

THIS DOCUMENT IS:

CONTROLLED BY: BOEING DEFENSE & SPACE GROUP,
SUBSYSTEM PROCESSING, 9-5584
ALL REVISIONS TO THIS DOCUMENT SHALL BE APPROVED
BY THE ABOVE ORGANIZATION PRIOR TO RELEASE

PREPARED UNDER **CONTRACT NO.** NAS9-18878, Task 3.2.2
 IR&D
 OTHER

PREPARED ON: MAC II/MS WORD, VER. 4.0 **FILED UNDER:**

DOCUMENT NO. **MODEL:**

TITLE: **FAULT TOLERANT TESTBED EVALUATION**
 - PHASE I FINAL REPORT

ORIGINAL RELEASE DATE 9/30/93

ISSUE NO. **TO** **DATE**

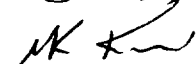
- THE INFORMATION CONTAINED HEREIN IS NOT PROPRIETARY.**
- THE INFORMATION CONTAINED HEREIN IS PROPRIETARY TO THE BOEING COMPANY AND SHALL NOT BE REPRODUCED OR DISCLOSED IN WHOLE OR IN PART OR USED FOR ANY DESIGN OR MANUFACTURE EXCEPT WHEN SUCH USER POSSESSES DIRECT, WRITTEN, AUTHORIZATION FROM THE BOEING COMPANY.**

**ANY ADDITIONAL LIMITATIONS IMPOSED ON THIS DOCUMENT
WILL BE FOUND ON A SEPARATE LIMITATIONS PAGE.**

PREPARED BY: V. Caluori Jr.
 T. Newberry



CHECKED BY: M. Kosai



SUPERVISED BY: M. Raftery
 I. White



APPROVED BY: G. Lee



SIGNATURE ORGN DATE

(NASA-CR-188275) **FAULT TOLERANT
TESTBED EVALUATION, PHASE I Final
Report (Boeing Defense and Space
Group)** 26 p

N94-29035

Unclas

ABSTRACT

In recent years, avionics systems development costs have become the driving factor in the development of space systems, military aircraft, and commercial aircraft.

A method of reducing avionics development costs is to utilize state-of-the-art software application generator (autocode) tools and methods. The recent maturity of application generator technology has the potential to dramatically reduce development costs by eliminating software development steps that have historically introduced errors and the need for re-work.

Application generator tools have been demonstrated to be an effective method for autocoding non-redundant, relatively low-rate input/output (I/O) applications on the Space Station Freedom (SSF) program; however, they have not been demonstrated for fault tolerant, high-rate I/O, flight critical environments. This contract will evaluate the use of application generators in these harsh environments.

Using Boeing's quad-redundant avionics system controller as the target system, Space Shuttle Guidance, Navigation, and Control (GN&C) software will be autocoded, tested, and evaluated in the Johnson [Space Center] Avionics Engineering Laboratory (JAEL). The response of the autocoded system will be shown to match the response of the existing Shuttle General Purpose Computers (GPCs), thereby demonstrating the viability of using autocode techniques in the development of future avionics systems.

KEY WORDS

Autocode

Controller

Fault Tolerance

Flight Critical

Guidance

Integration

Navigation

Redundancy

Software

Synchronization

Technology

Validation

Verification

Voting

TABLE OF CONTENTS

1.	INTRODUCTION	5
1.1	Objective	5
1.2	Multi-year Approach.....	5
1.3	Phase 1 Overview.....	6
2.	PHASE 1 TEST REPORT.....	8
2.1	Standalone Development Process.....	8
2.2	Digital Autopilot (DAP) on Fault Tolerant System.....	9
2.3	Closed-Loop Preparation / 1553 Integration.....	16
3.	1994 PLANS	18

APPENDICES

APPENDIX A	Equipment Descriptions
APPENDIX B	DAP Standalone Simulation—Integration Notes
APPENDIX C	Standalone Testing Memory Map
APPENDIX D	Autocoded Main Program with Scheduler
APPENDIX E	SA_Uilities Ada Module
APPENDIX F	SVM Mission Tables

LIST OF FIGURES

1.2-1	Multi-Year Approach	7
1.3-1	Development Process Summary	8
2.1-1	Standalone Development Process.....	9
2.2-1	Standalone Testing Environment.....	11
2.2-2	Initialization Frame Timing	12
2.2-3	Ascent Frame Timing.....	13
2.2-3	Lab Testing Schedule.....	15
2.3-1	Closed-Loop Simulation Environment.....	17
2.3-2	1553B Communications Test Environment.....	18
A1	ASC Prototype and Interfaces.....	20
A2	Fault Injection System Configuration	22

DEFINITIONS

- EG NASA JSC internal code for the Engineering Directorate's Navigation Control and Aeronautics Division (NCAD)
- EG NASA JSC internal code for the Engineering Directorate's Flight Data Systems Division (FDSD)

ACRONYMS, ABBREVIATIONS, AND SYMBOLS

ASC	Avionics System Controller
CAD	Computer Aided Design
Comm.	Communication
COTS	Commercial Off-the-Shelf
DAP	Digital Autopilot
ETDM	Embedded Target Debug Monitor
FSSR	Functional System Software Requirements
HOL	High-Order Language
I/O	Input/Output
ISI	Integrated Systems Incorporated
JAEL	JSC Avionics Evaluation Laboratory
JSC	Johnson Space Center
MB	Megabyte(s)
NASA	National Aeronautics and Space Administration
RSEL	Real-time System Engineering Laboratory
SVM	Sync and Voter Module
SOW	Statement of Work
VHM	Vehicle Health Monitoring

1. INTRODUCTION

This report documents the work performed on NASA research contract NAS9-18878, Task 3.2.2. Under this task Boeing delivered (on a loan-in basis) and demonstrated a state-of-the-art quad redundant controller compliant with open systems interface standards and commercial development tools. As part of this task the NASA/Boeing team demonstrated the compatibility of the fault tolerant system with automatic "spec-to-code" generated software. In recent years industry has matured and made available computer aided design (CAD) tools that enable control algorithm specification pictorially, using block diagrams. From this block diagram representation, the tools can automatically generate High Order Language (HOL) software. The goal on this project was to execute autocoded Ada software in Boeing's fault tolerant target system with minimal modifications to the autocoded software. In areas where modifications to the autocode were required recommendations were given and are documented here.

This work represents one piece of an overall development process that features a high degree of automation and enables rapid prototype and development of entire avionics systems.

1.1 Objective

The multi-year objective is to develop, test and demonstrate a highly automated process suitable for future avionics systems development. This process will span from requirements definition and systems design tasks to detailed hardware/software design, integration, and verification tasks. Successful development of this process (and the infrastructure of tools needed to implement the process) will result in substantial reductions in development and test costs as well as compression of project schedules. Reduction estimates approach 50 percent for cost and schedule compression.

1.2 Multi-year Approach

The multi-year task flow is shown in figure 1.2-1. A graduated build-up of capability and maturity in the new process is achieved that culminates in flight demonstration testing and

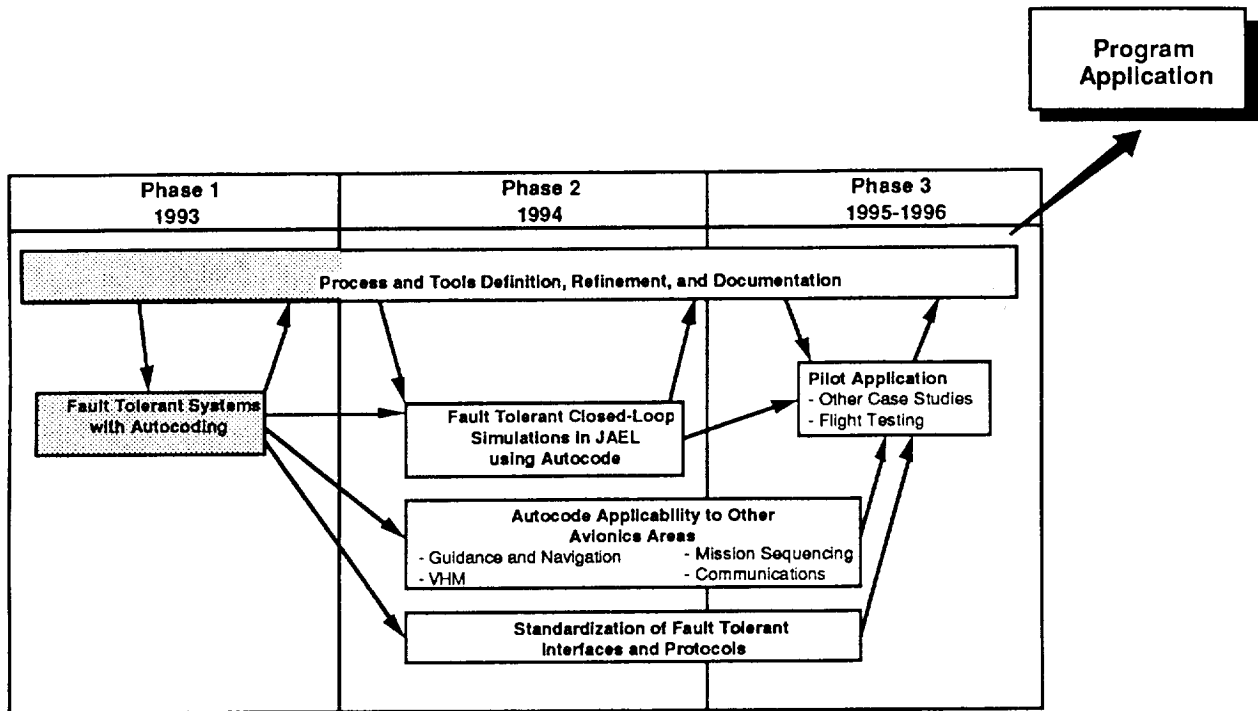


Figure 1.2-1 Multi-Year Approach

eventual full-scale application. Phase 1 completion is shown as shaded in the figure. In phase 2 the autocoding process in conjunction with the standardized fault tolerant system will be subjected to more rigorous testing in the Johnson Avionics Evaluation Lab (JAEL). Closed-Loop simulations will be developed and tested using Space Shuttle ascent flight control as the first application. Also in phase 2, other avionics areas will be investigated for development automation including: Vehicle Health Monitoring (VHM), Guidance, Navigation, and Communications. In phase 3, the rapid prototyping process and tools, along with the standard fault tolerant system, will be finalized and verified. The process will be applied to actual flight testing demonstrations and areas other than Shuttle flight will be explored to ensure the universal applicability of the new rapid prototyping process.

1.3 Phase 1 Overview

Figure 1.3-1 shows a simplified view of the development process required to achieve hardware-in-the-loop simulations. As shown in the figure, the phase 1 efforts focused on the shaded portion of the process. The significant aspect of the process depicted in figure 1.3-1 is that the product from the algorithm development stage is not a detailed software development specification (or functional system software requirements (FSSR) document). Instead, the product from that stage is automatically generated HOL software that, with minor modifications, can execute on a flight processor target system.

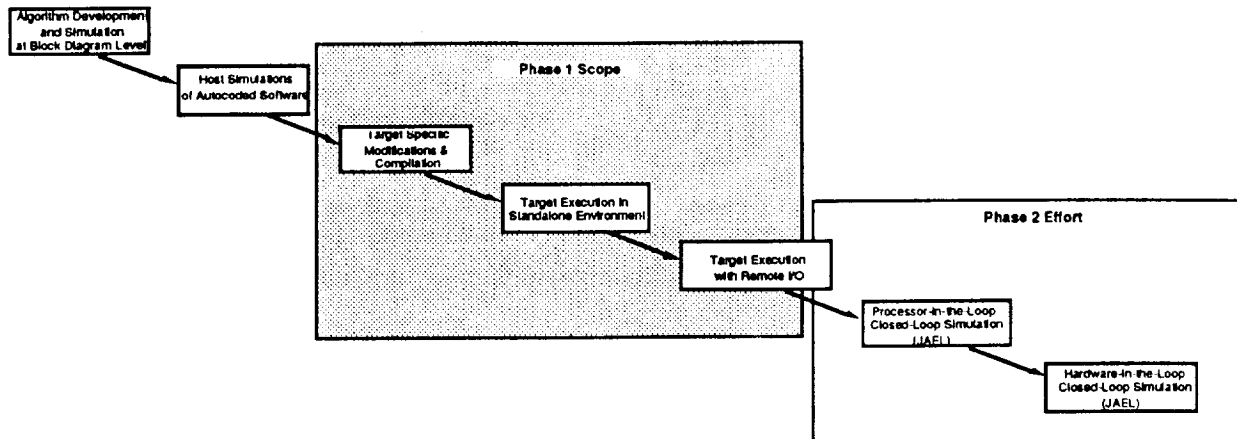


Figure 1.3-1 Development Process Summary

The goal of the phase 1 effort was to execute autocoded Ada software in Boeing's fault tolerant voting system with minimal hand modifications to the autocoded software. Areas where hand modifications were needed were identified and recommendations to address these areas given.

For this effort the Space Shuttle Digital Autopilot (DAP) control algorithm was used as a representative application. The NASA EG group had previously specified the DAP using autocode techniques and tools supplied by Integrated Systems Incorporated (ISI). NASA EK and Boeing personnel then transformed the autocoded software to a version compatible with the Boeing fault tolerant avionics system controller (ASC). This manual translation involved minor modifications and a working version was complete in less than one week. The translated code was then ported to the Boeing system and executed in a standalone, real-time simulation. All inputs and outputs to the DAP software were voted using the Boeing Voter Module and stored in ASC local memory. This allowed for post-simulation analysis and verification. It was shown that the ASC was capable of executing the DAP in a multi-channel synchronous configuration. It was also demonstrated that the ASC fault tolerant characteristics did not impinge on the autocoding process. All manual translation steps that were made would be required for execution on any real-time target.

In preparation for future closed-loop simulations in the JAEL, the NASA / Boeing team integrated the ASC with a simulated avionics subsystem. The subsystem was simulated by a personal computer and linked to the ASC as a remote terminal via a MIL-STD-1553B bus. DAP inputs and outputs were passed over the bus and voted at the boundary of the ASC. This is a key stepping stone to the phase 2 closed-loop simulation work.

2. PHASE 1 TEST REPORT

This section of the report documents phase 1 progress made in the JSC Real-time System Engineering Laboratory (RSEL). For this effort Boeing placed its fault tolerant avionics system controller (ASC) in the RSEL and the NASA/Boeing team worked jointly over a six week period on the tasks described here. Section 2.1 describes the detailed process necessary to generate, load, and test autocoded software on the Boeing target. Section 2.2 describes progress made on standalone execution of the Shuttle DAP on the target. Finally, section 2.3 describes progress made in preparation for closed-loop simulation in the JAEL.

2.1 Standalone Development Process

A detailed view of the process required to generate autocoded software and execute it on the Boeing target is shown in figure 2.1-1 and described here.

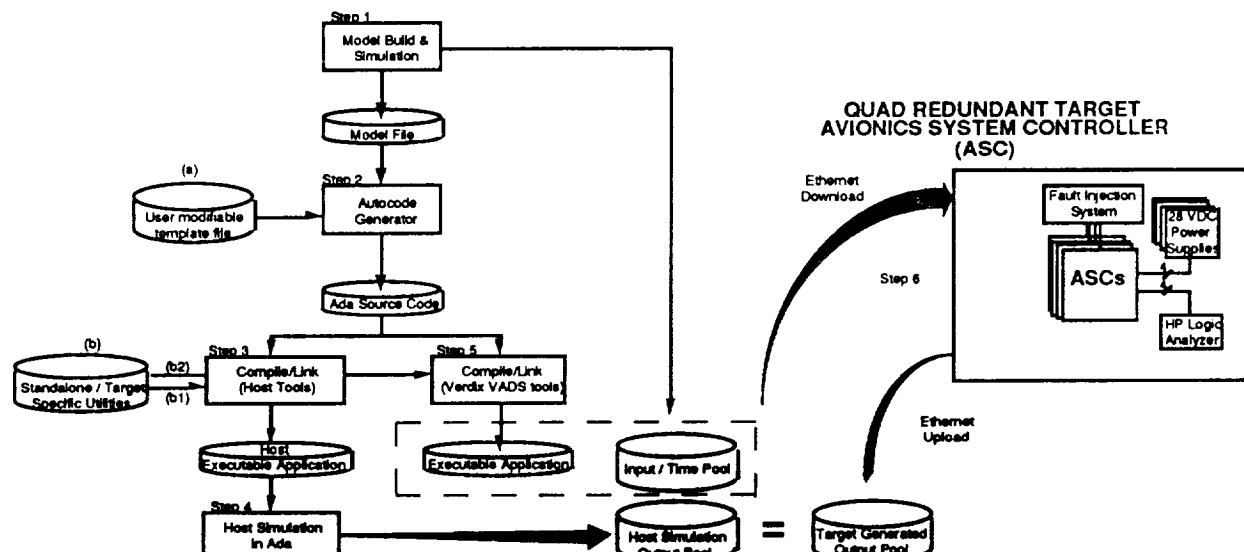


Figure 2.1-1 Standalone Development Process

The first step uses ISI's MatrixX build and simulation tools to develop a model file. The model file is an ASCII file that describes the pictorial representation of the control algorithm being developed—in this case the Shuttle DAP. Generation of the model file was performed by NASA personnel.

In the second step of figure 2.1-1 the model file and the ISI provided template file are input to the autocode generator to produce an Ada source code file automatically. The template file contains key elements such as the ISI provided task scheduler. This is a simple rate-monotonic scheduler capable of scheduling and dispatching multi-rate periodic tasks as well as triggered, or asynchronous, tasks. Appendix D contains a full listing of the Ada source code generated from combining the DAP model and the template file elements, including the scheduler¹.

The Ada source file is then combined with a host version of the ISI provided utilities file (b1 in figure 2.1-1) and compiled to generate an executable module (step 3). The utilities file contains

¹Note that the DAP was constructed as a single 25 Hz task. Future versions will update the DAP to run multiple rates including 12.5 and 6.25 Hz tasks.

platform specific functions including external I/O. The external I/O functions move data between the autocode internal data structures and external I/O elements. This is platform specific and usually requires one-time modifications.

In step 4 the executable module runs on the host development station to generate a host simulation output file. This output is then verified against original simulation runs from within the Matrix-X simulation tools to ensure validity. The above steps also create an input file to be used in later steps.

In the fifth step the Ada source code and a second version (b2) of the ISI provided utilities file are compiled and linked using Verdex Ada development tools. Appendix E contains a complete listing of the utilities file. The module produced can then be loaded on the Boeing target for execution.

In the sixth step the executable code and input file are loaded onto the ASC target and executed in real-time. The simulation run on the target produces an output file that is uploaded to the host system for comparison to the simulation output file.

The goal of the above process is to enable rapid development of control systems using block diagram level specifications. The template file and utilities file are intended to be modified one time only to meet user needs. Theoretically, once those files are set, the process becomes quite streamlined and enables rapid development studies as well as long-term maintenance at the block diagram level. Control system changes can be made to block diagrams on the host and tested on the target system in a matter of hours.

This project concentrated on steps 5 and 6 to study the impacts of a specific target, and its fault tolerant aspects, on the autocoding process. The goal was to understand the minimum changes required to achieve target executable code. Section 2.2 describes the work performed in this area.

2.2 Digital Autopilot (DAP) on Fault Tolerant System

Standalone Test Objectives. The specific objectives for this task were:

- Verify correct DAP autocode execution on fault tolerant target system and identify compatibility issues between autocode and fault tolerance
- Verify and measure timing aspects of DAP execution including
 - Deterministic execution across redundant channels
 - Execution time of DAP, scheduler, and I/O functions on target system
 - Verify and document memory usage on target system
- Measure and document processes and their flow-times

Standalone Test Environment. The standalone test environment, shown in figure 2.2-1, consists of a host Sun workstation and the Boeing quad redundant target controllers². Not shown in the figure are a Boeing fault injection system and a logic analyzer for gathering instrumentation data. The Sun is used for developing the Shuttle DAP software and the input data to be used for real-time execution on the target. The fault tolerant target contains four replications of 1) a 32-bit reduced instruction set computer (RISC) that consists of a R3000 CPU, R3010 floating-point accelerator, 16 MBytes of global main memory accessible over a

² Detailed descriptions of the equipment can be found in appendix A.

standard VME bus, and 128KBytes each of instruction and data cache; and 2) a Synchronization and Voter Module (SVM). The R3000 is used for executing the DAP software in real-time and the SVM is used for maintaining redundant channel synchronization and for voting all inputs and outputs to the R3000 processor.

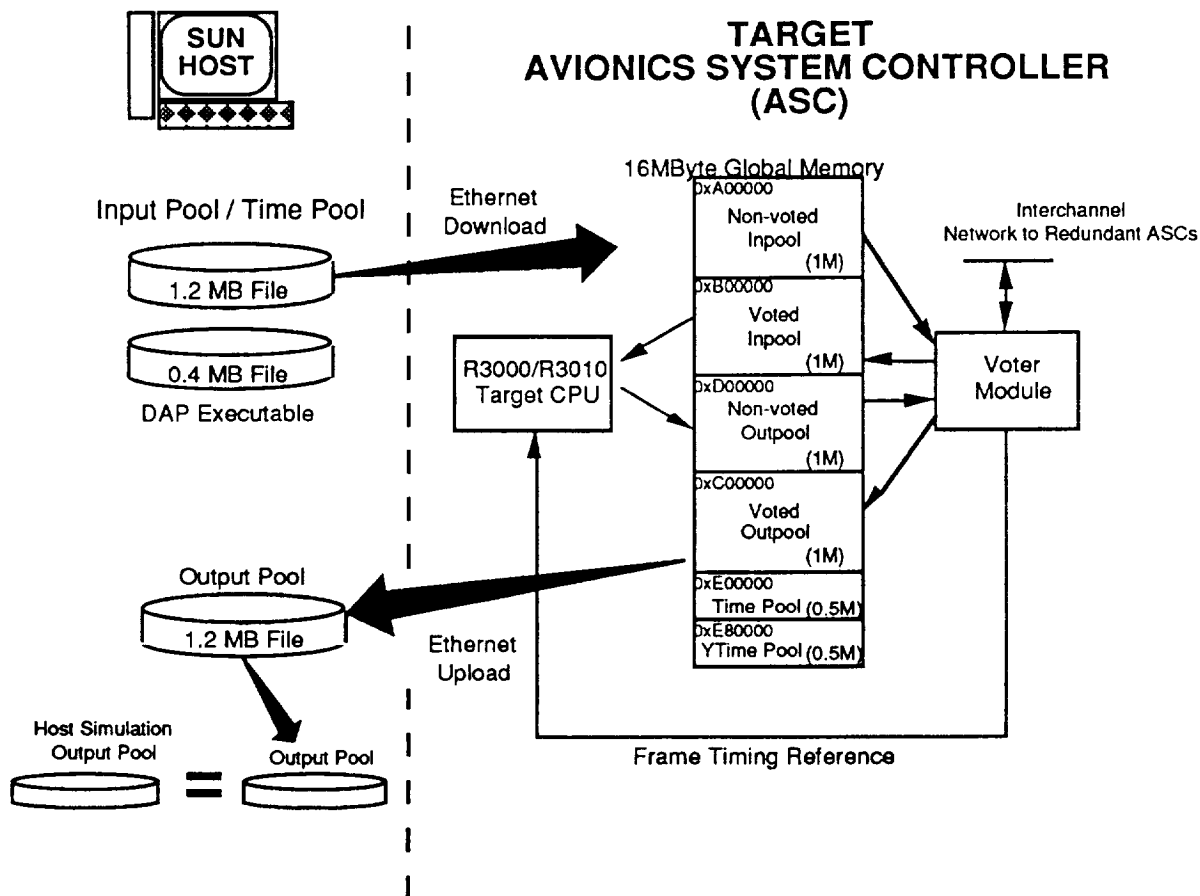


Figure 2.2-1 Standalone Testing Environment

Standalone Test Description. As shown in figure 2.2-1, in order to execute the DAP on the target an input pool, timing pool, and executable module were downloaded from the host to the target prior to execution. At run-time the pools resided in global memory and the DAP executed in real-time "standalone" mode. Standalone means that I/O data transfers for the DAP were performed, locally, within the target computer. All data transfers normally required to move data into and out of the R3000 processor were performed, including all voting functions. However, in standalone mode the end steps of moving data into and out of an actual I/O device (e.g. 1553 bus module) were replaced with transfers to and from global memory space within the target system. This is shown with the global memory map³ and the data flow arrows of figure 2.2-1. In this way, all input and output for a two and one-half minute simulation run were stored in controller memory and made available for post-run analysis and verification. After program execution, the newly generated voted output pool was uploaded to the host, then compared (bit-for-bit) with a previously generated host simulation output pool to assure valid target execution (see section 2.1-1).

³A complete memory map can be found in appendix C.

Figures 2.2-2 and 2.2-3 show timelines for real-time execution of the standalone test. As shown, the standalone test scenario consisted of two phases: an initialization phase and an ascent phase. The following paragraphs describe the timing of activity during these phases.

The initialization phase occurred as the first cycle on the SVM and prior to any cycles on the R3000. During this phase the autocode scheduler (executing on the R3000) idled, waiting for cycle, or frame, interrupts. The SVM retrieved the first frame's input data from the non-voted inpool, voted that data and stored it in the voted inpool. This enabled the DAP software to start execution of the ascent phase on the next cycle with input data in its buffers.

Following initialization, the ascent phase ran for two and one-half minutes. In order to run in real-time the SVM generated and supplied to the R3000 a minor cycle timing reference with a period of 40 ms⁴⁵. On a frame-by-frame basis the SVM retrieved DAP required input data (19 32-bit values) from the non-voted inpool, voted the data and placed it in the voted inpool. Also, the SVM retrieved DAP generated output data (19 32-bit values) from the non-voted outpool, voted the data and placed it in the voted outpool. The autocoded software executed on the R3000 and each frame it retrieved inputs from the voted inpool, scheduled task execution for the current frame, placed outputs from the previous frame of task execution in the non-voted outpool, and dispatched and executed the DAP control software as a single task. After the DAP task ran to completion the scheduler resumed execution and waited for the next frame interrupt.

After the mission completed the voted outpool was uploaded to the host (Sun) over ethernet and compared to the host simulation output pool.

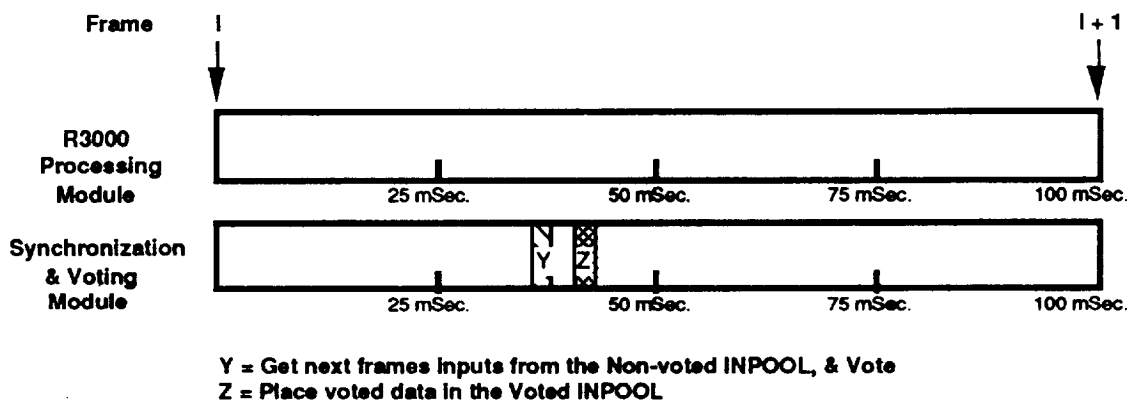


Figure 2.2-2 Initialization Frame Timing

⁴The cycle time as well as the timing of data transfers on the SVM are determined from an ASCII mission data load file (loaded into the SVM prior to execution). This file can be easily changed to accommodate different scenarios and cycle times.

⁵The SVM ensures that this reference is synchronized with the redundant channels' reference so that the DAP executes synchronously across channels.

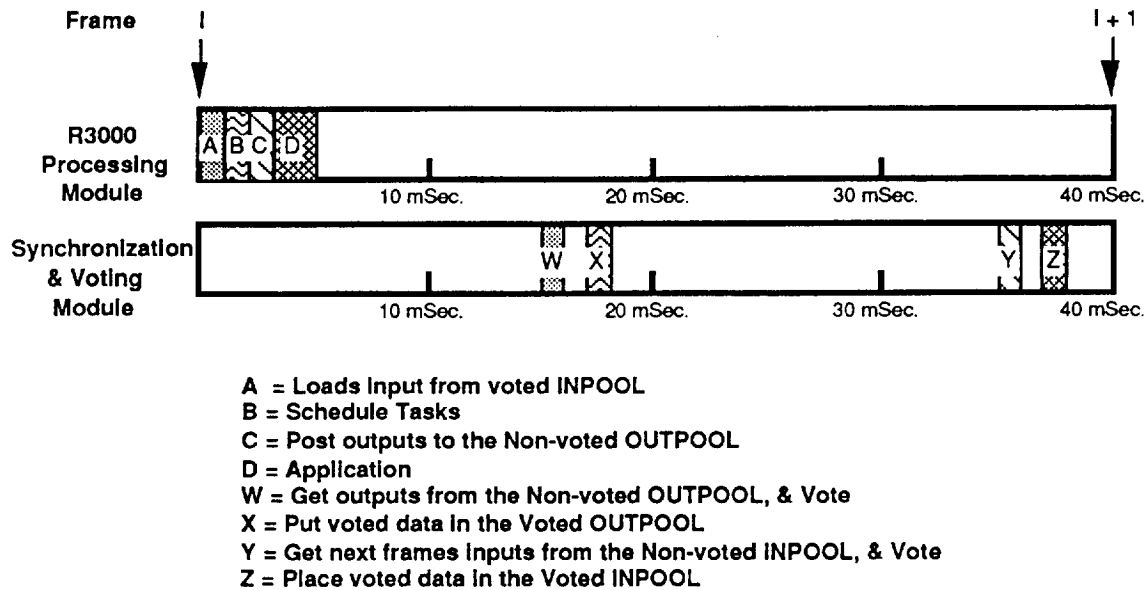


Figure 2.2-3 Ascent Frame Timing

Conclusions / Lessons Learned. In summary, the autocode derived DAP software executed on the fault tolerant system as expected and the outputs were verified as correct against a host simulation generated output file. The following paragraphs describe key conclusions/lessons learned during the course of the standalone tests.

Autocode and Fault Tolerance Compatibility. Overall, the fault tolerant target required no special modifications to the autocode process. The hand modifications made to the autocode generated software would be required to execute it on any particular target. The modifications that were made to the autcoded software fall under three headings:

1. Modifications to supply a timing reference to the task scheduler
2. Modifications to move input/output between internal data structure and global I/O buffers.
3. Modifications to enable instrumentation of software execution.

The timing reference modification was made in the main task scheduler program and took form as a polling function (see appendix D). The preferred implementation for this modification would be as a true processor interrupt to the background function in the utilities file. The vendor supplied O/S and debug tools did not easily accomodate handling of more than one backplane interrupt. Given the tight schedule of this contract polling was implemented as a temporary alternate solution. Future studies will modify the polling implementation to a true hardware interrupt implementation.

The second modification above was needed to enable the transfer of information from the application software internal data structures to the target specific I/O buffers. These modifications take form, primarily, as address mappings. These changes were very straightforward. It took less than one week to define and put in-place a simple mapping structure. Additionally, an improved mapping structure was defined and tested during the latter stages of the lab testing. The improved version was viewed as more memory efficient as well as more amenable to standardization. The simple mapping structure uses large arrays to store I/O of entire simulations while the newer version uses a simple mail box scheme that fills a

single small array in real-time. The mail box scheme arrays are only as large as one cycle's worth of data.

Again, the above modifications would be required to get the autocode to run on any target. It should be noted here that, for the Boeing target, one of the target-specific buffers was for voting internal state information (e.g. navigation state data). However, this buffer was specified just as any I/O buffer would be declared.

It was noted by the team that certain attributes of the system don't currently show up in the block diagram level specifications. These attributes include latency and jitter requirements of key data objects as well as certain redundancy management (RM) aspects. The Boeing target intentionally keeps RM transparent to the application software. Team members expressed a concern that certain parameters might want to be made available to the application such as: minimum redundancy levels, status on resource exhaustion, error strike count thresholds that determine when resources are reconfigured, and execution rate of RM logic. It was noted that the implementations may be target specific but that specification of the key parameters should possibly be made available at the block diagram level of the process depicted in figure 2.1-1. Future studies will determine how best to approach these areas.

One manual step not shown in the process flow of figure 2.1-1 is the generation of the SVM mission file that specifies cycle timing, data transfer times, and phase schedules to the SVM. This is an ASCII file that is easily changed. Nevertheless, it represents an area that may be streamlined by incorporation into the automatic development process. Boeing has previously developed a pictorial / tabular interface for creation of this file. Future studies will determine how best to incorporate this aspect into the overall development process.

Autocode for Time Critical Embedded Systems. Keys to enabling the use of autocode techniques are the emerging throughput and memory size capabilities of today's processing cards. The generation of Ada autocode from block diagrams results in larger, less efficient code. Previous processor and memory technology would not be able to store the programs as generated much less execute them in real-time. Approximate R3000 memory utilization for this experiment is shown in appendix C and key block sizes are repeated here:

- Executable file containing DAP, scheduler, external I/O	300KByte
- Vendor supplied O/S using standard Ada tasking libraries and services	105KByte
- Target PROM Start-Up	<100KByte
- Ethernet Debug Monitor Code	150KByte
- Standalone simulation inputs file	1.8MByte
- Standalone simulation outputs file	1.8MByte
- <u>Heap/Stack space</u>	<u>> 4MByte</u>
TOTAL	> 8MByte

These numbers are approximate but give a good feel for the amount of memory needed to enable a rapid prototype and development environment. Arguments can be made that the standalone I/O arrays are not true requirements for flight and that the heap space could be compressed considerably. However, the application for this experiment ($\approx 300K$) represents only a fraction of the functionality needed for actual flight. A minimum requirement might be set at 4 MBytes while a safe requirement might be set at greater than 8 MBytes.

In terms of execution time, the DAP control software executed in approximately 500 μs on the R3000/R3010 CPU. At certain mission times the execution time was observed at approximately 1 ms. This variance was expected by NASA personnel and attributed to various

events that occur during Shuttle ascent. Detailed questions about the application can be directed to NASA EG personnel.

One aspect of the autocoded software that needs future consideration is the fact that all outputs from completed periodic tasks are delayed by up to a minor frame's worth of time before being transferred from internal data structure to external I/O buffers. This introduces undesired latency into the information flow. A number of ways to solve this problem exist and are being studied which include:

- Specify a higher frequency of execution to the scheduler—this may be viable due to the reserve throughput capability on the target
- Develop a user code block that takes outputs and posts them externally immediately upon task completion

Finally, for this project, the DAP was only developed as single task running at 25 Hz. The DAP should be broken up and executed as it really executes on the Shuttle—as a series of multi-rate tasks. An issue with the autocode scheduler is that for multi-rate tasks, all I/O for all tasks are performed every minor frame. This represents a large amount of unnecessary overhead. Future studies will address this issue.

Process Improvement. Overall, the process—although still in need of refinements—looks very promising. The lab development schedule moved along at a very quick pace as shown in figure 2.2-4.

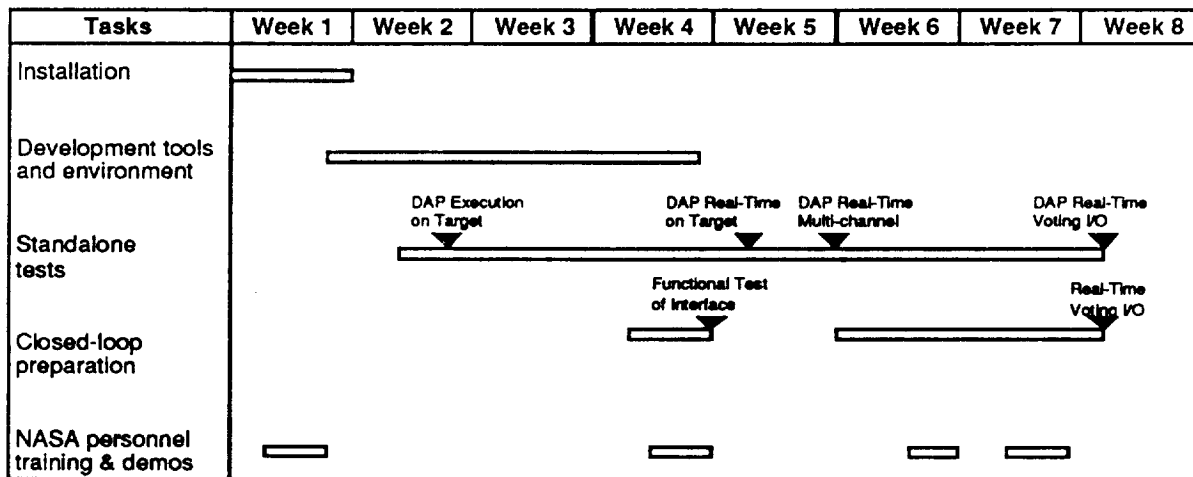


Figure 2.2-4 Lab Testing Schedule

In summary, the keys to enabling the success illustrated include:

1. The ASC is built using processors and modules with interfaces that comply to industry VME standards.
2. The ASC processor is supported by commercial software development tools.
3. The ASC fault tolerant architecture is designed to accommodate standard modules.
4. The throughput capability and memory size of the ASC target enabled the use of autocoding techniques. And,
5. The use of autocoding to generate the scheduler as well as application tasks.

Examination of the flow times in figure 2.2-4 shows the two segments of time with greatest duration were 1) the time from nominal DAP execution to DAP execution in real-time; and 2) the time from DAP real-time to DAP with voting of I/O. The largest contributor of flow time associated with (1) was the attempt at using backplane interrupts to generate timing while also using the debug monitor—which required backplane interrupts as well. The vendor supplied interrupt tables and services were not directly compatible with multiple backplane interrupts and this aspect caused 2 to 3 weeks of delay. It should be noted that once an alternate polling approach was adopted real-time execution was achieved in about one day of flow time. This area represents an implementation detail that, once addressed, should not recur. The largest contributor of flow time with (2) was also related to the debug monitor. The debug monitor links the host to the target via ethernet. When the ethernet link remains a public (versus a private) link, the target debug monitor hardware generates excessive interrupts at non-deterministic times to the R3000, thus causing large execution skews and deadline overruns. The nature of this problem is also that of a one-time occurrence. Solutions involve either of a) running without the debug tools installed for real-time tests, or b) switch the host-to-target link to a private link.

Elimination of these pathfinding type problems will result in a process that probably only requires from less than two days to a maximum of one week to move from application changes at the block diagram level to standalone fault tolerant target execution tests. This kind of schedule compression would represent a significantly new way of doing business.

2.3 Closed-Loop Preparation / 1553 Integration

In closed-loop operations, the Boeing fault tolerant controller will interface with the JAEL simulation environment over MIL-STD-1553B data buses. This is shown in figure 2.3-1.

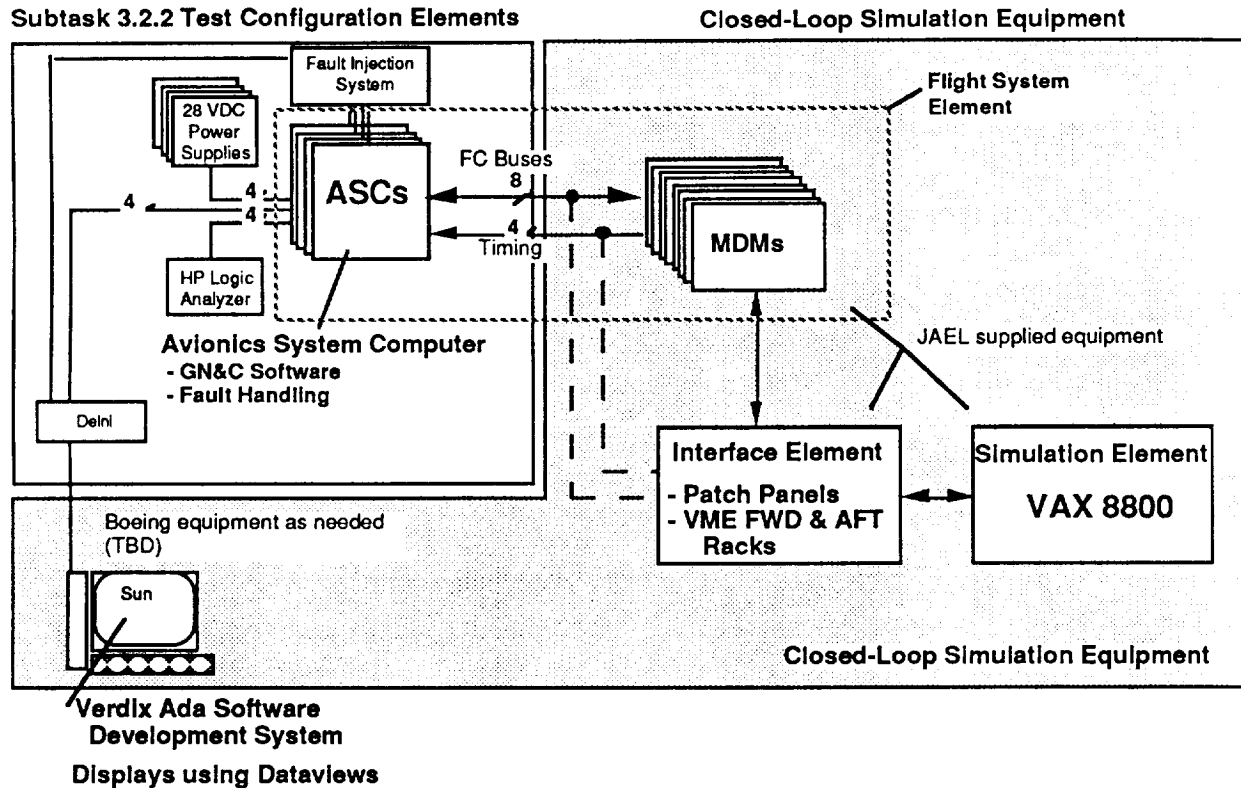


Figure 2.3-1 Closed-Loop Simulation Environment

In order to prepare for the 1553 data bus system operations in the JAEL the objectives of this task were to:

- Link the Boeing target with a single subsystem via a MIL-STD-1553B data bus
- Verify real-time communications with the subsystem using known data pools
- Execute the DAP in real-time, while receiving inputs from the remote subsystem and sending command outputs to the remote subsystem. All I/O shall be voted by the SVM in real-time.

Figure 2.3-1 shows the test environment for 1553B communications in real-time. For this test a 386 based PC was used as a remote subsystem. Before run-time, the inpool was loaded from disk to the PC memory in the remote subsystem; the outpool was loaded from the Sun into global memory space on the Boeing target. During run-time the simulation ran for two and one-half minutes using 40 millisecond minor cycles in the Boeing target. Each frame the inpool was requested from the subsystem remote terminal (RT), voted, and placed in global memory space. Additionally, the outpool was retrieved from global memory space, voted, and sent from the Boeing 1553 bus controller (BC) to the RT. At the conclusion of the test two steps were performed to verify proper operation. First, at the Sun the inpool that had been received via 1553 was uploaded and compared to a host version of the inpool. Second, at the remote subsystem (PC) the outpool that had been sent via 1553 was compared to a host version of the outpool.

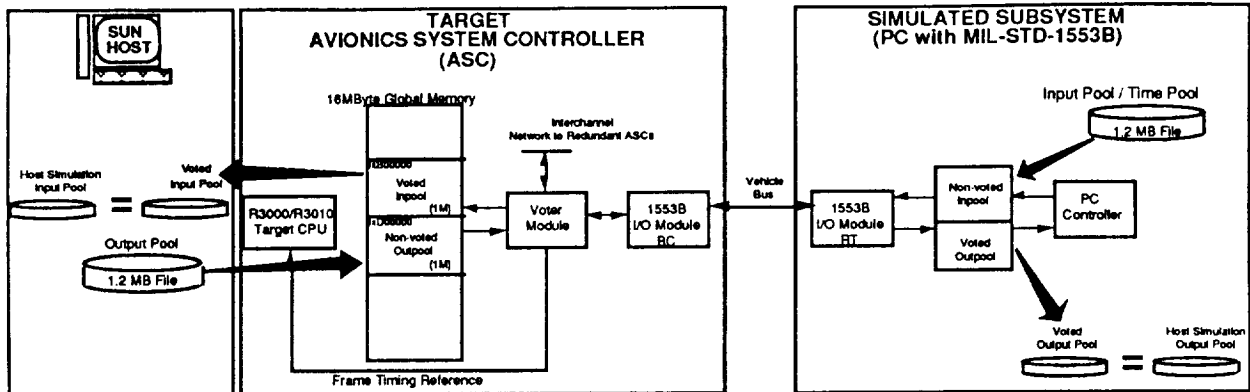


Figure 2.3-2 1553B Communications Test Environment

Results. The results from the two comparisons were that both the voted inpool at the Boeing target and the 1553B outpool at the PC were bit-for-bit identical with their respective data files. This represented a key stepping stone in preparation for closed-loop testing.

It should be noted that, for this test, the R3000 CPU remained idle. NASA personnel will continue to work the next step of the integration process—running the DAP on the R3000 while performing remote I/O with the PC based subsystem... Due to the success of the test described here, the fact that it was done in real-time, and the results of the standalone DAP testing, the team expects quick success for the final integration step of the DAP.

3. 1994 PLANS

In 1994 the NASA / Boeing team will continue to work jointly on the rapid prototyping process definition as well as the standard interface requirements that the process demands. Testing of the process will expand to include areas beyond flight control including, vehicle health monitoring (VHM), guidance, navigation, mission sequencing, and communications. Additionally, testing will move to a new level of fidelity in the JAEL closed-loop simulations.

The statement of work (SOW) for 1994 is available upon request and outlines the tasks to be performed during the next year.

APPENDIX A Equipment Descriptions

The Avionics System Computer (ASC) is a modular, redundant system that contains a 32-bit processor module (PM); a synchronization and voter module (SVM); and a 1553B data bus interface module (BIM) in each of its four redundant channels. Figure A1 shows the interfaces between a prototype computer and the simulation system and fault injection system. In all, there are four separate prototype computers in four separate boxes.

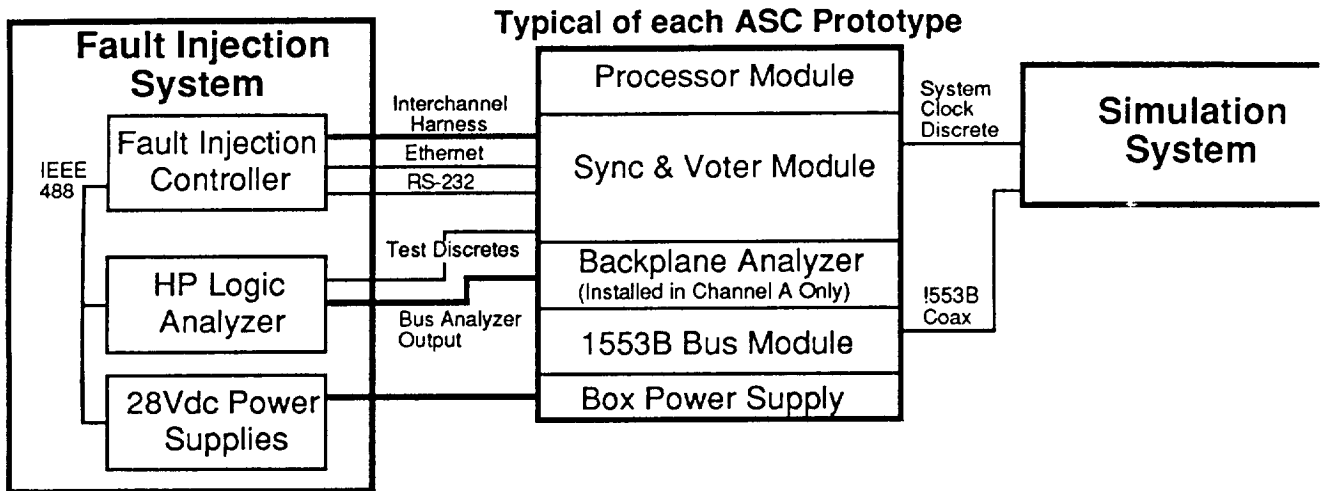


Figure A1. ASC Prototype & Interfaces.

ASC Prototype Software Overview

The ASC modularity has been carefully partitioned so that the application programmer is presented with the simplest possible view of the processor. First, redundancy management of the ASC's themselves are handled by the SVM logic. Second, application tasks communicate using a simple mailbox scheme that operates directly out of local memory.

(Application software here refers to such things as Guidance, Navigation, Flight Controls, Propulsion Controls, etc.)

The application software resides on the R3000 PM. The software is run using a simple rate monotonic scheduling approach. A 40ms minor frame defines the base frequency of the system. Timing interrupts, input/output, and redundancy management, however, are provided by the SVM. The SVM provides identical (voted) inputs to the processors and votes the resulting output commands. This avoids channel specific application software paths which greatly simplifies the code and associated testing/debugging.

The Ada application software is combined with an R3000 Ada kernel which provides the interfaces to the processor board, including interrupts. For the JSC autocode tests software functions are executed based on the 40 millisecond minor frame.

Application software development efforts are supported using a Verdex Ada software development system.

R3000 Processor Module

The processor module is based on the MIPS Inc. 32-bit R3000 Reduced Instruction Set Computer. The module contains an R3000 CPU, R3010 floating point coprocessor, 16 megabytes of main memory, 128 kilobytes of instruction cache memory, 128 kilobytes of data cache memory, 512 kilobytes of boot PROM, and 128 kilobytes of EEPROM. The operating frequency of the module is 25 MHz and it delivers up to 20 VAX equivalent MIPs of computing power. The R3000 module is used to execute guidance, navigation, flight and propulsion control application software.

Synchronization & Voter Module

The synchronization and voter module (SVM) houses Boeing's advanced technology for performing fault tolerant hardware voting functions. The module contains an Intel 80386 processor, 8 megabytes of main memory, and the synchronization & voter circuits that interface with redundant ASCs to perform clock synchronization and data voting. The module is used by the ASC to perform ASC redundancy management.

1553B Data Bus Module

This module provides a full function, slave interface between the MIL-STD-1553B bus and the ASC computer system. The interface to the ASC is mechanized through 64 kilowords of RAM memory. The RAM is partitioned into control register storage, data buffers, and control block lists. Control block lists enable the module to perform multiple 1553B transactions without host ASC intervention. The module will be used in Bus Controller mode to manage 1553B bus traffic.

Interchannel Wire Harness

The interchannel wire harness is used to connect ASC units into redundant configurations. The harness contains point-to-point interconnects that SVMs use to share information during synchronization and voting operations.

Fault Injection System Overview

The fault injection system is a tool for testing the redundancy management circuits and logic of the ASC. Faults will be modeled as errors at the fault containment region boundary, and errors will be injected between fault containment regions. The types of faults to be modeled will be: single permanent physical, single transient physical, Byzantine, non/near coincidental, timing and synchronization, and power. Figure A2 shows the Fault Injection System and its interface to the ASCs.

Fault Injection System Software

The fault injection system will be setup through menu selections prior to running a mission or experiment. Also, faults can be injected "on-the-fly" for demonstration purposes. As the experiment is running, a fault will be injected at a precise time as programmed during setup.

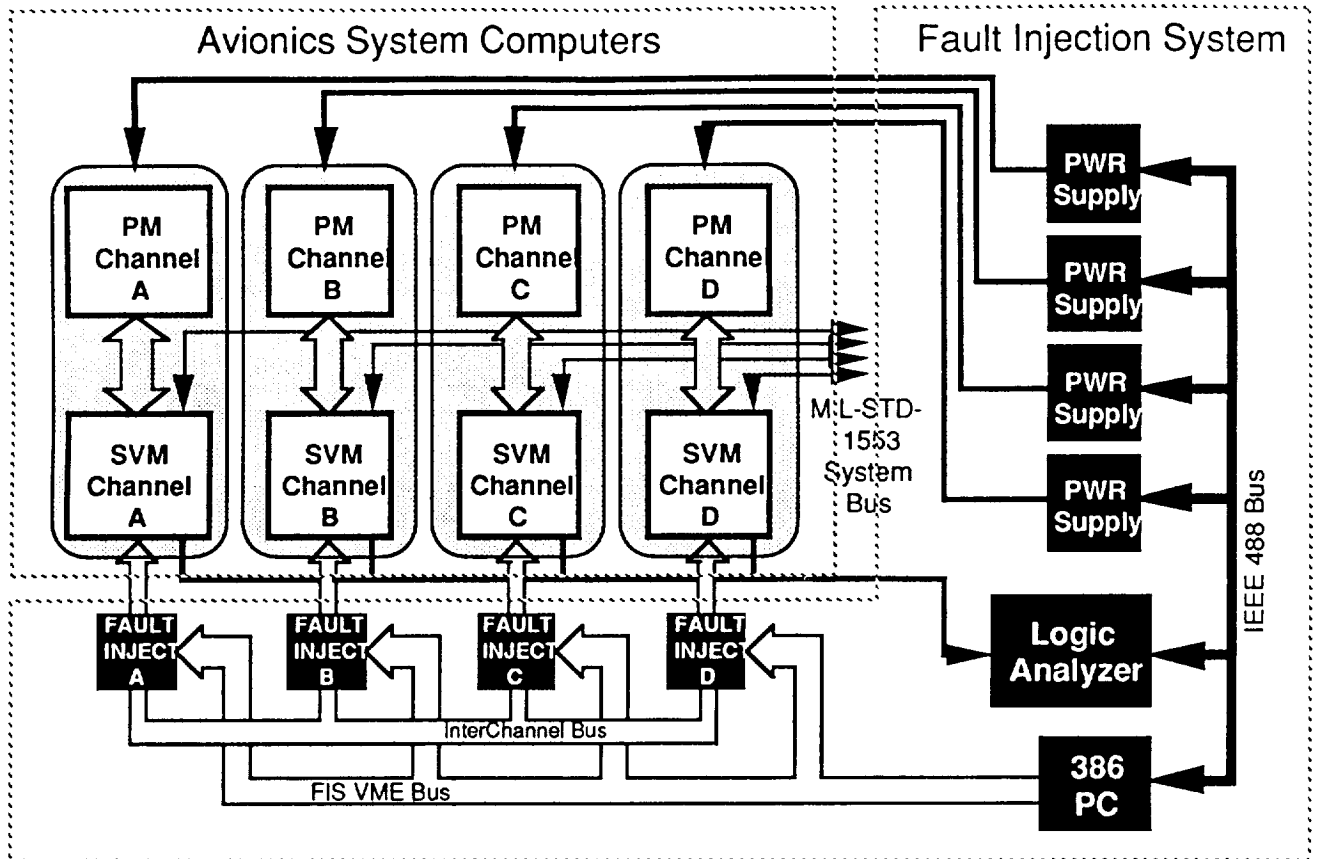


Figure A2. Fault Injection System Configuration

Fault Injection System Hardware

The fault injection system is a VME based system containing: a Radisys 386 PC computer with a VGA monitor, an IEEE 488 controller, an ethernet controller, and four Boeing Fault Injection Modules. The four Fault Injection Modules are plugged in-line with the interchannel cable (see figure A2). This approach is non-obtrusive with the flight computer in normal operation, and can inject faults at precise times.

APPENDIX B DAP Standalone Simulation—Integration Notes

Step 1 was for the target application to perform within a 'for loop'. The for loop substituted the real-time mechanism (frame interrupts) which would be generated by the Sync. & voter module (VME timing interrupt) in a later step. In this mode, the R3000 target executes independently of frame interrupts and voting. The R3000 got the inputs from the Non-voted inpool, and placed the outputs in the voted Outpool. After the mission/application has completed, the Voted Outpool was uploaded thru the ethernet link for comparison with the simulation output pool. The results from this step were that the R3000 repeatedly placed outputs in the Voted Outpool that matched bit-for-bit with the simulated Output pool on the Sun workstation.

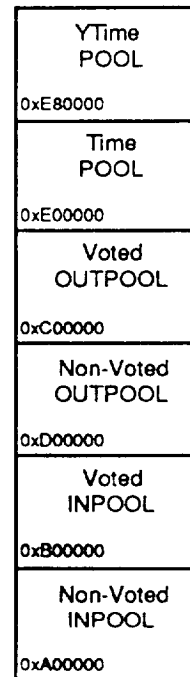
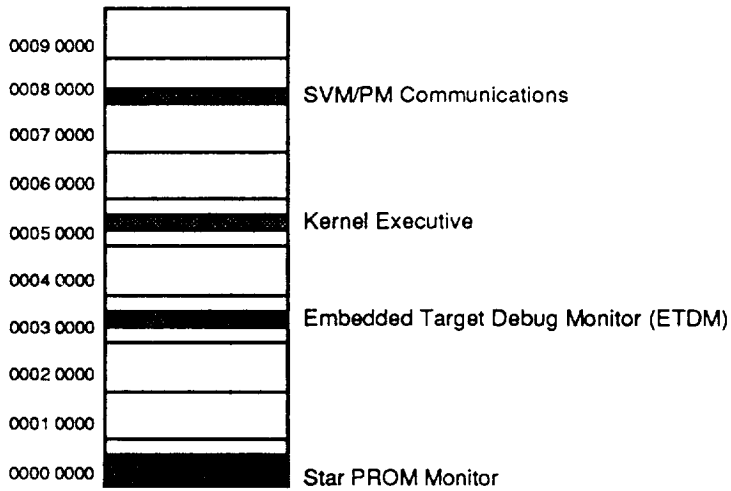
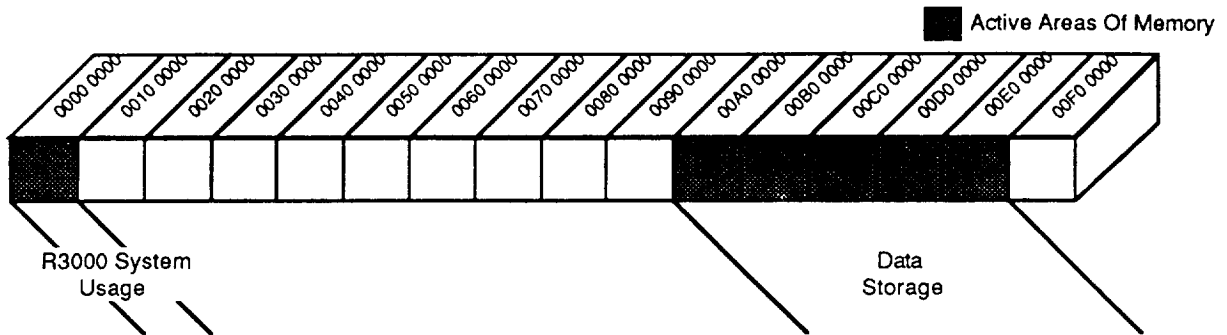
Step 2 was to remove the 'for loop' from the R3000 application and have the Sync. & voter module generate frame interrupts (VME interrupt #7). Therefore, the software running on the R3000 would perform an executive / scheduler and application (DAP) in real-time with respect to 40m sec. frames. Due to certain limitations within the Verdix kernel and Embedded Target Debug Monitor (ETDM), we were unable to quickly attach VME interrupts to the kernel. Faced with schedule constraints, it was decided to investigate interrupts at a later date and implement an alternative approach (frame flags) explained in the next step.

Step 3 was to replace VME interrupts with 'frame flags'. Simply put, the R3000 would poll a specific address in global memory. The SVM (sync. & Voting Module) would set that address at the beginning of the minor frame (40m sec.). The R3000 would see that address (frame flag) set, it would clear the frame flag, then execute the DAP. Note: this step does not involve the voting of inputs or outputs. The inputs are loaded (prior to execution) in the voted inpool from the Sun, and the outputs from the DAP are placed in the voted outpool. After the mission/application has completed, the Voted Outpool was uploaded thru the ethernet link for comparison with the simulation output pool. The results from this step were that the R3000 repeatedly placed outputs in the Voted Outpool that matched bit-for-bit with the simulated Output pool on the Sun workstation.

Step 4 was to implement voting of the inputs and outputs in real-time as outlined in the real-time execution section. As shown in the open loop environment figure (fig. 2.2.2-1), the inpool is downloaded (over ethernet) to the Non-voted inpool. On a frame by frame basis, the SVM will vote the non-voted inpool and place the voted inputs into the voted inpool. The DAP will get each frames inputs from this pool, execute and load outputs to the non-voted outpool. The SVM will vote the non-voted outpool and place the voted outputs into the voted outpool. Upon completion of execution, the voted outpool was uploaded to the sun for comparison with the verified output file.

APPENDIX C Standalone Testing Memory Map

R3000 Memory Map



APPENDIX D Autocoded Main Program with Scheduler

Available on request.

APPENDIX E SA_Utilities Ada Module

Available on request.

APPENDIX F SVM Mission Tables

Available on request.