupted until an entire iteration has been completed.

Next the Local System Services sequencer task, the FDIR/Time Manager, is scheduled. This task is responsible for the execution of the periodic Local System Service functions: Fast FDIR, Transient FDIR and the Local Time Manager. The FDIR/Time Manager is a high priority, periodic task and presently executes every 40 milliseconds. The periodicity of this task is dependent on the particular application running on the GPC; it may be different for each GPC site. The FDIR/Time Manager and the three processes it controls are discussed in Sections 4.2.1, 4.2.2 and 6.1.1.2.3.

Finally, the GPC Status Reporter display tasks are scheduled. There are three tasks for CRT display and one task for Macintosh display. Like the main task, these tasks are of the lowest priority and are time-sliced with each other and the main program. The GPC Status Reporter/display tasks are described in Section 5.2.2. The time slice interval is equal to the period of the fastest periodic task in the system.

At this point all of the Local System Services tasks have been scheduled and the FTP is in a known and operational state. I/O System Services tasks and any application tasks may now be scheduled and/or application procedures called.

The main program finally enters the self test loop where it executes continually at the lowest priority the self tests in order to uncover latent faults. The self tests are described in Section 4.2.1.3. If needed, applications may call procedures within this loop also.

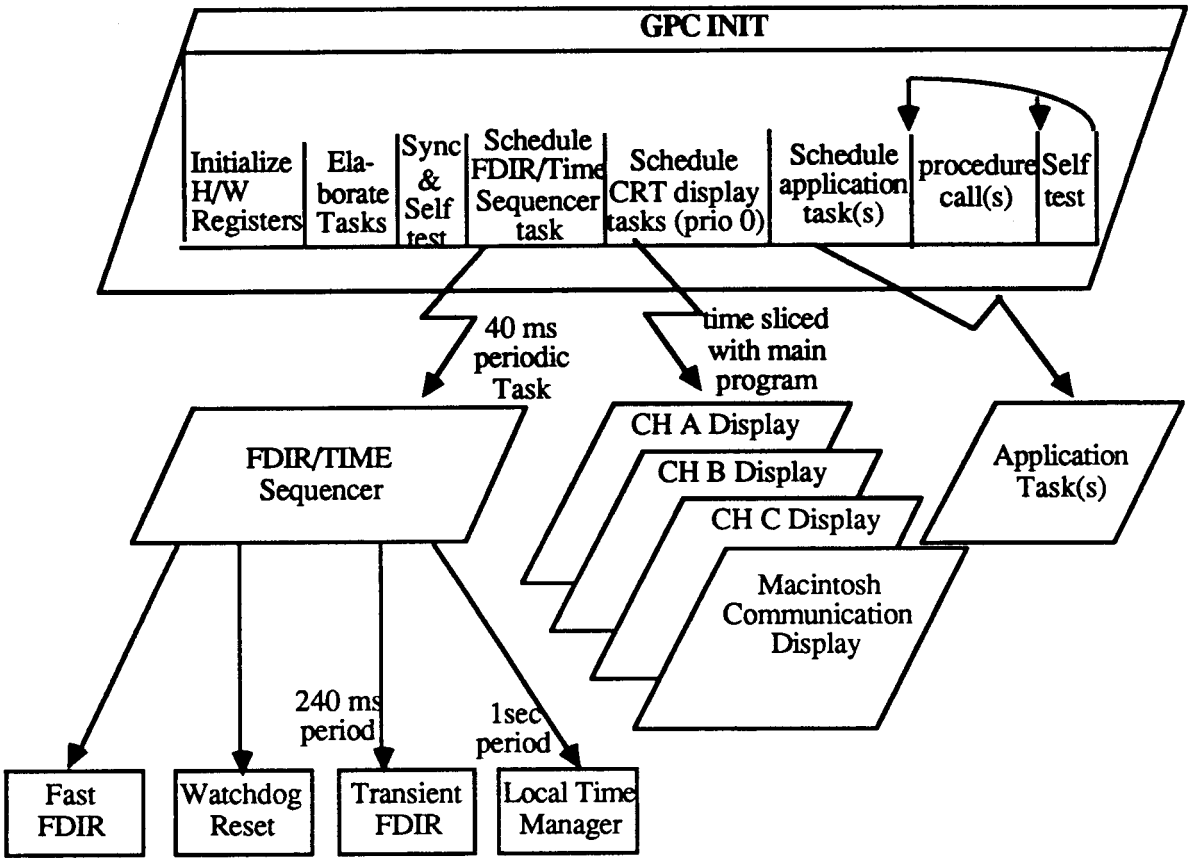


Figure 11. GPC INIT Control Flow Diagram

3.0 REAL TIME OPERATING SYSTEM

The foundation of the system software for AIPS is a real-time, multi-tasking operating system providing mechanisms for task scheduling, inter-task communication, memory management, and interrupt handling. The AIPS operating system consists of the vendor-supplied Ada Run Time System (RTS) along with those extensions needed to implement the functions given below. The extensions are written in Ada with time critical sections done in assembly language to reduce system overhead. The AIPS operating system resides on every IOP and CP.

3.1 Real Time Operating System Functional Requirements

The AIPS operating system provides the basic system services necessary to support the other application software tasks (see Figure 12). These services include:

1. Task execution management, including scheduling according to priority, time and event occurrence; dispatching (context switching); task suspension and termination.
2. Memory management, including global data allocation, local data allocation/deallocation and shared data access routines (protected read and write).
3. Intertask communication, including both synchronous and asynchronous (mailbox) communication.
4. Software exception handling.

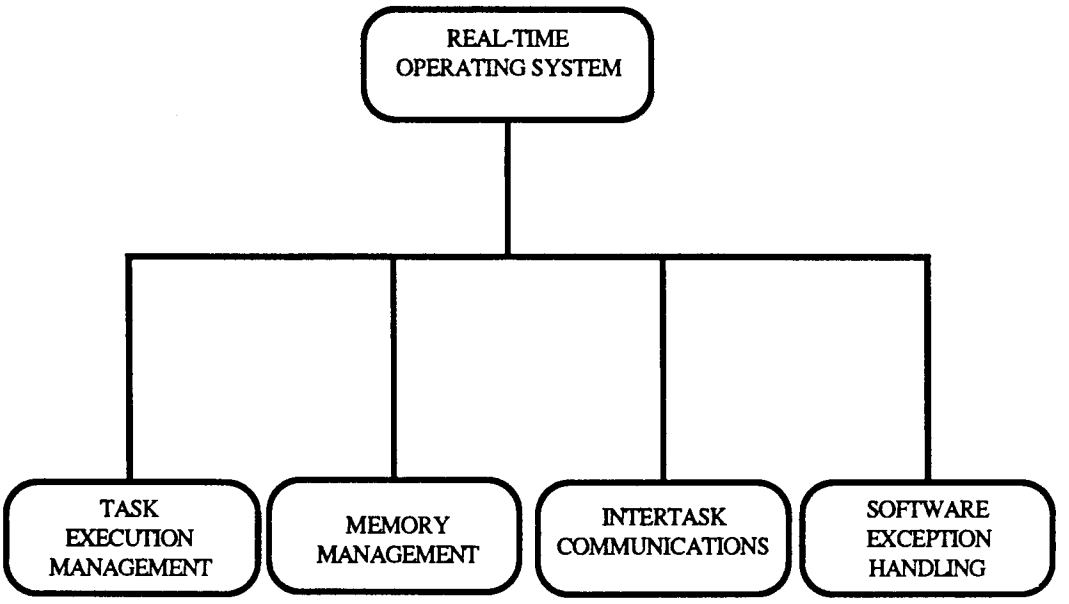


Figure 12. AIPS Real Time Operating System

3.1.1 Task Execution Management

The basic execution unit is the Ada "task". The management of task execution includes scheduling according to priority, time and event occurrence, dispatching, task suspension and termination. A static priority is associated with each task. Task execution order is according to priority and proceeds in a run-until-blocked mode until one of the following conditions occurs:

1. task completion,
2. self suspension,
3. preemption by hardware interrupt,
4. explicit scheduling of higher priority task, or
5. time slicing of equal priority task.

A task is capable of scheduling itself or other tasks according to time (one-shot or periodic) or upon the occurrence of a software defined event. The option also exists to bracket the region of a task's periodicity by time or events; i.e. provide start/stop times or events.

The RTS requirements of Ada provide minimal user control of task scheduling with the Ada rendezvous and the "delay" statement. Scheduling by event is not an Ada requirement. So extensions to the Ada RTS are necessary in order for the task to schedule itself or another task cyclically, as the result of an event, at an absolute time, immediately, or to deschedule a task (remove all scheduling requirements).

AIPS requires two additional functions that are not provided by Ada RTS vendors: (1) preserving the data exchange receiver register for each task during a context switch, and (2) voting the program counter (PC) after every interrupt. Preserving the data exchange receiver register is necessary because a data exchange requires two instructions, a write to the transmitter followed by a read from the receiver, and an interrupt could occur after the write but before the read. When a task is suspended, therefore, the receiver value must be saved in the task control block; when a task is resumed, this value must be restored to the receiver. In the laboratory demonstration implementation, which uses a Motorola 68010 processor with an 8 MHz clock, restoring the receiver register takes approximately 5 μs.

Voting the PC after every interrupt is necessary because an interrupt could bring unsynchronized channels back into sync, thus masking the faulty condition. The state of each channel at the time of the interrupt can be determined by voting the PC; when a channel's PC is different from the majority value the operating system sets a flag for the FDIR task. This procedure ensures that a fault is detected as early as possible. In the laboratory demonstration implementation, which uses a Motorola 68010 processor with an 8 MHz clock, the PC check takes approximately 43 μs, of which 5 μs is used for the data exchange.

3.1.2 Memory Management

There are two types of memory in the FTP, local memory and shared memory. Local memory is memory on the private bus and can be accessed only by a single processor. Shared memory is memory on the shared bus and can be accessed by both processors in a channel.

Local memory is used for task execution. The operating system is responsible for allocating and deallocating all local memory space from heaps and stacks. The user must be able to specify the amount of local memory to be allocated for tasks and data objects. Also, routines are provided for the controlled access (protected read and writes) to data objects shared between multiple application tasks resident on the same processor.

Shared memory can be accessed by tasks resident on different processors and by several tasks resident on the same processor. Routines are provided for controlled access (protected read and writes) to data objects shared between multiple application tasks resident on different processors.

3.1.3 Intertask Communication

Local intertask communication is communication that takes place between tasks executing on the same processor. The operating system supports two methods of local inter-task communication: synchronous and asynchronous. Synchronous communication requires the communicating tasks to be at a specified synchronization point and is implemented in Ada by the "rendezvous". In local asynchronous inter-task communication, the communicating tasks are not required to be at a specified synchronization point; rather the communication is via data "mailboxes" using the local memory controlled access routines.

Remote inter-task communication is communication that takes place between tasks executing on different processors. The operating system supports two types of remote inter-task communication: remote task release and asynchronous "mailbox" communication. Remote task release means that a task running on one processor can start or release a task on the other processor. The remote inter-task mailbox communication is supported by using the shared memory controlled access routines.

3.1.4 Software Exceptions

Software exceptions not explicitly handled by the applications tasks are intercepted by the operating system and the task is purged.

3.2 Real Time Operating System Software Specifications

3.2.1 Task Execution Management Process Descriptions

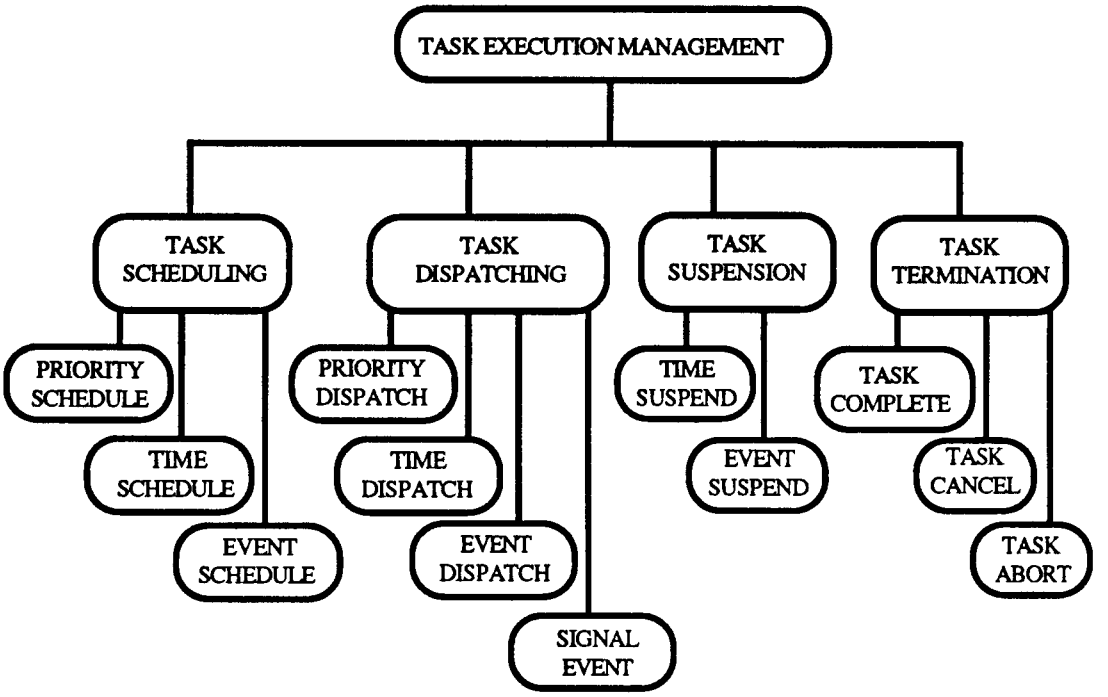


Figure 13. Task Execution Management

3.2.1.1 Process Name: PRIORITY_SCHEDULE

Inputs:

- GENERAL_TASK_ID

Outputs:

- GENERAL_TASK_ID.TASK_NOT_WAITING

Implementation Requirements:

- Initiation:
- On demand

Process Description:

If task to be scheduled is not waiting at a scheduling synchronization point, the TASK_NOT_WAITING indicator is set and an immediate return to the caller is performed.

Otherwise, if priority of task to be scheduled (GENERAL_TASK_ID.PRIORITY) is greater than the priority of the active task (ACTIVE_TASK_ID.PRIORITY), execution of the scheduled task is initiated via the PRIORITY_DISPATCH process; otherwise, the task being scheduled is placed on the ready queue according to its priority and a return to the active task is performed.

3.2.1.2 Process Name: TIME_SCHEDULE

Inputs:

- GENERAL_TASK_ID
- GENERAL_TASK_ID.REPETITION_TIME
- GENERAL_TASK_ID.COMPLETION_TIME
- GENERAL_TASK_ID.COMPLETION_EVENT
- INITIATION_TIME

Outputs:

- None

Implementation Requirements:

- Initiation:
  - On demand

Process Description:

Task to be scheduled (GENERAL_TASK_ID) is placed on the time queue according to the specified start time (INITIATION_TIME).

If task is initial task on time queue, the time queue interval timer is updated.

3.2.1.3 Process Name: EVENT_SCHEDULE

Inputs:

- GENERAL_TASK_ID
- GENERAL_TASK_ID.COMPLETION_TIME
- GENERAL_TASK_ID.COMPLETION_EVENT
- EVENT

Outputs:

- None

Implementation Requirements:

- Initiation:
  - On demand

Process Description:

Task to be scheduled (GENERAL_TASK_ID) is placed on the event queue indicated by the specified event(EVENT).

3.2.1.4 Process Name: PRIORITY_DISPATCH

Inputs:

- None

Outputs:

- None

Implementation Requirements:

- Initiation:
  - ready task priority > active task priority
  - active task suspension
  - active task termination

Process Description:

If a task is active (ACTIVE_TASK_ID /=null), it is preempted and placed on the ready queue according to its priority (ACTIVE_TASK_ID.PRIORITY).

Execution of the highest priority task on the ready queue is initiated.

3.2.1.5 Process Name: TIME_DISPATCH

Inputs:

- None

Outputs:

- TIME_TASK_ID.TASK_NOT_WAITING

Implementation Requirements:

- Initiation:
  - Time queue interval timer interrupt

Process Description:

If task at the head of the time queue (TIME_TASK_ID) is not waiting at a scheduling synchronization point, the TASK_NOT_WAITING indicator is set and dispatching of the task is not performed.

Otherwise, if priority of time task (TIME_TASK_ID.PRIORITY) is greater than priority of active task (ACTIVE_TASK_ID.PRIORITY), the PRIORITY_DISPATCH process is invoked; otherwise, the time task is placed on the ready queue according to its priority.

If the time task is periodic and the current time is less than TIME_TASK_ID.COMPLETION_TIME or TIME_TASK_ID.COMPLETION_EVENT has not occurred, then the time task is replaced on the time queue according to its period (TIME_TASK_ID.REPETITION_TIME). Otherwise, the task is removed from the time queue.

The time queue interval timer is set to interrupt for the task now at the head of the time queue.

3.2.1.6 Process Name: SIGNAL_EVENT

Inputs:

- EVENT

Outputs:

- GENERAL_TASK_ID

Implementation Requirements:

- Initiation:
  - On occurrence of event

Process Description:

All tasks (GENERAL_TASK_ID) on event queue indicated by the specified event (EVENT) are dispatched via the EVENT_DISPATCH process.

3.2.1.7 Process Name: EVENT_DISPATCH

Inputs:

- GENERAL_TASK_ID

Outputs:

- GENERAL_TASK_ID.TASK_NOT_WAITING

Implementation Requirements:

- Initiation:
  - Invoked by SIGNAL_EVENT process

Process Description:

If event task (GENERAL_TASK_ID) is not waiting at a scheduling synchronization point, the TASK_NOT_WAITING indicator is set and dispatch of the task is not performed.

Otherwise, if priority of event task (GENERAL_TASK_ID.PRIORITY) is greater than priority of active task (ACTIVE_TASK_ID.PRIORITY), the PRIORITY_DISPATCH process is invoked; otherwise, the event task is placed on the ready queue according to its priority.

If current time is greater than GENERAL_TASK_ID.COMPLETION_TIME or GENERAL_TASK_ID.COMPLETION_EVENT has occurred, the event task is removed from the specified event queue. Otherwise, the task is left on the queue.

3.2.1.8 Process Name: TIME_SUSPEND

Inputs:

- DELAY_TIME

Outputs:

- None

Implementation Requirements:

- Initiation:
  - On demand
- This process is implemented in Ada by the "delay" statement.

Process Description:

The active task (ACTIVE_TASK_ID) is preempted and the TIME_SCHEDULE process is invoked to place task on the time queue for the specified delay time (DELAY_TIME).

Ready task execution is initiated by invoking the PRIORITY_DISPATCH process.

3.2.1.9 Process Name: EVENT_SUSPEND

Inputs:

- EVENT

Outputs:

- None

Implementation Requirements:

- Initiation:
  - On demand

Process Description:

The active task (ACTIVE_TASK_ID) is preempted and the EVENT_SCHEDULE process is invoked to place task on the event queue indicated by the specified event (EVENT).

Ready task execution is initiated by invoking the PRIORITY_DISPATCH process.

3.2.1.10 Process Name: TASK_COMPLETE

Inputs:

- None

Outputs:

- None

Implementation Requirements:

- Initiation:
  - On demand
- This process is implemented in Ada by the normal task completion processing of the run time system.

Process Description:

The active task (ACTIVE_TASK_ID) is deactivated. The ready task (READY_TASK_ID) is initiated by invoking the PRIORITY_DISPATCH process.

3.2.1.11 Process Name: TASK_CANCEL

Inputs:

- GENERAL_TASK_ID

Outputs:

- None

Implementation Requirements:

- Initiation:
  - On demand

Process Description:

The specified task (GENERAL_TASK_ID) is removed from the time queue and any applicable event queues. If the task has been preempted (i.e. is on the run queue), it is allowed to run to completion.

3.2.1.12 Process Name: TASK_ABORT

Inputs:

- GERNERAL_TASK_ID

Outputs:

- None

Implementation Requirements:

- Initiation:
  - On demand
- This process is implemented in Ada by the "abort" statement.

Process Description:

The specified task (GENERAL_TASK_ID) and any dependent tasks are immediately removed from all queues (ready, time and event) in a manner least disruptive to the system as a whole. If the aborted task is the active task (GENERAL_TASK_ID = ACTIVE_TASK_ID), the PRIORITY_DISPATCH process is invoked.

3.2.2 Memory Management Process Descriptions

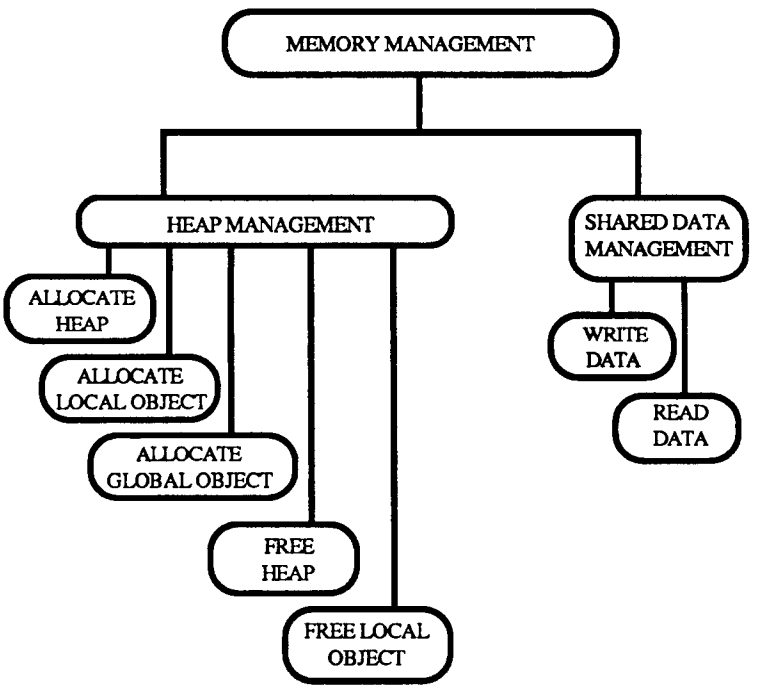


Figure 14. Memory Management

3.2.2.1 Process Name: ALLOCATE_HEAP

Inputs:

- GENERAL_TASK_ID
- GENERAL_TASK_ID.HEAP_SIZE

Outputs:

- GENERAL_TASK_ID.HEAP_BASE

Implementation Requirements:

- Initiation:
  - Task elaboration
- A STORAGE exception shall be raised if the requested amount of memory is not available.
- This process is implemented by the vendor supplied run time system at task elaboration.

Process Description:

The specified amount (GENERAL_TASK_ID.HEAP_SIZE) of consecutive heap memory is allocated to the requesting task for local stack space and for allocating local data objects.

The base location address (GENERAL_TASK_ID.HEAP_BASE) of memory allocated is returned to the process user.

3.2.2.2 Process Name: ALLOCATE_LOCAL_OBJECT

Inputs:

- LOCAL_OBJECT_SIZE

Outputs:

- LOCAL_OBJECT_LOCATION

Implementation Requirements:

- Initiation:
  - On demand
- A STORAGE exception shall be raised if the requested amount of memory is not available.
- This process is implemented in Ada by the "new" allocator.

Process Description:

The specified amount (LOCAL_OBJECT_SIZE) of the requesting task's local heap space is allocated for use by the requesting task as a data object.

The object location address (LOCAL_OBJECT_LOCATION) is returned to the process caller.

3.2.2.3 Process Name: ALLOCATE_GLOBAL_OBJECT

Inputs:

- GLOBAL_OBJECT_SIZE

Outputs:

- GLOBAL_OBJECT_LOCATION

Implementation Requirements:

- Initiation:
  - On demand
- A STORAGE exception shall be raised if the requested amount of memory is not available.

Process Description:

The specified amount (GLOBAL_OBJECT_SIZE) of the global memory is allocated for use by all tasks as a data object.

The object location address (GLOBAL_OBJECT_LOCATION) is returned to the process caller.

3.2.2.4 Process Name: FREE_HEAP

Inputs:

- GENERAL_TASK_ID
- GENERAL_TASK_ID.HEAP_SIZE
- GENERAL_TASK_ID.HEAP_BASE

Outputs:

- None

Implementation Requirements:

- Initiation:
  - Task Completion

Process Description:

The specified amount (GENERAL_TASK_ID.HEAP_SIZE) of local heap memory (GENERAL_TASK_ID.HEAP_BASE) previously allocated to the indicated task (GENERAL_TASK_ID) is released.

3.2.2.5 Process Name: FREE_LOCAL_OBJECT

Inputs:

- LOCAL_OBJECT_SIZE
- LOCAL_OBJECT_LOCATION

Outputs:

- None

Implementation Requirements:

- Initiation:
  - On demand
- This process is implemented in Ada by the UNCHECKED_DEALLOCATION generic.

Process Description:

The specified amount (LOCAL_OBJECT_SIZE) of the requesting task's local heap space (LOCAL_OBJECT_LOCATION) previously allocated is released.

3.2.2.6 Process Name: WRITE_DATA

Inputs:

- SHARED_DATA
- ACTIVE_TASK_ID.OUT_DATA

Outputs:

- None

Implementation Requirements:

- Initiation:
  - On demand

Process Description:

The following actions are performed when writing to a shared data area (local or global):

If the shared data area (SHARED_DATA.AREA) is locked (SHARED_DATA.LOCKED = true), the task is suspended (via process EVENT_SUSPEND) until shared data is unlocked (SHARED_DATA.LOCKED = false).

Lock data area (SHARED.DATA.LOCKED = true).
Write data (ACTIVE_TASK.OUT_DATA).
Unlock data area (SHARED_DATA.LOCKED = false).

3.2.2.7 Process Name: READ_DATA

Inputs:

- SHARED_DATA

Outputs:

- ACTIVE_TASK_ID.IN_DATA

Implementation Requirements:

- Initiation:
  - On demand

Process Description:

The following actions are performed when writing to a shared data area (local or global):

If the shared data area (SHARED_DATA.AREA) is locked (SHARED_DATA.LOCKED = true), the task is suspended (via process EVENT_SUSPEND) until shared data is unlocked (SHARED_DATA.LOCKED = false).

Lock data area (SHARED.DATA.LOCKED = true).

Read data (ACTIVE_TASK.IN_DATA).

Unlock data area (SHARED_DATA.LOCKED = false).

3.2.3 Intertask Communication Process Descriptions

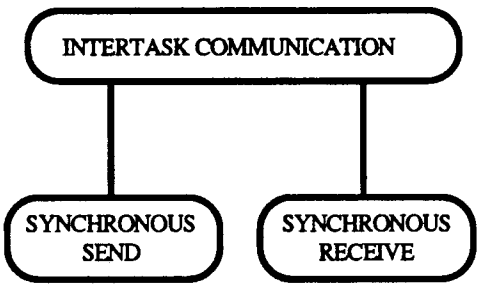


Figure 15. Intertask Communication

3.2.3.1 Process Name: SYNCHRONOUS_SEND

Inputs:

- RECEIVER_TASK_ID
- SENDER_TASK_ID.OUT_DATA

Outputs:

- SENDER_TASK_ID.IN_DATA

Implementation Requirements:

- Initiation:
  - On demand
- A TASKING exception is raised if a receiver task acknowledgement is not received.
- This process is implemented in Ada by the "rendezvous" entry call.

Process Description:

The SYNCHRONOUS_SEND process includes the following actions during synchronous intertask communication:

Obtain data from sender task (SENDER_TASK_ID.OUT_DATA) and store for receiver task (RECEIVER_TASK_ID.IN_DATA).

Initiate communication event with the specified receiver task.

Suspend sender task (via the EVENT_SUSPEND process) until receiver task acknowledges communication event completion (via the SIGNAL_EVENT) process.

Obtain data from receiver task (RECEIVER_TASK_ID.OUT_DATA) and store for sender task (SENDER_TASK_ID.IN_DATA).

3.2.3.2 Process Name: SYNCHRONOUS_RECEIVE

Inputs:

- RECEIVER_TASK_ID.IN_DATA

Outputs:

- RECEIVER_TASK_ID.OUT_DATA

Implementation Requirements:

- Initiation:
  - On demand
- This process is implemented in Ada by the "rendezvous" accept statement.

Process Description:

The SYNCHRONOUS_RECEIVE process includes the following actions during synchronous intertask communication:

> If a sender task has not initiated a communication event, the receiver task is suspended (via the EVENT_SUSPEND process) pending that event.
>
> Obtain input data from sender task (RECEIVER_TASK_ID.IN_DATA).
>
> Perform communication processing.
>
> Store output data for sender task (RECEIVER_TASK_ID.OUT_DATA).
>
> Acknowledge communication event completion via the SIGNAL_EVENT process.

3.2.4 Software Exception Process Description

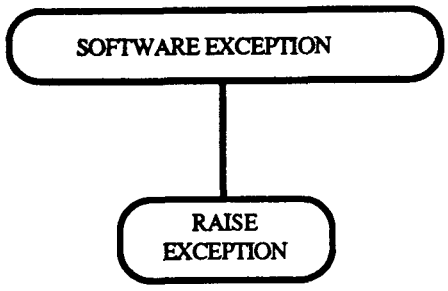


Figure 16. Software Exception

3.2.4.1 Process Name: RAISE_EXCEPTION

Inputs:

- GENERAL_TASK_ID
- EXCEPTION_ID

Outputs:

- None

Implementation Requirements:

- Initiation:
  - Detection of software exception
- This process is implemented in Ada by the vendor supplied run time system

Process Description:

Upon detection of a software exception (EXCEPTION_ID), the exception occurrence is recorded and the appropriate exception handler (GENERAL_TASK_ID.EXCEPTION_-HANDLER (EXCEPTION_ID) specified by the executing task is performed. If there is no specified exception handler, the task is purged.

4.0 GPC FAULT DETECTION, IDENTIFICATION AND RECONFIGURATION

The AIPS FTP uses hardware redundancy with fault detection and masking capabilities to provide fault tolerance. The design of that hardware is based on theoretical principles and is described in detail in Appendix A. The fault tolerance provided by the hardware is greatly enhanced by the Fault Detection, Identification and Reconfiguration (FDIR) functions which are part of the FTP local operating system. While the hardware alone in a triplex FTP could sustain one fault, the FDIR software allows it to sustain multiple successive faults and identifies the fault(s) for an operator, thus making the FTP much more robust and serviceable.

4.1 GPC Fault Detection, Identification and Reconfiguration Functional Requirements and Design

GPC FDIR is a process which is part of the local operating system in each AIPS processing site. The primary purpose of GPC FDIR is to keep the GPC and its external devices functioning correctly in the presence of any number of hardware faults. To achieve this, FDIR has two main functions:

- identifying a failed channel, i.e., detecting a fault, isolating it to a single channel, masking the faulty channel's inputs, and disabling its outputs.

- recovering a failed channel, i.e., determining that the fault no longer exists, bringing the channel into line with the two synchronized channels, accepting the channel's inputs, and enabling its outputs.

These functions must consume a minimum of the processing resources of the FTP under both fault and no-fault conditions. The tasks and procedures used by FDIR to implement these functions are summarized in Figures 12 and 13 and described in detail in the following sections.

A secondary purpose of FDIR is to report the status of the local GPC to the System Manager via the GPC Status Reporter. Therefore, faults and reconfiguration events are logged so that the GPC Status Reporter may transfer the information to the System Manager, which may be running on another GPC.

PROCESS	DESCRIPTION
FAULT DETECTION AND ISOLATION	
Fast FDIR	Periodic, high-priority task
Watchdog Timer Reset	Periodic, high-priority task
Background Selftests	As-time-available, low-priority task
M68010 Exception Handler	On-demand procedure
Reconfigure	On-demand procedure
Log Error	On-demand procedure
Log Reconfiguration	On-demand procedure
Log Non-Congruent Event	On-demand procedure
CHANNEL RECOVERY	
Transient FDIR	Periodic, high-priority task
Lost Soul Sync	On-demand high-priority task
Restart	On-demand procedure
Reconfigure	On-demand procedure
Log Reconfiguration	On-demand procedure
Log Non-Congruent Event	On-demand procedure

Figure 17. Summary of FDIR Tasks and Procedures

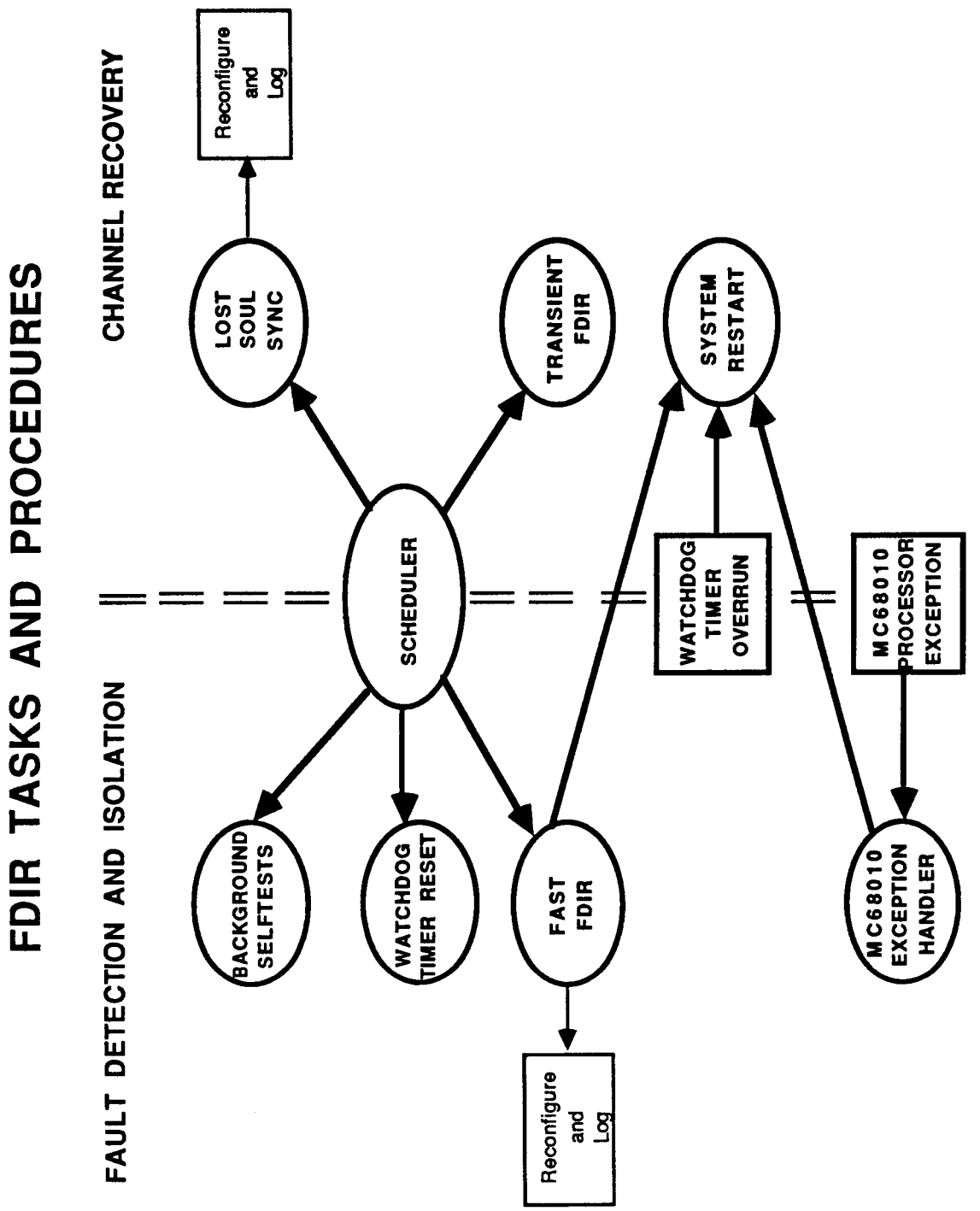


Figure 18. FDIR Tasks and Procedures

4.1.1 Fault Detection and Identification

Fault detection mechanisms are implemented in both hardware and software, while the identification mechanisms are implemented solely in software. Instruction-level synchronization together with bit-for-bit comparison of redundant data makes it possible to isolate a fault to a single channel.

There are four main processes which detect and identify faults:

- a periodic, high-priority task (Fast FDIR) which checks for failure of a companion, an unsynchronized channel, a failure in the data exchange hardware, and failure of a fault-tolerant clock;

- a periodic, high-priority task (Watchdog Timer Reset) which taps the watchdog timer within the given time bounds so that the timer does not overrun and cause a hardware reset;

- a low-priority task (Background Selftests) which does tests to uncover latent faults in memory, voting circuitry and error latches, and the real-time clock;

- a procedure for handling M68010 hardware exceptions such as an illegal instruction or addressing error.

After a channel is identified as being faulty, the GPC must be reconfigured so that the faulty channel does not affect GPC operation. The errors generated by the channel must be masked and its outputs must be stopped. This is done by a procedure (RECONFIGURE) which:

- sets a software variable which identifies the channel as failed;
- disengages the monitor interlock so that outputs from the faulty channel are disabled.
- logs the fault and the reconfiguration for later examination by an operator.

4.1.1.1 Fast FDIR

Fast FDIR is one of four tasks which detect and isolate errors. It is a high-priority task which runs every 40 ms. It checks for:

- a failed companion processor
- an unsynchronized channel
- a failure detected but not reconfigured around by the selftests

- a data exchange failure, i.e., either failure of the interstage or of any link in the data exchange hardware
- failure of a fault-tolerant clock.
- a missing companion processor

Error detection is done in the order given above. If any particular test uncovers a failure, the remaining tests are not done during that iteration of Fast FDIR. When an error is detected, the error is logged, the GPC is reconfigured to exclude the faulty channel, and the reconfiguration is logged.

Failed Companion

When one of the two processors in a channel (CP or IOP) detects a failure, the companion processor is notified. The companion processors must fail their member in the same channel, even though it may be fault free. This ensures that the CPs and IOPs always have the same configuration.

Unsynchronized Channel

An unsynchronized channel is detected by means of two tests: (1) the inconsistent PC check and (2) the presence test.

The inconsistent PC check is designed to handle the case where processors are out of sync but are brought back together by an interrupt, which thus masks the out of sync condition. Each time an interprocessor or timer interrupt occurs, a consensus value for the PC at the time of the interrupt is obtained by doing a FROM_ALL data exchange. Each channel then compares its own PC to the consensus value; a channel with a different value sets a flag which is subsequently checked by Fast. Checking the flag causes the inconsistent channel to diverge so that the two synchronized channels see it as missing when they do the presence test.

The presence test detects an unsynchronized channel by sending a unique pattern from each channel through the data exchange. If the result read from the data exchange receiver is not the expected pattern, the channel originating the exchange is judged not present and therefore out of sync.

Selftest Error

The background selftests attempt to detect latent errors, that is, errors which currently exist but have not yet caused data exchange errors or a channel to become unsynchronized. These tests are discussed fully in a later section. The selftest task does not reconfigure when it detects an error, however. It merely passes the pertinent information to the Fast FDIR task, which does the actual reconfiguration.

Data Exchange Failure

Failures in the data exchange are detected by analyzing the error latches set by the data exchange hardware. These latches are set during a data exchange if a miscompare is detected during the voting of data received from the interstages. The latches identify the type of exchange and the channel at fault when the miscompare occurred. A failure can be isolated either to (1) an interstage (including the transmitter-to-interstage link) , or (2) any link other than the transmitter-interstage link (i.e., transmitter-transmitter links and interstage-receiver links).

Fault-Tolerant Clock Failure

Failures in the fault-tolerant clocks are detected by analyzing the error latches set by the clock hardware. There are two sets of error latches: one which is set when the clock interstage detects faulty input from a clock element, and a second which is set when a clock element detects faulty input from a clock interstage. Failure of a fault-tolerant clock is merely logged; it does not cause the channel to be considered failed.

Missing Companion

A processor which has restarted because of a watchdog timer-caused reset or because of hardware or software exceptions needs to so notify its companion so that the companion may also restart. One way for such notification to occur is to have each processor periodically notify its companion that it is still running and at the same time check that its companion is still running. If a processor finds its companion missing, it assumes its companion has restarted and therefore does a software restart itself.

4.1.1.2 Watchdog Timer Reset

The second fault detection process is the Watchdog Timer Reset process. This process does not perform fault detection functions in quite the same way, however, as other processes in this category, i.e., by responding to a specific fault. Rather, the failure of this task to execute at its scheduled period would indicate a critical fault in either hardware or software and would cause a hardware reset.

The watchdog timer is a hardware component whose purpose is to prevent infinite software loops or hardware faults from hanging up the system. After it has been started, the watchdog must be cleared periodically within a set time window; if it is not cleared within this window (i.e., either too early or too late) a hardware reset occurs. On the AIPS FTP this window is 60-120 milliseconds plus or minus 10%. The Watchdog Timer Reset process performs the function of clearing the watchdog timer.

4.1.1.3 Background Self Test

The third fault detection task is the Selftest task, a low-priority task which runs when the GPC has no other important work to do. Its job is to uncover latent faults, that is, faults which exist but which have not yet caused data exchange errors or desynchronization of a channel. This task does tests on four hardware elements:

- Memory
- Voter circuitry and error latches
- Monitor interlock
- Real-time clock

The memory tests include the following:

- PROM sum check. This test verifies that all channels have identical values in ROM by doing a sum check.
- RAM scrub. This test checks each memory location to ensure that the values are identical among the three channels.
- RAM pattern test. This test checks the functionality of each location. It tests each bit's ability to hold both a 1 and a 0 by writing specific patterns to each word.
- Shared memory scrub. This test checks each memory location in shared memory to ensure that the values are identical among the three channels.

Voter circuitry and error latches are tested by writing normal and faulty patterns of data to the voters. After these votes, both the resulting values and the error latches are checked to confirm that all errors were properly latched and corrected.

The monitor interlock is tested by reading the current value and ensuring that it is identical among the three channels.

The real-time clock is tested by reading the current value and ensuring that it is identical among the three channels.

4.1.1.4 Hardware Exception Handler

The exception handler is the fourth fault detection process. It is invoked when there is a hardware exception such as an illegal instruction or an address error. The type of exception and relevant information such as the program counter and selected registers are logged in the non-congruent log. A presence test is done to determine if the exception was caused by a hardware error or a generic software error. If the results of the presence test show that a processor is alone, this implies a hardware error. If the presence test shows that a processor is with others, this implies a generic software error. In either case, the exception and

the results of the presence tests are logged in the non-congruent log, and the processor(s) are restarted.

4.1.2 Channel Recovery

As mentioned earlier, the reliability of the FTP is greatly enhanced if channels previously diagnosed as faulty but currently operating without faults can be brought back into the FTP configuration. How a channel is recovered depends on the type of failure, i.e., whether or not the fault has caused the channel to fall out of sync. A failure in the data exchange hardware does not desynchronize a channel, while other kinds of failures do. A GPC has recovered from a fault, therefore, when

- the failed channel can be resynchronized, or
- the failed channel no longer shows errors in the data exchange hardware.

When a channel has been recovered, the GPC must be reconfigured so that the recovered channel participates in the GPC operation and another fault can be tolerated.

There are three main processes involved in channel recovery. Transient FDIR distinguishes between transient and hard failures when a channel recovery is being attempted in order to balance competing system needs. Lost Soul Sync is responsible for resynchronizing an unsynchronized channel, i.e., synchronizing it to the instruction level and making its internal state the same as the duplex processors. Finally, the Restart process is invoked when a second fault or a common-mode failure occurs. These faults result in a fail-safe condition, which the AIPS FTP responds to with a system restart.

4.1.2.1 Transient FDIR

When recovering a failed channel, system resources are used most efficiently if a distinction is made between transient failures and hard failures. Transient failures are assumed to be caused by some temporary environmental condition (e.g., a power surge). By definition, they are expected to disappear with time. Hard failures, on the other hand, are caused by breakdowns of the FTP hardware that must be physically repaired.

The attempt to recover a failed channel could be made automatically (i.e., the software periodically tests the channel to determine its current state) or it could be made solely under operator direction (i.e., the operator enters a command indicating the channel has been repaired). The first method satisfies the need to recover the channel as quickly as possible while the second method satisfies the need to not waste system resources by repeatedly testing a channel with a hard failure. Transient FDIR strikes a balance between these two needs by initially assuming that any particular fault is transient (it has been observed that 50 to 80 percent of all faults in computer systems are transient) and automatically attempting a recovery. As time passes without the channel being recovered, it becomes

more likely that the fault is a hard failure rather than a transient, and Transient FDIR makes the recovery attempt less often. After a certain period it can reasonably be assumed that the failure is a hard failure and Transient FDIR either waits for an operator signal or tests the channel only at some infrequent interval such as its mean time to repair.

Additionally, it has been noted that hard failures tend to manifest themselves sporadically. A channel may be recovered according to the above criteria, but may immediately fail again. Transient FDIR attempts to prevent this situation by regarding a recovered channel as recovered only on a trial basis. If the channel passes its trial period without further errors, it is regarded as fully recovered and can be added back into the FTP configuration. Intermittent failures which occur at infrequent intervals (i.e., after the trial period has passed) will not be handled by this scheme, however, but will be regarded as new faults.

This distinction between transient and hard failures thus defines the two functions of Transient FDIR:

- It decides when it is appropriate to attempt to recover a failed channel.
- Once a channel is seen as fault-free, it monitors its health for a brief probation period before declaring it fully recovered.

Attempting Channel Recovery

The initial response to all detected faults is to mask the fault and disable all outputs from the faulty channel. Thereafter, the status of the failed channel is periodically "sampled" to determine if the fault is transient. Immediately after a failure, a recovery attempt is made and a sampling of the channels health is taken. If the attempt fails (i.e., the unsynchronized channel cannot be found or the data exchange latches still show errors), the time between successive attempts is doubled, until Mean Time To Repair (MTTR) is reached. This time delay between successive recovery tries and the samplings of the status of a failed channel is a function of state variables representing the "health" of the channel. The "health" variable, in turn, is a function of the error history of the particular channel with many recent fault observations for the channel indicating "poor" health and declining fault observations representing "good" health. The time between recovery attempts is doubled following each status sampling which indicates the fault is still present. This sampling sequence is repeated until either the fault status changes to indicated the fault is no longer present or an upper threshold on the retry time is crossed at which point the fault is deemed "hard". From this point on recovery will be attempted only when another MTTR period has passed or after an operator signals that the channel has been repaired.

Probation Monitoring

After a channel has been recovered, it must undergo a trial period before being declared fully recovered and functional. The length of this period is a function of the "health" of the

channel and depends on the number and type of faults. A channel with multiple faults in quick succession (i.e., the channel fails before it has passed its trial period) will have a longer trial period than if it only had a single fault. Faults that desynchronize a channel require a longer trial period than data exchange errors.

Additional Considerations

The work of Transient is complicated by several factors.

(1) AIPS is a bi-processor system. This has two ramifications.

(a) The CP and IOP must maintain identical configurations within the skew of one Fast cycle. Yet because of performance requirements, Transient runs only 25% as often as Fast. Identical configurations must be maintained because otherwise one side might see a failure as a single fault while the other side would see it as a double fault and go into fail-safe mode.

(b) When to execute the Channel Sync routine must be very closely coordinated between the CP and IOP so that neither side wastes time waiting at the handshake.

(2) IOS dual-ported memory and registers must be re-initialized after a recovery. Because of performance requirements and its low criticality, this process should not run uninterrupted and should not prevent Transient from making the other checks it must make during its periodic execution.

These factors led to the following design of the Transient FDIR process.

- All decision-making in the task is done by the CP. The CP then communicates with the IOP via shared memory as to what to do, specifically whether to schedule the Lost Soul Sync task or the IOSS Restore task (see below).

- The Channel Sync routine is run as part of a separate task (Lost Soul Sync) and is scheduled for an absolute time, which is calculated so that the IOP and CP spend minimal time waiting for each other. The Channel Sync routine sets a flag for Transient indicating whether or not it picked up the faulty channel.

- The IOS re-initialization is run as a separate task. The CP tells the IOP when to start it; the IOP signals the CP when the task is complete. Provision is made for the case where a second error occurs before the IOS re-initialization is finished, since there is no way to stop a task in midstream.

- After a channel has passed its probation period and reinitialized the IOS, it is permanently enabled by Fast FDIR rather than Transient. CP Transient, which

makes the decision to reenable, passes the necessary information to CP Fast, which in turn passes it to IOP Fast.

4.1.2.2 Lost Soul Sync

Lost Soul Sync is the process of attempting to resynchronize a previously failed channel (a "lost soul") and, if successful, bringing it to the same state as the two good channels. This process has two main steps:

- resynchronizing the channel, i.e., synchronizing it to the instruction level with the other two channels, and

- aligning the channel, i.e., making its volatile memory and registers the same as those of the other two channels. This ensures that after the code execution is synchronized, only a fault could cause a channel to lose synchronization, rather than, for example, a memory location that contained an incorrect value.

This task is an on-demand task scheduled by Transient FDIR when Transient deems it appropriate to attempt recovery of a failed channel. Because of the handshaking required between the CP and IOP, the task is scheduled to run at an absolute time. The following describes the two main steps of Lost Soul Sync.

The resynchronization function is performed in a loop where single-source exchanges are initiated with predetermined, constant data patterns being sourced. The exchanges executed in one iteration of the loop include one single-source exchange from each FTP channel. Divergent channels must be executing this loop at the same time in order to become synchronized. As a given channel performs the data exchanges within the sync loop, a comparison is made of the data received for a given exchange against the data expected for a successful exchange. A match indicates that the given channel is operating in synchronism with the channel at the source of the exchange; a mismatch indicates a lack of synchronism. Thus after every iteration a channel knows whether it is alone or synchronized and with whom. A channel remains in this loop as long as it is alone; channels which are together leave the loop either immediately (if they have become a triplex) or after a specified number of iterations (if they are a duplex).

Obviously, repetitive executions of the exact same loop of instructions by two unsynchronzed channels will never bring them to the point of synchronous operation. After each iteration, the two channels must delay their execution of the next iteration of the loop by different amounts of time in an attempt to phase shift into sync with one another. The length of the delay depends on the configuration of the channel(s), i.e., each channel or combination of channels delays for a different amount of time. All other channels in this configuration will delay the same amount and thereby remain synchronized with each other.

Since each channel of the FTP may have multiple processors, a coordination of processor activity within each channel is required before and during the resynchronization process. Only one of the processors in each FTP channel executes the sync loop code described above. Before doing so, the processors within each channel handshake with one another to signal their readiness to operate together with their counterparts in other channels. This handshake is implemented using shared memory. The processors not executing the sync loop described above must suspend operation until the sync loop is exited and the result of the synch attempt is known. This suspension must be implemented such that on resumption of processor operation following a successful synch attempt, not only will the set of inter-channel processors which actually executed the synch code be in synch, but their companion (intra-channel) processors will also execute in synch with their inter-channel counterparts. This is accomplished by locking the shared bus; a reference to any component on the shared bus (in particular, shared memory) results in a suspended bus cycle which completes only when the shared bus is unlocked by the locking processor. Thus those processors not executing the actual synch code will be suspended on a shared memory access with the processors executing the synch code clearing the lock only after completion of the synch attempt. The result of the synch attempt is passed to the companion processor via shared memory.

It is imperative that once synchronous operation between channels is established, no conditional changes in program flow are made based upon data not congruent across channels. Immediately following a synch attempt in which an a lone channel is added to the configuration, all hardware elements must be made congruent (i.e., 'aligned'). Any hardware element for which an incongruent state could cause a desynchronizing of channels must be aligned. Each processor within the channel must align its own local hardware resources. However, only one processor in a channel needs to align the shared resources within the channel (e.g., shared memory, real-time clock).

The same resynchronization and alignment processes are used for synchronizing three divergent channels at system startup.

4.1.2.3 System Restart

Certain faults which are detected may be of such a magnitude that they are unsustainable and may be recovered from only by restarting the system. Examples of such faults are a second fault detected by Fast FDIR and common-mode faults. The restart process accomplishes the system restart without requiring operator intervention.

4.1.3 Reconfiguration

The reconfiguration routine is called by the fault detection and identification tasks when a fault has been identified and by the recovery tasks when a channel has been repaired. During a reconfiguration a channel is either removed from the configuration because it was found to be faulty, or added in because the fault no longer exists.

When a channel is identified as being faulty, the errors generated by that channel must be masked and its outputs must be stopped. This is done by (1) setting a software variable which identifies the channel as having failed, and (2) disengaging the monitor interlock so the channel's outputs are disabled.

When a channel has recovered from a failure, its inputs must be accepted and its outputs enabled. This is done by the reverse process, i.e., (1) setting the software variable to say that the channel is now functional, and (2) engaging the monitor interlock so the channel's outputs are enabled.

When a channel has recovered from a failure, it is considered part of the configuration only by FDIR until its probation period has expired. This is done by setting a software variable to say that the channel is enabled on a trial basis.

4.1.4 Logging

Failures and reconfigurations are recorded in logs which may then be examined by a system operator or passed to the system FDIR manager. Certain failures and recovery actions are also entered in the non-congruent log, which contains unique information for each channel and is preserved by a software restart. The logs may be displayed by entering appropriate commands on a CRT or Macintosh computer.

4.2 GPC Fault Detection, Identification and Reconfiguration Software Specifications

4.2.1 Fault Detection and Identification Process Descriptions

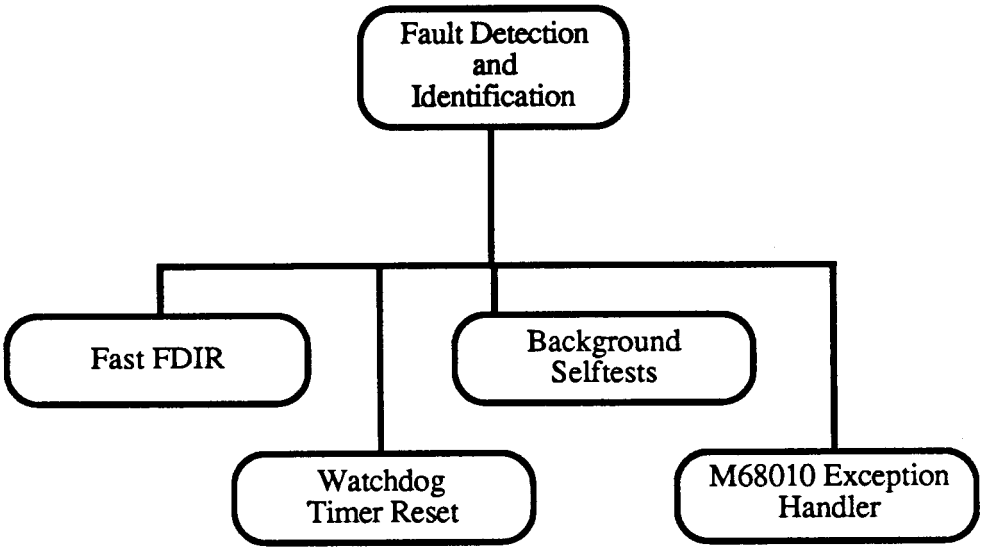


Figure 19. Fault Detection and Identification Functions

4.2.1.1 Fast FDI

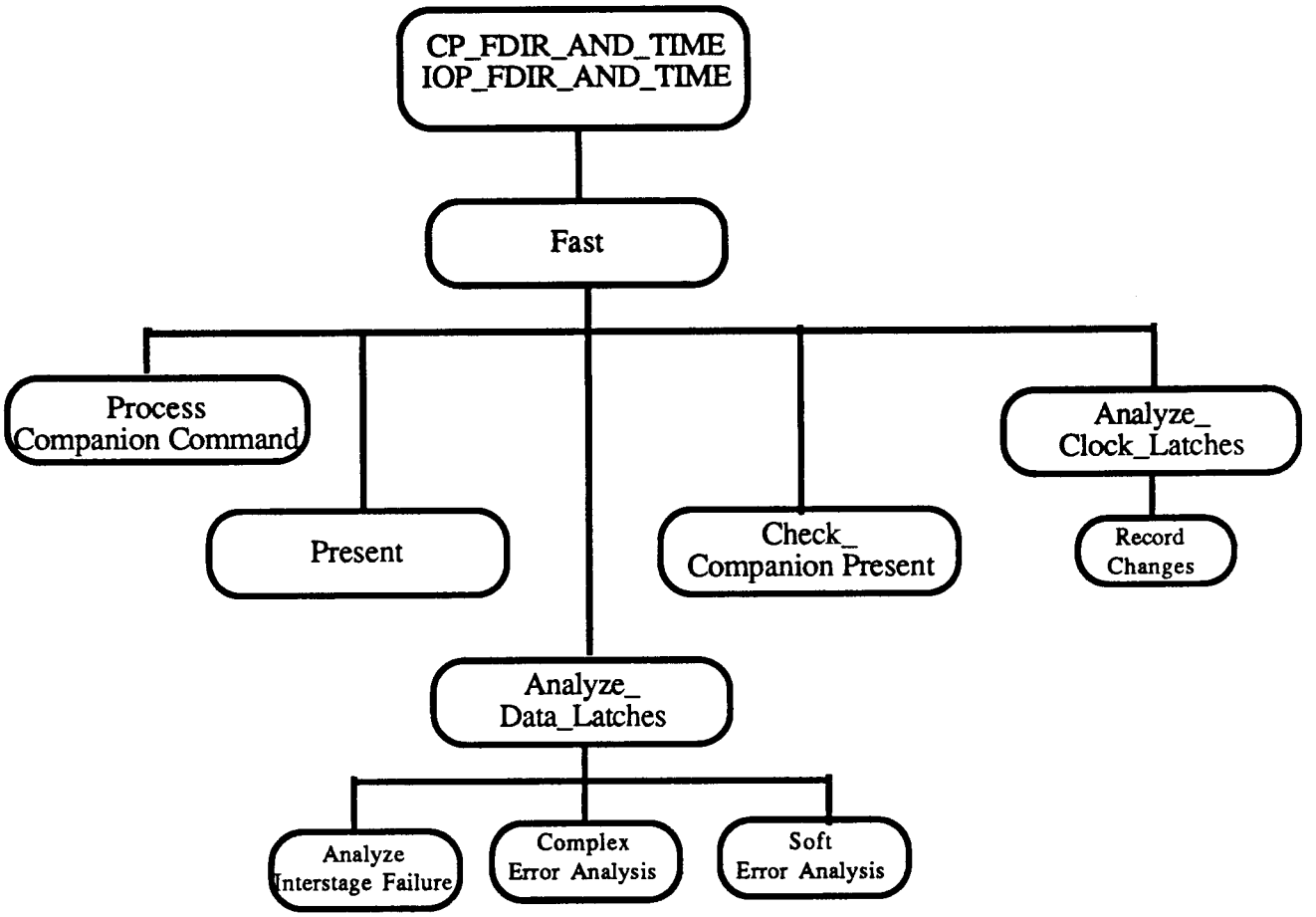


Figure 20. Fast FDI Processes

4.2.1.1.1 Process Name: CP_FDIR_AND_TIME IOP_FDIR_AND_TIME

Inputs:

- None

Outputs:

- None

Description:

These tasks implement the periodic FDIR functions and the Time Manager function for the CP and IOP, respectively. Although Fast FDIR, Transient FDIR, and the Time Manager are tasks, conceptually speaking, they were not actually implemented as separate Ada tasks. Rather, only one task, a combined FDIR_AND_TIME task, was implemented. This task is scheduled to run at the frequency of the highest priority function, Fast FDIR. It executes Fast FDIR every time it is scheduled and then uses a counter to maintain the proper periodicity of the other functions, i.e., Transient FDIR is executed only four times a second and the Time Manager only once. The motivation for the combined FDIR_AND_TIME task was to save the overhead time required for switching between separate tasks, especially since under no-fault conditions the time required for Transient FDIR is only a few microseconds. Interrupts are disabled while this task is executing.

4.2.1.1.2 Process Name: FAST

Inputs:

- Companion present flag from companion processor
- Enable command from companion (if IOP) or from Transient FDIR (if CP)
- Disable command from companion
- Disable command from selftest task

Outputs:

- Disable command to companion processor
- Enable command to companion (if CP)
- Companion present flag

Description:

This is the main procedure of the Fast FDIR process, and is called by the FDIR_AND_TIME task every 40ms. This procedure checks for the following faults:

- failure of a companion processor, i.e., a disable command from the companion
- a channel out of sync, i.e., either

  - an inconsistent PC, which indicates to a lone channel that it was out of sync at a timer or interprocessor interrupt, or
  - incorrect results from the presence test
- a fault detected but not reconfigured around by the selftests
- a data exchange error, i.e., either failure of the interstage or of any link in the data exchange hardware
- failure of a fault-tolerant clock
- absence of a companion processor, i.e., the companion has ceased execution

In addition, the procedure checks for an enable command, which indicates that a previously failed channel has been recovered.

The above list indicates the priority of the tests. Once a fault is detected, the remaining tests are bypassed during that execution of the procedure, because they would fail anyway. For example, if a processor is out of sync, that would naturally cause the data exchange error latches to be set.

After an error is detected, it is logged. The GPC is reconfigured to exclude the faulty channel, and the reconfiguration is logged. A disable command is written in shared memory to notify the companion processors of the failure. For the appropriate types of failures (i.e., all errors except data exchange errors), the faulty channel now goes to the Channel Sync procedure . If a channel determines that it is alone and that the GPC has degraded from a duplex to a simplex (i.e., a second fault has occurred), it goes to a FAIL_SAFE procedure.

An exception to this procedure occurs when a fault-tolerant clock failure is detected. This failure is not reconfigured around; it is merely logged.

The last step in the Fast FDIR procedure is to check for a Channel Enable command. The CP gets this command from Transient FDIR; if there is one it reconfigures, logs the reconfiguration and notifies the IOP by writing in shared memory. The IOP gets the Channel Enable command from the CP; if there is one it reconfigures and logs the reconfiguration.

Implementation Notes

The shared bus is locked during any access to shared memory (i.e., reading or writing the inter-processor commands). This is done to prevent these commands from simultaneous accesses. This implementation was selected over two alternative approaches:

1) using the shared read/write services of the Ada runtime system, and
2) associating a lock variable with each shared memory command so that a specific command could be locked rather than the shared bus.

The first alternative was rejected because the runtime system puts a task to sleep when a desired object is not available, and this is not an acceptable consequence for Fast FDIR. The second alternative was rejected because, while the amount of time that the shared bus was locked decreased, the amount of time to execute Fast FDIR increased. This thus became a tradeoff between taking time from the local processor and taking it from the companion processor. Locking the shared bus was then chosen because it is a simpler and more direct implementation.

Similar reasoning was used in the decision to use the shared data exchange, rather than the dedicated, for the presence test. Using the shared data exchange means the test can be very simple, but it required locking the shared bus. Using the dedicated data exchange does not require locking the shared bus but the test is more complex and takes longer. The shared exchange implementation was chosen because of its simplicity.

Finally, the shared bus is locked for the following sequence of tests:

- Checking for a disable command from the companion
- Checking for an inconsistent PC
- Presence test
  - Doing the data exchanges
  - Writing disable command to companion if error detected

This is done to ensure that when a processor falls out of sync, the error will be reported in a predictable way. Often a processor drops out of sync because its companion dropped out first. Locking the shared bus during the above sequence ensures that one side (and only one side) will report a presence failure while the other side will report a companion error.

4.2.1.1.3 Process Name: PROCESS_COMPANION_COMMAND

Inputs:

- Reconfig_cmd record

Outputs:

- Entry in error log
- Entry in FAST_INFO array

Description:

This procedure disables a channel when one set of processors (CP or IOP) has been notified that a companion has failed. The GPC is reconfigured to eliminate the faulty channel, and the error and the reconfiguration are logged. For errors other than data exchange errors, the companion to the failed processor drops out of sync at this point and goes to the Channel Sync procedure.

4.2.1.1.4 Process Name: PRESENT

Inputs:

- None

Outputs:

- Integer indicating channels in sync

Description:

This routine performs the presence test, i.e., it checks for missing (unsynchronized) channels. It does this by sending a unique pattern from each channel through the data exchange. If the result read after a particular exchange is not the expected pattern, the channel originating the exchange is judged to be not present and therefore out of sync.

As mentioned in the Implementation Notes for the Fast process, the patterns exchanged are long words and are transmitted using the shared data exchange. When a word is exchanged using the dedicated exchange, the data exchange hardware is sufficiently slow such that any particular word will be in the receiver for more than one FTC cycle. This would allow a channel only slightly behind to see the expected word in the receiver and incorrectly conclude that it is in sync. Successive exchanges of long words, however, result in the long word being in the receiver for only one FTC cycle, so a tardy channel could not conclude that it was in sync.

Interrupts are disabled and the shared bus is locked before this routine is called.

4.2.1.1.5 Process Name: CHECK_COMPANION_PRESENT

Inputs:

- Shared memory signal from companion

Outputs:

- Shared memory signal to companion

Description:

A processor which has reset because of the watchdog timer or because of hardware or software exceptions has no way to notify its companion of this fact. If one set of processors (i.e., CP or IOP) in all three channels reset in this way, they will never restart because they will all be waiting in the synchronization routine to handshake with their companions, who know nothing of their restart.

One solution to this problem is to have each processor periodically notify its companion that it is still running by writing a predefined value in shared memory. In addition to writing its own value, the processor determines if the companion is still executing by checking for the companion's value. The companion is allowed to miss one cycle before the processor assumes it has reset and does a reset itself.

4.2.1.1.6 Process Name: ANALYZE_DATA_LATCHES

Inputs:

- Data exchange error latches

Outputs:

- None

Description:

This routine analyzes the error latches set by the data exchange hardware and identifies the particular element of the hardware which has failed (i.e., the specific interstage or link).

Each channel gets its own copy of the four error latches (one for each type of exchange) and combines them into one word. This word is then sent to the other two channels. All three channels do a software vote to get a consensus value of the three individual latches.

Only the CP checks for link and interstage failures, because these failures will be manifested in both processors. Both the CP and IOP check for "soft" errors (i.e., non-congruent data in a FROM_ALL exchange), since this type of error would be unique to the particular processor. Each type of failure causes the error latches to be set in a known way, which allows the error to be identified unambiguously. Figure 21 summarizes the types of failures and the error latch settings for each type.

If only two channels are up, an error cannot be unambiguously identified. All that can be done in this case is to check for a second failure by ignoring any errors caused by the failed channel and then seeing if any other bits in the latches are set. To ignore errors caused by a failed channel X, three things must be masked:

- any latches set during a FROM_X exchange,
- the bits representing channel X in the other types of exchanges.
- the exchanged copy of channel X's error latches.

A second failure results in a fail-safe condition in which the system restarts.

After a fault has been identified, the LOG_AND_RECONFIG subroutine is called to create the reconfiguration record and error log entry which will be used by the main Fast FDIR procedure.

FAILURE MODES AND RESULTING ERROR LATCH PATTERNS

Fault Type	Reporting Channel											
	A				B				C			
	From All	From A	From B	From C	From All	From A	From B	From C	From All	From A	From B	From C
a -> * or A -> a	A	A	A	A	A	A	A	A	A	A	A	A
a -> B	—	—	—	—	A	A	A	A	—	—	—	—
A <=> B	—	B	A	—	—	B	A	—	—	B	A	—
A -> B	—	B	—	—	—	B	—	—	—	B	—	—
SOFT(A)	A	—	—	—	A	—	—	—	A	—	—	—

NOTATION	MEANING
A	Transmitter or receiver
a	Interstage
->	One way failure
<=>	Bidirectional Failure
*	All Channels
SOFT	Noncongruent data

Figure 21. Data Exchange Error Latch Settings

4.2.1.1.7 Process Name: ANALYZE_INTERSTAGE_FAILURE

Inputs:

- Voted value of error latches from the three channels

Outputs:

- None

Description:

This routine checks for a hard interstage failure. When an interstage has failed, all channels will show an error in that channel for all types of exchanges. This is illustrated in Figure 22. A failure in the link between a channel's transmitter and its interstage will cause the error latches to be set in the same way. The failure of this link and the failure of an interstage are indistinguishable.

After an interstage failure has been identified, the LOG_AND_RECONFIG subroutine is called to create the reconfiguration record and error log entry which will be used by the main fast FDIR procedure.

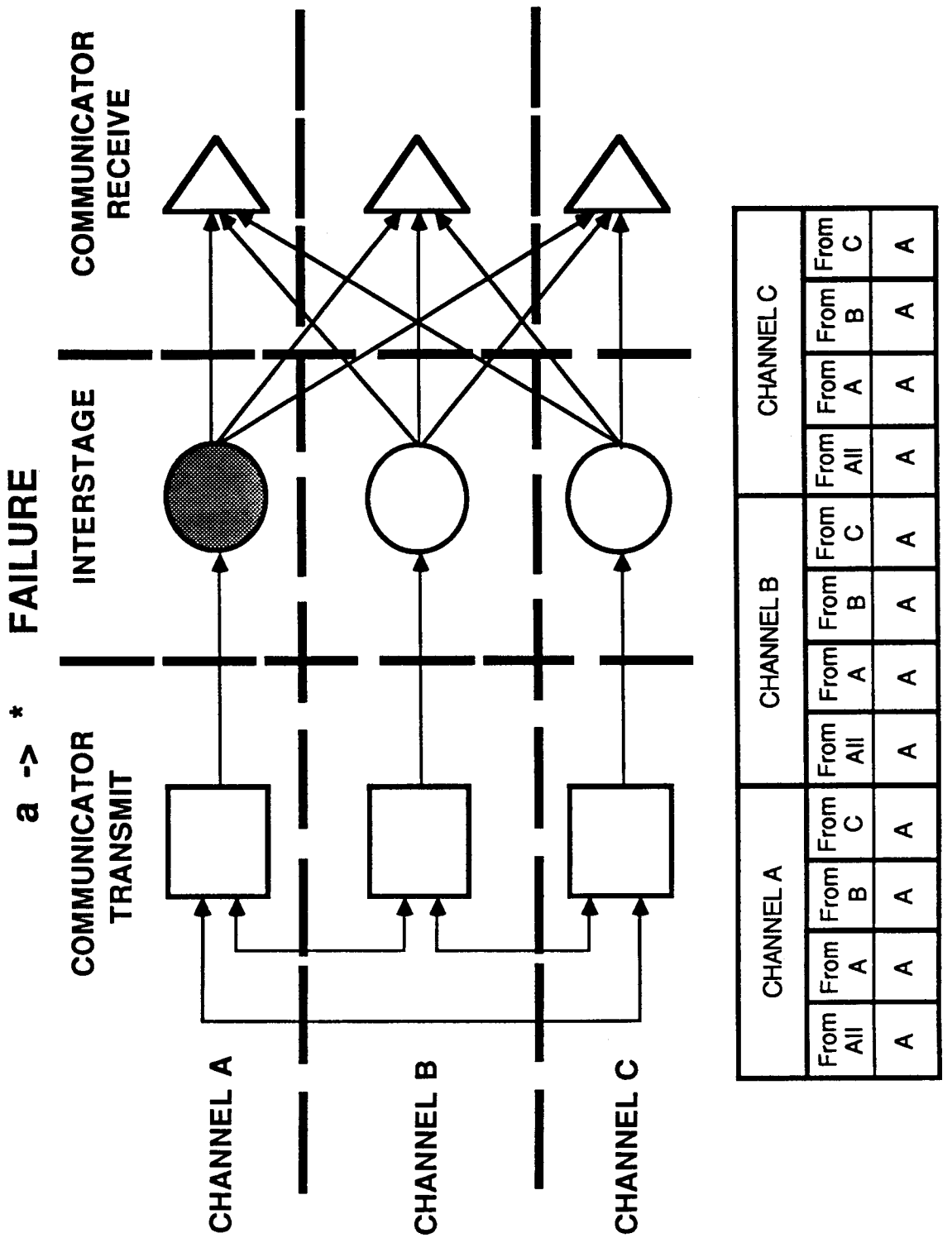


Figure 22 Latch Settings For Interstage Failure

4.2.1.1.8 Process Name: COMPLEX_ERROR_ANALYSIS

Inputs:

- Local and voted values of error latches from the three channels

Outputs:

- None

Description:

This subroutine checks for three types of link failures:

- a failure in the link between an interstage and any one channel's receiver;
- a bidirectional failure in the link between two channels' transmitters;
- a one-way failure in the link between two channels' transmitters.

These failures are illustrated in the following figures.

A failure in the link between an interstage and any one channel's receiver will cause error latches to be set only on the receiving channel. This channel will show errors in the channel originating the faulty link for all types of exchanges.

A bidirectional failure in the link between two channels' transmitters will set the same error latches on all three channels. All exchanges from the first channel will show an error in the second channel, and all exchanges from the second channel will show an error in the first channel.

A one-way failure in the link between two channels' transmitters will set the same error latches on all three channels. A failure from Channel 1 to Channel 2 will show errors in Channel 2 on From_Channel_1 exchanges.

After a link failure has been identified, the LOG_AND_RECONFIG subroutine is called to create the reconfiguration record and error log entry which will be used by the main Fast FDIR procedure.

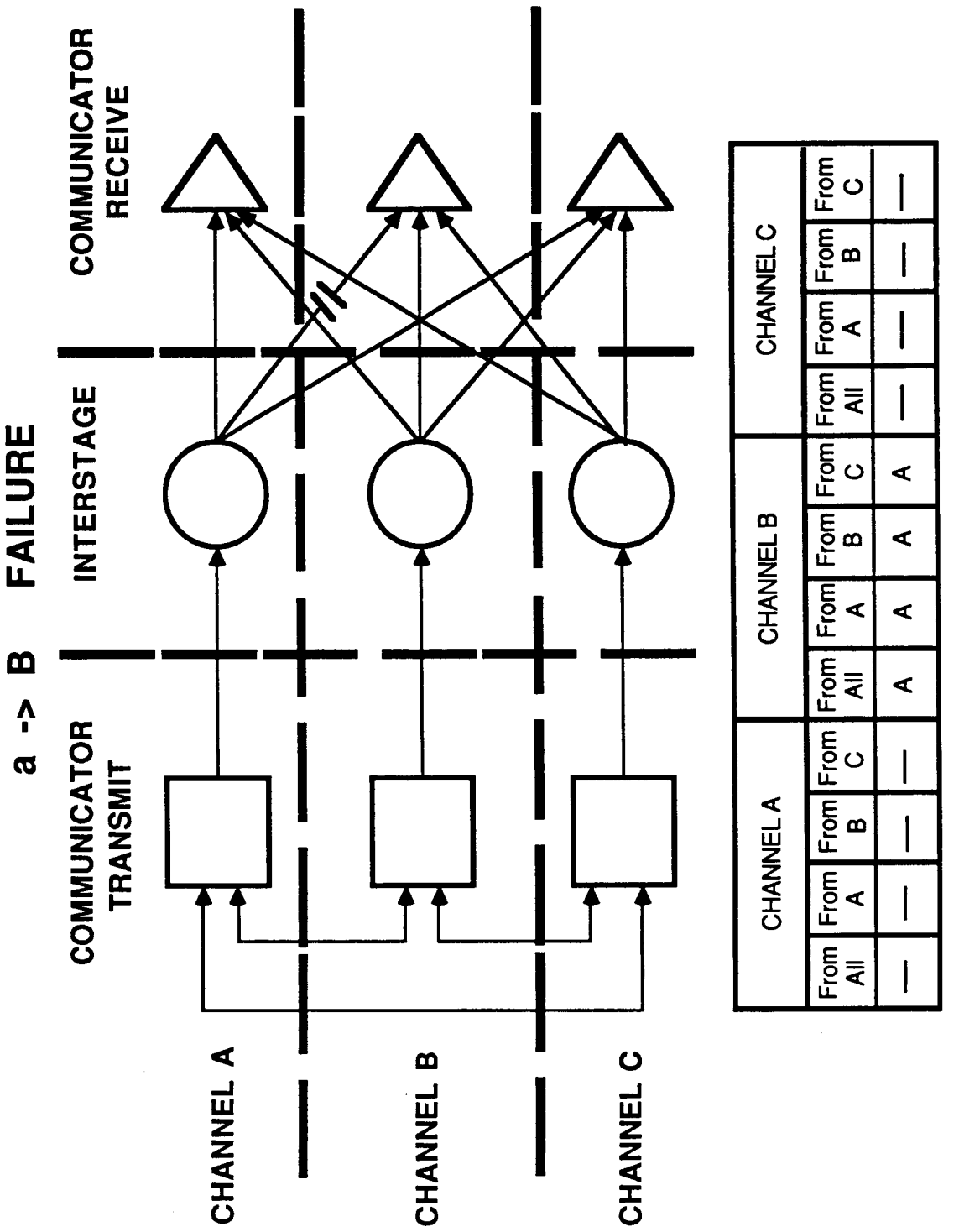


Figure 23. Latch Settings For Interstage - Receiver Link Failure

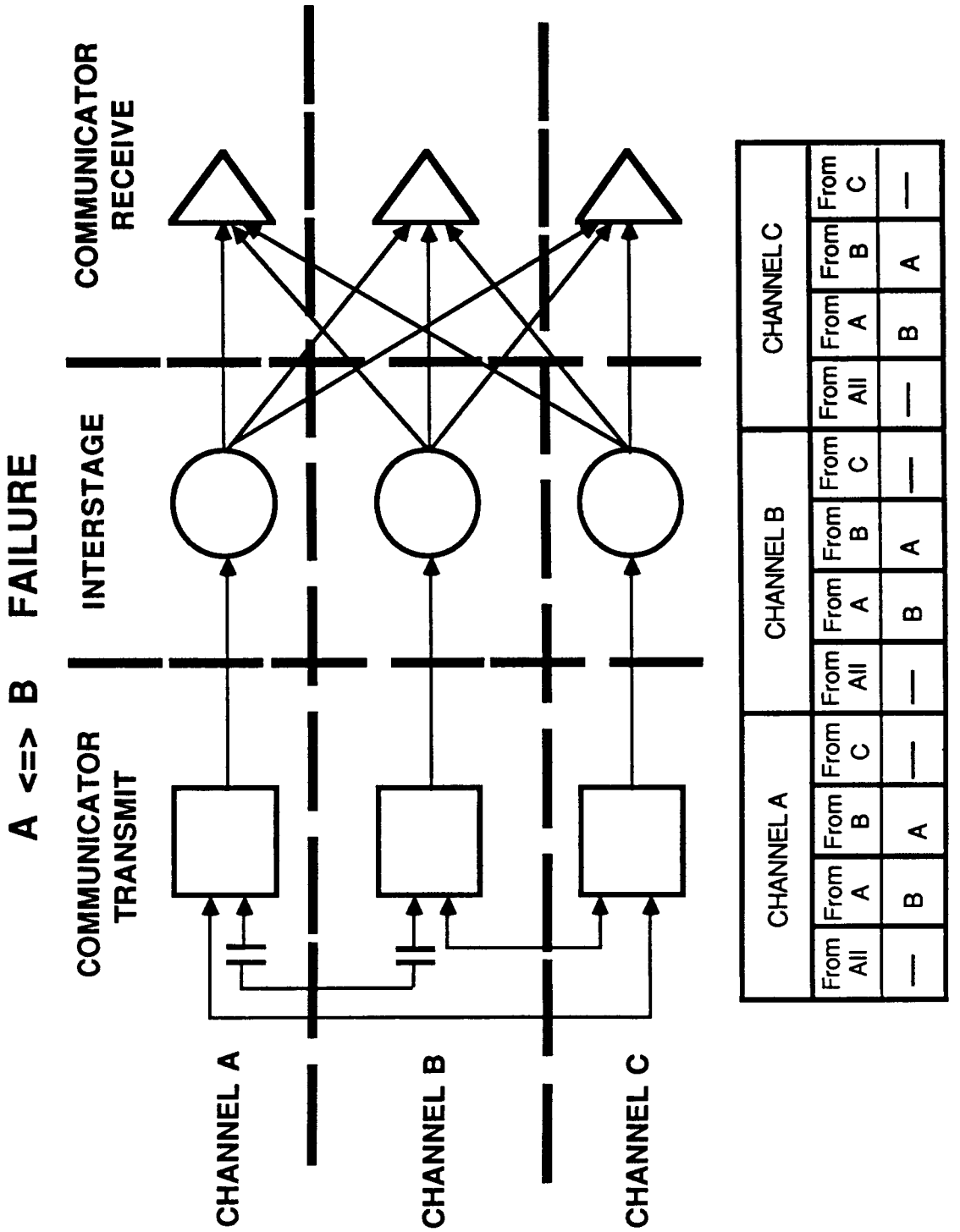


Figure 24. Latch Settings For Bidirectional Transmitter - Transmitter Failure

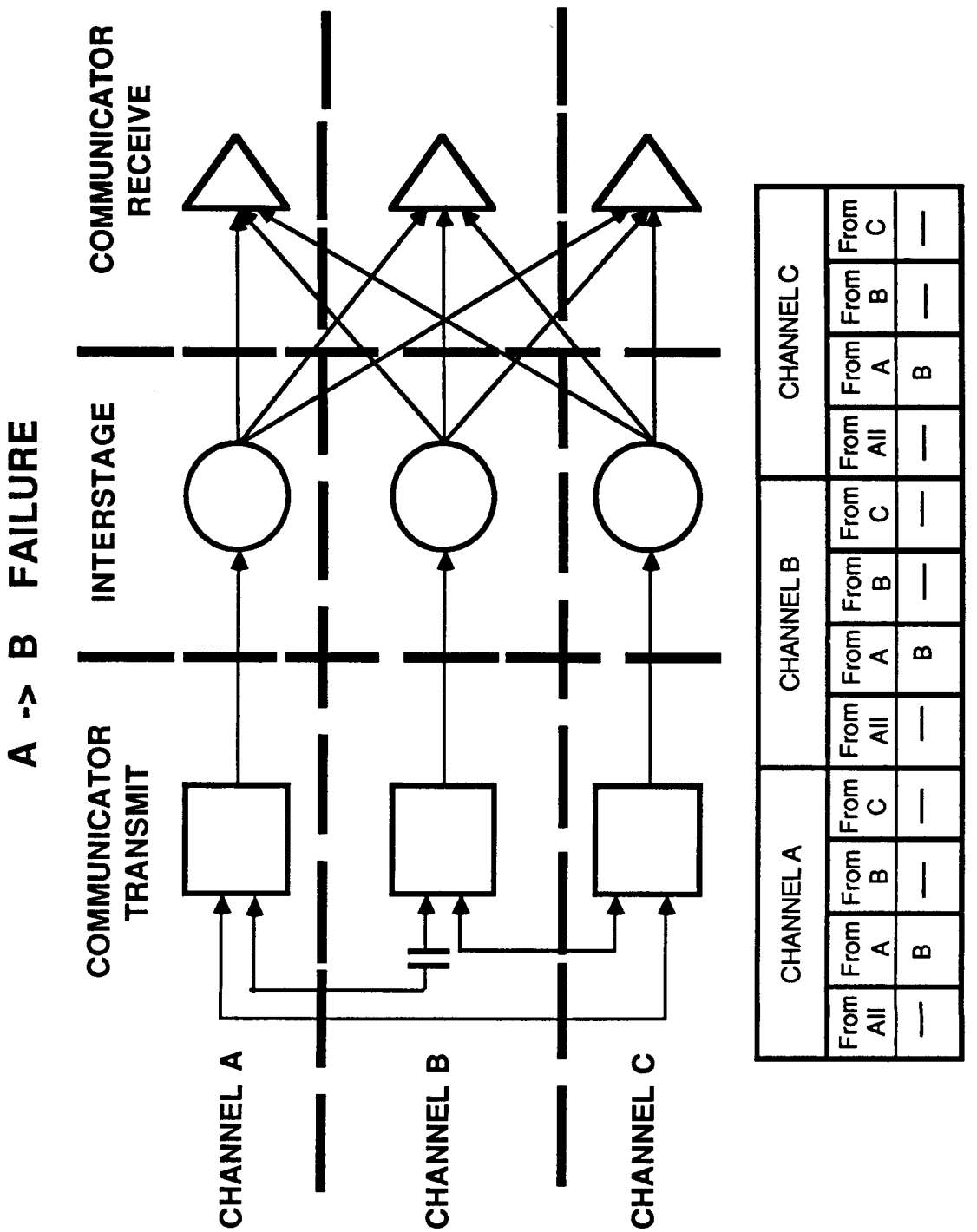


Figure 25. Latch Settings For Uni - Directional Transmitter - Transmitter Failure

4.2.1.1.9 Process Name: SOFT_ERROR_ANALYSIS

Inputs:

- Voted value of error latches from the three channels

Outputs:

- None

Description:

This subroutine checks for a "soft" error, i.e., non-congruent data sent during a FROM_ALL exchange. This type of error will set the same error latches on all three channels. The channel sending the non-congruent data will be identified as faulty on a FROM_ALL exchange (refer to the following figure).

After a "soft" error has been identified, the LOG_AND_RECONFIG subroutine is called to create the reconfiguration record and error log entry which will be used by the main Fast FDIR routine.

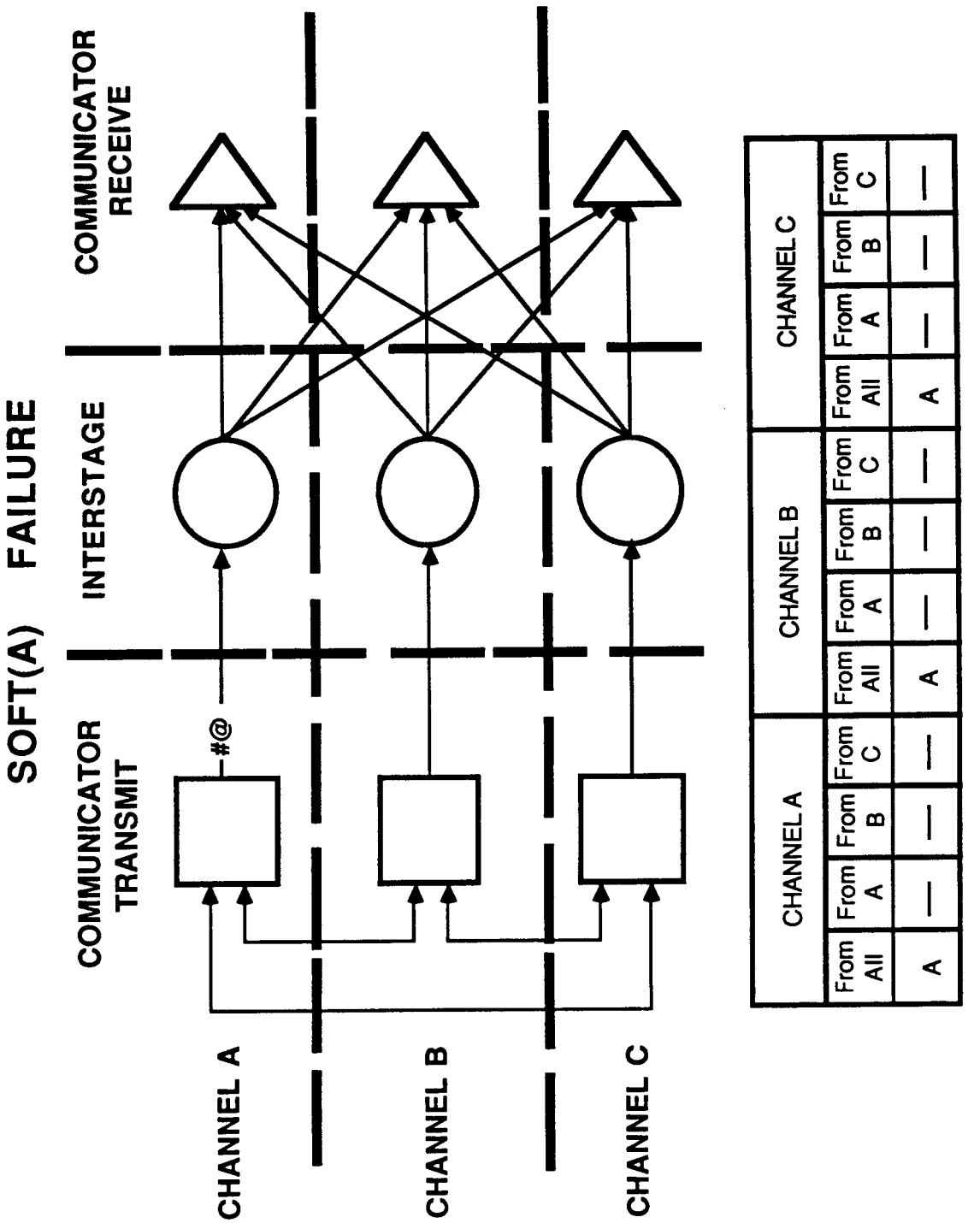


Figure 26. Latch Settings For "Soft" Failure

4.2.1.1.10 Process Name: ANALYZE_CLOCK_LATCHES

Inputs:

- Fault-tolerant clock error latches
- Fault-tolerant clock interstage error latches

Outputs:

- None

Description:

This routine analyzes the error latches set by the fault-tolerant clock hardware and identifies which clock element or clock interstage has failed.

Each channel gets its own copy of the two error latches (one from the interstage, one from the clock element) and combines them into one word. This word is then sent to the other two channels. All three channels do a software vote to get a consensus value of the three individual latches.

Each set of latches identifies errors about the other clock unit. Thus, the interstage latches identify faulty inputs from the clock elements. If Channel A's clock element fails, the interstage latches of all channels will show a fault in A. Conversely, the clock element latches identify faulty inputs from the interstage; if Channel A;'s interstage fails, the clock element latches of all channels will show a fault in A. These failures are illustrated in the following diagram,

This routine checks the current value of the latches against their previous value. If the value has changed, the routine determines whether a clock has failed or recovered, and makes an appropriate entry in the error log or the reconfiguration log.

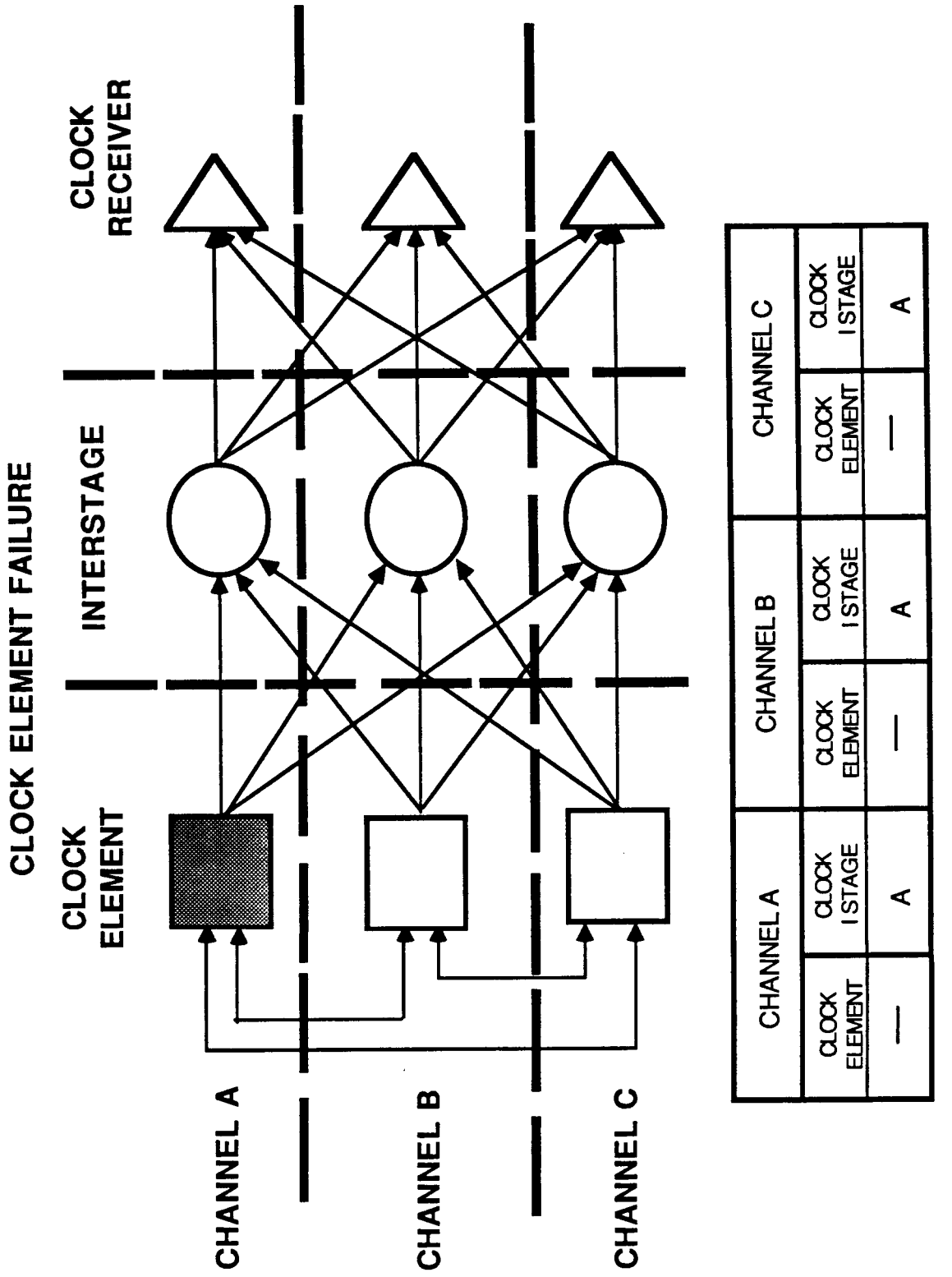


Figure 27. Latch Settings For Clock Element Failure

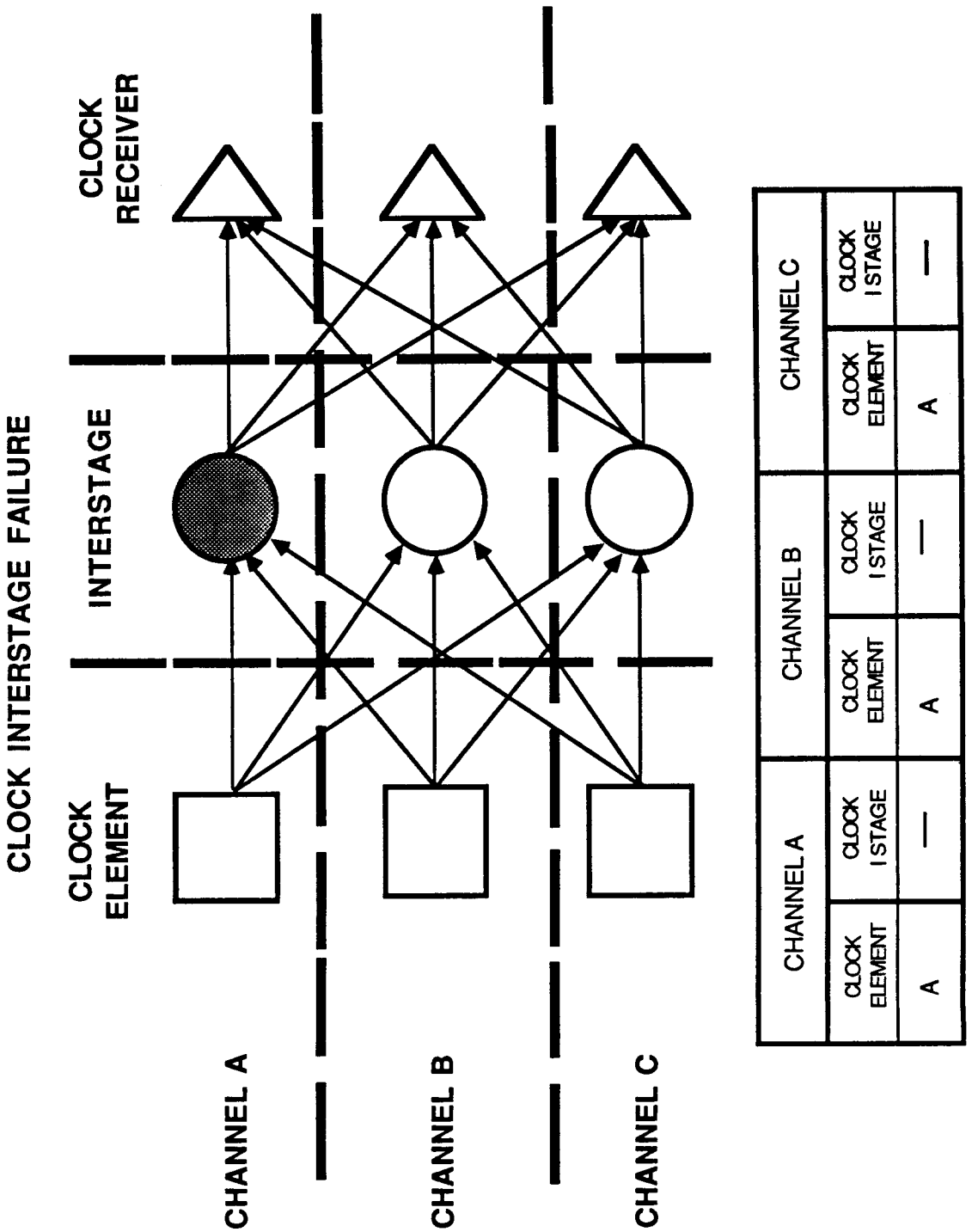


Figure 28. Latch Settings For Clock Interstage Failure

4.2.1.1.11 Process Name: RECORD_CHANGES

Inputs:

- Voted value of clock and clock interstage latches from the three channels
- Previous voted value

Outputs:

- Entry in error log or reconfiguration log

Description:

This subroutine examines differences between the current and previous values of the FTC latches and determines that either a clock has failed or recovered. If a clock has failed, an entry is made in the error log; if a clock has recovered an entry is made in the reconfiguration log. Differences between the current and previous values of the FTC interstage latches are also examined and recorded.

4.2.1.2 Watchdog Timer Reset

The watchdog timer is a hardware component whose purpose is to prevent infinite software loops or hardware faults from hanging up the system. After it has been started, the watchdog must be cleared periodically within a set time window; if it is not cleared within this window (i.e., either too early or too late) a hardware reset occurs. On the AIPS FTP this window is 60-120 milliseconds plus or minus 10%.

Two methods are required to reset the watchdog: 1) a periodic task which regularly clears the watchdog during normal system operation, and 2) a special procedure which is used during Lost Soul Sync when interrupts are disabled and the periodic task cannot run. These procedures are described on the following pages.

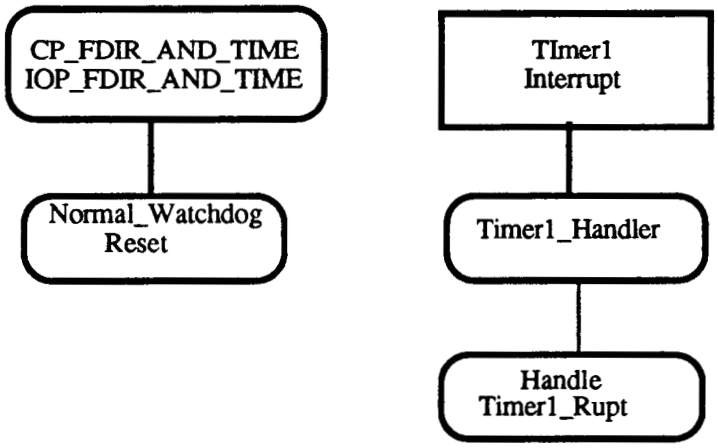


Figure 29. Watchdog Timer Reset Processes

4.2.1.2.1 Process Name: NORMAL_WATCHDOG_RESET

Inputs:

- Channel_Sync_State indicator
- Alternate_Iteration indicator

Outputs:

- Channel_Sync_State indicator
- Alternate_Iteration indicator

Description:

This routine is called by the highest priority task, the FDIR_AND_TIME task, to periodically clear the watchdog timer. This routine assumes that it is being called on every iteration of this task (i.e., every 40 ms) and that by clearing the timer on every other iteration it will be satisfying the timer requirements (refer to previous page). Clearing the watchdog in this way guarantees:

1) that timer interrupts are enabled;
2) that the runtime system is not in an infinite loop with interrupts off;
3) that the FDIR_AND_TIME task is being scheduled at its correct interval;
4) that the FDIR_AND_TIME task is not in an infinite loop;
5) that a processor which hangs waiting for a DTACK because it has referenced a non-existent address will be reset.

Special provision has been made for clearing the watchdog timer while the Channel_Sync routine is running, because Channel_Sync cannot be interrupted and the amount of memory to be aligned when a lone channel is being picked up is so great that it cannot possibly be completed within 120 milliseconds. During Channel_Sync, one of the unused interval timers is used to periodically cause an interrupt, at which time the watchdog is cleared. After Channel_Sync is completed, the job of clearing the watchdog reverts to the periodic routine, Normal_Watchdog_Reset. However, its alternate iteration test is now invalid; for example, it may be the alternate iteration but the watchdog may have been cleared only a few milliseconds ago while Channel_Sync was running. Thus, immediately after Channel_Sync has completed, this routine must use a time comparison to determine when it should clear the watchdog. After the watchdog has been reset once using this method, the alternate iteration method becomes appropriate again.

The alternate iteration method is the preferred method because this guarantees that the FDIR_AND_TIME task is being scheduled at its correct interval, whereas always using a time comparison to clear the watchdog would not guarantee this.

4.2.1.2.2 Process Name: Timer1_Handler
Handle_Timer1_Rupt

Inputs:

- Channel_Sync_State indicator

Outputs:

- Time watchdog last reset
- New value in timer

Description:

As mentioned in the process description for Normal_Watchdog_Reset, under normal conditions the watchdog timer can be cleared by a periodic task. Special provision must be made, however, for clearing the watchdog while the Channel_Sync routine is running, because Channel_Sync cannot be interrupted and the amount of memory to be aligned when a lone channel is being picked up is so great that it cannot possibly be completed within 120 milliseconds. During Channel_Sync, one of the unused interval timers is used to periodically cause an interrupt, at which time the watchdog is cleared.

These routines are invoked when the timer interrupt occurs. They clear the watchdog and set the timer to go off again. After Channel_Sync is complete, the job of clearing the watchdog reverts to the periodic procedure, and so an interrupt which occurs after Channel_Sync is finished is ignored and the timer is not reset.

Two routines are required here because the interrupt handler must conclude with an RTE instruction and must therefore be written in assembler, while the actual work of checking flags and clearing the watchdog is more conveniently done in Ada. Thus the Timer1_Handler routine is an outer shell which ends with an RTE instruction, while the Handle_Timer1_Rupt routine does the actual work required.

4.2.1.3 Background Selftests

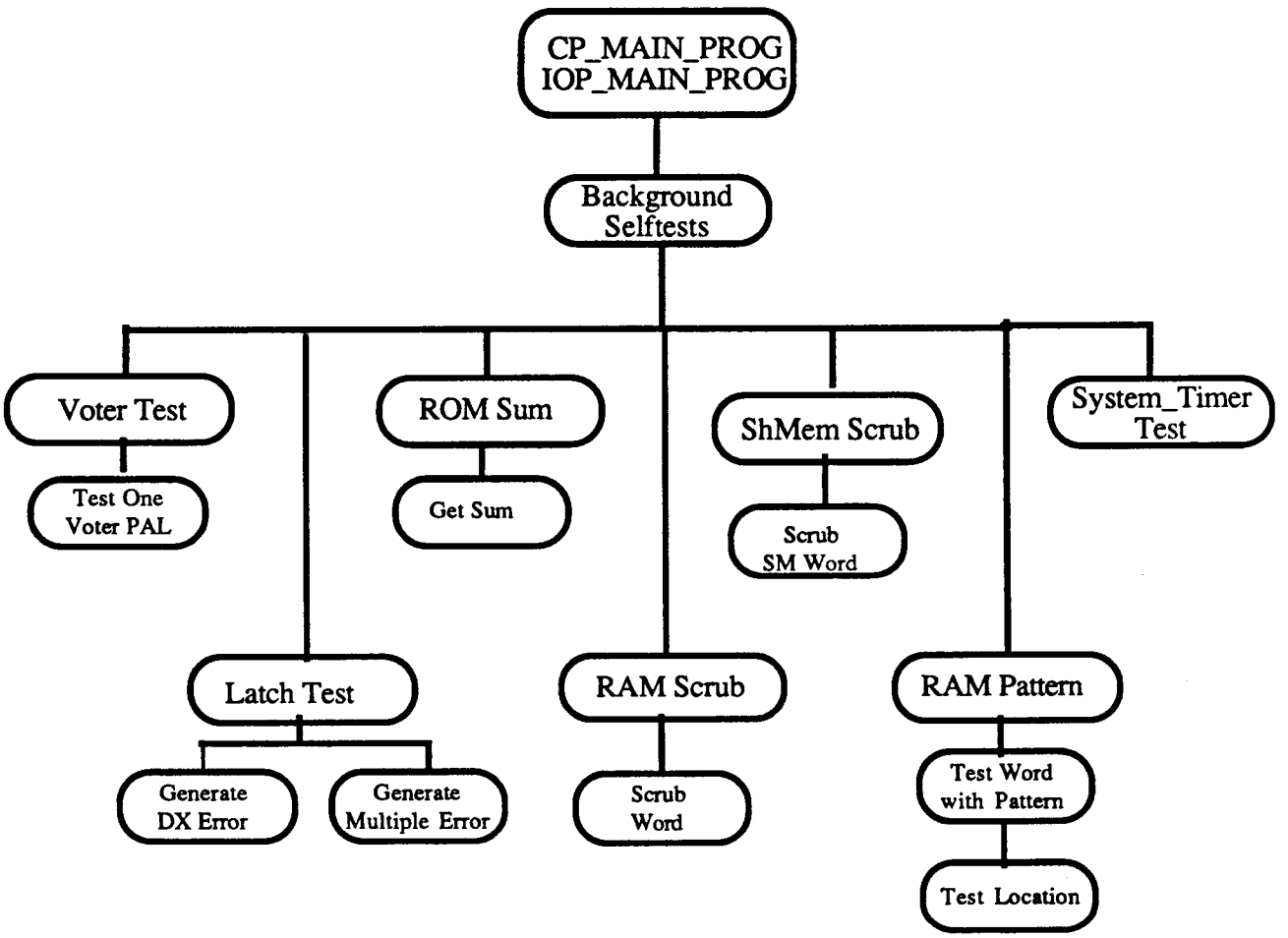


Figure 30. Background Selftest Processes

4.2.1.3.1 Process Name: BACKGROUND_SELFTESTS

Inputs:

- None

Outputs:

- None

Description:

This is the main procedure of the selftest process and is run as part of the main program (CP_MAIN_PROG or IOP_MAIN_PROG). It calls appropriate procedures to run the following tests:

- Data exchange voter test
- Data exchange error latch test
- ROM sum check
- RAM scrub
- Shared memory scrub
- Pattern test on offcard RAM
- Pattern test on oncard RAM
- System timer test

When an error is detected, the reconfiguration is not done immediately, but Fast FDIR is signaled that a reconfiguration is required and the necessary information is passed in a global variable. Further testing is then delayed until FAST has done the reconfiguration. Testing resumes from the point at which the error was detected.

This procedure is also executed at system startup when the entire suite of selftests is run without interruption before any other tasks are started. In the laboratory testing environment, this startup execution is optional and is controlled by a flag set by the operator in RAM.

4.2.1.3.2 Process Name: VOTER_TEST

Inputs:

- Current configuration

Outputs:

- Patterns to be used in each test

Description:

This selftest exercises the voting circuitry and error latches in the data exchange hardware. It verifies that when the voters receive non source-congruent data they produce a correct (majority) result and that the error latches are correctly set to indicate the dissenting channel.

The test is done by calling a subroutine which does a FROM_ALL exchange in which one channel transmits a pattern different from the other two channels. After the exchange, the called subroutine verifies that the correct result is read by all channels and that all channels correctly identify the dissenting channel on a FROM_ALL exchange. (Refer to following section on TEST_ONE_VOTER_PAL for more details.)

In order to thoroughly test the voters, the basic test must be repeated with a series of patterns. The data exchange mechanism acts on a 16-bit word. The voters, however, consist of eight 2-bit wide PALs, i.e., the 16-bit word is fed into the voters two bits at a time. This means that it is not necessary to test every possible combination of the bits in a 16-bit word. Rather, the bits can be tested in 2-bit units, i.e., first bits 0-1, then bits 2-3, then bits 4-5, etc. The bits being tested must exercise every possible combination of bits (4 for a 2-bit unit), but the value of the rest of the word is irrelevant.

Thus, a test of any particular 2-bit unit consists of the 16 combinations of patterns shown in the following table.

Concurring Channels	Dissenting Channels
00	00
	01
	10
	11
01	00
	01
	10
	11
10	00
	01
	10
	11
11	00
	01
	10
	11

Figure 31. Test on a Two-Bit Unit

This test is repeated eight times, once for each 2-bit unit in the 16-bit word. Since there are three channels, the entire sequence described above is repeated three times, with each channel having a turn at being the dissenter. This routine generates the concurring and dissenting patterns which are used for each test.

The voter test is not started unless all three channels are in sync. If the TEST_ONE_VOTER_PAL subroutine reports that an error was detected, the voter test is discontinued and the next selftest starts execution.

4.2.1.3.3 Process Name: TEST_ONE_VOTER_PAL

Inputs:

- Data exchange receiver values and error latches

Outputs:

- Reconfig_cmd record

Description:

This routine does the actual data exchange required to test a particular PAL in the data exchange voters and analyzes the results. This exchange is a FROM_ALL exchange in which one channel transmits a pattern different from the other two channels. After the exchange the result read back by all channels should be the pattern sent by the two concurring channels, and all error latches should say that there was an error in the dissenting channel on a FROM_ALL exchange. If these conditions are not met, the test is considered to have failed.

For each FROM_ALL exchange done during the test, a "good" pattern (transmitted by the concurring channels) and a "bad" pattern (transmitted by the dissenting channel) are generated. The channels send out the different patterns by using their id as an index to an array containing the address of the desired pattern. Three different arrays, corresponding to who is the dissenting channel at the time, are used. Thus when A is the dissenter, addresses are chosen from a table where

```
table (1) = ^bad_pattern      -- Ch A uses this address
table (2) = ^good_pattern     -- Ch B uses this address
table (3) = ^good_pattern     -- Ch C uses this address
```

Note: Care must be exercised when using channel id as an index to an array. The current definition of the channel_id type gives a value of 0 for A, 1 for B, and 2 for C. When Ada computes addresses within an array, it treats an index of 0 as a special case. Thus if the channels used their id as an index, channel A would drop out of sync. The solution is to convert the id from 0, 1, and 2 to 1, 2 and 3, and then have a 0th entry in the table which is unused.

Verifying the Expected Results

To verify that each channel has obtained the correct result after the FROM_ALL exchange, each channel distributes its result to the other two. The result from each channel is then compared to the expected result; any unequal result indicates an error. After the exchange, the error latches are also analyzed to verify their correct setting. Each channel exchanges

its copy of the FROM_ALL error latch; these values are then compared to the expected latch value. Any unequal result indicates an error.

A channel may appear to fail this test when actually it has dropped out of sync but has not yet been detected as missing by Fast FDIR. Thus when a particular channel fails a voter test, it is not immediately identified as having a voter error. Instead, the routine does its own presence test using the dedicated data exchange to determine if the channel is still synchronized. If it is, a reconfiguration record and error log entry are created for fast FDIR; if the channel is not synchronized, the error is ignored. In either case, the voter test is then discontinued and the next selftest starts executing.

Additional Considerations

1) The test is not done if the FROM_ALL error latch is set already.

2) Interrupts must be disabled during the critical period which includes doing the FROM_ALL exchange, reading the result, and reading the FROM_ALL error latch.

4.2.1.3.4 Process Name: LATCH_TEST

Inputs:

- Current configuration

Outputs:

- None

Description:

This selftest tests the error latches in the data exchange hardware for all types of exchanges. It verifies that when non source-congruent data is received by the voters, the error latches correctly indicate the channel at fault for the particular type of exchange.

The test is done by calling subroutines which cause two channels to do one type of exchange while the third channel does a different type. The subroutines verify that single and multiple errors are correctly latched. (Refer to following sections on GENERATE_DX_ERROR and GENERATE_MULTIPLE_ERROR.)

The latch test is not started unless all three channels are in sync. If either subroutine reports that an error was detected, the latch test is discontinued and the next selftest starts running.

4.2.1.3.5 Process Name: GENERATE_DX_ERROR

Inputs:

- Data exchange error latches

Outputs:

- Reconfig_cmd record

Description:

This routine verifies that single data exchange errors are correctly latched. This is done by causing two channels to do one type of exchange while the third channel does a different type. The results of the exchange do not need to verified, since this type of fault would be caught by the voter selftest. However, the error latches are checked to ensure that they correctly identify the dissenting channel on the particular type of exchange. If the error latches do not contain the expected value, the test is considered to have failed. By reading the error latches without clearing them, the routine guarantees that a particular fault will not be attributed to the wrong latch because of an address decoding problem.

The table on the following page shows the tests that are run and the expected results of each test.

In order for different channels to perform different exchanges without going out of sync, the test is table-driven, with each channel using its own id as an index into the table. (Refer to previous section on the voter test for cautions when using channel id as an index.) The table identifies the following for each test.

- Address of the DX transmitter to be used by each channel in doing the exchange. Each channel accesses this address by using its own id as an index.
- The pattern to be exchanged. Each channel accesses this pattern by using its own id as an index.
- Expected setting of each channel's error latches for this test
- Expected setting of each channel's error latches for the multiple-error test (refer to following section)

After the exchange is done, the error latches from each channel are read and distributed. If the latches do not equal the value expected for each channel, the test is considered to have failed.

Test	Dissenting Channel	Concurring Channel	Error Latches		
			Ch A	Ch B	Ch C
	A	B,C			
(1)	From_V	From_B	--	From_B = A Faulty	From_B = A Faulty
(2)	From_V	From_C	--	From_C = A Faulty	From_C = A Faulty
(3)	From_A	From_V	From_A = A Faulty	--	--
	B	A,C			
(4)	From_V	From_A	From_A = B Faulty	--	From_A = B Faulty
(5)	From_V	From_C	From_C = B Faulty	--	From_C = B Faulty
(4)	From_B	From_V	--	From_B = B faulty	--
	C	A,B			
(4)	From_V	From_A	From_A = C Faulty	From_A = C Faulty	--
(5)	From_V	From_B	From_B = C Faulty	From_B = C Faulty	--
(4)	From_C	From_V	--	--	From_C C Faulty

Figure 32. Data Exchange Error Latch Tests and Expected Results

A channel may appear to fail this test when actually it has dropped out of sync but has not yet been detected as missing by Fast FDIR. Thus when a particular channel fails a latch test, it is not immediately identified as having a latch error. Instead, the routine does its own presence test using the dedicated data exchange to determine if the channel is still synchronized. If it is, a reconfiguration record and error log entry are created for Fast FDIR; if the channel is not synchronized, the error is ignored. In either case, the latch test is then discontinued and the next selftest starts executing.

Additional Considerations

1) The test is not done if any of the error latches are already set.

2) Interrupts must be disabled during the critical period which includes doing the exchange and reading the appropriate error latch.

4.2.1.3.6 Process Name: GENERATE_MULTIPLE_ERROR

Inputs:

- Data exchange error latches

Outputs:

- Reconfig_cmd record

Description:

This routine verifies that multiple data exchange errors are correctly latched. This is done by repeatedly causing two channels to do one type of exchange while the third channel does a different type. This test is very similar to the single-error test described in the preceding section, and the reader should read this section before proceeding here. This test differs from the single-error test only in that all errors for a particular channel are generated before the error latches are read. The errors are actually generated twice before the latches are read, so that the test verifies that both the same error and different errors can be correctly OR'ed into the latches.

4.2.1.3.7 Process Name: ROM_SUM

Inputs:

- Starting and ending addresses of the memory to be summed

Outputs:

- Reconfig_cmd record

Description:

This test verifies that all three channels have identical values in ROM. Due to the static nature of ROM, it is not necessary to check every individual location. Rather, each channel computes a checksum of all PROM locations and then sends its own sum to the other two channels. A consensus value is obtained by doing a software vote of the individual values; each individual channel's sum is then compared to the voted sum to identify any errors. Although there is a small probability that two errors will cancel each other and the sum check will not detect them, that probability is small enough to be neglected.

A channel may appear to fail this test when actually it has dropped out of sync but has not yet been detected as missing by Fast FDIR. Thus when a particular channel fails the ROM sum test, it is not immediately identified as having a ROM error. Instead this routine does its own presence test using the dedicated data exchange to determine if the channel is still synchronized. If it is, the ROM sum error is considered to be a legitimate error and a reconfiguration record and error log entry are created for Fast FDIR; otherwise the error is ignored.

4.2.1.3.8 Process Name: GET_SUM

Inputs:

- Starting and ending addresses of the memory to be summed

Outputs:

- PROM checksum

Description:

This routine adds the contents of each 16-bit word in PROM and returns a 16-bit word containing the result.

4.2.1.3.9 Process Name: RAM_SCRUB

Inputs:

- Starting and ending address of area to be tested

Outputs:

- None

Description:

This procedure does a RAM scrub on the specified memory area. This ensures that the values in each location are congruent among the three channels. The subroutine SCRUB_WORD is called for each word in the area to be scrubbed (see following section).

4.2.1.3.10 Process Name: SCRUB_WORD

Inputs:

- Address of 16-bit word to be scrubbed

Outputs:

- Reconfig_cmd record

Description:

This procedure checks the specified memory location to ensure that the values are congruent, or identical, among the three channels. After the location has been read, each channel currently in the configuration sends its value to the other two channels. The value from each channel is then compared to a voted value.

A channel may appear to fail this test when actually it has dropped out of sync but has not yet been detected as missing by Fast FDIR. Thus when a particular channel fails the test, it is not immediately identified as having a RAM error. Instead, this routine does its own presence test using the dedicated data exchange to determine if the channel is still synchronized. If it is not, the error is ignored. If the channel is still synchronized, an error log entry identifying the location and value of the difference is created for fast FDIR and the test is repeated. If the word is still different, a voted value is written into the location and then read back. If the correct value is not read back, the failure is considered a hard failure and a reconfiguration record notifying Fast of the error is created.

There are some RAM locations whose contents may legitimately differ from channel to channel, e.g., the channel id and non-congruent data from external sources. Those variables which are known in advance as potentially different are kept in a non-congruent data area which is not scrubbed. But since not all the different RAM locations can be predicted (because, for example, of interrupts causing registers to be stored on the stack), at the present time only static variables are scrubbed.

4.2.1.3.11 Process Name: SHMEM_SCRUB

Inputs:

- Starting and ending address of area to be tested

Outputs:

- None

Description:

This procedure scrubs the dual-ported memory shared by the IOP and CP to ensure that the values in each location are congruent among the three channels. The subroutine SCRUB_SM_WORD is called for each word in the area to be scrubbed (see following section). This procedure is only executed by the CP.

4.2.1.3.12 Process Name: SCRUB_SM_WORD

Inputs:

- Address of 16-bit word to be scrubbed

Outputs:

- Reconfig_cmd record

Description:

This procedure checks each memory location to ensure that the values are congruent, or identical, among the three channels. After a particular location has been read, each channel currently in the configuration sends its value to the other two channels. The value from each channel is then compared to a voted value.

A channel may appear to fail this test when actually it has dropped out of sync but has not yet been detected as missing by Fast FDIR. Thus when a particular channel fails the test, it is not immediately identified as having a RAM error. Instead, this routine does its own presence test using the dedicated data exchange to determine if the channel is still synchronized. If it is not, the error is ignored. If the channel is still synchronized, a reconfiguration record and error log entry identifying the location and value of the difference are created for later use by Fast FDIR. No attempt is made, as in the RAM scrub, to write a voted value and repeat the test because the IOP (this test is only done by the CP) could have written something into that location that the CP would be destroying.

4.2.1.3.13 Process Name: RAM_PATTERN

Inputs:

- Starting and ending addresses of area to be tested

Outputs:

- None

Description:

This procedure tests the functionality of each word in the specified RAM area by writing a particular pattern to each word and reading it back. These patterns test the ability of each bit in the word to hold both 0 and 1 values. The subroutine TEST–WORD–WITH–PATTERN is called for each word in the area to be tested (refer to following section).

4.2.1.3.14 Process Name: TEST_WORD_WITH_PATTERN

Inputs:

- Address of 16-bit word to be tested

Outputs:

- Reconfig_cmd record

Description:

This procedure tests the functionality of each word in RAM by writing a particular pattern to each word and reading it back. After a word has been written and read, each channel currently in the configuration sends its result from the test to the other two channels. The channels then verify that the test was passed on each individual channel.

Six different patterns, which represent the standard sequence of marching ones surrounded by zeros and marching zeros surrounded by ones, are written to each word. These patterns test each bit's ability to hold both a 0 and a 1 no matter what the setting of its neighbor bits.

A channel may appear to fail this test when actually it has dropped out of sync but has not yet been detected as missing by Fast FDIR. Thus when a particular channel fails the test, it is not immediately identified as having a RAM error. Instead, this routine does its own presence test using the dedicated data exchange to determine if the channel is still synchronized. If it is not, the error is ignored. Otherwise, it is assumed to be a legitimate RAM error and an error log entry is created for Fast FDIR. The test is then repeated; if the error occurs again, the routine assumes the error represents a hard failure and a reconfiguration record is created for Fast FDIR.

The actual reading and writing of the test pattern is done by calling an assembly language subroutine which saves the contents of the location in a register before testing it (refer to following section on TEST_LOCATION).

C-2

4.2.1.3.15 Process Name: TEST_LOCATION

Inputs:

- Address of word to be tested
- Pattern to be written

Outputs:

- Value read after test pattern was written

Description:

This is an assembly language function which writes a specified pattern to a specified word in RAM, then reads it back and returns the value read to the caller.

The current value of the word to be tested is saved in a register before the test is started and restored when the test is finished. In the laboratory testing environment, where program code probably resides in RAM, the routine must take care not to write over its own instructions. The beginning and end of the critical section of code are identified with labels, and any locations within this area are not tested.

4.2.1.3.16 Process Name: SYSTEM_TIMER_TEST

Inputs:

- Local and voted values of the system timer

Outputs:

- Reconfig_cmd record

Description:

This test verifies that the system timers in all three channels have the same value. The channels first obtain a consensus value for the timer by doing a FROM_ALL exchange. Each channel currently in the configuration then sends its own timer value to the other channels and these individual values are compared to the voted value to identify any errors.

A channel may appear to fail this test when actually it has dropped out of sync but has not yet been detected as missing by Fast FDIR. Thus when a particular channel fails the test it is not immediately identified as having a timer error. Instead, the routine does its own presence test using the dedicated data exchange to determine if the channel is still synchronized. If it is, a reconfiguration record and error log entry are created for Fast FDIR; if the channel is not synchronized, the error is ignored.

4.2.1.3.17 Process Name: GET_LATEST_CONFIG

Inputs:

- Current configuration at runtime
- Startup configuration

Outputs:

- Array identifying up channels

Description:

This routine converts the current configuration from a record of three booleans to an array of three booleans. The selftest routines can work more easily with the configuration represented as an array rather than a record. The input configuration record will either be the global configuration record maintained in the CONFIG package during normal system execution, or it will be the startup configuration record, which is a record local to the TEST package and which is used when the selftests are running uninterrupted at system startup.

4.2.1.3.18 Process Name: CREATE_RECONFIG

Inputs:

- Startup_selftest flag
- Faulty channel id and fault id

Outputs:

- Reconfig_cmd record

Description:

This routine creates the reconfiguration record that is passed to Fast FDIR when a fault is detected. This record is not created, however, if a fault is detected while the selftests are running at system startup.

4.2.1.3.19 Process Name: DELAY_UNTIL_FAST

Inputs:

- Startup_selftest flag

Outputs:

- None

Description:

This subroutine is called after a fault has been detected and a reconfiguration record created for Fast FDIR. Its function is to suspend the selftest task until Fast FDIR has been able to reconfigure around the faulty channel. It does not want to do this, however, if the selftests are running at system startup, since at that time they are running with interrupts disabled and Fast will never execute.

4.2.1.3.20 Process Name: CHANNEL_PRESENT

Inputs:

- Channel id

Outputs:

- Boolean indicating if specified channel is synchronized

Description:

This routine determined if the specified channel is synchronized with at least one other channel by doing a presence test using the dedicated data exchange registers. While a presence test using the shared data exchange would be simpler (refer to Section 4.2.1.1.4), it requires locking the shared bus, which is not an acceptable action for this task. Since any word that is transmitted using the dedicated exchange registers will remain in the exchange receiver for more than one FTC cycle, it is possible that a channel only slightly behind would see the expected word and incorrectly conclude that it was in sync. A valid presence test using the dedicated exchange must therefore do two rounds of exchanges: (1) each channel transmits a unique pattern and determines what other channels it sees, and (2) each channel transmits a unique pattern indicating who it saw. A software-voted value of who was seen is then obtained and compared to who the particular channel saw. If a majority vote could not be obtained or who the particular channel saw does not agree with the voted value, the particular channel itself is unsynchronized and returns FALSE. If the particular channel is synchronized, it compares who each individual channel saw with the voted value to determine if the specified channel is synchronized or not.

4.2.1.4 68010 Exception Handler

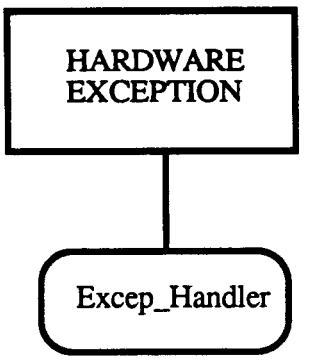


Figure 33. Exception Handler Processes

4.2.1.4.1 Process Name: EXCEP_HANDLER

Inputs:

- Program counter, status register and vector offset from the stack

Outputs

- Entry in the non-congruent log

Description:

This routine receives control when a hardware exception such as an illegal instruction or addressing error occurs.

The type of exception and relevant information such as the program counter, status register, and selected registers are logged in the non-congruent log. A presence test is done to determine if the exception occurred synchronously on all channels or if only one channel was affected, and the result is logged. The processor then does a software restart.

4.2.2 Channel Recovery Process Descriptions

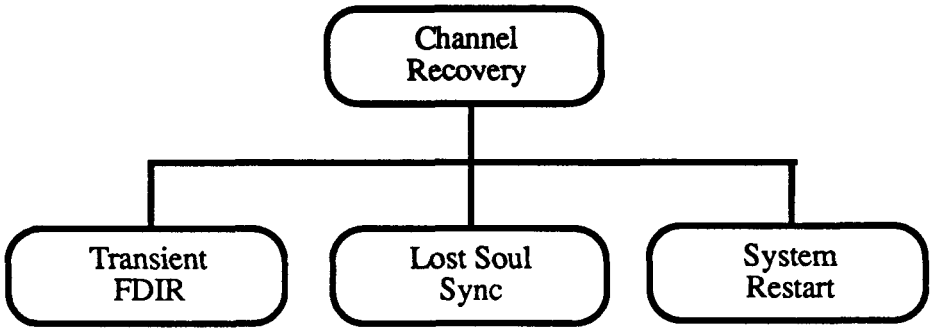


Figure 34. Channel Recovery Functions

4.2.2.1 Transient FDIR

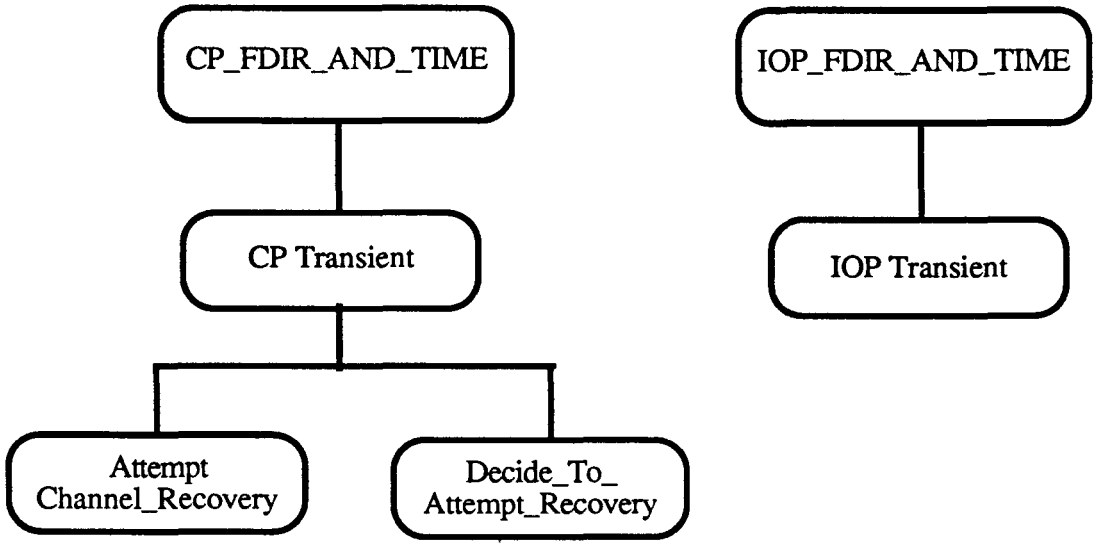


Figure 35. Transient FDIR Processes

4.2.2.1.1 Process Name: CP_TRANSIENT

Inputs:

- FAST_INFO array set by Fast FDIR
- CHANNEL_STATUS array
- CHANNEL_ON_TRIAL array
- LOST_SOUL_STATUS boolean
- IOSS_RESTORE_CMD boolean

Outputs:

- CHANNEL_STATUS array
- Channel_enable_cmd for CP Fast
- Sched_lost_soul command in shared memory
- IOSS_restore_cmd boolean

Description:

This is the main procedure of the TRANSIENT_FDIR process on the CP and is called by the CP_FDIR_AND_TIME task. It does all the decision making for both the CP and the IOP and notifies the IOP when either a resynchronization is to take place or the IOS must be reinitialized. It notifies CP Fast FDIR when a channel is to be permanently enabled.
To support this decision making, this procedure maintains four variables about each channel:

- FAIL_LEVEL: the amount of time the channel must operate correctly after it has recovered from a fault (i.e., the length of its probation). This variable is incremented when a fault has been detected and decremented as the probation period passes.

- RETRY_BACKOFF: the amount of time between recovery attempts. This variable is set when a recovery is attempted and is doubled with each succeeding attempt until MTTR is reached.

- RETRY_TIME: the amount of time remaining until another recovery attempt can be made (assuming the previous one failed). This variable is set with the RETRY_BACKOFF value when a recovery is attempted; if the recovery fails, it is decremented as time passes.

- CHANNEL_ON_TRIAL: whether or not a channel is on probation. This variable is set when a channel is initially recovered, reset when the probation period is over and the IOP is signalled to start its IOSS restore task, and finally cleared when the IOP signals that the IOSS restore task is complete.

This procedure has work to do only when a channel is down. It makes the following sequence of tests; as soon as any condition is met, the current iteration of Transient is terminated.

- Checks for a new error, i.e., one detected by Fast since the last iteration of Transient. If there has been a new error, the FAIL_LEVEL for the channel is updated and an attempt is made to recover the channel.

- Checks if a channel is on probation. If so, it counts down the probation time by decrementing FAIL_LEVEL. When the probation time has expired, the IOP is notified to start the IOSS_RESTORE task and the CHANNEL_ON_TRIAL variable is set to WAITING_FOR_IOP.

- Checks if the LOST_SOUL_SYNC task was previously scheduled, but has not run yet. If so, there is nothing to do.

- Checks if the LOST_SOUL_SYNC task was previously scheduled and has picked up the failed channel. If so, it begins the channel's probation period by setting the CHANNEL_ON_TRIAL variable to yes.

- Checks if there was a previously detected failure (i.e., CHANNEL_ON_TRIAL = NO) whose RETRY_TIME must be decremented. If so, RETRY_TIME is decremented; when it goes to zero, another attempt is made to recover the channel.

- Checks if the channel has passed its probation period and is waiting for the IOP to finish the IOSS_RESTORE task. If the IOSS_RESTORE task is finished, Fast FDIR is notified that the channel an be permanently enabled.

The above checks must be made in the given order because of their priority. The check for a new error must be made first because this condition cancels all previously existing states, for example, a channel being on trial. Checking on the status of the LOST_SOUL_SYNC task must be done before checking for CHANNEL_ON_TRIAL = NO so that LOST_SOUL_SYNC does not get scheduled multiple times.

4.2.2.1.2 Process Name: ATTEMPT_CHANNEL_RECOVERY

Inputs:

- FAST_INFO array
- Error latches from Fast FDIR

Outputs:

- Sched_lost_soul record in shared memory
- LOST_SOUL_STATUS boolean
- Scheduling of Lost_Soul_Sync task

Description:

This procedure is executed when a channel's RETRY_TIME (the time between successive recovery attempts) has expired.

First the variables governing the recovery attempt (RETRY_BACKOFF and RETRY_TIME) are set for the next attempt. Then the procedure checks for the type of failure (synchronization or data exchange errors).

To recover an unsynchronized channel, the LOST_SOUL_SYNC task must be scheduled and a scheduling command written in shared memory for the IOP. The results of the LOST_SOUL_SYNC task will not be known until a subsequent iteration of the TRANSIENT task.

To recover from data exchange errors, the data exchange error latches are checked for the absence of errors. If no errors are detected, a command to temporarily enable the channel is sent to Fast FDIR and its probation period begins.

4.2.2.1.3 Process Name: DECIDE_TO_ATTEMPT_RECOVERY

Inputs:

- INITIAL_RECOVERY_ATTEMPT boolean
- RETRY_TIME
- CHANNEL_RECOVERY_MODE
- CHANNEL_REPAIRED

Outputs:

- RETRY_TIME
- CHANNEL_REPAIRED

Description:

When a fault is first detected, it is regarded as a transient and recovery is attempted immediately. If the attempt fails, the time between successive attempts increases until MTTR is reached. If the channel has not been recovered by this time, the fault is regarded as a hard failure. Action at this point depends on the user-specified recovery mode: either recovery is attempted every MTTR or recovery is attempted only when an operator indicates the channel is repaired. (The default in the laboratory testing environment is to attempt every MTTR).

4.2.2.1.4 Process Name: IOP_TRANSIENT

Inputs:

- Sched_IOSS_restore command from CP
- Sched_lost_soul command from CP

Outputs:

- IOSS_RESTORE_SCHEDULED boolean

Description:

This is the main procedure of the TRANSIENT_FDIR process on the IOP and is called by the IOP_FDIR_AND_TIME task. It has work to do only when a channel is down. It checks for commands from the CP, which makes all the decisions of the TRANSIENT_FDIR process It can receive two commands from the CP:

- to schedule the LOST_SOUL_SYNC task, which tries to pick up an unsynchronized channel, or
- to schedule the IOSS_RESTORE task, which re-initializes the IOS dual-ported memory and registers.

The command to schedule the LOST_SOUL_SYNC task specifies the absolute time at which the task should run. This is necessary so that neither the CP nor the IOP spend unnecessary time waiting in the LOST_SOUL_SYNC task to do their handshake.

The command to schedule the IOSS_RESTORE_TASK is cleared by the task itself to indicate that it is completed. This procedure uses the IOSS_RESTORE_SCHEDULED boolean in FDIR_GLOBALS to prevent itself from scheduling the task multiple times.

4.2.2.1.5 Process Name: FDIR_GLOBALS

Inputs:

- None

Outputs:

- None

Description:

This package contains types and variables shared by different tasks involved in the FDIR process.

- Fast FDIR must pass several different pieces of information to Transient FDIR. This information is written by Fast, then read and cleared by Transient.
  - The error latches read by the error latch analysis routine
  - An array of flags, one for each channel, to indicate whether a channel's failure caused it to drop out of sync or not.
- The Display task can accept a command from an operator that a failed channel has been repaired. It passes this information to Transient FDIR in an array of booleans, one for each channel.
- CP Transient and IOP Transient communicate through commands in shared memory. The type for the Lost_Soul_Sync scheduling command is defined here.
- IOSS Restore must indicate to IOP Transient when the IOSS Restore task is done. It does this by clearing a flag which IOP Transient has previously set.

4.2.2.2 Lost Soul Sync

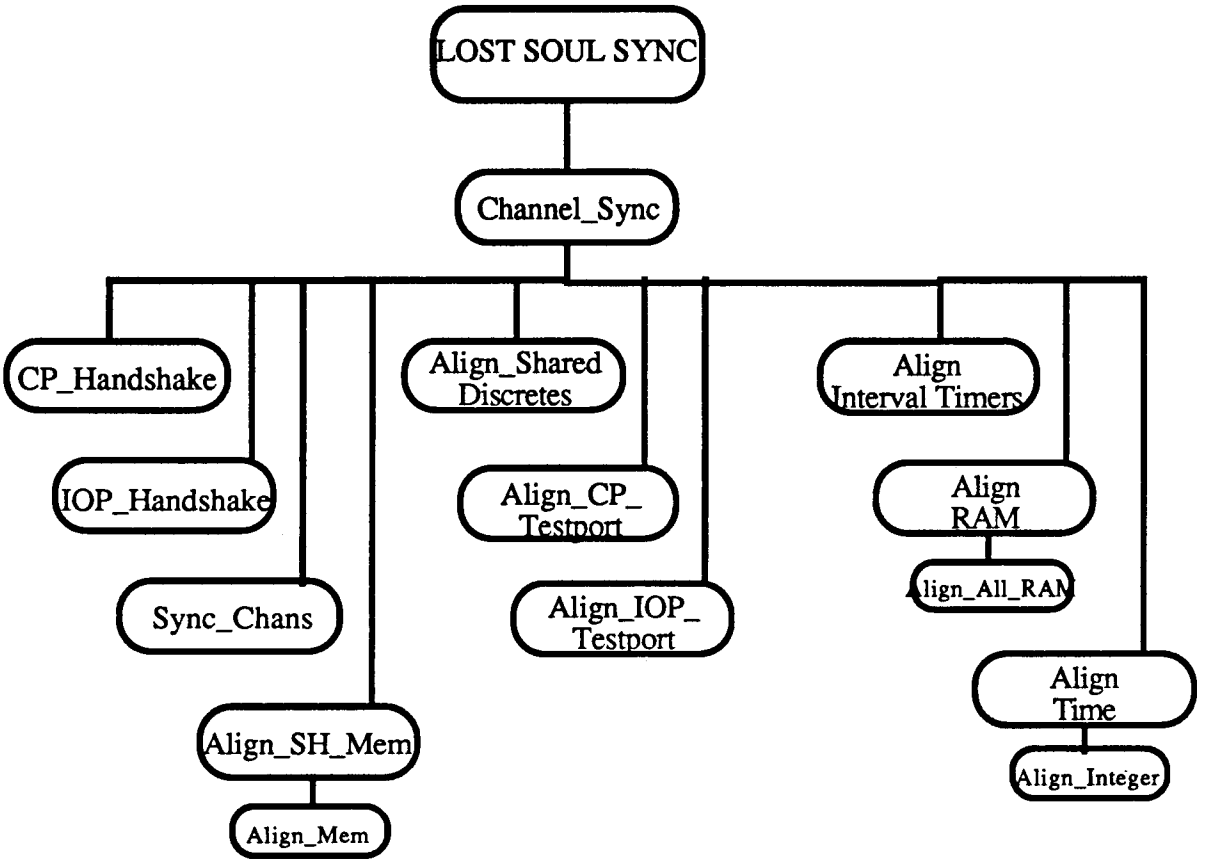


Figure 36. Lost Soul Sync Processes

4.2.2.2.1 Process Name: LOST_SOUL_SYNC

Inputs:

- None

Outputs:

- None

Description:

The LOST_SOUL_SYNC task is scheduled when Transient FDIR has determined that an attempt should be made to pick up a channel that has become desynchronized. The task is scheduled to run at an absolute time and is descheduled after one iteration. It calls the CHANNEL_SYNC routine which tries to pick up the lost channel.

4.2.2.2.2 Process Name: CHANNEL_SYNC

Inputs:

- Integer version of current configuration
- Reason for synching - initial or pickup
- Type of sync variable in global memory

Outputs:

- Entry in reconfig log
- Type of sync variable in global memory

Description:

This is the main procedure of the LOST_SOUL_SYNC task. It is also the procedure used to synchronize channels at system startup. It is where channels who are together and channel(s) who are alone find each other. Its main steps are:

- to synchronize the channels to the instruction level, and
- to align the channels' volatile memory and registers, i.e., make them identical amongst the synchronized channels.

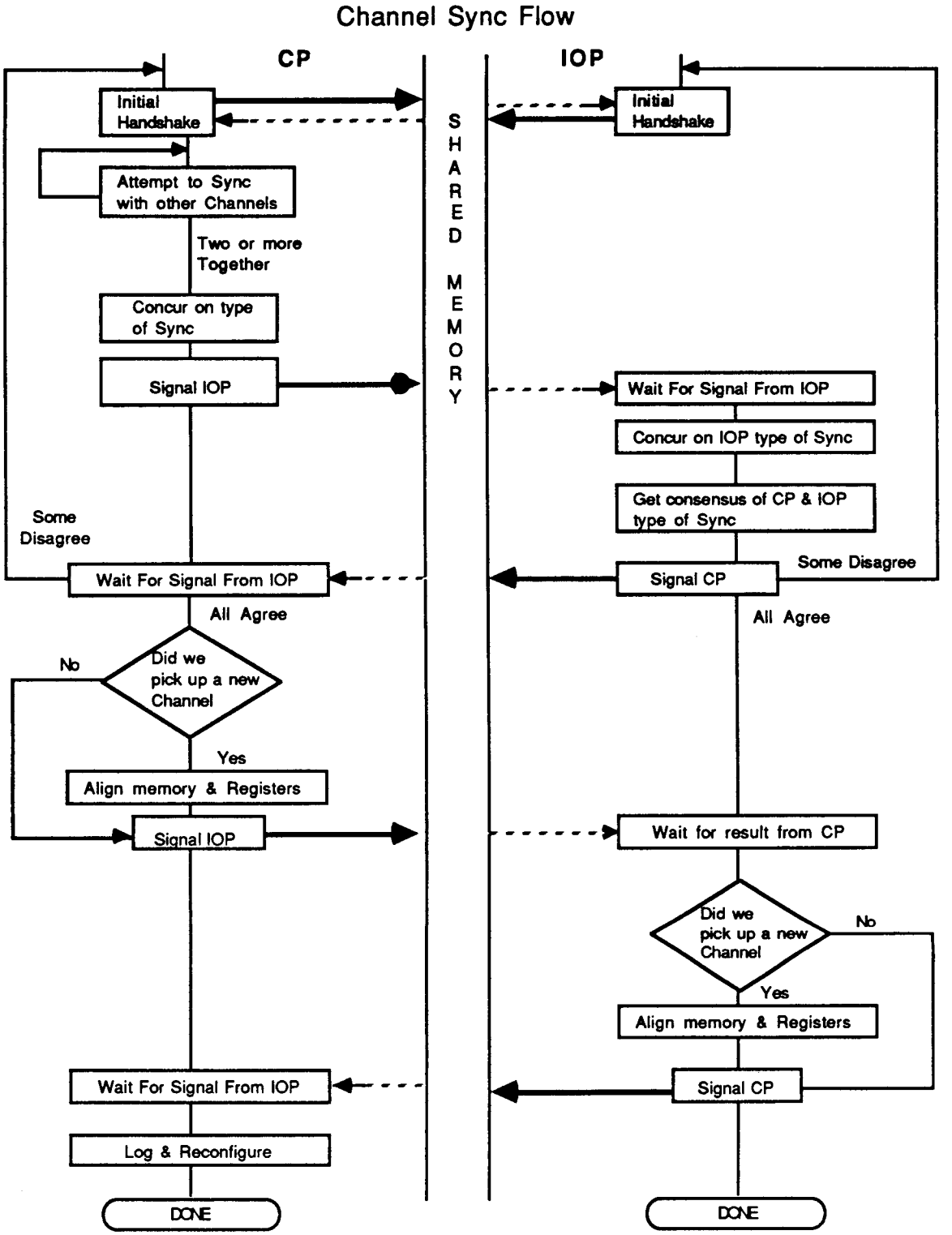


Figure 37. Channel Sync Flow Diagram

To be in sync, a processor must be synchronized to the instruction level with other processors of its type (i.e., CPs or IOPs). In addition, its companion must be synchronized with other processors of the companion's type, i.e., either both processors in a channel are good or they are both failed. The steps in the synchronization process and the interaction between CP and IOP are summarized below and shown in the following diagram.

- Initial Handshake
- CPs Sync
- CPs Concur on Type of Sync
- CPs Signal IOPs
- IOPs Concur on Type of Sync
- IOPs Determine Consensus of CP and IOP Types of Sync
- IOPs Signal CPs
- CPs Align
- CPs Signal IOPs
- IOPs Align
- IOPs Signal CPs
- CPs Reconfigure and Log

Initial Handshake

The purpose of the initial handshake is to coordinate the synchronization of the CPs with that of the IOPs, since a processor cannot be considered synchronized unless its companion is also synchronized. This handshake ensures that both sides are in the Channel Sync routine before the attempt to sync is made. It also means that only one side (the CPs were chosen arbitrarily) needs to sync using the data exchange hardware. As explained in more detail in the process descriptions for CP_HANDSHAKE and IOP_HANDSHAKE, the other side (the IOPs) will automatically sync while waiting for their companions.

CPs Sync

The CP locks the shared bus and makes a fixed number of attempts to synchronize with another channel(s) by doing data exchanges. As long as a CP is alone, it repeats this step.

CPs Concur on Type of Sync

This step and the next four steps concern the CP and IOP agreeing on what type of sync they are doing. This agreement is required in order for the system to recover from some catastrophic condition, such as a second failure or hardware or software common-mode fault, by automatically restarting. These catastrophic conditions could result in channels going to the Channel Sync routine in different states, which would make it impossible for

them to be aligned. Therefore we must check why a channel is in Channel Sync and ensure that at least two channels arrive there in the same state.

There are four possible types of sync:

- POWER_ON - an operator has restarted the system
- SOFT_RESTART - the software has restarted the system
- RECOVERY_CHANS_TOGETHER - two synchronized channels are trying to pick up a lone channel
- RECOVERY_LONE_CHAN - a lone channel is trying to be picked up

A consensus value for the type of sync is arrived at by the standard method of bit-for-bit voting. If there is no majority, there will be no consensus, with one exception: a vote of POWER_ON and SOFT_RESTART results in a consensus value of SOFT_RESTART. This is because POWER_ON and SOFT_RESTART produce identical states in the processor; their only difference is that SOFT_RESTART preserves the existing non-congruent logs. The action taken when there is no consensus depends on the particular situation; this is discussed in more detail below.

The first step in arriving at a consensus value for type of sync is for the CPs to agree among themselves. After two or more CPs have synchronized, they all exchange their type of sync and obtain a voted value. If there is no consensus or if the majority are lone processors trying to be picked up, they have different states and cannot be aligned. They must restart in order to achieve identical states.

CPs Signal IOPs

After the CPs examine their type of sync, they notify the IOPs by setting the TOIOP variable in shared memory. They can set it to two possible values:

- 6 - The CPs could not achieve a consensus and/or had to restart.
- 7 - The CPs did achieve a consensus. The consensus value is written in the TYPE_OF_SYNC variable in shared memory.

The CPs now wait for a return signal from the IOPs.

IOPs Concur on Type of Sync

The IOPs reach a consensus on their type of sync in the same way as the CPs. They all exchange their type of sync and obtain a voted value. If there is no consensus or if the majority are lone channels trying to be picked up, they must restart so that they can achieve identical states.

IOPs Determine Consensus of CP-IOP Type of Sync

If the IOPs did not have to restart, they now compare their voted type of sync to the CPs' type from shared memory. There are two possible results.

- A consensus is reached. This will happen either because the two types are the same or because one is a Power_On and one is a Soft_Restart.

- No consensus is reached. This will be the case when one side has started over (either a power_on start or a software restart), but the other side has not. In this situation, the side which did not restart must now do so. One side cannot continue from where it previously left off when the other side is starting afresh.

IOPs Signal CPs

After the IOPs examine their types of sync and the CPs' type of sync, they notify the CPs by setting the TOIOP variable in shared memory. They can set three possible values:

- 8 - The CPs must do a restart. The IOPs will wait for them at the Initial Handshake.
- 9 - The IOPs must do a restart. The CPs should wait for them at the Initial Handshake.
- 10 - A consensus value has been reached. It is written back into the TYPE_OF_SYNC variable in shared memory. Both sides now use the consensus value as their type of sync.

The IOPs now start Interval Timer 1, which is used to reset the watchdog timer during the resynchronization process and wait for a signal from the CPs as to whether they need to align.

CPs Align

Having agreed on a type of sync, the CPs now check to see if they have picked up any new channel(s). If they have, they start Interval Timer 1, they then align their volatile memory and registers. These include:

- interval timers
- RAM (oncard and offcard)
- testport memory
- oncard discretes
- shared discretes
- LMN data exchange control register

Finally the data exchange error latches and fault-tolerant clock error latches are cleared.

CPs Signal IOPs

The CPs notify the IOPs of the results of the above step by setting the TOIOP variable in shared memory. They can set it to two possible values:

- 2 - The CPs have picked up a channel and aligned. The IOPs must align also.
- 3 - The CPs did not pick up anyone new. The IOPs do not need to align.

IOPs Align

If the CPs picked up a new channel, the IOPs must now align their volatile memory and registers. These are the same as on the CP side, with two additions:

- shared memory
- system timer

Finally, the data exchange error latches and fault-tolerant clock error latches are cleared.

IOPs Notify CPs

Whether or not the IOPs needed to align, they clear the TOIOP variable to signal the CP that they are done.

CPs Log and Reconfigure

Finally the CPs reconfigure and log the reconfiguration. This step is done here because the system timer is initialized/aligned by the IOPs; a meaningful time is not available until this point.

4.2.2.2.3 Process Name: CP_HANDSHAKE IOP_HANDSHAKE

Inputs

- TOIOP variable in shared memory

Outputs:

- TOIOP variable in shared memory

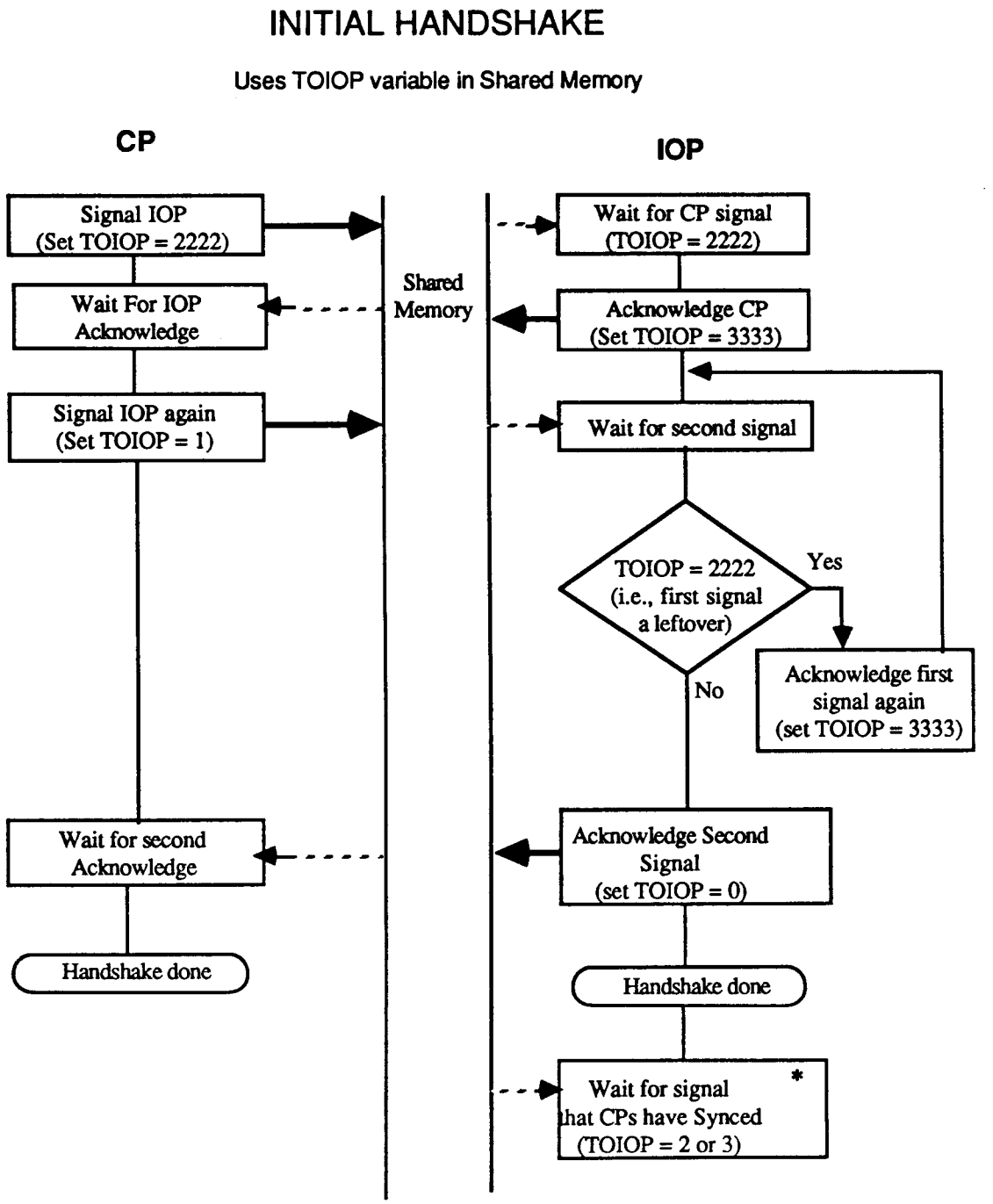
Description:

These two routines execute the handshake between the CP and IOP which is required before an attempt to synchronize multiple channels is made.

The basic sequence of the handshake is for the CP to write some value in the TOIOP variable in shared memory and for the IOP to clear it. After the IOP has cleared it, the CP attempts to sync using the data exchange hardware, first locking the shared bus. When they have successfully synchronized, the CPs write another value in the TOIOP variable and unlock the shared bus. Meanwhile, the IOP, having cleared TOIOP, is waiting for the variable to change to some other value. The IOPs will automatically synchronize here because, since the shared bus is locked, they will all be suspended at the instruction referencing shared memory.

One complication involves the initial setting of the TOIOP variable. If the IOP looks for a particular value, but that value is left over from a previous execution, the IOP will incorrectly conclude that its companion is waiting at the handshake. This problem was solved by having the handshake be a two-step process. In between the first and second steps, the IOP can determine if the value it read was a leftover value or if its CP is really waiting for it.

The actual steps of the initial handshake are shown in the following diagram.



* IOPs will synchronize here waiting for shared bus to be unlocked

Figure 38. CP and IOP Handshake Flow

4.2.2.2.4 Process Name: SYNC_CHANS

Inputs:

- None

Outputs:

- Integer representing the channels in sync

Description:

This is an assembly language routine which synchronizes channels at the instruction level by sending a unique pattern from each channel through the data exchange. If the result read after a particular exchange is the expected pattern, the channel originating the exchange is judged to be present and in sync.

The patterns exchanged must be long words and must be transmitted using the shared exchange registers. When a word is exchanged using the dedicated registers, the data exchange hardware is sufficiently slow such that any particular word will be in the receiver for more than one FTC cycle. This would allow a channel only slightly behind to see the expected word in the receiver and incorrectly conclude that it is in sync. Successive exchanges of long words using the shared registers, however, result in the long word being in the receiver for only one FTC cycle, so a tardy channel could not conclude that it was in sync.

The routine does three data exchanges: FROM_A, FROM_B, and FROM_C. After each exchange it analyzes the result to see if the source channel is present. When all three channels are present, it returns immediately to the calling routine; otherwise it iterates the exchanges a number of times, attempting to pick up all channels. This basic data exchange loop is shown in the following N-S diagram.

The execution time of this data exchange loop is of critical importance. If the three channels are all executing this code but have started at different times, they will never synchronize if the basic exchange loop takes the same number of FTC cycles on each channel. The loop must take a different (prime) number of FTC cycles on each channel. This is done by having each unsynchronized channel delay for a different amount of time at the end of each iteration, while synchronized channels have no delay. In the laboratory demonstration implementation, which uses a Motorola 68010 processor with an 8 MHz clock, synchronized channels take 15.5 FTC cycles for one iteration, channel A alone takes 17.5 cycles, channel B alone takes 18.5 cycles, and channel C alone takes 20.5 cycles.

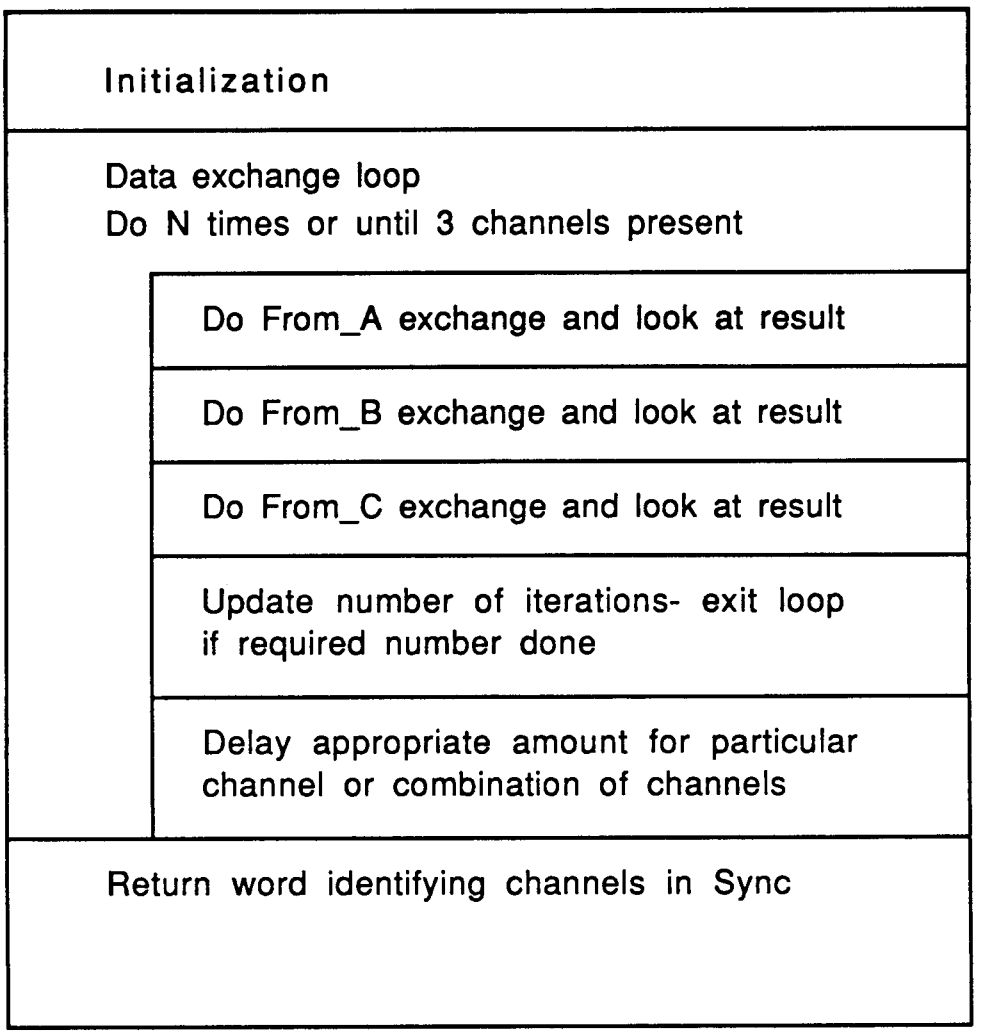


Figure 39A. Sync Chans N-S Diagram

4.2.2.2.5 Process Name: START_TIMER1

Inputs:

- Current tine
- Time watchdog timer was last reset

Outputs:

- Countdown value in Timer 1
- Handler address for Timer 1 interrupt

Description:

Since the Channel_Sync routine runs uninterrupted by other system tasks, when a recovered channel is being aligned the amount of time used will exceed the limit of the watchdog timer. During Channel_Sync, therefore, one of the unused interval timers (specifically, Timer 1) is used to periodically cause an interrupt, at which time the watchdog is reset.

This subroutine sets Interval Timer 1 the first time by calling a system subroutine with an initial value and the address of the interrupt handler. The initial value must be calculated based on the time the watchdog was last reset, since the watchdog cannot be cleared either too soon or too late.

This subroutine is called by the CP only after it has determined that it has picked up the lone channel, but it is called by the IOP immediately after the IOP has determined a consensus Type_of_Sync. This is necessary because the CP does not signal whether it has picked up anyone until after it has aligned, by which time the IOP's watchdog would have overrun.

4.2.2.2.6 Process Name: ALIGN_SH_MEM

Inputs:

- None

Outputs:

- None

Description:

This procedure aligns the channel's shared memory. There are eight sections of shared memory, as follows:

F87000 - F87FFE
F8F000 - F8FFFE

F97000 - F97FFE
F9F000 - F9FFFE

FA7000 - FA7FFE
FAF000 - FAFFFE

FB7000 - FB7FFE
FBF000 - FBFFFE

The alignment is done by calling the assembly language routine, ALIGN_MEM (refer to following section).

4.2.2.2.7 Process Name: ALIGN_MEM

Inputs:

- Starting and ending addresses of memory to be aligned

Outputs:

- A voted value written to each location

Description:

This assembly language routine does the actual alignment of the specified memory area by doing a FROM_ALL exchange of each long word and then writing the result back into memory. For greater efficiency, preparations for the next exchange are interleaved with the writing and reading of the current exchange. The alignment is accomplished by doing long word exchanges via the shared exchange registers (therefore the CP and IOP must align in sequence), rather than by doing word exchanges on the dedicated registers (in which case the CP and IOP could align in parallel). The first method was judged to be faster in total time and simpler.

4.2.2.2.8 Process Name: ALIGN_SHARED_DISCRETES

Inputs:

- Local value of shared discretes register
- Voted value of shared discretes register

Outputs:

- Interprocessor interrupt bits in shared discretes

Description:

This procedure aligns the shared discretes, specifically, the CP-to-IOP interrupt and IOP-to-CP interrupt. If the majority of channels have the particular interrupt set, it is set on all channels; otherwise it is cleared on all channels. The CP sets the interrupt-IOP bit but clears its own interrupt; similarly the IOP sets the interrupt-CP bit but clears its own interrupt.

4.2.2.2.9 Process Name: ALIGN_CP_TESTPORT

Inputs:

- None

Outputs:

- None

Description:

This procedure is executed by the CP to align its testport memory. All of testport memory can be accessed by both the CP and IOP, but it has been divided into separate sections for convenience. There are two sections of testport memory for the CP, as follows:

C27000 - C27FFE
C2F000 - C2FFFE

The alignment is done by calling the assembly language routine, ALIGN_MEM (refer to earlier section).

4.2.2.2.10 Process Name: ALIGN_IOP_TESTPORT

Inputs:

- None

Outputs:

- None

Description:

This procedure is executed by the IOP to align its testport memory. All of testport memory can be accessed by both the CP and IOP, but it has been divided into separate sections for convenience. There are two sections of testport memory for the IOP, as follows:

C37000 - C37FFE
C3F000 - C3FFFE

The alignment is done by calling the assembly language routine, ALIGN_MEM (refer to earlier section).

4.2.2.2.11 Process Name: ALIGN_TIMER0

Inputs:

- None

Outputs:

- Interval Timer Control Register
- Interval Timer value

Description:

This routine aligns Interval Timer 0 by setting the timer to an arbitrarily small value. A system subroutine is called to set this value since the shared bus must be accessed in between writing the most significant byte and least significant byte of the timer. This access ensures that the timers are set away from a rising or falling edge of the fault tolerant clock and therefore the resulting interrupts will be asserted on the same pulse of the fault tolerant clock..

4.2.2.2.12 Process Name: ALIGN_RAM

Inputs:

- None

Outputs:

- None

Description:

This routine aligns oncard and offcard ram (excluding the non-congruent data area). One subroutine is called to align both. Since the non-congruent area is forced to be at an absolute address and is therefore located at the high end of offcard RAM, there is no need to align anything following this area.

The align is done by calling the assembly language subroutine ALIGN_ALL_RAM (refer to following section).

4.2.2.2.13 Process Name: ALIGN_ALL_RAM

Inputs:

- Starting and ending addresses of memory areas to be aligned

Outputs:

- A voted value written to each location

Description:

This assembly language routine aligns the specified areas of oncard and offcard RAM by doing a FROM_ALL exchange of each long word and then writing the result back into memory. For greater efficiency, preparations for the next word are interleaved with the reading and writing of the current word. The alignment is accomplished by doing long word exchanges via the shared exchange registers (refer to description of ALIGN_MEM routine). Oncard and offcard RAM must be aligned in a single routine so that it does not matter whether variables and data reside oncard or offcard or both.

4.2.2.2.14 Process Name: ALIGN_TIME

Inputs:

- None

Outputs

- None

Description:

This routine calls the Time Manager's Align_Time routine (refer to Section 6.2.1.1.5.1) and then aligns the SYSTEM_TIMER_OFFSET value in shared memory.

4.2.2.2.15 Process Name: ALIGN_INTEGER

Inputs:

- address of long word to be aligned

Outputs:

- a voted value written to the specified address

Description:

This assembly language routine aligns a single long word by doing a FROM_ALL exchange using the shared data exchange and writing the result back into the specified address.

4.2.2.2.16 Process Name: INITIAL_SYNC_SIMPLEX

Inputs

- TOIOP flag in shared memory

Outputs:

- TOIOP flag in shared memory
- Entry in reconfiguration log

Description:

This procedure is executed on simplex channels at system startup. Since there is only one channel, no synchronization or alignment takes place. Agreeing on type of sync is easier than in a triplex, since the only two possible types are POWER_ON and SOFT_RESTART. The routine does a handshake between the CP and IOP and initializes certain variables (the configuration, system time) and registers (LMN_DX control registers, system timer).

4.2.2.3 System Restart Processes

4.2.2.3.1 Process Name: XRESTART

Inputs:

- None

Outputs:

- Entry in non-congruent log
- Type_of_sync variable

Description:

This routine is called when a critical (i.e., unrecoverable) fault is detected and the only way to recover is to completely restart the system. Examples of critical faults are a second fault detected by Fast FDIR, or a common-mode hardware or software fault. A channel may be alone or with other channels when it decides to restart. This routine logs the occurrence of the restart and sets the Type_of_Sync flag to SOFT_RESTART_REQUEST. It then jumps to the beginning of the load module.

4.2.3 Reconfiguration Process Descriptions

4.2.3.1 Process Name: CONFIG Package

Description:

This package is concerned primarily with maintaining the software records of each channel's status: present or not present. Some of the procedures and functions in this package are used by other processes throughout the system, while others are used only by FDIR processes. These functions and procedures are described on the following pages.

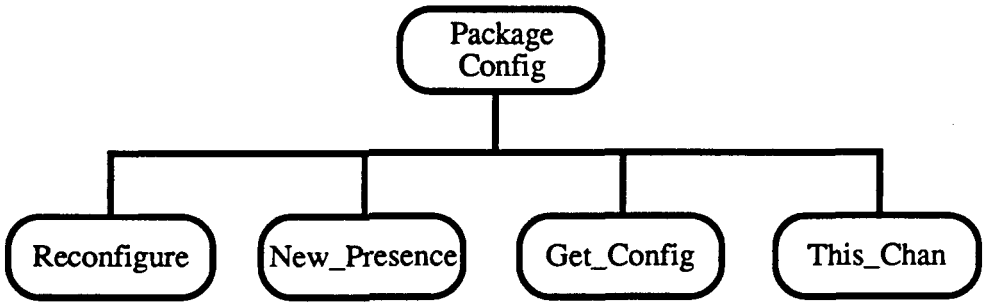


Figure 40. Reconfiguration Processes

4.2.3.1.1 Process Name: RECONFIGURE

Inputs:

- Reconfig_cmd record identifying the channel and whether it is failed or recovered

Outputs:

- Configuration record
- Integer version of configuration record
- STATUS_INFO array for display task
- Monitor interlock

Description:

This procedure is called in four different situations:

- system startup
- a channel has failed
- a channel has been recovered after a failure (trial recovery)
- a channel has passed its probation period after recovery (permanent recovery)

In each situation, appropriate actions are taken to update variables reflecting the current configuration and enable/disable the Monitor Interlock.

At system startup:

- THE_CONFIG, TRIAL_CONFIG and CHANS_IN_SYNC are set to reflect the channels which successfully synchronized.
- INT_CONFIG, INT_TRIAL_CONFIG and INT_CHANS_IN_SYNC are set to correspond to THE_CONFIG, TRIAL_CONFIG and CHANS_IN_SYNC, respectively.
- The STATUS_INFO array is set to indicate whether each channel is failed or active and the time it failed or became active.
- The Monitor Interlock is engaged for the active channels and disengaged for all others.

When a channel has failed:

- THE_CONFIG and TRIAL_CONFIG are updated to reflect the failed channel.
- If the failure was other than a data exchange failure, CHANS_IN_SYNC is also updated to reflect the failed channel.
- The STATUS_INFO array is updated to reflect the failed channel.

- The failed channel's Monitor Interlock is disengaged.
- INT_CONFIG, INT_TRIAL_CONFIG and INT_CHANS_IN_SYNC are set to correspond to THE_CONFIG, TRIAL_CONFIG and CHANS_IN_SYNC, respectively.

When a channel has been recovered after a failure (trial recovery);

- CHANS_IN_SYNC and TRIAL_CONFIG are updated to reflect the recovered channel.
- INT_CHANS_IN_SYNC and INT_TRIAL_CONFIG are set to correspond to CHANS_IN_SYNC and TRIAL_CONFIG, respectively.

When a channel has passed its probation period after recovery (permanent recovery);

- THE_CONFIG is updated to reflect the recovered channel.
- The STATUS_INFO array is updated to reflect the recovered channel.
- INT_CONFIG is set to correspond to THE_CONFIG.
- The recovered channel's Monitor Interlock is engaged.

4.2.3.1.2 Process Name: NEW_PRESENCE

Inputs:

- Integer value of old configuration
- Integer value of new configuration

Outputs:

- Reconfig_cmd record identifying the channel that is different and whether it was added or deleted

Description:

This function examines two integer versions of the configuration and figures out which channel is different between the two versions. If a channel has been added it creates a reconfig_cmd record for the channel with a reason of ENABLE_T; if a channel has been deleted it creates a reconfig_cmd record with a reason of DISABLE.

4.2.3.1.3 Process Name: GET_CONFIG

Inputs:

- Integer version of current configuration

Outputs:

- Record version of current configuration

Description:

This function is called by local system service and I/O system service routines that want to know the current configuration. The function takes the integer version of the current configuration and converts it to a record of type configuration.

It is necessary to obtain the current configuration by calling this routine rather than by simply making a local copy of the current configuration record. This is because Ada generates multiple instructions to move the configuration record, rather than using a single MOVE.L instruction. Thus if a routine were to be interrupted while moving the record, when it got control again the record may have changed, due to an intervening execution of Fast FDIR, Transient FDIR, or Lost Soul Sync. To make sure a consistent copy of the configuration record is obtained, it is necessary to use this function.

4.2.3.1.4 Process Name: THIS_CHAN

Inputs:

- Channel id from the Monitor Interlock Register

Outputs:

- Channel id of type channel_id

Description:

This function reads the channel id from the Monitor Interlock register and converts it to a variable of type channel_id with a value of A, B, or C.

4.2.4 Logging Process Descriptions

4.2.4.1 User Error Logs

Logging routines are included in the following packages:

RECORD_FDIR_ERRORS
FDIR_ERROR_LOG
RECORD_RECONFIGS
RECONFIG_LOG
NON_CONGRUENT_LOG

The non-congruent log itself is contained in the NON_CONGRUENT_DATA package.

4.2.4.1.1 Process Name: RECORD_FDIR_ERRORS.REPORT_ERROR

Inputs:

- Information to be entered in the error log

Outputs:

- None

Description:

This routine logs faults detected by the Fast_FDIR task. It calls a subroutine, FDIR_ERROR_LOG.ENTER, to make the log entry.

4.2.4.1.2 Process Name: FDIR_ERROR_LOG.ENTER

Inputs:

- Information to be entered in the error log

Outputs:

- Entry in the error log
- Counters to indicate number of entries and next available slot in the error log

Description:

This routine creates a new entry in the FDIR error log. This log has room for 20 entires; after all available slots are filled the next entry goes at the top of the log. Indices to the oldest and newest entries in the log are maintained, along with a count of the number of entries.

4.2.4.1.3 Process Name: RECORD_RECONFIGS.REPORT_RECONFIG

Inputs:

- Information to be entered in the reconfiguration log

Outputs:

- None

Description:

This routine logs all reconfigurations that occur. It calls a subroutine, RECONFIG_LOG.ENTER, to make the log entry.

4.2.4.1.4 Process Name: RECONFIG_LOG.ENTER

Inputs:

- Information to be entered in the reconfiguration log

Outputs:

- Entry in the reconfiguration log
- Counters to indicate number of entries and next available slot in the log

Description:

This routine creates a new entry in the reconfiguration log. This log has room for 20 entires; after all available slots are filled the next entry goes at the top of the log. Indices to the oldest and newest entries in the log are maintained, along with a count of the number of entries.

4.2.4.1.5 Process Name: NON_CONGRUENT_LOG.LOG_NONCONGRU_EVENT

Inputs:

- Strings identifying logger and information to be logged

Outputs:

- Entry in non-congruent log

Description:

This routine creates a new entry in the non-congruent log. This log has room for 16 entries; after all available slots are filled the next entry goes at the top of the log. An index to the next available slot is maintained, along with a count of the number of entries.

In order to determine the next available slot, two special subroutines must be called, since different channels will be at different positions in their respective logs and computing a new position should not throw them out of sync. Refer to INCR_NC_LOG_INDEX and SET_UP_CURRENT_ENTRY which follow.

4.2.4.1.6 Process Name: NON_CONGRUENT_LOG.INCR_NC_LOG_INDEX

Inputs:

- Current NC_LOG_INDEX

Outputs:

- Updated NC_LOG_INDEX

Description:

The non-congruent log has room for 16 entries; the index varies between 0 and 15. This routine increments it and resets it to 0 when it gets past 15. This must be done without actually checking the value because different channels may have different indices, and such a check might throw them out of sync. This assembly language routine uses an AND instruction to set a 16 to 0.

4.2.4.1.7 Process Name: NON_CONGRUENT_LOG.SET_UP_CURRENT_ENTRY

Inputs:

- Index to the non-congruent log

Outputs:

- Address of next available position in the non-congruent log

Description:

This routine takes an index value and converts it to an actual address within the non-congruent log. This special routine is required because the Ada implementation of this operation treats an index of 0 as a special case. Since different channels may have different indices to the non-congruent log, this special case could throw a channel out of sync. This routine also speeds up the computation by using a shift instruction rather than a multiply.

4.2.4.1.8 Process Name: NON_CONGRUENT_DATA Package

Description:

This package contains data items whose values may legitimately differ across channels and which are therefore kept in this special area which is neither aligned nor scrubbed. Limitations of the Ada compiler force this package to reside at an absolute address; the high end of RAM was chosen for convenience. There are three types of data in this package:

1) channel id
2) input values which may differ across channels
3) the non-congruent log

The channel id is obtained at system startup by reading the value from the hardware register and converting it to the channel_id type defined in the Config package.

Input values which may differ across channels include local copies of the error latches, local results of selftests, and data read from input devices.

The non-congruent log contains information about selected faults which occurred. This log is not reinitialized after a software restart so that information about the cause of the restart is available.

4.2.4.2 Debugging Logs

The FDIR software also has the facility to record ongoing events (both normal and error conditions) in a log which cannot be displayed but must be inspected manually by the system engineer. The subroutine to update this log is described in the following section.

4.2.4.2.1 Process Name: DEBUG_LOG

Inputs:

- Two identifying words passed by the caller

Outputs:

- Either a new entry in the log, or an addition to the count field of the previous entry

Description:

This routine maintains a trace log so that the sequence of events in the FTP at any particular time may be known. This log is circular; after all available slots are filled the next entry goes at the top of the log.

In addition to recording the caller id and an additional word of information, which are passed as parameters, the routine maintains a count of the number of times it has been called in succession with identical parameters. Thus, for example, the selftest procedure to write RAM patterns calls this routine once for every word that it tests. There are not n entries in the log, however; rather there is only one entry with a field to indicate that the entry was made n times in succession.

The current value of the system timer is also saved for each entry.

This procedure is written in assembly language to minimize its impact on system performance.

5.0 GPC STATUS REPORTER

The purpose of the GPC Status Reporter is to collect and disseminate information about the status of the various AIPS components. Hardware status information is provided by the various redundancy managers, while software status information is provided by the operating system. The collected information is made available to an operator, for manual decision-making, and to the System Manager software function, for automated decision-making. The three major functions of the Status Reporter are shown in Figure 41.

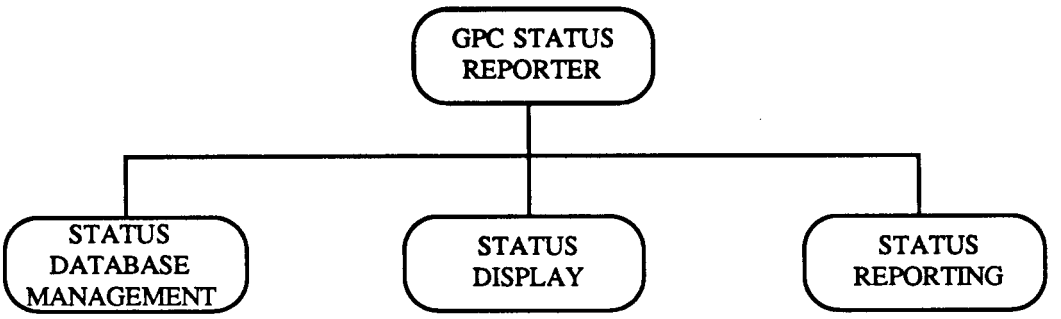


Figure 41. GPC Status Reporter Functions

5.1 GPC Status Reporter Functional Requirements and Design

AIPS is a distributed system with redundancy management functions at each site. Some of these functions are located at every site (e.g., GPC FDIR, IOS FDIR and ICIS FDIR) while other functions are located only at selected sites (e.g., the I/O Network Manager and the IC Network Managers). The GPC Status Reporter, which resides on both the CP and IOP at every site, must collect and disseminate status information provided by every function resident at its own particular site.

Figure 42 shows the flow of the status information. The shaded circles in this figure indicate functions executed only at selected sites, while the unshaded circles indicate functions executing at every site.

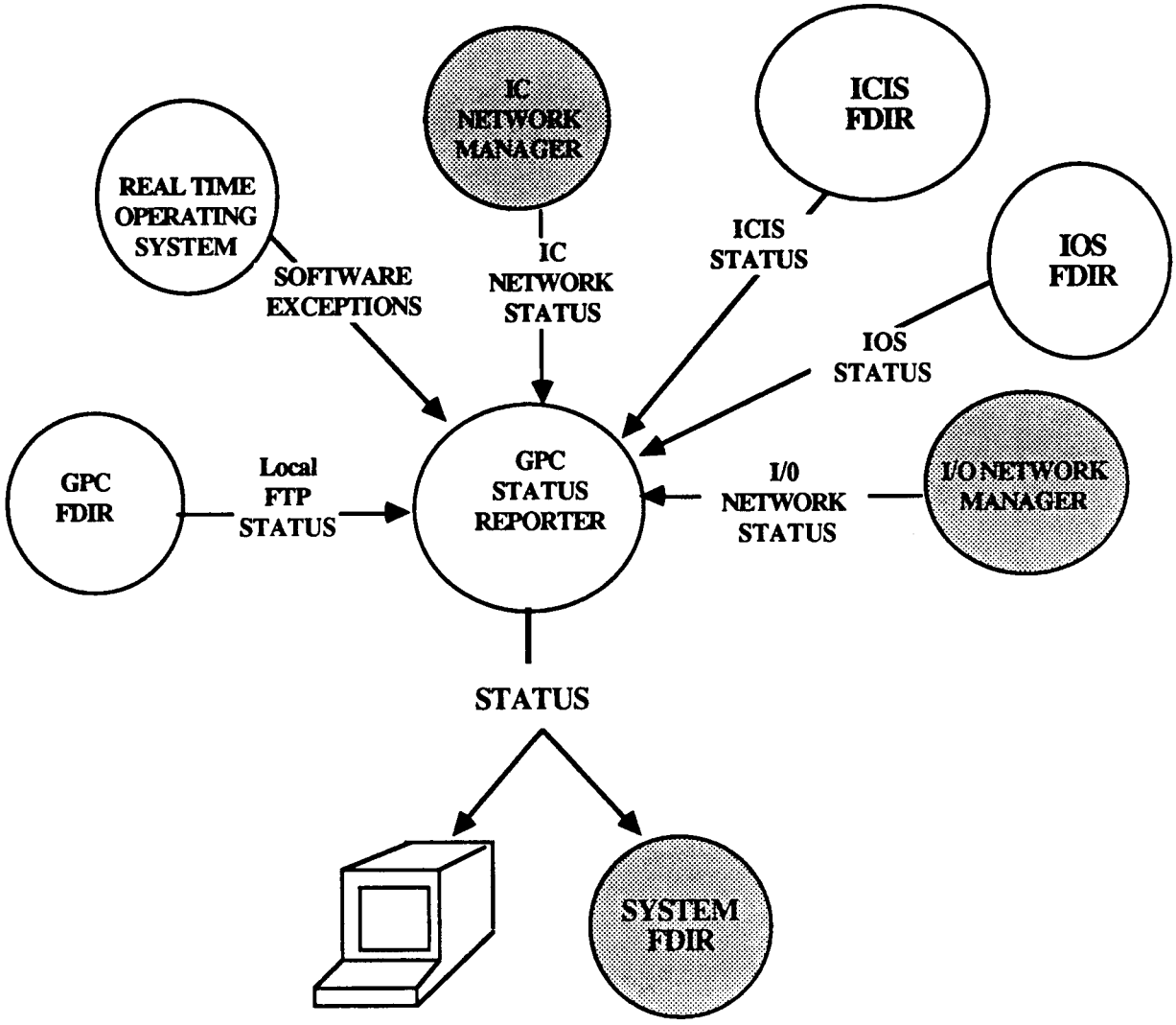


Figure 42. GPC Status Reporter Data Flow

5.1.1 Status Database Management

The status database consists of hardware and software status information kept in logs and variables which are maintained by the various redundancy managers and the operating system. This information is segmented so that the redundancy managers do not have access to the entire database, but only to their own particular segment. The logs and variables used by each function are summarized in the following table. Access to the logs is controlled using the lock-out feature of the Ada runtime system, which requires tasks to "lock" the desired log before using it. A task which finds the log already locked is automatically suspended and then resumed when the log is unlocked.

REDUNDANCY MANAGER	STATUS DATABASE SEGMENT
GPC FDIR	FTP Configuration FDIR Error Log FTP Reconfiguration Log Non-Congruent Log
Real Time Operating System	Software Exception Log
IC Network Manager	IC Network Configuration IC Network Error Log IC Network Reconfiguration Log
ICIS FDIR Manager	ICIS Configuration ICIS Error Log ICIS Reconfiguration Log
IOS FDIR Manager	IOS Configuration IOS Error Log IOS Event Log
I/O Network Manager	I/O Network Configuration I/O Network Error Log I/O Network Event Log

Figure 43. Status Database

5.1.2 Status Display

The Status Display software displays the status information currently available about a particular hardware/software component in response to an operator query. Each processor card in a GPC has a Dual Universal Asynchronous Receiver/Transmitter (DUART) chip with two channels, which allows the processor to communicate with a display device such as a CRT or personal computer over two RS-232 lines. Two types of displays are currently available, one on a CRT and one on a Macintosh computer, and each uses its own RS-232 line. The functions which provide these displays are shown in Figure 44.

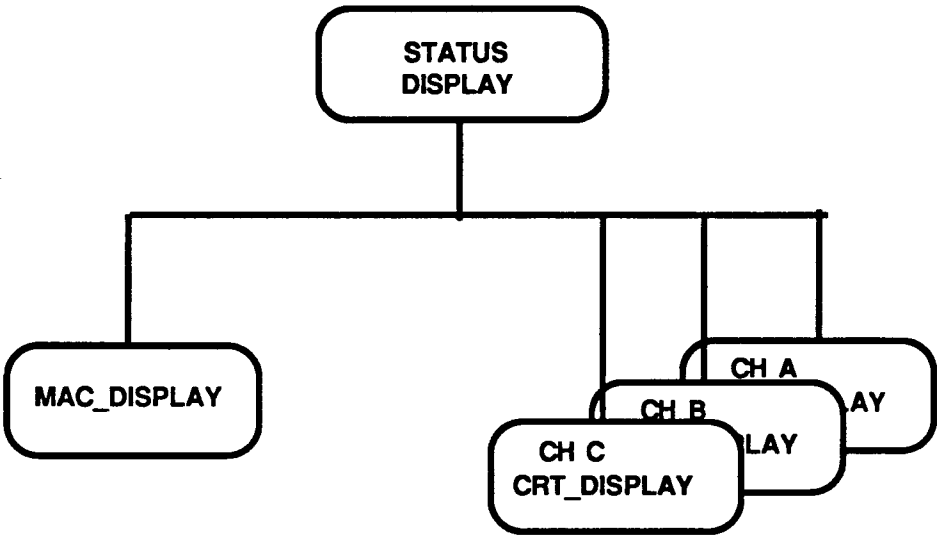


Figure 44. Status Display

The CRT_Display function allows an operator to communicate with a GPC via a VT220 or VT240 terminal. There are three instances of the CRT_Display task, one to serve each channel, so that a terminal may be connected to each channel and three different displays called up and monitored by an operator if desired. The three tasks run at the lowest priority. All three tasks execute on all three channels but the input to any one task is only from the channel being served by the particular task. Specifically, the CRT_Display task for Channel A executes on all channels but its input comes from Channel A's DUART and must be distributed to the other channels by means of the data exchange before any processing is done; output is then sent only to Channel A's DUART. Similarly, the CRT_Display task for Channel B executes on all channels, but its input, which comes from Channel B's DUART, is distributed before being processed, and output is sent only to Channel B's DUART.

The Mac_Display function allows an operator to communicate with a GPC via a Macintosh computer. The Mac_Display task sends the status information to a Macintosh computer where it is formatted into a pictorial or tabular display. This moves the complex and time-consuming routines necessary to create user-friendly displays from the GPC to the Macintosh.

There is only one instance of the Mac_Display task, so communication can take place with only one FTP channel at any given time. The channel that will receive and transmit data is chosen according to channel identification and availability. Channel A has the highest priority and channel C the lowest. If Channel A is active, its DUART is responsible for the Macintosh link. If the channel currently driving the Macintosh display fails, the MAC_Display software automatically switches to the next available DUART. The operator requests the desired displays from the FTP by using Macintosh menus; several

displays may be called up and monitored simultaneously by using Macintosh windows. The Macintosh may communicate with the GPC over any distance, and remote users may connect their Macintosh to a GPC using a modem and telephone line.

5.1.3 Status Reporting

The Status Reporting function sends all currently available status information to the System Manager, either periodically or in response to a fault. Since the System Manager has status information from all sites, it can form a global view of things and identify faults in a way in which the individual redundancy managers, with only their local viewpoint, cannot. The System Manager is thus able to identify transient faults, isolate the cause of ambiguous faults, and determine the need for function migration.

5.2 GPC Status Reporter Software Specifications

5.2.1 Status Database Management Process Descriptions

The status database management processes are specific to each of the database segments and are described in the section pertaining to the appropriate redundancy manager. These processes are summarized in the following table.

REDUNDANCY MANAGER	STATUS DATABASE SEGMENT	UPDATE PROCESS	WHERE DESCRIBED	READ PROCESS	WHERE DESCRIBED
GPC FDIR	FTP Configuration	Reconfigure	4.2.3.1.1	Get_Config	4.2.3.1.3
	FDIR Errors	Report_Error	4.2.4.1.1	Get_Log	
	FTP Reconfigurations	Report_Reconfig	4.2.4.1.3	Get_Log	
	NonCongruent Log	Log_Noncongru_Event	4.2.4.1.5		
Real Time Operating System	Software Exception Log	Log_Exception		Not implemented	
IC Network Manager	IC Network Configuration	Not implemented			
	IC Network Errors				
	IC Network Reconfigurations				
ICIS FDIR	ICIS Configuration	Not implemented		Not implemented	
	ICIS Errors				
	ICIS Reconfigurations				
IOS FDIR	IOS Configuration	Network Hardware Status	3.4.1 [4]	Network Hardware Status	3.4.1 [4]
	IOS Errors	IOS Error Logs	3.5.1 [4]	IOS Error Logs	3.5.1 [4]
	IOS Reconfigurations	IOS Event Logs	3.5.2 [4]	IOS Event Logs	3.5.2 [4]
I/O Network Manager	I/O Network Configuration	Network Hardware Status	3.4.1 [4]	Network Hardware Status	3.4.1 [4]
	I/O Network Errors	I/O Network Error Logs	3.5.1 [4]	I/O Network Error Logs	3.5.1 [4]
	I/O Network Reconfigurations	I/O Network Event Logs	3.5.2 [4]	I/O Network Event Logs	3.5.2 [4]

Figure 45. Database Management Process Descriptions

5.2.2 Site Status Display Process Descriptions

5.2.2.1 Terminal Displays

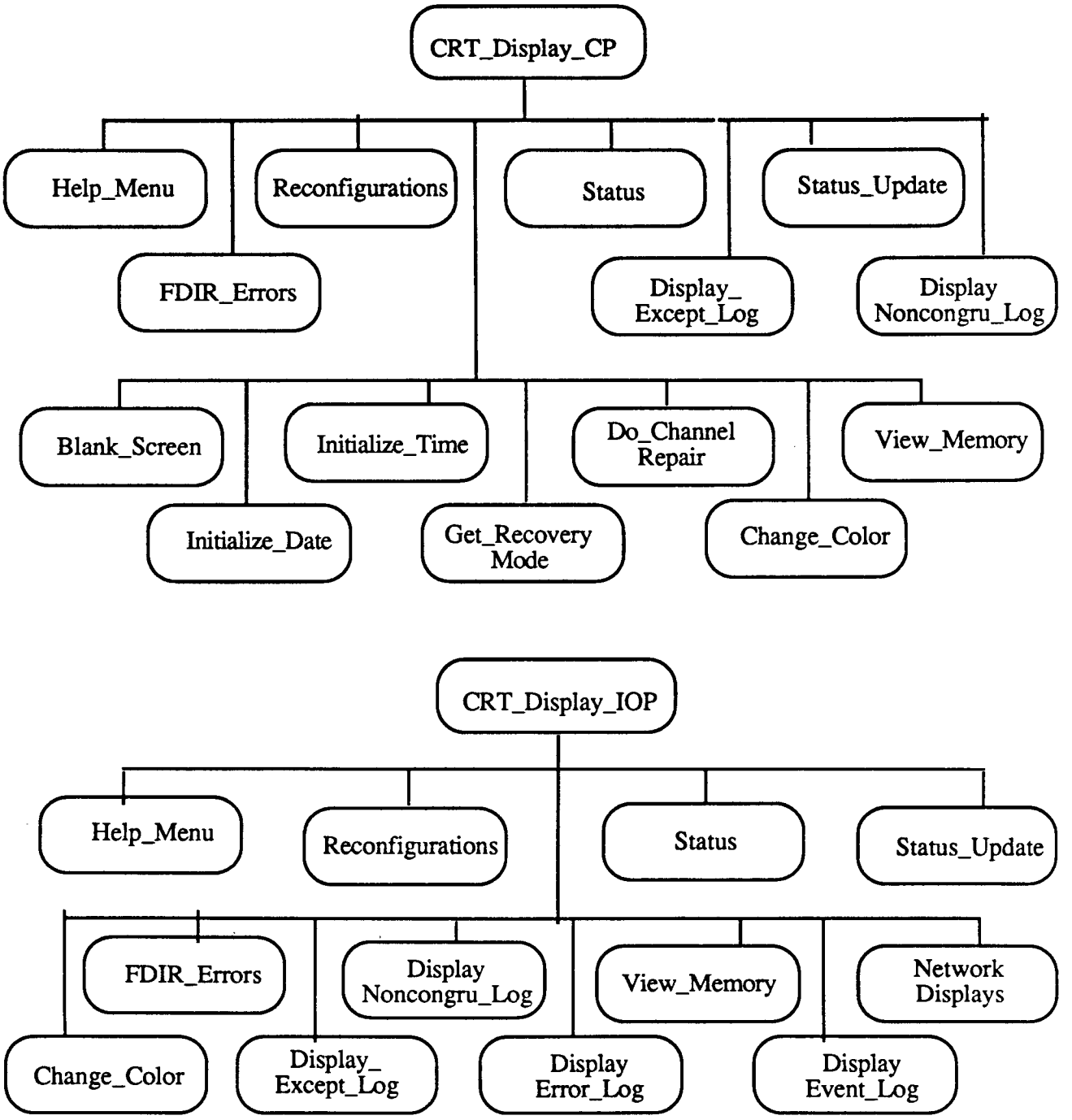


Figure 46. Terminal Display Processes

5.2.2.1.1 Process Name: CRT_Display_CP

Inputs:

- Channel id
- User commands from a terminal

Outputs:

- Invalid command message to terminal

Description:

This task provides a means by which an operator may communicate with the FTP, specifically the CP, via a VT220 or VT240 terminal connected to an RS232 port. There are three instances of this task, one for each channel, so that simultaneous input from three different channels may be processed.

This task runs at the lowest priority and executes periodically. However, the periodicity is not achieved by using the Scheduler functions, but rather by a delay statement at the end of the task's main loop. The task is initially started by a rendezvous with the main program which identifies to each instance of the task which channel it is serving.

Within its main loop, the task calls the procedure INPUT_WAITING to determine if new input has come in from the RS232 port. If it has, the input command is read and an appropriate routine is called to process it. If there is no new input and the last command entered was an ST command (Status Display), the time and date on the status display are updated.

5.2.2.1.2 Process Name: Help_Menu

Inputs:

- None

Outputs:

- CRT display

Description:

This routine responds to the HE (Help) command and displays a list of valid input commands. Valid CP commands are:

CR -- Repair Channel
CC -- Change Color
CL -- Clear Screen
DA -- Set Date
FD -- Display FDIR Error Log
HE -- Help
NC -- Display Noncongruent Log
OP -- Open Memory
RE -- Display Reconfiguration Log
RM -- Set Recovery Mode
SE -- Display Software Exceptions
ST -- Display FTP Status
TI -- Set Time

Valid IOP commands are:

CC -- Change Color
CL -- Clear Screen
FD -- Display FDIR Error Log
HE -- Help
NC -- Display Noncongruent Log
OP -- Open Memory
RE -- Display Reconfiguration Log
R1/R2/R3 -- Display I/O Network Errors
SE -- Display Software Exceptions
ST -- Display FTP Status
V1/V2/V3 -- Display I/O Network Events

5.2.2.1.3 Process Name: FDIR_Errors

Inputs:

- FDIR Error Log

Outputs:

- CRT display

Description:

This routine responds to the FD command (Display FDIR Errors). It formats the FDIR error log and outputs it to a CRT. For each entry in the log, the type of error, the faulty channel and the date and time of the error are displayed. Additional information is displayed for the memory selftest errors (RAM scrub, RAM pattern, shared memory scrub, and PROM sum check): the address containing the fault, the actual value at that address, and the expected address.

5.2.2.1.4 Process Name: Reconfigurations

Inputs:

- Reconfiguration log

Outputs:

- CRT display

Description:

This routine responds to the RE (Display Reconfigurations) command. It formats the reconfiguration log and outputs it to a CRT. For each entry in the log, the reconfiguration event (e.g., power on, disable, enable), the processor, the time and date of the event, and the channel(s) involved are displayed.

5.2.2.1.5 Process Name: Display_Except_Log

Inputs:

- Software exception log

Outputs:

- CRT display

Description:

This routine responds to the SE (Display Software Exceptions) command. It formats the software exception log and outputs it to a CRT. For each entry in the log, the date and time of the error, where it occurred (subroutine or task id), and a brief description of the exception are displayed.

This subroutine is declared as EXTERNAL so that it may be used by the runtime system to display the software exception log in the event of some catastrophic error which forces a system shutdown.

5.2.2.1.6 Process Name: Status

Inputs:

- Current configuration

Outputs:

- CRT display

Description:

This routine responds to the ST (Display FTP Status) command. It formats the initial display which shows the current configuration of the FTP (which channels are up, which are down, and failure times). After this initial display has been drawn, the STATUS_UPDATE routine (see following section) is periodically called to update the date, time and any change in channel status.

5.2.2.1.7 Process Name: Status_Update

Inputs:

- Current configuration
- STATUS_INFO array
- STATUS_CHANGED array

Outputs:

- Time, date and channel status fields in status display

Description:

This routine updates only the time, date and channel status fields in the status display. To make the routine as efficient as possible, the channel status is updated on the display only when it has changed. The STATUS_INFO and STATUS_CHANGED arrays are used to keep track of when the status changes (from online to failed and failed to online) and whether that changed status has been reflected in the display.

5.2.2.1.8 Process Name: Display_Noncongru_Log

Inputs:

- Non-congruent log
- Channel id whose log is to be displayed

Outputs:

- CRT display

Description:

This routine processes the NC command, which is used to display the non-congruent log for a particular channel. After the NC command has been entered, this routine prompts the operator for the specific channel id. The log is then copied from the resident channel to all channels using a single-source data exchange. For each entry in the log, the date and time of the error, where it occurred (subroutine or task id), and a brief description of the non-congruent event are displayed.

5.2.2.1.9 Process Name: Blank_Screen

Inputs:

- None

Outputs:

- Blank CRT screen

Description:

This command responds to the BL (Blank Screen) command. It causes the CRT screen to be cleared.

5.2.2.1.10 Process Name: Initialize_Date

Inputs:

- New value for date

Outputs:

- Local_GPC_Time variable in shared memory

Description:

This routine processes the DA command, which is used to enter the current date. After the DA command has been entered, this routine prompts the operator for the date, which must be entered in the format DD-MON-YYYY. It then calls the SET_DATE subroutine provided by the Time Manager package to update the global time record, which resides in shared memory.

5.2.2.1.11 Process Name: Initialize_Time

Inputs:

- New value for time

Outputs:

- Local_GPC_Time variable in shared memory

Description:

This routine processes the TI command, which is used to enter the current time of day. After the TI command has been entered, this routine prompts the operator for the time of day, which must be entered in the format HH:MM:SS. It then calls the SET_TIME subroutine provided by the Time Manager package to update the global time record, which resides in shared memory.

5.2.2.1.12 Process Name: Get_Recovery_Mode

Inputs:

- Recovery mode (automatic or on-command)
- Channel id

Outputs:

- Channel_Recovery_Mode array

Description:

This routine processes the RM command, which is used to indicate the recovery mode (either automatic or on-command) for a specified channel. After the RM command has been entered, this routine prompts the operator for the specific channel id. The information is then stored in the Channel_Recovery_Mode array in the FDIR_Globals package, where it will be referenced by Transient FDIR.

5.2.2.1.13 Process Name: Do_Channel_Repair

Inputs:

- Id of repaired channel
- Current configuration
- Channel_Recovery_Mode array

Outputs:

- Channel_Repaired array

Description:

This routine processes the CR command, which is used to indicate that a failed channel has been repaired. After the CR command has been entered, this routine prompts the operator for the specific channel id. An error message is displayed for the operator if the specified is not down or if its recovery mode is automatic. Otherwise, the Channel_Repaired array in the FDIR_Globals package is updated to reflect the repaired channel, and the information will be used subsequently by Transient FDIR.

5.2.2.1.14 Process Name: Change_Color

Inputs:

- Id of desired color scheme

Outputs:

- Command to terminal to set color characteristics

Description:

This routine processes the CC command, which is used to indicate the desired color scheme for a VT240 terminal. After the CC command has been entered, this routine lists codes for each of the possible color schemes and prompts the operator to make a selection. The correct bit pattern is then written to the terminal to select the desired colors.

5.2.2.1.15 Process Name: Open_Memory

Inputs:

- Memory address

Outputs:

- Voted and individual channel values for specified address

Description:

This routine processes the OP command, which allows an operator to display and modify the contents of specified memory addresses. After the OP command has been entered, this routine prompts the operator for an address and then displays both a voted value and individual channel values for the specified address. If a channel is down, '*****' is displayed for its value. If a channel disagrees with the voted value, its own value is displayed; otherwise '-----' is displayed to indicate its agreement with the voted value.

After the value of the specified address has been displayed, the operator types another command to indicate what he wants to do next. The possible actions are:

N	Display the next address
P	Display the previous address
A	Display a new address (operator will be prompted for the new address)
=	Modify current address (operator must type in new value at the repositioned cursor)
Q	Exit (Help screen will then be displayed)

5.2.2.1.16 Process Name: CRT_Display_IOP

Inputs:

- Channel id
- Terminal input

Outputs:

- Invalid command message to terminal

Description:

This task provides a means by which an operator may communicate with the FTP, specifically the IOP, via a VT220 or VT240 terminal connected to an RS232 port. There are three instances of this task, one for each channel, so that simultaneous input from three different channels may be processed.

This task runs at the lowest priority and executes periodically. However, the periodicity is not achieved by using the Scheduler functions, but rather by a delay statement at the end of the task's main loop. The task is initially started by a rendezvous with the main program which identifies to each instance of the task which channel it is serving.

Within its main loop, the task calls a procedure, INPUT_WAITING, to determine if new input has come in from the RS232 port. If it has, the input command is read and an appropriate routine is called to process it. If there is no new input and the last command entered was an ST command (Status Display), the time and date on the status display are updated.

Most of the commands handled by this task are also handled by the CRT_Display_CP task, and the routines to process these commands have been described earlier in this section. The reader is referred to previous subsections for information on the CC, CL, FD, HE, NC, OP, RE, SE, and ST commands.

5.2.2.1.17 Process Name: Display_Event_Log

Inputs:

- Event log for the specified I/O network

Outputs:

- CRT display

Description:

This routine responds to V1, V2 and V3 (Display Event Log n) commands. It formats the event log for the specified I/O network and outputs it to a CRT. Each entry in the log identifies the time and date of the event, the relevant network, the id of the subroutine logging the event, and a brief description of the event

5.2.2.1.18 Process Name: Display_Error_Log

Inputs:

- Error log for the specified I/O network

Outputs:

- CRT display

Description:

This routine responds to R1, R2 and R3 (Display Error Log n) commands. It formats the error log for the specified I/O network and outputs it to a CRT. Each entry in the log identifies the time and date of the fault, the relevant network, the channel and node reporting the fault, an identifier of the test which detected the fault, and a brief description of the fault.

5.2.2.1.19 Process Name: I/O Network Displays

Inputs:

- I/O Network Status Information

Outputs:

- CRT display

Description:

These subroutines use the information maintained by the I/O Network Manager about the networks to format a pictorial display of the I/O networks. These subroutines are described in Section 3.6 of the "Input/Output Network Management Software" document, May 1988.

5.2.2.2 Macintosh Displays

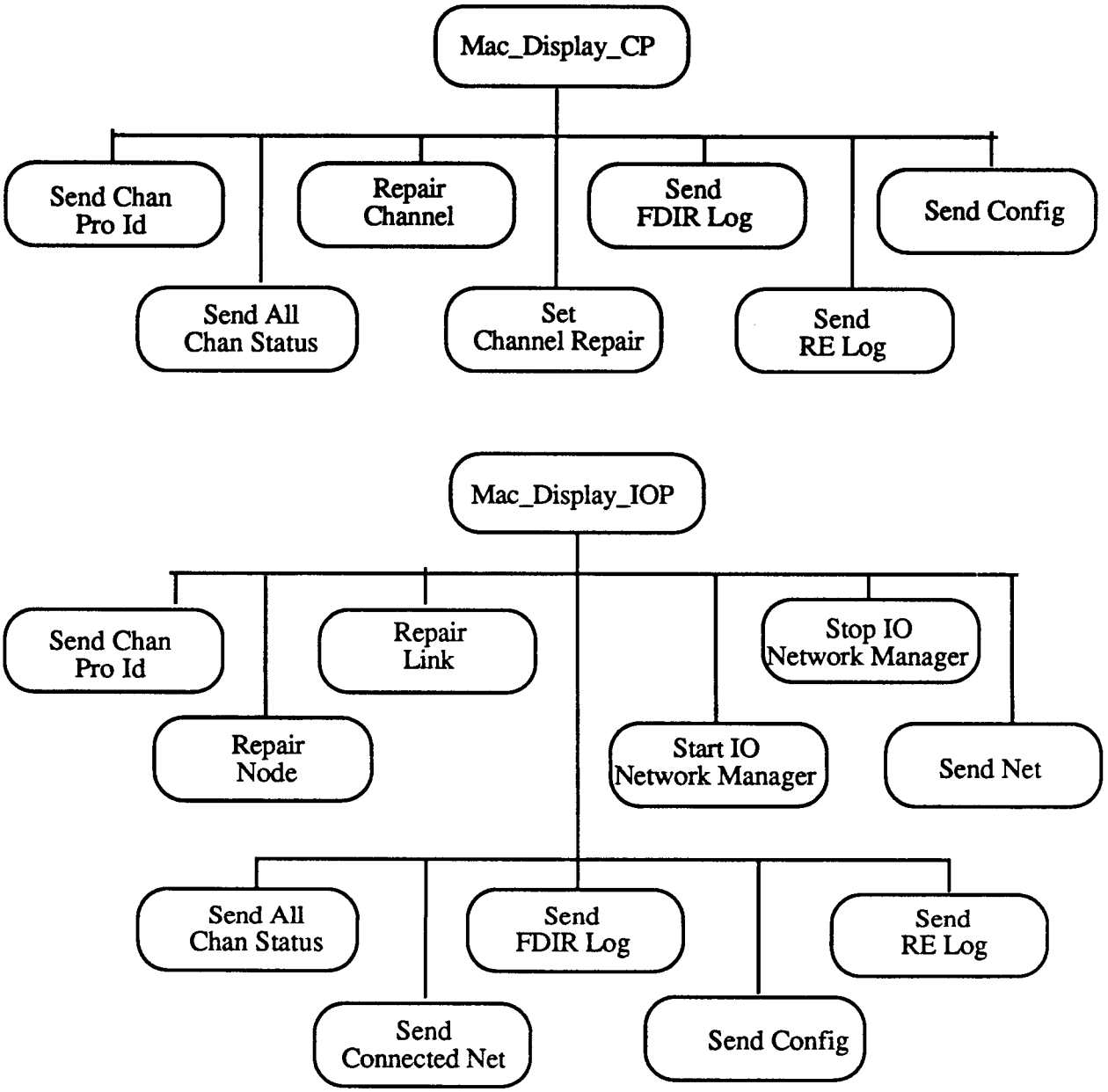


Figure 47. Macintosh Display Processes

5.2.2.2.1 Process Name: Mac_Display_CP

Inputs:

- Current configuration
- Commands from Macintosh

Outputs

- None

Description:

This task provides a means by which an operator may communicate with the FTP, specifically the CP, via a Macintosh computer. The communication takes place by means of an RS232 port, although it is a different port that that used by the terminal display task (refer to Section 5.2.2.1.1). Unlike the terminal display tasks, there is only one instance of this task, so commands from the Macintosh may come into only one channel at any given time.

This task runs at the lowest priority and executes periodically. However, the periodicity is not achieved by using the Scheduler functions, but rather by a delay statement at the end of the task's main loop.

Within its main loop, the task calls the procedure INPUT_WAITING to determine if new input has come in from the RS232 port. If it has, the input command is read and an appropriate routine is called to process it. If there is no new input, it sends the current configuration, date and time to the Macintosh.

5.2.2.2.2 Process Name: SendChanProId

Inputs:

- Processor id
- Channel id

Outputs

- Processor and channel id to Macintosh

Description:

This routine responds to the ID (Send Id) command. It transmits the processor id (CP or IOP) and channel id of the processor to which the Macintosh is connected.

5.2.2.2.3 Process Name: SendAllChanStatus

Inputs:

- None

Outputs

- None

Description:

This routine responds to the ST (Display FTP Status) command. It sends the status of all FTP channels to the Macintosh by calling the subroutine SendChanStatus, which sends the status (online or failed) of the individual channels.

5.2.2.2.4 Process Name: RepairChannel

Inputs:

- Current configuration
- Channel_Recovery_Mode array

Outputs

- Channel_Repaired array

Description:

This routine processes the CR command, which is used to indicate that a failed channel has been repaired. If the recovery mode for the specified channel is On-Command, the Channel_Repaired array is updated to reflect the repair; for the automatic recovery mode the command is ignored. The information in the Channel_Repaired array is subsequently used by Transient FDIR.

5.2.2.2.5 Process Name: SetChannelRepair

Inputs:

- Recovery mode (automatic or on-command)

Outputs

- Channel_Recovery_Mode array

Description:

This routine processes AU and OC commands, which are used to indicate the recovery mode to be used after a failure (either automatic or on-command). The same recovery mode is assigned to all channels, and the information is stored in the Channel_Recovery_Mode array.

5.2.2.2.6 Process Name: SendFDIRLog

Inputs:

- FDIR Error Log

Outputs

- Error log entries to Macintosh

Description:

This routine responds to the FD (Display FDIR Errors) command. It sends entries from the FDIR Error Log to the Macintosh. Rather than sending all entries currently in the log, the routine sends only those entries that have been made since the last time entries from the log were sent. For each log entry that is transmitted, the type of error, the faulty channel and the date and time of the fault are transmitted. Additional information is transmitted for the memory selftest errors (RAM scrub, RAM pattern, shared memory scrub, and PROM sum check): the address containing the fault, the actual value at that address and the expected value.

5.2.2.2.7 Process Name: SendConfig

Inputs:

- Integer version of current configuration

Outputs

- Integer version of current configuration to Macintosh

Description:

This routine transmits the integer version of the current FTP configuration to the Macintosh.

5.2.2.2.8 Process Name: SendRELog

Inputs:

- Reconfiguration log

Outputs

- Error log entries to Macintosh

Description:

This routine responds to the RE (Display Reconfigurations) command. It sends entries from the Reconfiguration log to the Macintosh. Rather than sending all entries currently in the log, the routine sends only those entries that have been made since the last time entries from the log were sent. For each log entry that is transmitted, the reconfiguration event (e.g., power on, disable, enable), the processor, the time and date of the event, and the channel(s) involved are transmitted.

5.2.2.2.9 Process Name: Mac_Display_IOP

Inputs:

- Current configuration
- Commands from Macintosh

Outputs

- None

Description:

This task provides a means by which an operator may communicate with the FTP, specifically the IOP, via a Macintosh computer. The communication takes place by means of an RS232 port, although it is a different port that that used by the terminal display task (refer to Section 5.2.2.1.16). Unlike the terminal display tasks, there is only one instance of this task, so commands from the Macintosh may come into only one channel at any given time.

This task runs at the lowest priority and executes periodically. However, the periodicity is not achieved by using the Scheduler functions, but rather by a delay statement at the end of the task's main loop.

Within its main loop, the task calls the procedure INPUT_WAITING to determine if new input has come in from the RS232 port. If it has, the input command is read and an appropriate routine is called to process it. If there is no new input, it sends the current configuration, date and time to the Macintosh.

Most of the commands handled by this task are also handled by the Mac_Display_CP task, and the routines to process these commands have been described earlier in this section. The reader is referred to previous subsections for descriptions of the routines which process the ST, FD, and RE commands.

5.2.2.2.10 Process Name: RepairNode

Inputs:

- Node id
- Port id

Outputs

- None

Description:

This subroutine processes the RN (Repair Node) command. It reads the node id and port id sent from the Macintosh and calls the Restore_Node_or_Link subroutine which is part of the I/O Network Manager. (The reader should refer to pp. 64 and 69-70 of "Input/Output Network Management Software, May 1988, for further information.)

5.2.2.2.11 Process Name: RepairLink

Inputs:

- Node id
- Port id

Outputs

- None

Description:

This subroutine processes the RL (Repair Link) command. It reads the node id and port id sent from the Macintosh and calls the Restore_Node_or_Link subroutine which is part of the I/O Network Manager. (The reader should refer to pp. 64 and 69-70 of "Input/Output Network Management Software, May 1988, for further information.)

5.2.2.2.12 Process Name: Start_IO_Network_Manager

Inputs:

- Network id

Outputs

- Network manager database

Description:

This routine is called when A1, A2 or A3 (Activate I/O Network) commands are received. This routine is part of the I/O Network Manager and is discussed in Section 3.1.1 of "Input/Output Network Management Software", May 1988.

5.2.2.2.13 Process Name: Stop_IO_Network_Manager

Inputs:

- Network id

Outputs

- Network manager database

Description:

This routine is called when D1, D2 or D3 (Deactivate I/O Network) commands are received.

5.2.2.2.14 Process Name: SendNet

Inputs:

- Network_Info array

Outputs

- Network status info to Macintosh

Description:

This procedure responds to N1, N2 and N3 commands, which are used to request a complete display of a particular network. Using the Network_Info array, which is maintained by the I/O Network Manager, this procedure transmits the status of every node and link in the specified network to the Macintosh.

5.2.2.2.15 Process Name: SendConnectedNet

Inputs:

- Active_Nets array

Outputs

- Id of connected network(s) to Macintosh

Description:

This routine responds to the CN (Send Connected Network) command. It sends the id of all the networks currently connected to the FTP.

5.2.3 Status Reporting Process Descriptions

Status reporting has not been implemented for the centralized AIPS configuration and so is not discussed in this section of the document.

6.0 LOCAL TIME MANAGER

Maintaining and disseminating the current time is the responsibility of the Local Time Manager, which resides on each GPC. This task maintains the current time by periodically updating a software variable to reflect a time consistent across all AIPS sites as well as to reflect elapsed time on the local GPC. This task also provides the current time to all software processes, including the runtime system. These responsibilities are divided between two functions, the Local Timekeeper and the System Timekeeper, as shown in Figure 48.

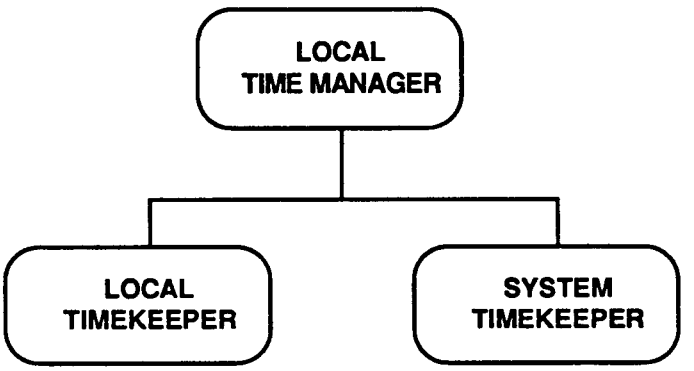


Figure 48. Local Time Manager

6.1 Local Time Manager Functional Requirements and Design

The Local Timekeeper function of the Local Time Manager is required for both the centralized and distributed configurations of AIPS. The System Timekeeper function, however, is used only in the distributed configuration. The flow of information through these two functions is shown in Figure 49.

6.1.1 Local Timekeeper

The Local Timekeeper maintains the current time on each GPC. It updates the current time by periodically adding the elapsed time as measured by the hardware timer. If the GPC is part of a distributed AIPS configuration, it also updates the current time based on calculations made by the System Timekeeper so that the time is consistent across all GPCs in the configuration. The Local Timekeeper also provides the current time, either in 24-hour clock time, clock ticks or millisecond time, to all software processes that request it.

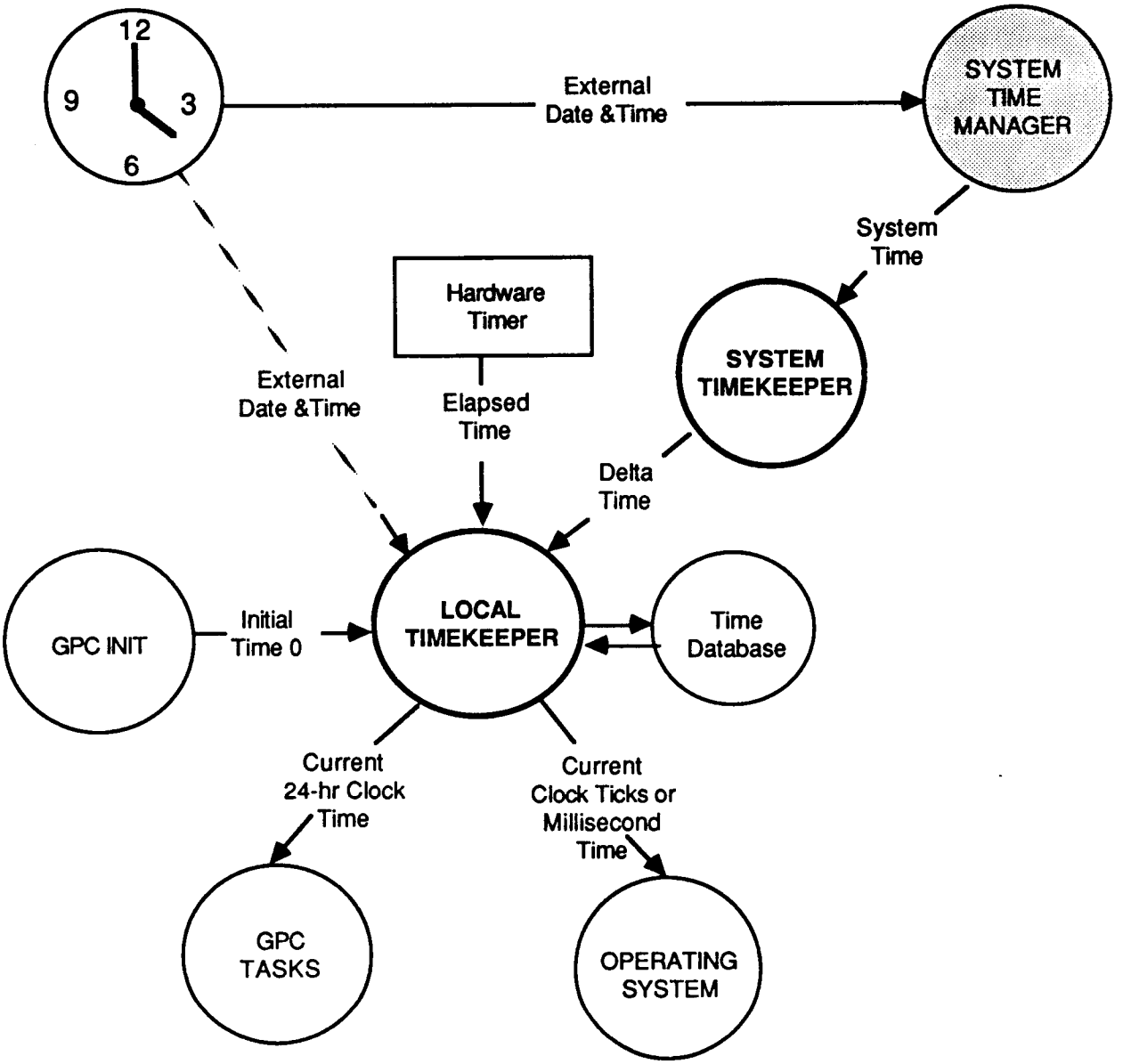


Figure 49. Information Flow through the Local Time Manager

The Local Timekeeper has three main components, the Time Database, the Time Database Manager and Calendar Extensions, which are shown in Figure 50.

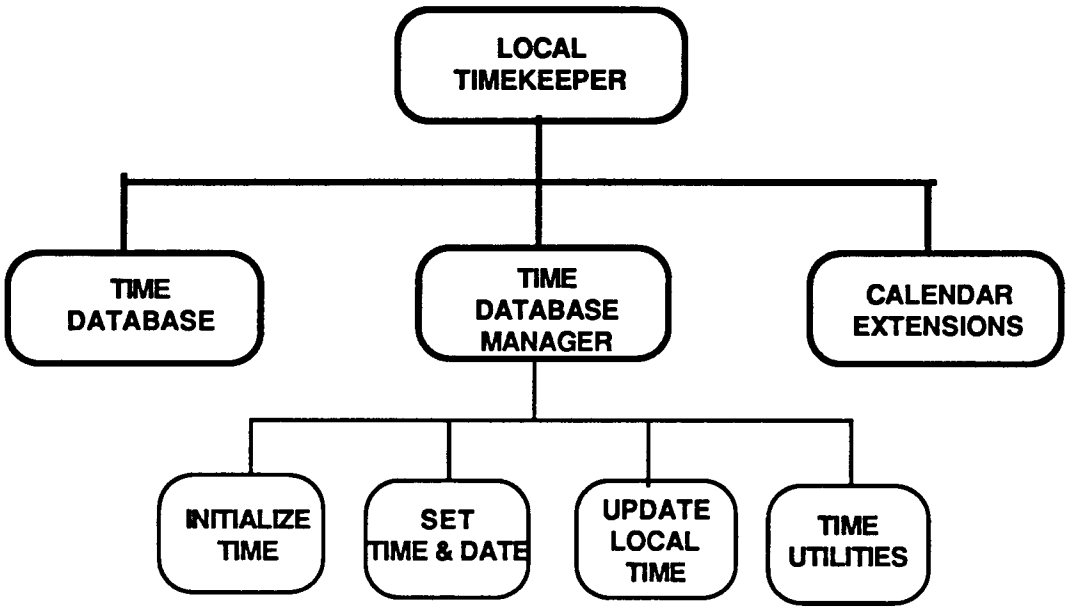


Figure 50. Local Timekeeper Components

6.1.1.1 Time Database

The Time Database consists of the hardware timer and software data structures used to maintain the current time. The hardware timer is located on the shared bus, and the data structures are located in shared memory, so the database may be accessed by both the CP and IOP.

The hardware timer is implemented as a 32-bit counter driven by the GPC's fault-tolerant clock. Since the fault-tolerant clock signal is generated every 4.125 μsec, each tick of the hardware timer represents 4.125 μsec.

The following data structures are used to maintain the current time.

Local_GPC_Time. This record contains the most recent value of the current time as computed by the Time Database Manager, along with the value of the hardware timer corresponding to that time. This time is represented internally by a record containing the year, julian day, and seconds of the day. It is initialized at power on to a default value provided by the operating system and is set to an actual (i.e., external) time and date after the GPC has been synchronized.

Timer Offset. Because the hardware timer cannot be set to any value other than 0, a software variable is required so that the timer may be aligned among unsynchronized channels. The offset represents some previous value of the hardware timer and must be added to the current timer value to obtain a true timer reading.

Delta Time. This value is provided by the System Timekeeper. It is used to correct the local GPC time so it corresponds to the universal time.

Average Drift Rate. This value is also provided by the System Timekeeper and reflects the extent to which local GPC time is drifting from the universal time.

6.1.1.2 Time Database Manager

The Time Database Manager updates the Time Database with the current time and provides the current time to any software process, either application or operating system, that requests it. Applications typically need the current time relative to the start of the mission and to a granularity of seconds, while the runtime system needs the current time relative to only an hour ago, but to a granularity of clock ticks or milliseconds. The Time Database Manager provides both types of current time; it also provides functions to manipulate clock tick and millisecond time such as comparing or differencing two times.

Updates to the current time (the Local_GPC_Time variable) are made by only one function (Update_Local_Time) on only one processor (the IOP). The update is made in such a way that no other task or processor can read the variable while it is being updated. After the variable has been updated, it is copied to each processor's local RAM, where it is referenced whenever the current 24-hour clock time is requested. The variable is read in such a way that a task will not be interrupted before the read is completed; this ensures that a reading task will always obtain a logically consistent time.

As seen in Figure 28, the Time Database Manager has four functions:

- Initializing the time and date, i.e., setting the time and date to an initial value at power-on;
- Setting the time and date, i.e., setting the time and date to some actual (i.e., external) time and date after the GPC has been initialized and synchronized;
- Updating local GPC time, i.e., using inputs from the hardware timer and the System Timekeeper to update the current time;
- Manipulating clock tick and millisecond time, for example, computing the difference between two successive values of the hardware timer, or comparing two times.

6.1.1.2.1 Initialize Time

The Initialize Time routine sets the date segment of the Local_GPC_Time record in the Time Database to the default initialization value supplied by the operating system. It sets the time segment of the Local_GPC_Time record to zero and clears and restarts the hardware system timer. Interrupts are disabled and the shared bus locked during this update. The Initialize_Time routine is called only once by the operating system and on only one processor (the IOP).

6.1.1.2.2 Immediate Set Time and Date

The Set Time and Set Date routines set the time and date segments, respectively, of the Local_GPC_Time record to the values supplied by the caller. Interrupts are disabled and the shared bus locked during this update. The Set_Time and Set_Date routines are also invoked by the System Timekeeper function. They are executed on only one processor (the IOP).

6.1.1.2.3 Update Local Time

The current time is updated by adding to it the time elapsed since the current time was last computed. The elapsed time is computed by finding the difference between the value of the hardware timer when the current time was last updated and its current value. The computation of current time is summarized as follows:

Local_GPC_Time = Local_GPC_Time + (current_timer_value - old_timer_value)

Because the time is stored in shared memory and because the time may be requested by a user (via the Ada CALENDAR.CLOCK function) at an unpredictable frequency, thus incurring a potentially large overhead in shared memory access time, the calculation of current GPC time is divided into two phases. In the first phase, one processor (the IOP) computes Local_GPC_Time once a second and stores it in the Time Database. A copy of Local_GPC_Time is also saved in the processor's local memory. In the other processor, Local_GPC_Time is not calculated, but only copied from shared to local memory, also once a second. Local_GPC_Time is thus maintained in shared memory to within one second of the actual time. In the second phase, requests for the precise current time, which are made by calling the Ada CALENDAR.CLOCK routine, are satisfied by computing the actual time as above using the locally maintained copies of Local_GPC_Time. In this manner the overhead of accessing shared memory is reduced, and the cost of computing the precise time is incurred only by those routines which request it.

The Update_Local_Time routine must correctly account for the rollover of the hardware timer. It must also correctly account for the rollover of the 24-hour time values, such as seconds at the end of the day and julian days at the end of the year. The CLOCK routine in

the Ada CALENDAR package must also account for rollover when computing exact time on demand.

In the Distributed AIPS system, the Update_Local_Time routine must occasionally adjust the value of the time to account for hardware timer drift among the local GPCs. A value to be used to modify accordingly the result of the above time computation is maintained in the Time Database by the Adjust_Time routine of the System Timekeeping function.

6.1.1.2.4 Time Utilities

The Time Database Manager also contains a number of auxiliary functions to provide time services to the operating system. These include (1) an align timer function which is called by the Lost Soul Sync task in order to update the timer offset variable and then clear and restart the hardware timer; (2) a function to access the hardware timer; and (3) functions to provide the current time directly from the hardware timer (either in clock ticks or in clock ticks converted to milliseconds) and functions to add and compare time presented in this format. These functions are required for operating system scheduling services.

6.1.1.3 Calendar Extensions

Calendar Extensions provide the standard time functions specified in the Ada Language Reference Manual along with additional functions that allow AIPS application users to manipulate or examine time in additional ways. The Ada-supplied package CALENDAR (cf., MIL-STD-1815A, Section 9.6) includes the standard function CLOCK, which returns the current 24-hour time. Other functions extract the year, month, day, and seconds, as specified in the ADA Language Reference Manual.

The CLOCK function enables any application to obtain the current 24-hour time, which is computed in the following manner:

time = old_time + (current_timer_value - old_timer_value)

where time is calculated by adding the amount of time elapsed since time was last updated (old_time). The elapsed time is computed by figuring the difference between the current value of the hardware timer and its value when the time record was last updated.

The CLOCK routine must correctly account for the rollover of the hardware timer. It must also correctly account for the rollover of the internal time values, such as seconds at the end of the day and julian days at the end of the year.

The Calendar package implemented for AIPS includes additional functions that are not predefined by Ada, such as a function to determine the julian day number of the current day and a function to convert a variable of julian date type to a variable of the private type time.

6.1.2 System Timekeeper

The System Timekeeper function is part the distributed AIPS configuration, but not part of the centralized configuration. Therefore, none of the System Timekeeper software has been implemented in the engineering model of the AIPS centralized configuration. The System Timekeeper function resides on one processor (either the CP or the IOP) at each processing site. It ensures that the Time Database on the local site has a time that is consistent with that of the universal time source. The System Time Manager, which is part of the System Manager software, resides on one GPC which has been designated as the universal time source, and periodically broadcasts the universal time. The System Timekeeper receives this universal time. The initial broadcast is used to set the GPC's local time. Subsequent broadcasts are used to adjust the GPC's local time to account for drift between the local GPC and the universal time source. The subsequent broadcasts are also used to monitor the universal time being broadcast for the occurrence of large differences between it and the local time.

If the System Time Manager is co-resident on the GPC, the function of System Timekeeper depends on the time source used by the System Time Manager. If the time source is local, i.e. the timer hardware resident on the GPC, then the System Timekeeper function is not invoked since no adjustment is necessary for consistency. Initialization of the time data is done in the same manner as in the centralized AIPS, i.e., by an operator or some other external source. If the time source is remote, then the System Timekeeper is invoked, but it receives the time data from the source via an external interface rather than the intercomputer network. The System Time Manager then uses the local copy of the adjusted system time for its periodic broadcast.

Figure 51 shows the functions of the System Timekeeper.

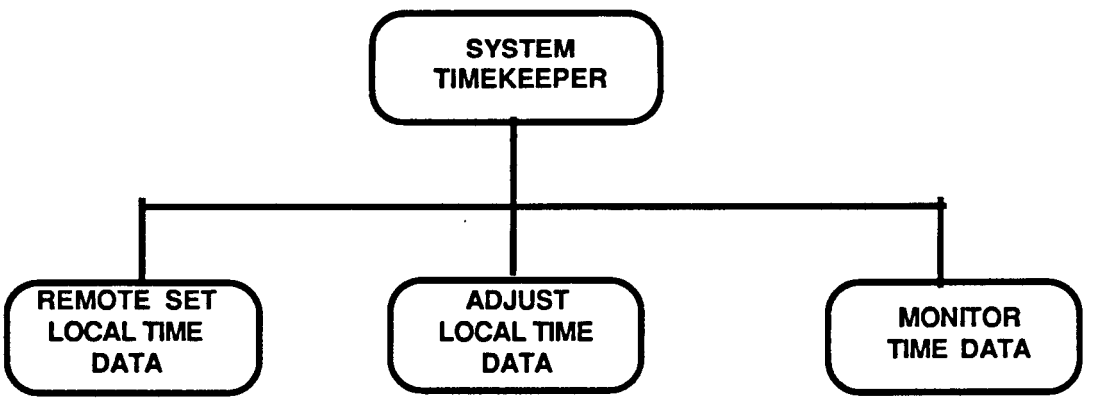


Figure 51. System Timekeeper

6.1.2.1 Remote Set Local Time Data

This process sets time to a value based on the time data received in the initial broadcast from the System Time Manager and the time stamps appended by Inter-Computer System Services. The algorithm that calculates the correct time values to initially set and synchronize the local time with the universal time source is described below.

There are two main values to be considered: (1) the initial time (designated with a subscript 0) when the System Time Manager reads the system time source for broadcast and (2) the current time (designated with subscript 1) in the local GPC to be determined. In the following, time quantities in the local GPC are shown with a prime ('), while quantities in the global GPC are shown without the prime. Actual values are shown in upper case and estimates are shown in lower case. The problem is to find a value for $time_1'$, where $time_1'$ is the new time estimate in the local GPC that is required, so that:

$$|TIME_1 - TIME_1'| \leq \varepsilon$$

where	$TIME_1$	is the current time in the global GPC,
	$TIME_1'$	is the current time in the local GPC,
and	ε	is a time skew less than one millisecond.

To estimate $TIME_1'$ in the local GPC, one must estimate $TIME_1$. $TIME_1$ can be estimated as follows:

(1) $$time_1 = TIME_0 + DLB + DLX + DLA$$

where	$TIME_0$	is the time read for broadcast in the global GPC,
	DLB	is the delay before transmission in the global GPC,
	DLX	is the transmission delay,
and	DLA	is the delay after reception in the local GPC.

DLX will be estimated as a fixed constant (it is expected to be less than 20 microseconds). The other quantities can be calculated from the time broadcast as follows. Set_Time_Data includes the following:

	$TIME_0$	- the time read for broadcast in the global GPC,
	TTS	- the transmit time stamp (the time of transmission set by the global GPC),
and	RTS	- the receive time stamp (the time of reception set by the local GPC).

From these values, one can determine:

(2) $$DLB = TTS - TIME_0$$

Using CALENDAR.CLOCK one can obtain the following:

$TIME_{now}'$ - the current time while processing the time request in the local GPC.

From this value, one can determine:

(3) $DLA = TIME_{now}' - RTS$

The quantities from equations (2) and (3) are substituted into (1) to find $time_1'$. It is assumed that $time_1$ is equal to $time_1'$ within one millisecond.

After $time_1'$ has been determined the local copy of universal time source is set by invoking the Local Timekeeper routine, Immediate Set Time and Date.

6.1.2.2 Adjust Local Time Data

This process is responsible for adjusting the local copy of the universal time source based on the periodic time broadcast from the System Time Manager and the time stamps appended by Inter-Computer System Service. The adjustment value is used to maintain a consistent time value throughout the AIPS. The following is a description of the algorithm that calculates the adjustment value, Delta Time, that is used to dedrift the local copy of the universal time source.

First, the Average Drift Rate is calculated as a moving average of some number (Drift_Readings) of the most recently measured values for the instantaneous Drift_Rate:

(1) $$\langle DR \rangle = (1/N) \sum_{i=1}^{N} DR_i$$

where $\langle DR \rangle$ is the Average_Drift_Rate,
N is the number (Drift_Readings) of readings included,
and DR_i is the i^{th} instantaneous Drift_Rate.

The method for calculating the instantaneous Drift Rate is as follows. Each time the time broadcast occurs and the time data is received, a value is calculated for Delta Time, the correction needed so the time in the local GPC agrees with the broadcast system time.

If the time has not been dedrifted (by Update Local Time) since the last time broadcast, then:

(2a) $$DR = (DEL_{now} - DEL_{prior}) / (TIME_{now} - TIME_{prior})$$

where DR is the instantaneous Drift_Rate,
DEL_{now} is the current value for Delta_Time,
DEL_{prior} is the previous value for Delta_Time,
$TIME_{now}$ is the current time,
and $TIME_{prior}$ is the previous time.

If the time has been dedrifted since the last time broadcast, then:

(2b) $$DR = DEL_{now} / (TIME_{now} - TIME_{DT})$$

where $TIME_{DT}$ is the last Dedrift_Time,
and the other quantities are as above.

The method for calculating Delta_Time based on the time data broadcast is as follows. There are two main values to be considered: (1) the time (designated with a zero subscript) when the System Time Manager reads the system time source for broadcast, and (2) the current time (designated with a one subscript) when Delta_Time is to be determined. In the following, time quantities in the local GPC are shown with a prime ('), while quantities in the global GPC are unprimed. The problem is to find a value for DEL, where DEL is the correction to time in the local GPC that is needed, so that:

$$|TIME_1 - TIME_1'| \leq \varepsilon$$

where $TIME_1$ is the current time in the global GPC,
$TIME_1'$ is the current time in the local GPC,
ε is a time skew less than one millisecond.

To estimate DEL in the local GPC, one must estimate $TIME_1$, and measure $TIME_1'$, then:

(3) $$DEL = time_1 - TIME_1'$$

where $time_1$ is an estimate of $TIME_1$.

$TIME_1$ can be estimated as follows:

(4) $$time_1 = TIME_0 + DLB + DLX + DLA$$

where $TIME_0$ is the time read for broadcast in the global GPC,
DLB is the delay before transmission in the global GPC,
DLX is the transmission delay,
and DLA is the delay after reception in the local GPC.

DLX will be estimated as a fixed constant (it is expected to be less than 20 milliseconds). The other quantities can be calculated from the time broadcast as follows:

$TIME_0$ - the time read for broadcast in the global GPC
TTS - the transmit time stamp,
and RTS - the receive time stamp.

From these values, one can determine:

(5) $$DLB = TTS - TIME_0$$

Using CALENDAR.CLOCK, one can obtain the following:

$TIME_1'$ - the current time in the local GPC.

From this value, one can determine:

(6) $$DLA = TIME_1' - RTS$$

The quantities from equations (5) and (6) are substituted into (3) and (4) to find DEL. The current value for DEL is then substituted into (2) to find DR, and that is in turn substituted into equation (1) to find the moving average, <DR>.

These values are stored as part of the system time database. The Local Timekeeper function, Update Local Time, uses the Delta_Time value to dedrift the local copy of the universal time source, thus maintaining a consistent time.

6.1.2.3 Monitor Time Data

This process monitors the drift between the broadcast time data and local time data. If the time drift is large enough to indicate errors, this timer drift error is reported to the System Time Manager.

6.2 Local Time Manager Software Specifications

6.2.1 Local Timekeeper Process Descriptions

6.2.1.1 Time Database Manager Process Descriptions

6.2.1.1.1 Process Name: INIT_TIME

Inputs:

- Year, Julian Date, Time of day (in seconds)

Outputs:

- None

Description:

The Local_GPC_Time in the System Time Database is initialized to the supplied date and time. This routine is invoked at power-on by only one processor (the IOP). The date is set to January 1, 1901. Interrupts are disabled and the shared bus is locked while Local_GPC_Time is being modified.

6.2.1.1.2 Process Name: SET_DATE

Inputs:

- Year, Julian Date or
  Year, Month, Day

Outputs:

- None

Description:

The date portion of the Local_GPC_Time in the System Time Database is set to the supplied date. Interrupts are disabled and the shared bus is locked while Local_GPC_Time is being modified. This routine is restricted to one processor (the IOP).

6.2.1.1.3 Process Name: SET_TIME

Inputs:

- Time of day (in seconds)

Outputs:

- None

Description:

The time portion Local_GPC_Time in the System Time Database is set to the supplied time. Interrupts are disabled and the shared bus is locked while Local_GPC_Time is being modified. This routine is restricted to one processor (CP or IOP).

6.2.1.1.4 Process Name: UPDATE_LOCAL_TIME

Inputs:

- Current_timer_valuefrom the system timer
- System_Time_Database

Outputs:

- Local_GPC_Time to CP and IOP

Implementation Requirements:

- Periodicity: 1 sec.

Description:

The Local_GPC_Time in the System Time Database is updated by the routine Update_Local_Time. In the IOP, the current time is computed according to the expression:

Local_GPC_Time = Local_GPC_Time + (current_timer_value - old_timer_value)

where time is calculated by adding the old time (the time of the last time update) and the difference between the current timer value and the old timer value (the system timer value of the last time update). It is then stored along with the current timer value in the System Time Database in shared memory. A copy of the data is preserved in local memory for use by the Ada CALENDAR.CLOCK function. In the CP, the current time is not calculated, but local copies are made of the values of the Local_GPC_Time and current timer which are in shared memory . When the current time is requested by an application via the CLOCK function as defined in the Ada CALENDAR package, routines local to both the CP and IOP compute the current time according to the above algorithm, using the locally stored copies of Local_GPC_Time and Old_Timer_Value. In this manner, the overhead of storing the current time in shared memory is reduced to the periodicity of the Update_Local_Time (no more frequently than once per second), while the overhead of computing the exact current time is incurred only by those routines which call CLOCK on demand.

6.2.1.1.5 Time Utilities

The following routines provide access to the GPC hardware timer. The timer is implemented as a 32-bit counter register driven by the GPC's fault-tolerant clock. The fault-tolerant clock signal is generated every 4.125 μsec. therefore, each tick of the hardware timer represents 4.125 μsec. Due to hardware limitations (specifically, the hardware timer cannot be set to any value other than zero), it is necessary to maintain a 32-bit offset value, which is added to the value of the hardware timer to give the actual current time value.

6.2.1.1.5.1 Process Name: ALIGN_TIME

Inputs:

- None

Outputs:

- None

Description:

The system timer offset is updated with the sum of the current value of the system timer offset and the value of the GPC hardware timer. The hardware timer is then cleared and restarted. Due to hardware limitations (specifically, the timer register cannot be set to any value other than zero), it is necessary to maintain a 32-bit offset value which is added to the value of the timer register to give the actual time value. By saving the current time value in the system timer offset and clearing the hardware register, the invoking routine is able to align the the GPC time by aligning only the timer offset which can be set to the appropriate value. It is assumed that interrupts have been disabled by the invoking routine. This routine is called only by Lost Soul Synch when recovering a channel.

6.2.1.1.5.2 Process Name: READ_TIMER_AND_OFFSET

Inputs:

- None

Outputs:

- Time duration (in seconds)

Description:

The GPC hardware timer is read and added to the value of the system timer offset. The result (in seconds) is returned to the caller. Due to hardware limitations (specifically, the timer register cannot be set to any value other than zero), it is necessary to maintain a 32-bit

offset value which is added to the value of the timer register to give the actual time value. This function may be called indirectly by an application routine via the CLOCK function of the Ada CALENDAR package, and may therefore be invoked with interrupts enabled. It would then be possible for the system timer offset to be aligned while the GPC hardware timer was being read, resulting in an unmatched set of values; to insure against this possibility, the offset is read, the hardware timer is read, and the offset is read once again. If the two values of the offset agree, a consistent set of values has been obtained; if not, the procedure is repeated. The resulting time value (in 4.125 μsec. clock tick units) is multiplied by 4.125e-6 to convert it to seconds and is returned to the caller.

6.2.1.1.5.3 Process Name: MS_TIME

Inputs:

- Time (in seconds)

Outputs:

- Time (in milliseconds)

Description:

The specified time in seconds is converted to milliseconds. This routine is needed to convert a time value of the Ada fixed point type DURATION to the integer derived type MS_TIME_T which is used by the Operating System to manipulate values expressed in milliseconds.

6.2.1.1.5.4 Process Name: MS_ADD

Inputs:

- Time1, Time2 (in milliseconds)

Outputs:

- Time (in milliseconds)

Description:

The sum of the input values, modulo the maximum value of the GPC hardware timer, is returned. The maximum value of the hardware timer is 2**31 * 4.125e-6 sec.

6.2.1.1.5.5 Process Name: MS_DIFFERENCE

Inputs:

- Time1, Time2 (in milliseconds)

Outputs:

- Time (in milliseconds)

Description:

The difference of the input values, modulo the maximum value of the GPC hardware timer, is returned. The maximum value of the hardware timer is 2**31 * 4.125e-6 sec. If Time1 is before Time2, the difference is returned. If Time1 is later than Time2, zero is returned. For efficiency, this routine is implemented so as to avoid the use of divide instructions.

6.2.1.1.5.6 Process Name: MS_CLOCK

Inputs:

- None

Outputs:

- Time (in milliseconds)

Description:

The value of the GPC timer, converted to milliseconds, is returned. The GPC time in seconds is determined using the Read_Timer_and_Offset function (6.1.1.5.2), converted to milliseconds using MS_Time (6.1.1.5.3), and returned to the caller.

6.2.1.2 CALENDAR Extensions Process Descriptions

6.2.1.2.1 Process Name: CLOCK

Inputs:

- None

Outputs:

- Current Time

Description:

The local GPC time is periodically updated by the routine Update_Local_Time. This time is maintained in the Local_GPC_Time data structure in the System Time Database. A local copy of this data is also preserved by both the CP and IOP at the time of update. When the current time is requested by an application via the CALENDAR.CLOCK function, the current time is calculated according to the expression

time = Local_GPC_Time + (current_timer_value - old_timer_value)

using the locally stored copies of Local_GPC_Time and Old_Timer_Value. In this manner, the overhead of storing the current time in shared memory is reduced to the periodicity of the Update_Timer_Routine, while the overhead of computing the exact current time is incurred only by those routines which call CLOCK on demand.

Since the local copy of the Local_GPC_Time data structure consists of several data values, a single move-multiple machine instruction is used to read these values into registers in order to insure consistency of the data without disabling interrupts.

6.2.1.2.2 Process Name: JULIAN_DATE

Inputs:

- Current Time

Outputs:

- Julian Date

Description:

The Julian date corresponding to the supplied time is returned to the caller. This function is necessary since the Julian date is a component of the Ada private type TIME and cannot be accessed directly.

6.2.1.2.3 Process Name: TIME_OF

Inputs:

- Year, Julian Date, Time of Day (in seconds)

Outputs:

- Time

Description:

The supplied date (expressed in terms of the Julian day instead of the month and day as defined in the standard Ada package CALENDAR) and time are converted to the Ada private type TIME. This function is necessary since the input values are components of the Ada private type TIME and cannot be assigned directly.

6.2.2 System Timekeeper Process Descriptions

The System Timekeeper has not been implemented for the centralized AIPS configuration and therefore is not discussed in this section of the document.

7.0 CONCLUSIONS AND RECOMMENDATIONS

Local System Services for a centralized configuration of the AIPS engineering model have been designed, implemented and tested. This software provides the redundancy management and operating system support for the core GPC. The responsibilities of this software include initialization of the GPC hardware and software, task execution and memory management, intertask communication, detection and isolation of hardware faults in the core GPC, collection and dissemination of local GPC status, and management and dissemination of the system time.

Additional functions of Local Systems Services required for a distributed AIPS configuration are presently in the design phase. These functions, including GPC Resource Allocation, Status Reporting, and System Timekeeping, will be documented when the distributed AIPS software is completed.

The Local System Services software is composed of 25,935 lines of Ada source code, which includes 11,150 Ada statements and comments. When compiled and linked into an executable module, this code (which includes instructions and global variables) requires 354,380 bytes of FTP memory.

7.1 Testing of Local System Services Software

Preliminary testing of Local System Services software was done by forcing hardware faults and by executing modified software that generated artificial error conditions. An unsynchronized channel, for example, could be caused by resetting a processor or turning off a channel's power. An interstage failure could be caused by flipping a specially constructed switch that grounded the interstage's power supply. The software that analyzed data exchange link failures, on the other hand, was tested by executing a modified version that selectively changed the values read from the error latches to values which represented a fault condition. Similarly, error analysis paths in the background selftests were exercised by modifying the results of selected tests to make it appear that a fault had occurred. The results of these preliminary tests were monitored by setting breakpoints, using the display routines described in Section 4.2.4.1, and using the debug log described in Section 4.2.4.2. The software correctly identified the fault and reconfigured the FTP in all tests.

Some performance metrics were gathered using both oscilloscopes and the debug log to make time measurements. Operating system overhead and FDIR overhead were measured, and the results are summarized in Figure 52. Since the FDIR tasks are periodic, their numbers represent the time required for each iteration.

Extensive systematic testing of the GPC hardware for performance and reliability under fault-free and degraded conditions remains to be done. A hardware fault injector will be used for much of this testing.

Function	AIPS FTP Overhead(Verdix 5.4)
CP FDIR (no fault conditions)	2150 μs
IOP FDIR (no fault conditions)	1800 μs
Regular Ada Delay	
Scheduling	11.8 msec
Dispatch	1200 μs
Modified Ada Delay	
Scheduling	1300 μs
Dispatch	1200 μs
Simple Ada rendezvous	500 μs
Return Again	1300 μs
Timer Dispatch	2100 μs
Context switch	480 μs
Local Event Dispatch	1000 μs
Remote Task (Event) Dispatch	
Signal (one CPU)	360 μs
Dispatch (Other CPU)	1100 μs
Total (some overlap)	1300 μs
Shared Memory Protected Access	
Access (one CPU)	650 μs
Rupt handler (Other CPU)	300 μs

Figure 52. FDIR and Operating System Overhead Measurements

7.2 Future Work

There are several areas of design and validation for the AIPS Fault Tolerant Processor that remain to be addressed in the AIPS program. These include:

- channel resynchronization after a transient fault or repair event,
- differentiation between transient and intermittent faults,
- additional monitor interlock functionality and fault detection,
- fault identification in a duplex FTP, and
- additional protection against common mode faults, i.e., faults that affect more than one fault containment region simultaneously.

7.2.1 Channel Resynchronization

After a transient fault or repair event, it is necessary to bring the previously faulty channel back into synchronism with the others. To assure identical states, the redundant channels must be synchronized to the instruction level and must have identical values for all volatile hardware control registers, memory and processor registers.

As discussed in Section 4.2.2.2.4, the synchronization of channels to the instruction level is done by an assembly language routine that makes use of the data exchange hardware and fault tolerant clock. In the current implementation, the instructions executed in this routine must take a known number of fault-tolerant clock cycles to be able to synchronize divergent channels. The routine is dependent on the instruction fetch and execution time of the particular microprocessor being used. A way must be found to implement the synchronization routine so that it is independent of the microprocessor instruction times. An algorithm must be developed which can also be rigorously proven to be correct.

A second problem with channel synchronization involves the time required to give all channels identical values for their volatile registers and memory, i.e., to 'align' them. Since AIPS software is very sophisticated and complex, it requires a large amount of memory for static and local variables. Because alignment is accomplished by transferring all values through the data exchange one 32-bit word at a time at a basic rate of 15 μsec per word, and because the entire synchronization process must run uninterrupted, the alignment process takes an unacceptably long time. Critical application tasks miss too many cycles. The resynchronization process cannot be omitted because it allows a GPC to sustain more than one fault, thus providing greater reliability.

Several alternatives can be explored to speed up alignment such as an alternative data transfer mechanism (not the present data exchange hardware) or by partitioning memory and only bringing back the critical application task first, or using a hardware memory consistency checker whose purpose is to identify sections of memory which have changed and need to be realigned. The performance and reliability of new fast realignment techniques can then be modeled.

7.2.2 Differentiation Between Transient and Intermittent Faults

When recovering a failed channel, system resources are used most efficiently if a distinction is made between transient failures and hard failures. The current design for making this distinction was previously described in Section 4.1.2.1. This design does not account completely, however, for intermittent faults, i.e., hard faults which occur sporadically. The probation concept used in this design will catch intermittent faults which occur at brief intervals (that is, within the probation period) but will not detect intermittent faults which occur at longer intervals (e.g., once every 10 minutes).

Two main areas need to be studied in this regard: (1) how do we distinguish between transient and intermittent faults, and (2) how do we treat the intermittent fault once it has been identified.

Intermittent faults may be identified by the periodicity and frequency of their occurrence. For example, a fault occurring once an hour may be arbitrarily designated as transient while one occurring once a minute would be designated as intermittent. History records could be kept, so that a fault which recurred three times within 24 hours might be designated as transient while one which recurred 100 times within this period would be designated as intermittent.

The second area to be studied concerns how an intermittent fault should be treated. When a fault is intermittent, is the reliability of the system greater if the channel is taken off line (leaving a less robust GPC) or if the channel is allowed to continue (with the chance of the intermittent fault occurring at the same time as a second fault)? At what point does continually recovering an intermittently faulty channel use more system resources than the additional reliability warrants?

We need to study the relative frequency of transient, intermittent and hard faults. The results of the study will be used to create/update Markov models of the AIPS system to include the effects of transient and intermittent faults as well as hard faults. Sensitivity analysis on these models can be performed by varying parameters such as the frequency of an intermittent fault. The results of this analysis will be used to improve the transient analysis algorithm to calculate the appropriate thresholds to use for the transient fault analysis. At present those parameters are guesses.

7.2.3 Additional Monitor Interlock Functionality and Fault Detection

At present, the engage/disengage function of the hardware Monitor Interlock is used to disengage or engage a channel's outputs and the watchdog timer function is used to reset a runaway processor. The Monitor Interlock also has a lock feature which provides more stringent control than does the engage mechanism. Once a faulty channel is locked by the other two channels, it cannot be reset other than by operator intervention. The channel remains locked even if the system is restarted or the power is cycled. This feature is not used presently. A study needs to be done to determine how the use of this feature will affect the reliability of the system and to determine what situations would require using the lock.

A reliable method for detecting latent faults in the monitor interlock is also needed. The monitor interlock is presently tested by comparing the three channels' discrete values. This does not effectively test the hardware, but extensive selftest of this hardware may not be possible in real time. To allow selftests of this hardware in real time, a complicated scheme requiring additional hardware and software must be developed to disengage the entire FTP from the rest of the system. A selftest needs to be developed that will thoroughly test the monitor interlock at preflight and a subset of this test could perhaps be added to the runtime selftests.

7.2.4 Duplex FTP Fault Isolation and Reconfiguration

According to fault tolerance principles, a quadruply redundant FTP can sustain two faults and detect a third, while a triply redundant FTP can sustain one fault and detect a second with 100% coverage. If a fault occurs in a duplex FTP, the failure can be detected with 100% coverage but cannot be isolated to one one of the two channels with high coverage. The present redundancy management algorithm simply goes to a fail-safe state by disengaging both channels. However, for those applications where it is not possible to downmode to a fail-safe mode because continued operation is absolutely essential, some attempt must be made to isolate the fault, albeit at less than 100% coverage.

There are several hardware features that can be used to develop the desired isolation algorithm. In particular, since the monitor interlock engage/disengage and watchdog timer discretes are available to all channels via point to point connections, these discretes can be accessed even if the channels are not in sync. A lone channel could perform self tests to determine if it was the faulty member of the duplex. If it fails the tests it could disengage itself. If it passes, it would check if the other channel was disengaged. (The coverage of this action is the coverage that can be obtained by selftests running on a simplex processor.) The other channel being disengaged would indicate that that channel was faulty and the lone channel would be able to continue as a simplex. Other algorithms to allow a duplex to degrade to an operational simplex need to be studied.

7.2.5 Common Mode Fault Protection

At present Local System Services provides a restart/reinitialization of the FTP as a protection for common mode faults, i.e., faults that affect more than one fault containment region simultaneously or almost simultaneously. Since all the manifestations or the errors generated from common mode faults are not known, the restart only protects the FTP when the error symptoms are the following: hardware exception, operating system software exception, desynchronization of channels, or watchdog timer reset.

Further work needs to be done to identify and categorize other potential sources of common mode faults and potential errors generated by those faults. Additional fault avoidance techniques such as proof-of-correctness and alternative recovery techniques such as function migration, software watchdog timers and multiversion software should be studied for practicality and effectiveness.

8.0 REFERENCES

1. T. B. Smith, III, "Fault Tolerant Processor Concepts and Operation", 14th International Symposium on Fault Tolerant Computing, Orlando, Florida, June, 1984.

2. J. H. Lala, "A Byzantine Resilient Fault-Tolerant Computer for Nuclear Power Plant Applications", 16th International Symposium on Fault Tolerant Computing, Vienna, Austria, July, 1986.

3. T. Massotto and L. Alger, "Advanced Information Processing System: Input/Output System Services", to be published.

4. G. Nagle, L. Alger and A. Kemp, "Advanced Information Processing System: Input/Output Network Management Software", NASA Contractor Report 181678, May, 1988.

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-181767		

4. Title and Subtitle	5. Report Date
Advanced Information Processing System: Local System Services	April, 1989
	6. Performing Organization Code

7. Author(s)	8. Performing Organization Report No.
L. Burkhardt, L. Alger, R. Whittredge, and P. Stasiowski	
	10. Work Unit No. 506-46-21-05

9. Performing Organization Name and Address	11. Contract or Grant No.
The Charles Stark Draper Laboratory, Inc. 555 Technology Square Cambridge, MA 02139	NAS1-18061
	13. Type of Report and Period Covered: Contractor Report

12. Sponsoring Agency Name and Address	14. Sponsoring Agency Code
National Aeronautics and Space Administration Langley Research Center Hampton, VA 23665-5225	

15. Supplementary Notes

F. L. Pitts was the Langley Technical Monitor.

16. Abstract

The Advanced Information Processing System (AIPS) is a multi-computer architecture composed of hardware and software "building blocks" that can be configured to meet a broad range of application requirements. The hardware building blocks are fault-tolerant, general-purpose computers, fault- and damage-tolerant networks (both computer and input/output), and interfaces between the networks and the computers. The software building blocks are the major software functions: local system services, input/output, system services, inter-computer system services, and the system manager.

The foundation of the local system services is an operating system with the functions required for a traditional real-time multi-tasking computer, such as task scheduling, inter-task communication, memory management, interrupt handling, and time maintenance. Resting on this foundation are the redundancy management functions necessary in a redundant computer and the status reporting functions required for an operator interface. This report documents the functional requirements, functional design and detailed specifications for all the local system services.

17. Key Words (Suggested by Author(s))

Ada Multi-Tasking Operating System
Fault Tolerant Processor System Software
Fault Tolerant Processor, Reconfiguration
Redundancy Management
Fault Detection and Identification
Advanced Information Processing System

18. Distribution Statement

Unclassified-Unlimited

Subject Category 62

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price
Unclassified	Unclassified	189	