P.31

NASA Technical Memorandum 89154

DOCUMENTATION OF THE CURRENT FAULT DETECTION, ISOLATION, AND RECONFIGURATION SOFTWARE OF THE AIPS FAULT-TOLERANT PROCESSOR

David T. Lanning
Allen W. Shepard
Sally C. Johnson

(NASA-TM-89154) DOCUMENTATION OF THE CURRENT FAULT DETECTION, ISOLATION AND RECONFIGURATION SOFTWARE OF THE AIPS FAULT-TOLERANT PROCESSOR (NASA) 31 p N90-10602
CSCL 01B G3/61 Unclas 0237136

August 1987

Date for general release August 31, 1989

NASA
National Aeronautics and Space Administration
Langley Research Center
Hampton, Virginia 23665-5225

TABLE OF CONTENTS

INTRODUCTION

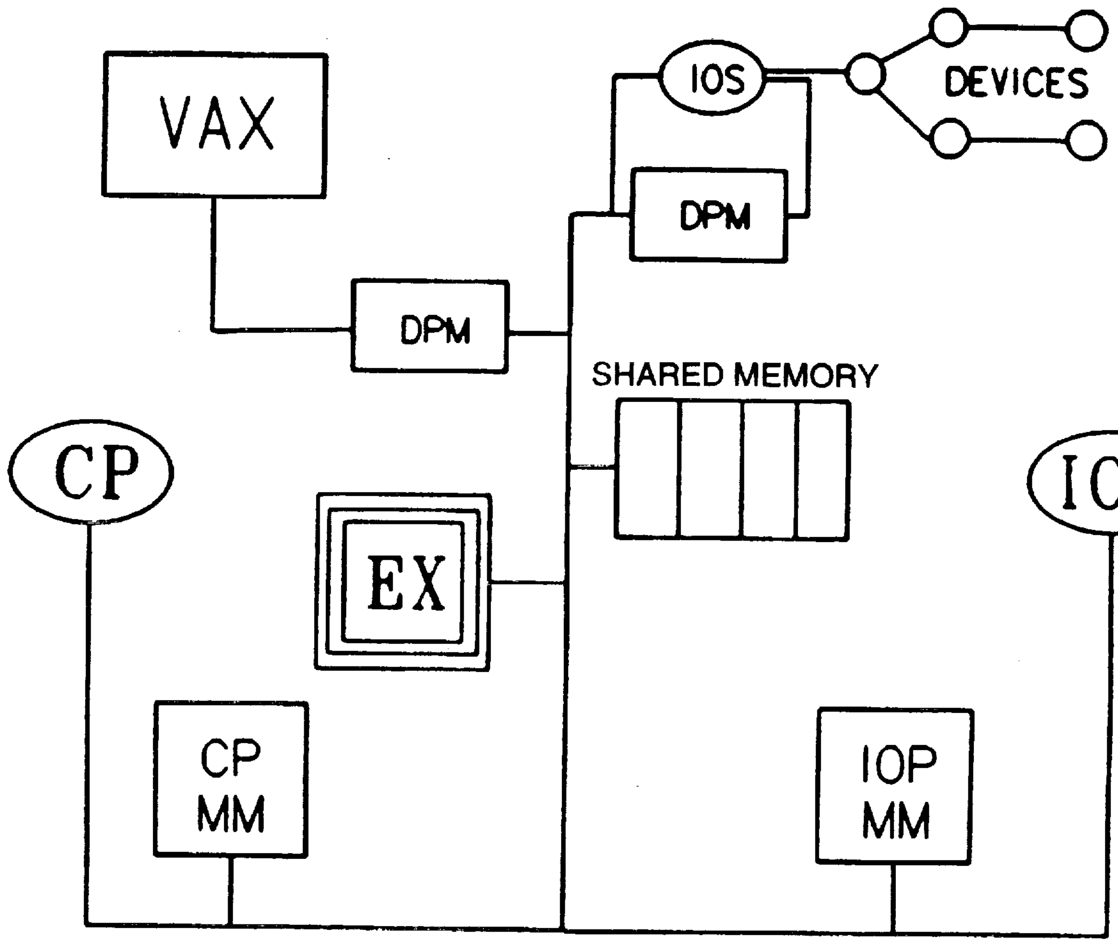
This report contains documentation of the December 1986 version of the Ada code for the fault detection, isolation, and reconfiguration (FDIR) functions of the Advanced Information Processing System (AIPS) Fault-Tolerant Processor (FTP). The FTP is a major component of the AIPS system being developed by the Charles Stark Draper Laboratory, Inc for NASA. This documentation was generated to aid in determining the current status of the FDIR software and to reveal in detail how the FTP FDIR functions are implemented. Because the FTP is still under development and the software is constantly undergoing changes, this should not be considered final documentation of the FDIR software of the FTP. The documentation was done by two Old Dominion University Students, David T. Lanning and Allen W. Shepard.

The FDIR routines are divided into "fast" FDIR functions and "slow" FDIR functions. The fast FDIR routines are purely for fault detection. The slow FDIR routines, which take longer to execute, hence the name "slow", are executed whenever a fault is detected. These routines isolate the detected fault and reconfigure the system to remove the failed processor. There are also slow FDIR routines for reconfiguring the system to bring back in a processor after it has been repaired or has recovered from a transient fault. The FDIR software was originally written in the C language, then was translated to Ada and modified by Gregory Greeley. Some of the functions are written in Motorola 68000 assembly language to facilitate bit-manipulations of variables used for writing to and reading from memory-mapped error latches.

The hardware architecture of the FTP is briefly described, then the software routines for the fault detection, isolation, and reconfiguration functions are documented, followed by the hardware interface and error logging routines. Appendix A is an index of the FDIR routines listing where each routine is discussed in this report. Appendix B is chart of the package dependencies.

THE HARDWARE ARCHITECTURE

The physical architecture of the FTP hardware is not as significant to the implementation of the software as is the software's view of the hardware, or the "virtual hardware". In the FDIR software, the triplex redundancy is transparent to each channel — each channel views itself as running alone. All communication between channels is handled via an "exchange register". Each of the three channels has one such exchange register (see figure 1). Each channel views the exchange register as being his alone, not as connected to the other channels through the voting hardware. The exchange register is viewed by the channel as a magic box, into which is put information, and out of which comes magically corrected information. Behind the exchange register is a large network of communication lines and voting hardware, as shown in figure 2. The network is redundant to provide fault tolerance, so that channel-to-channel communication can continue even in the event of a communication line failure. The communication network also provides partial error isolation. Recognizable patterns present in the receivers are used to help isolate the origin of hardware faults.



CP	Computational Processor
DPM	Dual-Port Memory
EX	Data Exchange
IOP	Input/Output Processor
IOS	Input/Output Sequencer
MM	Mass Memory

Figure 1. Hardware configuration of a single channel.

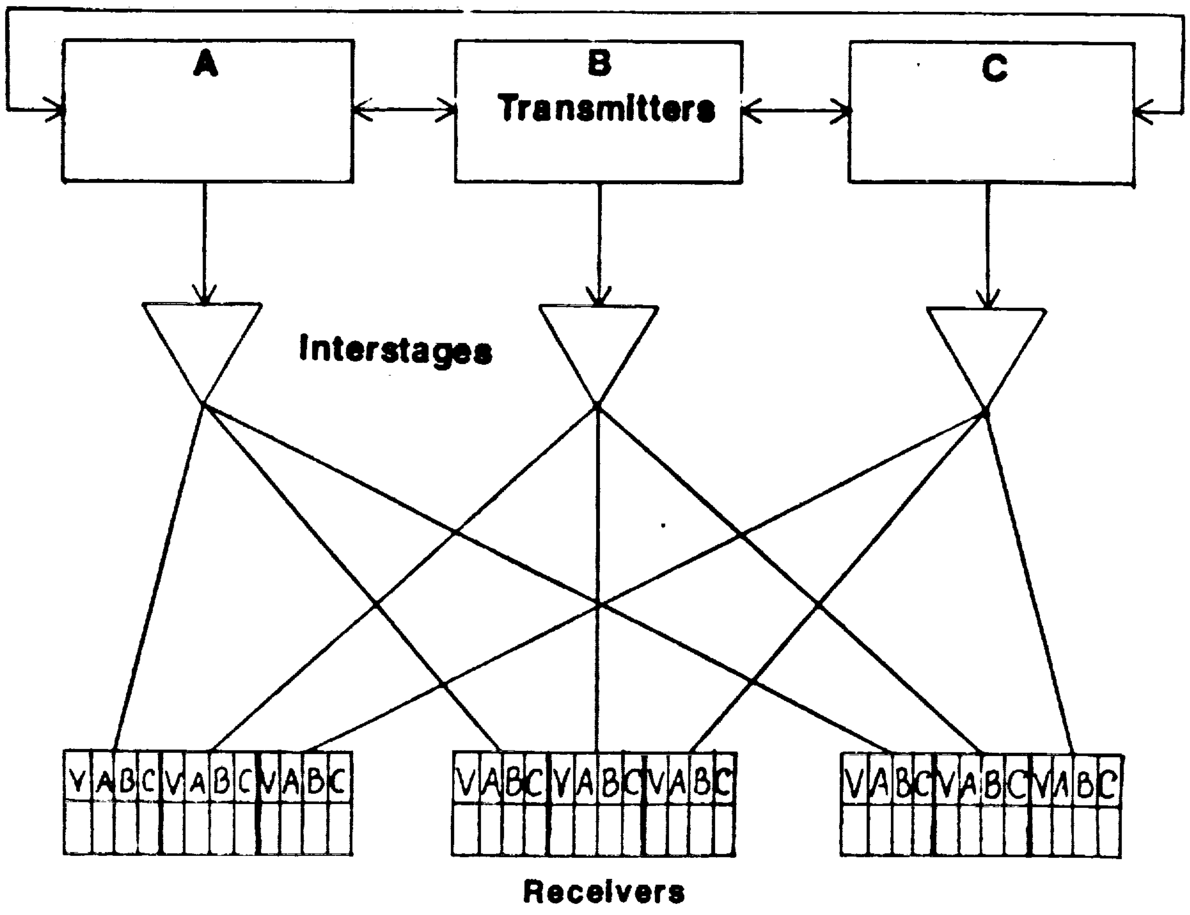


Figure 2. Interprocessor communication.

FAULT DETECTION

The routines responsible for detecting faults are divided into four packages: TEST, ERROR_LATCH, CLOCK_LATCH, and FDIR. Package TEST is the only fault detection package not constantly active seeking faults. TEST (as of now) is designed to be called during system start-up and disabled afterwards. Packages ERROR_LATCH and CLOCK_LATCH manipulate the hardware error flags, informing the FDIR software of errors detected by hardware. Package FDIR is the main fault detection package, driving the software error detection routines and handling all errors indicated by hardware latches.

Package TEST

OVERVIEW: This package contains the functions to correct RAM and check RAM and ROM. Calls are made to the FDIR_ERROR_LOG package to record errors. There are two functions for testing the voting hardware, but they have been commented out.

The test part of the FTP has four functions. It tests the CPU, I/O handler and voter, RAM and ROM for errors. All the functions run in the lowest priority task rate group, whenever the CPU has free time. The test functions are designed to catch latent faults before they build up and produce multiple coincident bad answers.

1 PROCEDURE:

1) MISCOMPARE insures that the voters, critical to finding these errors, are operating correctly. The function is given good data to send to the voters to make sure none of the bits are stuck. This procedure is commented out. It is not used.

   STEP BY STEP:
   1) The channels transmit the integer they were given to each other. They vote, and the voted value is saved.
   2) Each channel's transmitted value is compared to the voted value. If they agree then error flag is set to false and the procedure terminates.
   3) If a channel does not have the voted value, but all the channels have their channel values, then the error flag is set to false and the procedure terminates. The channel value is an integer flag marking that something succeeded. No indication is given as to how the value got there or what it means. It is not explained how the channel values might get into the channels.
   4) If a channel fails the first two tests but has the value called "connect" then the error flag is set to false and the procedure terminates. It is not explained how "connect" might get into the channels.

5) If the channels fail the first three tests then it is assumed there must be a fault in one of the channels.
   a) The value channel A transmitted is compared against the voted value. If they are unequal then A is the bad channel. The channel_id is set to A, the error flag is set to true and the procedure terminates.
   b) The same thing is repeated for channel B.
   c) Channel C is assumed to be the bad channel if A and B are good. The channel_id is set to C, the error flag is set to true and the procedure terminates.

1 VISIBLE FUNCTION:

1) SYSTEM_TEST, which is the driver, returns a reconfiguration flag.

6 LOCAL FUNCTIONS:

1) RAM_PATTERN makes sure memory is working, that each bit can hold either a zero or a one. It is given a start and end address that defines a block of memory to be checked. Two different patterns are written to each word in memory and tested. The original contents of each memory word are restored after the test.

   STEP BY STEP:
   1) Starts on the first even word boundary.
   2) Memory Utilities Package.TEST_LOCATION is called twice to write two different patterns to each word. The memory location is then read to insure the pattern was stored correctly. The previous contents of each word is restored after the test. TEST_LOCATION disables interrupts from the time the memory word is saved until it is restored.
   3) If the patterns can be written in memory then the test is good, else the test flag is set to bad.
   4) The three channels then tell each other how their tests went. If a test was bad and the channel is active then:
      a) The reconfigure flag is set to remove that channel.
      B) RECORD_FDIR_ERRORS.REPORT_ERROR is called to make the log entry. The time, channel and location are recorded.
      c) The function terminates, returning a reconfiguration flag to remove that channel.
   5) If the channels test good then the next memory word is tested. Steps two through five are repeated until the last word in the block is checked.
   6) A good configuration flag is returned if all the memory locations checked are good.

2) BAD_TEST is a simple function that is given a channel's ID and a system ID and builds a reconfiguration variable. The reconfiguration variable is returned.

3) GET_SUM is passed the start and ending addresses so it can sum a block of memory locations. It returns the sum total of the memory it checked. The function can be interrupted.

4) ROM_SUM tests the ROM by adding up all the ones in the memory segment

it is given and voting on the result. Since all three processors are doing this, one ROM is simply compared against the others. One fault will be found but two faults could slip by unnoticed, e.g. a one and a zero switched. All channels not in the configuration are kept from voting.

STEP BY STEP:
1) GET_SUM is called to sum up the rows in memory from the starting address to the ending address GET_SUM is given. The result is saved in local_sum.
2) The channels then vote on the value in local_sum.
3) The voted sum is recorded and put in a variable voted_sum.
4) Each channel, A then B then C, transmits its value of local_sum and compares the transmitted local_sums against the voted_sum.
5) If a channel's local_sum does not equal the voted sum and it is still active then bad_test is called.
   a) Bad_test is a local procedure that sets the reconfigure flag to remove a particular channel.
   b) RECORD_FDIR_ERRORS.REPORT_ERROR is called to log the time and channel ID that failed.
   c) The reconfiguration flag is returned and the function terminates.
6) If all the ROMs checked out okay, then a good reconfiguration flag is returned.

The last two functions have been commented out:

5) RAM_SCRUB does what ROM_SUM does plus RAM_SCRUB removes bad channels. It makes sure the different channels have the same sum of 1's in memory. It does not make sure they are in the same place though, so a double failure might slip by.

STEP BY STEP:
1) It is given a start and ending memory location. For each word in memory:
2) The word is read and put in local_result.
3) The value of local_result is voted on and saved in voted_value.
4) MISCOMPARE is called to use local_result and voted_value to test the channels.
5) If MISCOMPARE signals a problem then:
   a) voted_value is written into the memory location and local_value.
   b) MISCOMPARE is called with voted_value.
   c) If MISCOMPARE signals bad again then DECLARE_HARD is called with the bad channels ID. A reconfigure is returned by RAM_SCRUB and the function terminates.
6) If MISCOMPARE signals okay then the next memory word is checked.
7) If all the memory locations are good then a good test flag is returned.

6) VOTER_LATCH is a single line function that simply returns a flag meaning all is well. This function, and its call from SYSTEM_TEST have been commented out.

PACKAGE TEST STEP BY STEP: One function call does it all.

SYSTEM_TEST is the driver that calls all the other functions. It returns a reconfiguration flag when an error is found. The SYSTEM_TEST functions run in the lowest priority task rate group, whenever there is free CPU time.

1) SYSTEM_TEST calls RAM_PATTERN to check the RAM on the CPU processor card. Any error will send an error message to the system by setting the reconfigure flag. The function terminates.

2) RAM_PATTERN is called again to check the RAM on the memory card. If this goes well then RAM is working, else the reconfigure flag is set for that channel. The function terminates.

3) ROM_SUM is then called by SYSTEM_TEST. ROM_SUM calls GET_SUM to sum up all the rows in that channel's ROM memory for comparison. If an error is found a reconfigure flag, with that channel's ID is returned. The function terminates.

If all the tests were okay then a flag showing all the tests went well is returned.

Package ERROR_LATCH

OVERVIEW: This package contains the functions and procedures for checking, clearing and masking of the voting error latches. It also has the latest_latch variable showing the current state of the latches. The latches are memory mapped hardware locations that indicate detected errors. The latch array holding the state flags for the error latches is initialized to no_errors.

2 Procedures:

1) ADD_LATCH maintains an integer array of the latest six latch variables. The oldest set is overwritten when the seventh or more record is added.

2) CLEAR_LATCH clears the error flags of the A, B, C and V latches for all channels. The values of the error flags are lost, and not saved.

5 Functions:

1) ANY_ERRORS is given three integers, A, B, and C, representing the state of the three channel's latches. If there are any errors, then true is returned, else ANY_ERRORS returns false.

2) LATCHED_ERRORS is given three integers representing the three channel's latches. If a channel's error bit is set, then LATCHED_ERRORS returns a reconfigure flag to delete the bad channel. A channel is deleted if, and only if, all the active channels agree the channel is bad. This includes the bad channel knowing itself is bad. Only one channel can be deleted at a time. There is no visible error routine to catch a channel that thinks itself is bad, when the

other channels think the channel is good. It is left to the voting hardware to find that error. If only one channel is left, and thinks itself is bad, then it returns a reconfigure flag so it can be deleted.

STEP BY STEP:
1) CONFIGURATION.COUNT is called to determine the number of active channels.
2) All the channels not in the configuration are kept from voting by a case statement. Only the errors of the channels in the configuration are used.
3) If all the active channels agree a particular channel's error flag is set, then the reconfigure command is set to remove that channel and the function terminates. If only some of the channels agree a channel is bad, then nothing happens.

3) MAKE_MASK uses the configuration variable to create the mask that keeps bad channels from voting. It returns the mask as an integer.

4) READ_LATCH returns an integer indicating the errors on the XA, XB, XC, and XV data exchange registers.

5) READ_CLEAR calls READ_LATCH to obtain the current error states of data exchange latches A,B,C and V. It then clears the latches and returns an integer. The integer holds the previous error states of the latches.

Package CLOCK_LATCH

OVERVIEW: This package manipulates bit-mapped flags to indicate the systems' CPU and IOP clocks statuses. The procedures/functions utilize the voting hardware to detect errors and to insure that all channels agree on the state of the latches. The package also maintains mask values to eliminate erroneous input from channels.

2 PROCEDURES:

1) ADD_LATCH maintains a virtual stack of the latest six sets of bit mapped clock latch flags. If the stack is full, the oldest set of latch flags is discarded.

2) RECORD_CHANGES checks the bit flags representing the current latches for errors for each channel's CPU and IOP. If an error is indicated, RECORD_FDIR_ERRORS.REPORT_ERROR is called, passed the error type and channel ID. The procedure calls MEMORY_UTIL.MASK to manipulate the masks, and REPORT_ERROR to set system error flags.

1 FUNCTION:

1) READ_LATCH determines the present state of the clock latch flags and asks each active channel to vote on their value. This voted value becomes the current value of all active channels' latch flags. READ_LATCH then calls ADD_LATCH to record the current latch values. Each channels' latch flags are checked to determine if they have

changed since the function was last called. If so, RECORD_CHANGES is called (ACTUALLY IT IS ALWAYS CALLED) to record these new clock latches. The current state of the latches is then incorporated into the mask bit flags for each channel.

The package itself initializes the latches at the start to values that force the checking of each channel's latch values.

Package FDIR

OVERVIEW: Package FDIR contains the driver procedure for most of the Ada FDIR software (FAST), and 3 support procedures for the driver. These support procedures handle 1) the removal of a channel from the configuration, 2) initializing procedure FAST, and 3) running the system self-test. FDIR also contains the task FAST_FDIR which initializes the Ada FDIR system, and calls procedure FAST.

4 PROCEDURES:

1) REMOVE_CHANNEL records and requests a system reconfiguration. It also calls LOST_SOUL if the removal results in only one channel being left in the configuration. The last line in the procedure schedules the LOST_SOUL task.

   STEP BY STEP:
   a) Call RECORD_RECONFIGS.REPORT_RECONFIG to record the required reconfiguration.
   b) Call CONFIG.RECONFIGURE to reconfigure the system.
   c) Call CSDL_RSP.SHARED_WRITE_RECONFIG_CMD to record the reconfiguration in a shared data space for use later in this package.
   d) Call CONFIG.GET_CONFIG to update the current configuration.
   e) If only one channel is now running, call SYNC.LOST_SOUL.
   f) If the channel currently executing this code is the channel just removed from the configuration, call SYNC.LOST_SOUL. NOTE: This removes the channel from execution until it is in synchronization, allowing it to run without being in the voting configuration.
   g) Call CSDL_RSP.TSCHED to schedule the LOST_SOUL task for execution.

2) FAST is the driver procedure for most of the FDIR software. Due to it's complexity, it is presented here in pseudo-code. A flow chart diagram is contained in Appendix C.

```
STEP BY STEP:
1) If running in simplex mode, RETURN.
2) Call CONFIG.GET_CONFIG to get current system configuration.
3) If 2 channels are configured, and LOST_SOUL is NOT running;
   a) Call CSDL_RSP.TSCHED to schedule the LOST_SOUL task.
4) Call SYNC_UTLS.PRESENT and CONFIG.INT2CONFIG to get a configura-
   tion variable representing the channels currently synchronized.
5) If less than 2 channels are synchronized,
  a) Call CONFIG.NEW_PRESENCE to see if the configuration matches the
     channels in synchronization.
  b) If a channel should be in synchronization but isn't, call
```

RECORD_ERRORS.REPORT_ERROR to report the error, and REMOVE_CHANNEL to remove the channel.

c) Call CONFIG.GET_CONFIG to get the current configuration, call CONFIG.NEW_PRESENCE to get current channels in synchronization, and compare the two.

d) If a channel is not in synchronization, but is in the configuration, call RECORD_ERRORS.REPORT_ERROR to report the error, and REMOVE_CHANNEL to remove the channel.

e) Call SYNC.LOST_SOUL.

6) ELSE IF 2 or more channels are synchronized and the configuration is not equal to the channels synchronized,

a) Call CONFIG.NEW_PRESENCE to see if the third channel is synchronized and not in the configuration, or if channels are configured and not synchronized.

b) If a channel is configured but not synchronized, call RECORD_ERRORS.REPORT_ERROR to report the error, and REMOVE_CHANNEL to remove the channel.

c) RETURN.

7) Call ERROR_LATCH.CLEAR_LATCH to read the current state of the error latches, and clear the old copy.

8) Call ERROR_LATCH.MAKE_MASK to construct a mask preventing voting by channels not in the configuration.

9) Call EXCHANGE.TRANSMIT and MEMORY_UTILS.MASK to allow all configured channels to vote on the current state of the error latches.

10)Call ERROR_LATCH.LATCHED_ERRORS to determine if the hardware has discovered an error,if so (returned true-disable),

a) Call RECORD_ERRORS.REPORT_ERROR to report the error, and REMOVE_CHANNEL to remove the channel.

b) RETURN.

11)Call CLOCK_LATCH.READ_LATCH to allow all channels in the current configuration to vote on the status of the clock latches and record that value.

12)If the self test(which is not called) detected a channel that is in the current configuration but is faulty;

a) Set flags to indicate that selftest found no errors, call REMOVE_CHANNEL to remove the channel from the configuration, and RETURN.

13)Call CSDL_RSP.SHARED_READ_RECONFIG_CMD to read the last reconfiguration performed (for CP/IOP as appropriate).

14)If the last reconfiguration indicated that a channel was removed from the configuration,

a) Call CSDL_RSP.SHARED_WRITE_RECONFIG_CMD to set the flags in the last reconfiguration record to indicate no reconfigurations were performed.

b) Call RECORD_ERRORS.REPORT_ERROR to report the error, and REMOVE_CHANNEL to remove the last channel reconfigured again. NOTE: This effectively disables recently synchronized channels from voting.

c) RETURN.

15) END PROCEDURE FAST.

3) INIT initializes "last channel removed" variables to none(false-disable) for the CP/IOP, sets selftest_reconfig variable to

none(false-disable), and calls CSDL_RSP.TSCHED to schedule task FAST_FDIR.

4) SYSTEM_TEST_LOOP calls TEST.SYSTEM_TEST to check memory and system hardware. Sets variable selftest_reconfig to the reconfiguration indicated by TEST.SYSTEM_TEST.

TASK FAST_FDIR:
1) Call CSDL_RSP.INIT_RSP to initialize the Ada runtime support package.
2) Call CSDL_RSP.BLOCK to prevent FAST_FDIR from being called during it's own execution.
3) Call procedure FAST.
4) On any exceptions, call SYSTEM.REPORT_ERROR.

PACKAGE STEP BY STEP (Compilation initializations)
1) Call CSDL_RSP.INIT_SHARED_DATA to initialize memory mapped locations for "last channel removed" variables for both CP/IOP.
2) Set lost_soul_on flag to false.

ISOLATION

Fault isolation is handled by packages CONFIG and MEMORY_UTILITIES. Package MEMORY_UTILITIES actually provides support routines for packages FDIR and CLOCK_LATCH. MEMORY_UTILITIES provides calling procedures with the bit-manipulation ability necessary for the use of bit-mapped error latches. CONFIG isolates which channel is faulty, and disables the faulty channel from voting by removing it from a "configuration variable".

Package CONFIG

OVERVIEW: This package contains the procedures/functions necessary to maintain the current system configuration, to disable/enable a channel, and to reconfigure the system. The package also checks for channels going off or coming on line, and can reconfigure the system as necessary.

1 PROCEDURE:

1) RECONFIGURE checks the form of reconfiguration required (init, enable, disable) and acts accordingly. If init, the package config variable is modified so that it matches the reconfiguration record's config variable. If the reconfig command is ENABLE, the procedure checks which channel(s) to enable and sets the package config variable to true for that channel(s). If DISABLE is specified, the package config flag is set to false for the specified channel(s). RECONFIGURE then calls function CONFIG2INT to update the configuration variable.

8 FUNCTIONS:

1) COUNT returns the number of channels presently operational.

2) INT2CONFIG returns a boolean flag array representing the present configuration, derived from an integer passed in.

3) GET_CONFIG returns the present configuration, represented as a boolean flag set.

4) INT_CONFIG_ADDR returns the memory address of the variable INT_CONFIG, an integer representing the present configuration.

5) SIMPLEX_ADDR returns the memory address of the boolean variable SIMPLEX.

6) THIS_CHAN returns a value of the type CHANNEL_ID representing the channel currently in control c the shared channel communication lines. It calls a nested function, INTEGER_TO_CHANNELID which converts an integer representation of the channel to one of the type CHANNEL_ID.

7) NEW_PRESENCE, when passed the current configuration and the expected configuration, returns reconfiguration flags. "false disable channel A" indicates that the current configuration is proper. If current = expected, flags are set to indicate "false disable channel A". Else function COUNT is called to determine if more channels are running

than are expected. If so, flags are set to "true enable channel A". Else set flags to "true disable channel A". Procedure then checks configuration to determine which channel(s) is(are) not as expected, and sets the channel flag(s) accordingly.

8) CONFIG2INT accepts flags representing the present configuration of the system and returns an integer representing that configuration. NOTE: 7=all 3=a,b 5=a,c 6=b,c.

PACKAGE STEP BY STEP:

1) The package first sets the configuration flags to indicate all channels off-line.
2) The integer representation of the configuration is set to indicate all channels off-line.
3) The processor_type flag is set to CP or IOP accordingly.

Package MEMORY_UTILITIES

OVERVIEW: This package contains 68000 Assembly code to handle memory manipulations that Ada does not support.

5 FUNCTIONS:

1) MASK returns the logical AND of the bit patterns of its two arguments.

2) COMBINE returns the logical OR of the bit patterns of its two arguments.

3) CHANGE returns the logical XOR of the bit patterns of its two arguments.

4) INVERSE returns the logical NOT of the bit pattern of its argument.

5) TEST_LOCATION returns the integer result of the memory pattern test for the given location and pattern.

RECONFIGURATION

The routines for performing reconfiguration of the system, including the reintroduction of a processor after repair or disappearance of a transient fault are in packages TRANSIENT, SYNC_UTILITIES, and SYNC.

Package TRANSIENT

OVERVIEW: Transient handles the storage and manipulation of the channel hard/transient failure flags. It also keeps an unreliability index on all three channels, and a single mean-time-to-repair (mttr) indicator for any hard failures. The unreliability index is a numerical value, which is increased whenever a failure occurs, and decremented at regular intervals (10 times a second) during regular system operation.

A channel failure is assumed to be transient until it increases the channel's unreliability index past a set threshold (4 failures in one week). As soon as a channel is pushed past the threshold, it is declared a hard failure.

The following set of seven bottom level procedures/functions handle the manipulation of failure error flags without making any decisions as to when or how to remove/reinstate a failed channel.

4 PROCEDURES:

1) REMOVE_TICK subtracts one from each of the four unreliability indexes and from 'mttr' if they are >0.

2) ADD_FAILURE is called whenever a channel has suffered a failure. ADD_FAILURE adds a set number to the unreliability index of the indicated channel, and 'mttr' is set to a standard mean time to repair value.

3) DECLARE_HARD is called when a channel is discovered to be a hard failure. A local flag ('hard') is set to true for the specified channel. The variable 'mttr' is set to a standard mean-time-to-repair value.

4) ALL_TRANSIENT resets all local hard failure flags to false.

3 FUNCTIONS:

1) GET_HARD_FAILURES returns a list of which channels are considered up, and which channels are considered hard failures.

2) OVER_THRESHOLD compares the indicated channel's count variable to a set tolerable value. If the count in question is over the tolerable level, the function returns a true value.

3) TIME_TO_REPAIR checks to see if the mean time to repair has elapsed. If it has, the variable 'mttr' is reset to a set value (mean_time) and the function returns a true value. The true value indicates that the failed channel should have been repaired by this time.

Package SYNC_UTILITIES

OVERVIEW: The Sync_utilities package contains three procedures written in Motorola 68000 assembly code for:

1) Synchronizing the FTP channels,
2) Testing which FTPs are present.
3) Aligning a specified segment of memory to agree with the memory of the other channels,

3 PROCEDURES:

1) SYNC_CHANS Synchronizes the channels through a process of increasing/ decreasing delays between memory writes. This stops when a simultaneous write to a dedicated virtual receiver register is detected or when the number of tries the procedure is given is exhausted. The lengths of the delays are relatively prime numbers.

   STEP BY STEP:
   1) Halts all interrupts.
   2) Determines its channel i.d.
   3) Checks each dedicated virtual receiver register (DVRR) against the proper channel code(passed in) to determine a channels synchronized presence.
   4) If all three channels did not show present and there are some tries left, the procedure waits and then loops back to step 3.
   5) If all channels show present then a flag, an integer, is returned to show this. The interrupts are enabled and the procedure stops.
   6) If the number of tries ran out before all the channels synchronized then an integer flag is returned showing which channels made it and which did not. The interrupts are enabled and the procedure stops.

2) PRESENT returns a flag saying which channels are present, working.

   STEP BY STEP:
   1) Inhibits interrupts.
   2) Locks the shared bus.
   3) Writes A's unique pattern to the memory mapped hardware data exchange registers.
   4) Checks for the voted presence of A by comparing the receiver register with what is there when A is present.
   5) The results of the compare are used to update a configuration variable showing current configuration.
   6) Steps 3 - 5 are repeated, with new patterns, to test B and C.
   7) A flag, an integer, is returned showing the present configuration, the shared bus is unlocked and interrupts are enabled.

3) ALIGN_MEM is given the start and end address in RAM memory. It will correct memory inconsistencies one word at a time. A majority vote is taken on what a word should contain. The voted word is then copied back into the memory locations used for the majority vote. The process is then done for the next word in memory.

NOTE: Voting is done in hardware. When an error is found it is corrected by majority vote.

STEP BY STEP:
1) ALIGN_MEM is given a start address and an ending address. For each individual memory location:
   a) One word (16 bits) is read from memory and transferred to the data exchange register. The hardware does a majority vote on the word and sends it to each channels' receiver.
   b) The voted value of the word is then read from the receiver register and written back into the same location.
   c) Steps a and b, are repeated for every word of memory between the start and end addresses.

Package SYNC

OVERVIEW: SYNC contains several internal support procedures: ADD_CHANNEL, LOCK and UNLOCK_SHARED_BUS, ALIGN_INTERVAL_TIMERS, and three main procedures: ALIGN_SH_MEM, INITIAL, and LOST_SOUL. ALIGN_SH_MEM assures that all four sections in the shared memory bank contain the same information. LOST_SOUL re-synchronizes a channel that has fallen out of synchronization. INITIAL sets up the FDIR system for operation.

7 PROCEDURES and 1 TASK:

1) ADD_CHANNEL initializes a reconfigure variable to "OK" for the indicated channel. It then calls RECORD_RECONFIGS.REPORT_RECONFIG to log the system reconfiguration. Finally, CONFIG.RECONFIGURE is called to enable the indicated channel and to update the configuration flags.

2) LOCK_SHARED_BUS writes an integer flag directly to a memory location reserved for the SHARED_BUS flag.

3) UNLOCK_SHARED_BUS Clears a memory location reserved for the SHARED_BUS flag.

4) ALIGN_SH_MEM appears to align four sections of a single bank of memory. Calls LOCK_SHARED_BUS, calls SYNC_UTILITIES.ALIGN_MEM 8 times to align 4 sections of a bank of memory, calls UNLOCK_SHARED_BUS.

5) ALIGN_INTERVAL_TIMERS forces the values of each channel's interval timer to be congruent with the other channels. It is called after LOST_SOUL has been picked up in the LOST_SOUL procedure. NOTE: This implementation only aligns timer 1 of (0,1,2), and assumes a 2 byte binary count in mode 1. The timers will appear stopped from the time this procedure reads them until they are rewritten.

6) INITIAL synchronizes the CPs\IOPs, and initializes the reconfiguration flags and records.

```
STEP BY STEP:
Tests to see if CP or IOP.
If IOP
   Wait for CP to be ready
   If SIMPLEX,
      Set configuration flags to indicate which channel is up.
      Set RECONFIG variable to indicate initialization.
      Call RECORD_RECONFIGS.REPORT_RECONFIG to report initialization.
      Call CONFIG.RECONFIGURE to reconfigure the system.
      Call EXCHANGE.SET_SIMPLEX_EXCHANGE to preclude hardware voting.
      Call CONFIG.CONFIG2INT for integer representation of
         configuration.
      RETURN to calling procedure.
   Else if IOP and not SIMPLEX,
      Call LOCK_SHARED_BUS.
      Call SYNC_UTILITIES.SYNC_CHANS twice to try to get all channels
         synchronized.
      Set current_config to indicate all channels on line.
      Set RECONFIG variable to indicate initialization.
      Call RECORD_RECONFIGS.REPORT_RECONFIG to log initial
         reconfiguration.
      Call CONFIG.RECONFIGURE to reconfigure the system.
      Call CONFIG.CONFIG2INT for integer representation of
         configuration.
      Call UNLOCK_SHARED_BUS.
      Wait for CP to finish.
      If more than one channel up,
        Call SYNC_UTILITIES.ALIGN_MEM to align data exchange area.
        Call ERROR_LATCH.CLEAR_LATCH.
      Else call LOST_SOUL.
END IOP CLAUSE.
If process is a CP then,  (*NOTE: IOP WILL BE INITIALIZED FIRST.*)
   Wait for IOP to be ready.
   Signal IOP that CP is also ready.
   Wait for IOPs to become synchronized.
   If SIMPLEX,
      Set configuration flags to represent channel configured.
      Set RECONFIG variable to indicate initialization.
      Call RECORD_RECONFIGS.REPORT_RECONFIG to report initialization.
      Call CONFIG.RECONFIGURE to reconfigure the system.
      Call CSDL_RSP.RESET_SYSTEM_TIME.
      Call EXCHANGE.SET_SIMPLEX_EXCHANGE to preclude hardware voting.
      Call CONFIG.CONFIG2INT for integer representation of
         configuration.
      RETURN to calling procedure.
   Else if not SIMPLEX,
      Call LOCK_SHARED_BUS.
      Call SYNC_UTILITIES.SYNC_CHANS twice to attempt channel
         synchronization.
      Set configuration flags to represent current configuration.
      Initialize reconfiguration variable to indicate initialization.
      Call RECORD_RECONFIGS.REPORT_RECONFIG to record initial
         reconfiguration.
      Call CONFIG.RECONFIGURE to reconfigure system.
```

```
        Call CONFIG.CONFIG2INT for integer representation of
           configuration.
        Call UNLOCK_SHARED_BUS.
        If more than one channel up,
          Call SYNC_UTILITIES.ALIGN_MEM to align data exchange area.
          Call CSDL_RSP.RESET_SYSTEM_TIME.
          Call ALIGN_SH_MEM.
          Call ERROR_LATCH.CLEAR_LATCH.
        Else call LOST_SOUL.
   END PROCEDURE.
```

7) LOST_SOUL attempts to synchronize a stray channel with those currently running.

```
   STEP BY STEP:
   Calls CONFIG.GET_CONFIG to get current configuration.
   Repeats
      Call LOCK_SHARED_BUS.
      Call SYNC_UTILITIES.SYNC_CHANS to try to get the channels to
          synchronize.
      Call UNLOCK_SHARED_BUS.
   Until the number of channels up is greater than one.
   Sets flags to indicate the number of CPs/IOPs up.

   NOTE: THE REST OF THE PROCEDURE IS COMMENTED OUT.

   --If a "lost soul" is found,
   --      Call CONFIG.NEW_PRESENCE.
   --   Call CONFIG.RECONFIGURE.
   --   If the lost soul found was a CP,
   --      Call RESET_CLOCK and incorporate value into time state.
   --      Call CSDL_RSP.SET_SYSTEM_TIME.
   --      Call ALIGN_SH_MEM.
   --   End if CP.
   --   Call SYNC_UTILITIES.ALIGN_MEM to align data exchange area.
   --   Call ALIGN_INTERVAL_TIMERS.
   --   Call ERROR_LATCH.CLEAR_LATCH.
   --   Call I_MACHINE.RESTART.
   --End if.
```

1 TASK:

1) LOST_SOUL_SYNC

```
   STEP BY STEP:
   Call CSDL_RSP.INIT_RSP to perform initializations necessary for
      the csdl_rsp run-time support package procedures.
   Loop,
      Call CSDL_RSP.BLOCK to prevent task from being called during
         it's own execution.
      Call LOST_SOUL to attempt to synchronize the stray channel(s).
   End loop
   Exception, ( If LOST_SOUL fails to synchronize any channels.)
      When others=> call SYSTEM.REPORT_ERROR.
   End Exception, and task.
```

HARDWARE INTERFACE

The hardware interface package EXCHANGE allows processor pairs to operate independently of each other while still allowing communication between them. This is accomplished by giving the independent processor pairs access to hardware that transparently provides the necessary communication between the operating pairs.

Package EXCHANGE

OVERVIEW: This package facilitates the exchange of information between channels. The two functions provide for the exchange of information at both the transmitter-to-transmitter level (shared-data channels), and the interstage-to-receiver level (dedicated-data channels). The procedure allows for the voting hardware to be bypassed.

2 FUNCTIONS:

1) TRANSMIT accepts data from a channel in the form of an integer. This information is passed to the virtual receivers of all three channels. The function returns the hardware-voted value to the calling procedure.

2) RAW_TRANSMIT accepts data in the form of a long_integer. This data is passed to the other channels at the virtual transmitter level. The return value is the hardware-voted value of the long_integer.

1 PROCEDURE:

1) SET_SIMPLEX_EXCHANGE switches the registers used in the above functions to preclude hardware-voting.

   NOTE: There is no procedure to reset the virtual registers that SET_SIMPLEX_EXCHANGE switches. Therefore, there is no way of reactivating the voting hardware once it is disabled, short of rebooting the system.

ERROR LOGGING ROUTINES

The routines for logging errors are in packages RECONFIG_LOG, RECORD_FDIR_ERRORS, FDIR_ERROR_LOG, LATEST_FDIR_ERRORS, EXCEPTION_LOG, AND RECORD_RECONFIGS. These routines were written for debugging purposes, but they also provide a useful maintenance log of system performance.

Package RECONFIG_LOG

OVERVIEW: The purpose of this package is to maintain a reconfiguration log. RECONFIG.LOG contains only one procedure, ENTER. This procedure is identical to one in package FDIR_ERROR_LOG with the same name. The reason for the procedure duplication is so that two distinct logs, one for fault errors and one for reconfiguration data may co-exist.

1 PROCEDURE:

1) ENTER inserts a "log_entry" value into a "log_array" structure. The "log_array" is represented by a circular array of "log_entrys". The value is inserted in the next available empty position in the log array. Should none exist, the value is inserted in place of the oldest "log_array" entry.

Package RECORD_FDIR_ERRORS

OVERVIEW: The entire package is actually a dummy procedure header. When the only procedure, "REPORT_ERROR" is called, it passes control directly to package FDIR_ERROR_LOG procedure ENTER. This package provides isolation from the error log in package FDIR_ERROR_LOG by providing a different set of access privileges than those provided by package FDIR_ERROR_LOG. RECORD_FDIR_ERRORS allows no retrieval of log information, only the reporting of error conditions.

1 PROCEDURE:

1) REPORT_ERROR calls package FDIR_ERROR_LOG procedure ENTER. It does nothing else.

Package RECORD_RECONFIGS

OVERVIEW: The purpose of this package is to maintain a reconfiguration log. RECORD_RECONFIGS contains only one procedure, REPORT_RECONFIGS. This procedure calls RECONFIG_LOG.ENTER. The procedure RECONFIG_LOG.ENTER is identical to one in package FDIR_ERROR_LOG with the same name. The reason for the procedure duplication is so that two distinct logs, one for fault errors and one for reconfiguration data may co-exist. The purpose of this package is to limit access to the reconfiguration log created by RECONFIG_LOG to the entry of log entries only.

1 PROCEDURE:

1) REPORT_RECONFIG calls RECONFIG_LOG.ENTER, passing it an ITEM as defined in RECONFIG_TYPES, record LOG_ENTRY.

NOTE: ENTER inserts a "log_entry" value into a "log_array" structure. The "log_array" is represented by a circular array of "log_entries". The value is inserted in the next available empty position in the log array. Should none exist, the value is inserted in place of the oldest "log_array" entry.

Package FDIR_ERROR_LOG

OVERVIEW: This package contains one procedure and one function. Procedure ENTER updates the error log. Function GETLOG produces and returns a copy of the error log. These functions perform two error-log maintenance functions for the FDIR error log. They serve to isolate the error log from other packages and provide a data abstraction level allowing errors to be reported and logged from other packages regardless of error-log structure.

1 PROCEDURE:

1) ENTER inserts a log_entry value into the "log_array", a circular array. The value is inserted at the next empty position, or if the log_array is full, the new value overwrites the oldest existing entry.

1 FUNCTION:

1) GETLOG copies the "internal_log" into a "log" record, records the number of entries, and returns the log record with the oldest log entry in the first position.

Package LATEST_FDIR_ERRORS

OVERVIEW: This package serves to limit access privileges to the current copy of the FDIR error log contained in FDIR_ERROR_LOG. LATEST_FDIR_ERRORS allows read access but not write access to the error log.

1 FUNCTION:

1) GET_LOG calls FDIR_ERROR_LOG.GETLOG which returns a copy of the current FDIR error log.

Package EXCEPTION LOG

OVERVIEW: This package contains two procedures. Procedure ENTER updates the error log. These procedures perform two error-log maintenance functions for the exception log. Procedure GETLOG produces and returns a copy of the error log. They serve to isolate the exception log from other packages and provide a data abstraction level allowing exceptions to be reported and logged from other packages regardless of exception log structure.

2 PROCEDURES:

1) ENTER inserts a log_entry value into the "log_array", a circular array. The value is inserted at the next empty position, or if the log_array is full, the new value overwrites the oldest existing entry.

2) GETLOG copies the "internal_log" into a "log" record, records the number of entries, and returns the log record with the oldest log entry in the first position.

CONCLUDING REMARKS

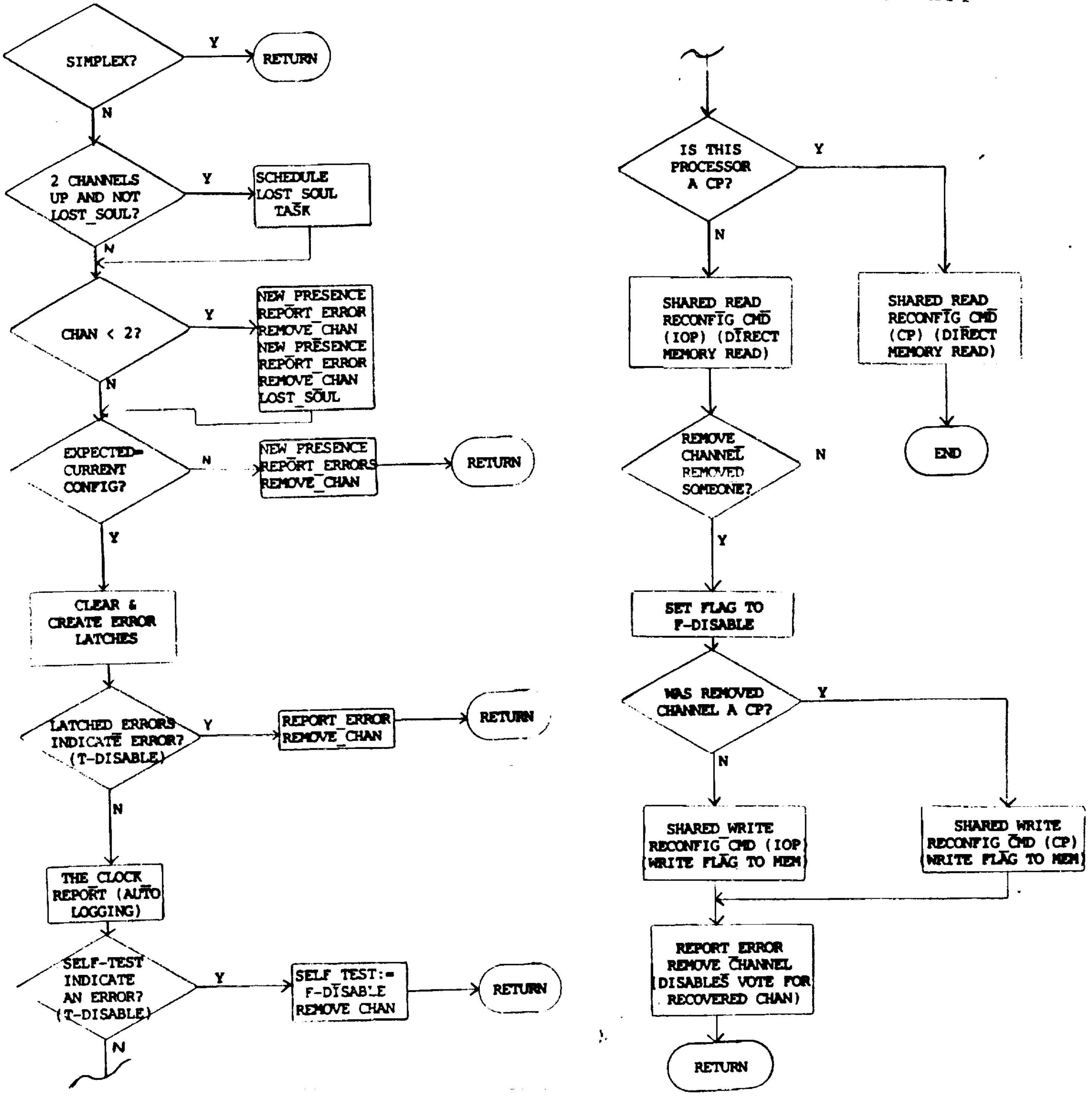
The Ada fault detection, isolation, and reconfiguration (FDIR) software of the AIPS Fault-Tolerant Processor that was current as of December 1986 was documented. Since the system is still under development, this report is intended only to aid in understanding of the FDIR functions and to aid in identifying changes or additions needed to the system. This report is not intended to take the place of final documentation of the FDIR functions.

APPENDIX A. INDEX OF PROCEDURES AND FUNCTIONS

Name	Package	Type	Page
ADD_CHANNEL	SYNC	Procedure	17
ADD_FAILURE	TRANSIENT	Procedure	15
ADD_LATCH	ERROR_LATCH	Procedure	8
ADD_LATCH	CLOCK_LATCH	Procedure	9
ALIGN_INTERVAL_TIMER	SYNC	Procedure	17
ALIGN_MEM	SYNC_UTILITIES	Procedure	16
ALIGN_SH_MEM	SYNC	Procedure	17
ALL_TRANSIENT	TRANSIENT	Procedure	15
ANY_ERRORS	ERROR_LATCH	Procedure	8
BAD_TEST	TEST	Function	6
CHANGE	MEMORY_UTILITIES	Function	14
CLEAR_LATCH	ERROR_LATCH	Procedure	8
COMBINE	MEMORY_UTILITIES	Function	14
CONFIG2INIT	CONFIG	Function	14
COUNT	CONFIG	Function	13
DECLARE_HARD	TRANSIENT	Procedure	15
ENTER	EXCEPTION_LOG	Procedure	22
ENTER	FDIR_ERROR_LOG	Procedure	22
ENTER	RECONFIG_LOG	Procedure	21
FAST	FDIR	Procedure	10
FAST_FDIR	FDIR	Task	12
GETLOG	EXCEPTION_LOG	Procedure	23
GETLOG	FDIR_ERROR_LOG	Procedure	22
GET_CONFIG	CONFIG	Function	13
GET_HARD_FAILURES	TRANSIENT	Function	15
GET_LOG	LATEST_FDIR_ERRORS	Function	22
GET_SUM	TEST	Function	6
INIT	FDIR	Procedure	11
INITIAL	SYNC	Procedure	17
INIT_CONFIG_ADDR	CONFIG	Function	13
INIT2CONFIG	CONFIG	Function	13
INVERSE	MEMORY_UTILITIES	Function	14
LATCHED_ERRORS	ERROR_LATCH	Function	8
LOCK_SHARED_BUS	SYNC	Procedure	17
LOST_SOUL	SYNC	Procedure	19
LOST_SOUL_SYNC	SYNC	Task	19
MAKE_MASK	ERROR_LATCH	Procedure	9
MASK	MEMORY_UTILITIES	Function	14
MISCOMPARE	TEST	Procedure	5
NEW_PRESENCE	CONFIG	Function	13
OVER_THRESHOLD	TRANSIENT	Function	15

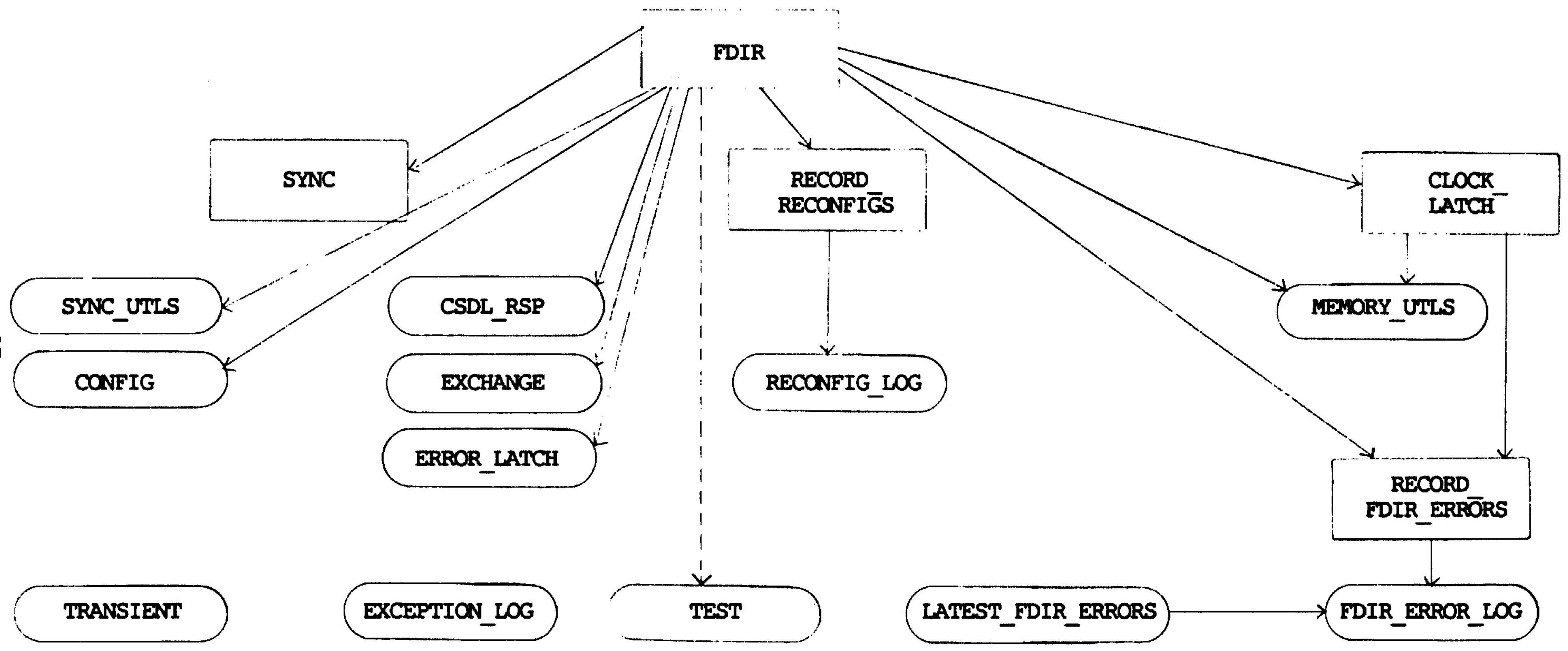
APPENDIX C. FLOWCHART OF PROCEDURE FAST

ORIGINAL PAGE IS OF POOR QUALITY



SIMPLEX?
Y
RETURN
N
2 CHANNELS UP AND NOT LOST_SOUL?
Y
SCHEDULE LOST_SOUL TASK
N
CHAN < 2?
Y
NEW_PRESENCE
REPORT_ERROR
REMOVE_CHAN
NEW_PRESENCE
REPORT_ERROR
REMOVE_CHAN
LOST_SOUL
N
EXPECTED= CURRENT CONFIG?
N
NEW_PRESENCE
REPORT_ERRORS
REMOVE_CHAN
RETURN
Y
CLEAR & CREATE ERROR LATCHES
LATCHED ERRORS INDICATE ERROR? (T-DISABLE)
Y
REPORT_ERROR
REMOVE_CHAN
RETURN
N
THE CLOCK REPORT (AUTO LOGGING)
SELF-TEST INDICATE AN ERROR? (T-DISABLE)
Y
SELF_TEST:= F-DISABLE
REMOVE CHAN
RETURN
N
IS THIS PROCESSOR A CP?
Y
N
SHARED_READ RECONFIG_CMD (IOP) (DIRECT MEMORY READ)
SHARED_READ RECONFIG_CMD (CP) (DIRECT MEMORY READ)
END
REMOVE CHANNEL REMOVED SOMEONE?
N
Y
SET FLAG TO F-DISABLE
WAS REMOVED CHANNEL A CP?
Y
N
SHARED_WRITE RECONFIG_CMD (IOP) WRITE FLAG TO MEM
SHARED_WRITE RECONFIG_CMD (CP) WRITE FLAG TO MEM
REPORT ERROR
REMOVE CHANNEL
(DISABLES VOTE FOR RECOVERED CHAN)
RETURN

APPENDIX B. FDIR PACKAGE DEPENDENCIES



Standard Bibliographic Page

1. Report No. NASA TM-89154	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle Documentation of the Current Fault Detection, Isolation, and Reconfiguration Software of the AIPS Fault-Tolerant Processor		5. Report Date August 1987
		6. Performing Organization Code
7. Author(s) David T. Lanning, Allen W. Shepard, and Sally C. Johnson		8. Performing Organization Report No.
9. Performing Organization Name and Address NASA Langley Research Center Hampton, VA 23665-5225		10. Work Unit No. 505-66-21-01
		11. Contract or Grant No.
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Washington, DC 20546		13. Type of Report and Period Covered Technical Memorandum
		14. Sponsoring Agency Code

15. Supplementary Notes

David T. Lanning and Allen W. Shepard, Old Dominion University, Norfolk, Virginia.
Sally C. Johnson, Langley Research Center, Hampton, Virginia.

16. Abstract

This report contains documentation of the December 1986 version of the ADA code for the fault detection, isolation, and reconfiguration (FDIR) functions of the Advanced Information Processing System (AIPS) Fault-Tolerant Processor (FTP). Because the FTP is still under development and the software is constantly undergoing changes, this should not be considered final documentation of the FDIR software of the FTP.

17. Key Words (Suggested by Authors(s)) Fault tolerance Fault tolerant processor Advanced Information Processing System	18. Distribution Statement [redacted] until August 31, 1989 Subject Category 61		
19. Security Classif.(of this report) Unclassified	20. Security Classif.(of this page) Unclassified	21. No. of Pages 30	22. Price

[redacted]