Computer Science Department Technical Report
Dependable Computing and Fault-Tolerant Systems Laboratory
University of California
Los Angeles, CA 90024-1596

MULTI-VERSION SOFTWARE DEVELOPMENT: A UCLA/HONEYWELL JOINT PROJECTR FOR FAULT-TOLERANT FLIGHT CONTROL SOFTWARE

Algirdas Avizienis
Michael R. Lyu
Werner Schutz

April 1988
CSD-880034

TABLE OF CONTENTS

Multi-Version Software Development:

A UCLA/Honeywell Joint Project for

Fault-Tolerant Flight Control Software

Algirdas Avižienis

Michael R. Lyu

Werner Schütz

UCLA Dependable Computing and Fault-Tolerant Systems Laboratory

University of California, Los Angeles, CA 90024, U.S.A.

1. Introduction: Origin and Scope of the Project

The investigation being reported here is the consequence of a coincidence of research interests in design diversity [Aviz82] at the UCLA Dependable Computing & Fault-Tolerant Systems (DC & FTS) Laboratory and at the Sperry Commercial Flight Systems Division of Honeywell, Inc., in Phoenix, Arizona (abbreviated as "H/S" in the following discussion).

Four of the long-range goals of UCLA research, which was initiated in 1975 [Aviz85b], are:

(1) The development of rigorous design guidelines (a *paradigm*) that will eliminate all identifiable causes of *related* design faults in two or more independently generated versions of a program or design.

(2) Testing for and detailed study of all potentially related design faults that actually produce similar errors in two or more versions independently generated from a given

specification.

(3) Development of qualitative criteria that allow the assessment of the *potential for diversity* through the study of a specification from which the versions are to be generated.

(4) Development of methods for the study of a set of multiple versions to determine to what extent diversity is actually present in the set, and search for the means to quantize the relative diversity of versions that originate from a given specification. The relative benefits of random vs. "enforced" diversity are also of great interest.

Honeywell/Sperry CFSD has been a very successful builder of aircraft flight control systems for over 30 years. A recent major product of H/S is the flight control system for the Boeing 737/300 airliner, in which a two-channel diverse design is employed [Will83].

The main research interest of H/S is the generation of demonstrably effective N-version software in an industrial environment, such as exists now and is being further developed by H/S. This objective includes all four above stated topics of UCLA research, referenced to the industrial environment, as well as the estimation of the effectiveness of N-version software and of its relative safety as compared to a single-version approach.

It was mutually agreed that an experimental investigation was necessary, in which H/S would supply an automatic flight control problem specification, specify H/S software design and test procedures, deliver an aircraft model and sets of realistic test cases, and also provide prompt expert consultation. The research was initiated in October, 1986 and carried out at the UCLA DC & FTS Laboratory, funded jointly by H/S and the State of California "MICRO" program. A six-version programming effort in which six programming languages were used and 12 programmers were employed took place during 12 weeks of the summer of 1987. An intensive evaluation followed, and is continuing as of February, 1988.

2. The Problem and the Constraints

2.1 The Global Scope of the Application

The control laws implemented in this particular experiment are typical of those certified for the automatic pitch control of commercial airliners, in the landing flight phase. The experiment features the following activities:

1. Simulation of the automatic pitch control of a transport aircraft, in the final approach to the airport. The elements of the control loop are control laws, the airplane, sensors mounted on the airplane, the landing geometry, and wind disturbances.

2. Programming of control laws by independent teams, based on software requirements document (i.e., the software specification).

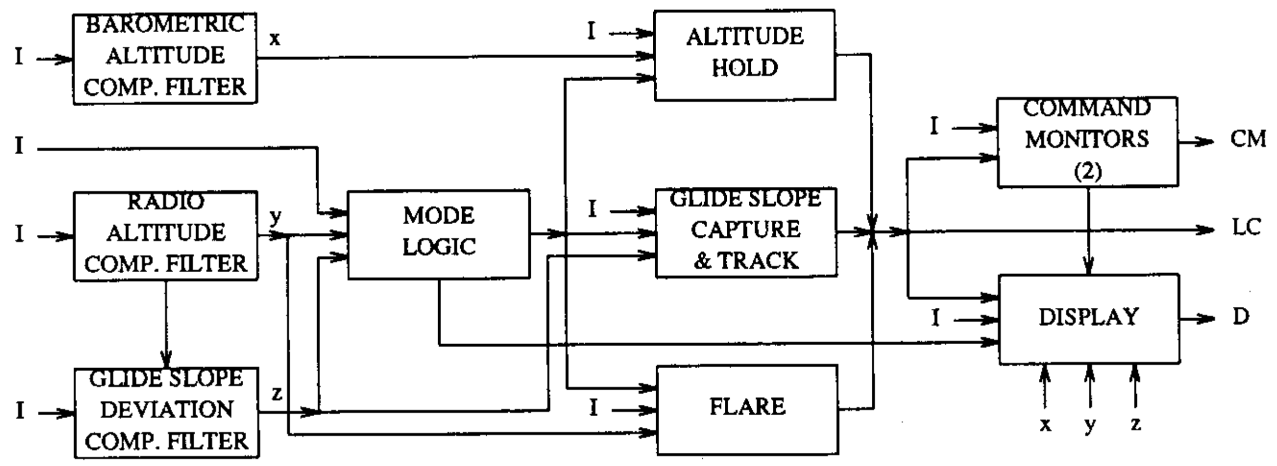
3. Modelling of airplane and wind turbulence on Unix environments.

An aircraft model definition document, which describes the global structure of the whole flight control system, was supplied by H/S. This document provided mathematical models for functions within the landing/approach control loop, but external to the control laws to be implemented by multi-version software. These models were programmed on the host computer in order to provide a suitable control problem for the experiment.

2.2 The Automatic Landing Problem

Automatic (computer-controlled) landing of commercial airliners is a flight control function that has been implemented by H/S and other companies. The specification used in the UCLA-H/S experiment is part of a specification used by H/S to build a 3-version Demonstrator System (hardware and software), employed to show the feasibility of N-version programming for this type of application. The specification can be used to develop a flight control computer

(FCC) for a real aircraft, given that it is adjusted to the performance parameters of a specific aircraft. All algorithms and control laws are specified by diagrams which have been certified by the Federal Aviation Administration (FAA). The *pitch control* part of the auto-land problem, i.e., the control of the vertical motion of the aircraft, has been selected for the experiment in order to fit the given budget and time constraints. The major system functions of the pitch control and its data flow are shown in Figure 1.



BAROMETRIC ALTITUDE COMP. FILTER
RADIO ALTITUDE COMP. FILTER
GLIDE SLOPE DEVIATION COMP. FILTER
MODE LOGIC
ALTITUDE HOLD
GLIDE SLOPE CAPTURE & TRACK
FLARE
COMMAND MONITORS (2)
DISPLAY
I
x
y
z
CM
LC
D

Legend: I = Airplane Sensor Inputs
LC = Lane Command
CM = Command Monitor Outputs
D = Display Outputs

Figure 1: Pitch Control System Functions and Data Flow Diagram

Simulated flights begin with the initialization of the system in the Altitude Hold mode, at a point approximately ten miles from the airport. Initial altitude is about 1500 feet, initial speed 120 knots (200 feet per second). Pitch modes entered by the autopilot-airplane combination, during the landing process, are: Altitude Hold, Glide Slope Capture, Glide Slope Track, Flare, and Touchdown.

The *Complementary Filters* preprocess the raw data from the aircraft's sensors. The *Barometric Altitude* and *Radio Altitude Complementary Filters* provide estimates of true altitude from various altitude-related signals, where the former provides the altitude reference for the Altitude Hold mode, and the latter provides the altitude reference for the Flare mode. The *Glide Slope Deviation Complementary Filter* provides estimates for beam error and radio altitude in the Glide Slope Capture and Track modes.

Pitch mode entry and exit is determined by the Mode Logic equations, which use filtered airplane sensor data to switch the controlling equations at the correct point in the trajectory.

Each Control Law consists of two parts, the Outer Loop and the Inner Loop, where the Inner Loop is very similar for all three Control Laws. The Altitude Hold Control Law is responsible for maintaining the reference altitude, by responding to turbulence-induced errors in attitude and altitude with automatic *elevator* control motion. (The elevator is the surface of an airplane that controls the vertical motion.) As soon as the edge of the glide slope beam is reached, the airplane enters the Glide Slope Capture and Track mode and begins a pitching motion to acquire and hold the beam center. A short time after capture, the track mode is engaged to reduce any static displacement towards zero. Controlled by the Glide Slope Capture and Track Control Law, the airplane maintains a constant speed along the glide slope beam. Flare logic equations determine the precise altitude (about 50 feet) at which the Flare mode is entered. In response to the Flare control law, the vehicle is forced along a path which targets a vertical speed of two feet per second at touchdown.

Each program checks its final result (elevator command) against the results of the other programs. Any disagreement is indicated by the Command Monitor output, so that the supervisor program can take appropriate action.

The Display continuously shows information about the FCC on various panels. The current pitch mode is displayed for the information of the pilots (Mode Display), while the results of the Command Monitors (Fault Display) and any one of sixteen possible signals (Signal Display) are displayed for use by the flight engineer.

Upon entering the Touchdown mode, the automatic portion of the landing is complete and the system is automatically disengaged. This completes the automatic landing flight phase.

The original specification of this application given to the programmers was a 64 page document (including tables and figures) written in English. Its development required about 10 weeks of effort by two members of the coordinating team, plus consultation by H/S experts. Its current version can be found in Appendix I, and its detailed evolution is presented in Section 4.

2.3 The Choice of Diversity Dimensions

Design diversity is a potentially effective method to avoid similar errors that are caused by design faults in N-version software systems. The choice of diversity dimensions in this experiment was based on the experience gained from (1) previous experiments at UCLA [Chen78, Kell83, Aviz84, Kell86], (2) recommendations from H/S, and (3) published work from other sites [Gmei79, Bish86, Knig86].

Independent programming teams are the baseline dimension for design diversity. This allows the diversity to be generated with an uncontrolled factor of "randomness". However, different dimensions of design diversity, including different algorithms, programming languages, environments, implementation techniques and tools, should be investigated and explored, and possibly used to assure a certain level of "forced" diversity [Aviz85b]. It was decided that different algorithms were not suitable for the scope of FCCs due to potential timing problems and difficulties in proving their correctness (guaranteed matching among them). The investigation of different programming languages was attractive since it provides protection from subtle compiler errors and avoids the need to certify compiler correctness. Moreover, although research had been initiated in this direction [Gmei79, Bish86], significant comparisons of different high order programming languages for the same critical application have not yet been reported.

Six programming languages were chosen to implement the application problem in this project. They included: two widely used conventional procedural languages (C and Pascal), two modern object-oriented programming languages (Ada and Modula-2), a logic programming language (Prolog), and a functional programming language (T, a variant of Lisp). It was hypothesized that different programming languages will force people to think differently about the application problem and the program design, which could lead to significant diversity of programming efforts. Choices of the Prolog and T versions presented challenges to this project, since it was thought that they might not be suitable for this computation-intensive application. Nevertheless, it was still considered to be worthwhile to investigate this unexplored area, especially to assess the impact of Prolog and T on the structure of the auto-land programs.

2.4 Requirements for Software Testing

H/S, along with other avionics suppliers, must adhere to the requirements of the document DO-178A [RTCA85], the industrial software design and test standard approved by the FAA. The following definitions apply to software testing, as specified in [RTCA85].

(a) Requirements-Based Tests (black box testing). Test cases are derived from the software requirements independent of the software structure. Primarily, these are the requirements specified in the Software Requirements Document (Software Specification), but further requirements may emerge during the design process (e.g., scheduler requirements). These tests demonstrate that the software performs its *intended functions*. Each software requirement should be traceable to an associated verification test or tests.

(b) Software Structure-Based Tests (white box testing). Test cases are derived from the software design itself. As such, they can address features of the implementation which may or may not be apparent from a requirements perspective. Typically, requirements-based tests are analyzed for structural coverage and augmented as necessary. In this sense the structure-based tests complement the requirements-based tests to provide sufficient test coverage. Such structure-based tests are necessary to provide some measure of protection from *unintended functions* in software that may pass all of its requirements-based tests. All of the software must be exercised to a degree commensurate with its software certification level.

Therefore, software errors are postulated to be caused by two types of human-made faults: *requirement* faults and *structural* faults. A requirement fault exists when a specified requirement is not or not completely complied with. A structural fault is the complement of the requirement fault, i.e., it is any fault which is not exposed by system testing based on the system specification.

2.5 The Rationale for Applying Design Diversity in Testing

Three categories of aircraft systems are distinguished by the FAA, namely *flight critical, flight essential,* and *non-essential,* with different testing efforts required for each. In general, avionics equipment is designated as "critical" when loss of the function provided by the equipment can cause a catastrophic aircraft failure. The probability of such an occurrence must be demonstrated by test or analysis to be 10^{-9} or less over the duration of the flight. Avionics equipment is designated as "essential" when loss of its function can significantly impact safety. For essential equipment the probability of loss of function must be demonstrated to be 10^{-5} or less over the duration of the flight.

The software portion of the critical equipment must, then, have a probability of failure less than 10^{-9} depending on the failure rates in remaining portions of the system. To protect against failures in single-version software that cause total loss of a critical function, a *structural-based testing* methodology is required in addition to *requirements-based testing*. Any fault will manifest itself identically in all redundant computation channels that use identical software; but this exhaustive testing procedure (Level 1) is assumed to assure the desired reliability. For software which can fail and cause loss of an essential function only, requirements-based testing alone is required (Level 2).

While requirements-based testing may be extensive, the number of test cases is bounded by the system requirements. Structure-based testing, on the other hand, is likely to be very extensive, possibly involving permutations of all inputs together with a rather subjective evaluation of each result. If more than a few inputs are involved, the time required to prepare and run the test, and to analyze the results becomes prohibitive and may present a serious scheduling and cost problem. Structural testing appears to be analogous to the hardware "failure modes and effects analysis" procedure with LSI circuits, which is acknowledged to be extremely difficult to implement fully [Trea82]. Therefore, the FAA encourages manufacturers, where practical, to reduce the level of testing by architectural means. The architectural techniques to reduce test levels that the FAA has accepted, or is likely to accept, employ design diversity as their central attribute. The application of threefold diversity in critical software is based on the conjecture that the likelihood of two identical, critical structural faults in 3-version software is, in the verified and validated release, substantially reduced from the likelihood of a critical structural fault in a single version; thus only Level 2 testing may be required in 3-version architectures. The FAA has recognized, however, that the conflicting requirements for design independence and of having the diverse elements perform the same function impose an important design constraint. Therefore, these systems must be shown to monitor each other under all forseeable conditions and critical modes of operation.

3. Guidelines and the Process of Multi-Version Programming

3.1 Personnel

The recruitment and interviewing of programmers started about 3 months before the 12-week version generation phase in June, 1987. The summer is an especially favorable time to recruit highly qualified personnel from the about 260 CS graduate students at UCLA, since about 20 Teaching Assistants (many of them from programming classes) and several fellowship holders are able to accept summer employment. About 20 candidates, most of them graduate students at UCLA, submitted applications. The final choice of 12 programmers and their assignment to six teams were made one month before starting the software generation. Table 1 shows the specialties, graduate standing, and qualifications of the programmers identified by their assigned languages. The data indicate a mature, experienced, and well qualified group of research programmers. The effort was directed by the Principal Investigator, and coordinated by a three-member coordinating team, who started the work of writing the specification and developing guidelines and procedures, with support of H/S personnel, in November, 1986. A senior staff expert in flight control computing from H/S maintained continuous contact and regularly made visits to UCLA.

3.2 Schedule of the Experiment

The software version generation for this experiment was conducted in six phases:

1. *Training meetings (five in total, 2-4 hours each)*: One project-introduction meeting was offered to all the applicants, and all other four meetings were held after the selection of personnel. H/S presented a discussion of flight control systems as background information. Introductory presentations were made summarizing the experiment's goals, requirements and the multiple version software techniques. Issues of different programming languages were also discussed. A kick-off meeting was

Team member	Degree held			CS standing in summer '87		Programming experience
	Field	Degree	Year	Program	Year	
Ada-1	CS	B.S	1984	M.S.	2nd	3 years
Ada-2	ECE ECE	B.S. M.S.	1982 1984	Ph.D.	2nd	3 years
C-1	IE CS	B.S. M.S.	1981 1983	Ph.D.	3rd	2 years
C-2	CS CS	B.S. M.S.	1982 1984	Ph.D.	2nd	5 years
Modula2-1	ECE ECE	B.S. M.S.	1982 1984	Ph.D.	2nd	3 years
Modula2-2	ECE	B.S.	1984	M.S.	2nd	2 years
Pascal-1	EE	B.S.	1984	M.S.	4th	6 years
Pascal-2	EECS ECE	B.S. M.S.	1984 1986	Ph.D.	2nd	2 years
Prolog-1	EE CS	B.S. M.S.	1984 1986	Ph.D.	2nd	3 years
Prolog-2	CS	B.S.	1986	M.S.	2nd	3 years
T-1	EECS	B.S.	1983	M.S.	3rd	2 years
T-2	CS	B.S.	1986	M.S.	2nd	3 years
Coord-1	ECE CS	B.S. M.S.	1981 1984	Ph.D.	4th	6 years
Coord-2	CS CS	B.S. M.S.	1984 1986	M.S.	2nd	3 years
Coord-3	ECE	B.S.	1986	M.S.	2nd	2 years

Table 1: Summary of the UCLA Programmer and Coordinator Background

held on the first day of the software development phase. At that meeting, the programmers were given the written specifications and documentation on system tools to start their 12-week effort. Rules and guidelines about schedules, deliverables, and communication protocols were also clearly defined. The programmers were strongly motivated and showed serious concerns about the project in these meetings.

The need for inter-team isolation was thoroughly discussed and clearly acknowledged by all programmers.

2. *Design phase (4 weeks)*: At the end of this four-week phase, each team delivered a design document following the guidelines and formats provided at the kick-off meeting. Each team delivered a design walkthrough report after conducting a walkthrough which was attended by UCLA and H/S principal investigators, the UCLA coordinating team, and an H/S software expert.

3. *Coding phase (3 weeks)*: By the end of this 3-week phase, programmers had finished coding, conducted a code walkthrough by themselves, and delivered a code development plan and a test plan. Code Update Report forms were distributed for them to record every change that was made after the code was generated.

4. *Unit testing phase (1 week)*: Each team was supplied with sample test data sets (generated by H/S) for each module that were suitable to check the basic functionality of that module. They had to pass all the unit testing data before they could proceed to the next phase. One week was allotted to this phase. At the end of this phase, each team conducted a coding/testing review with UCLA coordinators and H/S representatives to present their progress and testing experience.

5. *Integration testing phase (2 weeks)*: Four sets of partial flight simulation test data were produced by H/S and provided to each programming team for integration testing. This phase of testing was intended to guarantee that the software was suitable for the closed-loop simulation of the integrated system.

6. *Acceptance testing phase (2 weeks)*: Programmers formally submitted their programs. Each program was run in a test harness of nine flight simulation profiles. When a program failed a test it was returned to the programmers with the input case on which it failed, for debugging and resubmission. By the end of this two week phase, five programs had passed this acceptance test successfully. The T program

encountered difficulties in using the T interpreter and it was necessary to do additional work over the next month before that version passed the acceptance test.

All the participants of this project presented concluding talks and met each other socially at a final one-day workshop when the software generation phase ended. During that occasion programmers were free to talk with each other, exchange their experiences, and fill out a Post-Experiment Questionnaire. A large variety of experiences, viewpoints and difficulties encountered were brought out during this final workshop and following party.

All the forms and documents that were distributed in this experiment are listed in Appendix II.

3.3 The Programming Process

The software engineering process involved in this project included formal reviews, well-planned record keeping, isolation rules, a formal communication protocol, and carefully executed testing phases. This controlled process provided continuous interactions between the coordinators from UCLA and H/S, and each individual team.

The design review, the coding/testing review, and the final review and workshop were the three formal reviews within this project, all with the participation of H/S experts. These reviews were designed to follow industrial standards as much as possible. Moreover, they served as checkpoints to observe the progress of each programming team and to adjust the development process according to their feedback.

For the purpose of keeping a complete record, several "deliverables" were required from each team. These deliverables, representing the products of the project, included two "snapshots" of each separate module (before and after unit tests), four snapshots of the complete program (those before and after integration tests, and those before and after acceptance tests), two design documents (preliminary and final versions), program metrics, design walkthrough reports, and code update reports.

Since error reporting was considered extremely important for this project, each team was required to report all the changes made to their program, starting from the time when the program first compiled successfully. All changes had to be reported, no matter whether they were due to detected faults, efficiency improvement, specification updates, etc. For each change a "Code Update Report", a standardized form designed by the coordinating team, had to be turned in. If a code change was made because of a design change, a "Design Walkthrough Report" (another standardized form) had to be submitted as well. For the subsequent analysis, we consider only those changes that were done to correct faults in the programs.

The purpose of imposing isolation rules on the teams was to assure the ‘‘independent generation’’ of programs, which meant that programming efforts were carried out by individuals or groups that did not interact with respect to the programming process. In order to keep this constraint, the programming teams were assigned physically separated offices for their work. Additionally, programmers were strictly admonished not to discuss any aspect of their work with members of other teams. The coordinating team monitored the progress of each team. Work-related communications between programmers and the coordinating team were conducted only via a formal tool (electronic mail). The programmers directed their questions to the coordinating team, who then tried to respond as quickly as possible. Whenever necessary, the help of the H/S flight control experts was provided by phone calls and personal meetings to resolve questions.

Generally, each answer was only sent to the team that submitted the corresponding question. The answer was broadcast to all teams only if the answer led to an update or clarification of the specification, if there was an indication of a misunderstanding common to some teams, or if the answer was considered to be important or relevant for other teams for some other reason. In the first case, a broadcast constituted an official amendment to the original specification. This contrasts with the communication protocol used in the NASA experiment [Kell86] where the answers to *all* questions were broadcast, regardless of which team submitted the question. The resulting flood of messages proved to be a bothersome overload, that was avoided this time. The communication diagram among H/S experts, the UCLA coordinating team and the programming teams is presented in Figure 2.

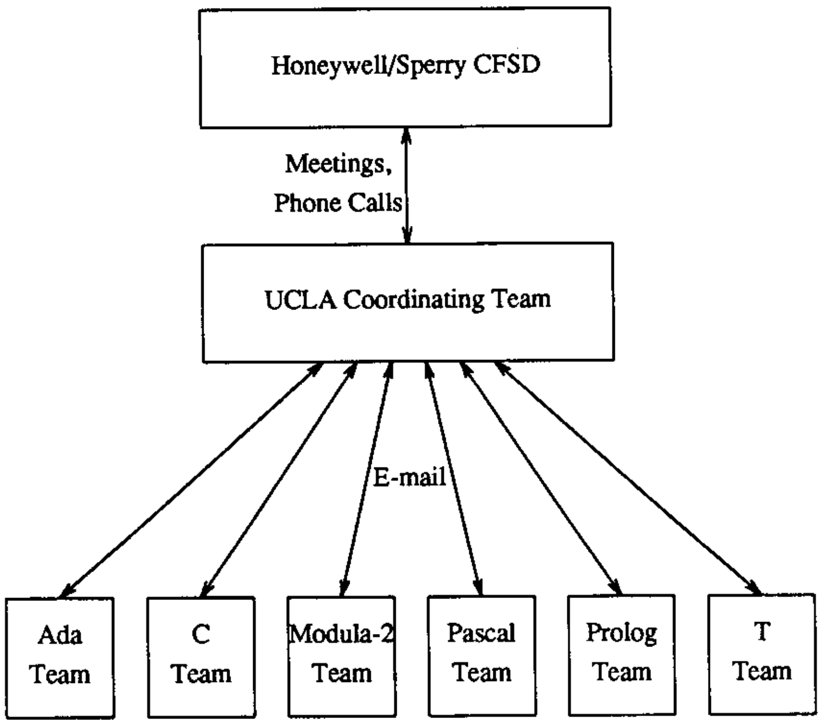


Figure 2: Communication Diagram of the Experiment

3.4 Experience with the Communication Protocol

The communication protocol was designed in order to: (1) prevent the ambiguity of oral communications; (2) give the coordinating team time to think and discuss before answering a question and to summon the help of H/S flight control experts, if necessary; (3) provide a record of the communication for possible analysis; (4) reduce the number of messages sent to each individual team; and (5) adhere to the principle of supplying only absolutely necessary information to the programming teams, aiming to avoid any bias on a team's design decisions by supplying unnecessary and/or unrequested information.

With respect to the first three goals the protocol was very successful although occasionally it was felt that it was more difficult to write the answer to a certain question, that oral communication would have been easier and more efficient in some cases. The communication with H/S was very efficient; thus it was possible to answer all questions within a short time, usually less than one day.

Altogether, about 120 questions were sent by the programming teams. The answers to only 30 of them were broadcast. The total number of broadcast messages was 40, three of which required an additional follow-up message, to provide further clarification or to correct errors in the original message. 10 broadcast messages were not triggered by a question; 5 of them were sent because either the coordinating team or H/S detected an error in the specification or for some other reason decided to update it, and 5 of them were a result of the Design Review at which some common misinterpretations of the specification were observed. The individual teams received between 53 and 64 messages; that constitutes a reduction by a factor of 2 in comparison with the number of messages that would have been received if the communication protocol of the NASA experiment [Kell86] had been used.

3.5 Exercise of the Phased Testing

To emphasize the importance of testing, three phases of testing, unit tests, integration tests, and acceptance tests, were introduced for error detection and debugging. Different strategies for program testing were provided during program generation phase in order to clean up programs before they were subjected to a final evaluation. Table 2 lists the differences among these phases.

category	unit test phase	integration test phase	acceptance test phase
test case generator	open loop by PC Basic	closed loop by PC Basic	closed loop by multiple languages
test data access	file i/o by each version	interfacing C routines	interfacing C routines
test procedure	by individual teams	by individual teams	by coordinating team
tolerance level	0.01 for degrees	0.01 for degrees	0.005 for degrees (except prolog)

Table 2: Different Schemes Used in the Testing Phases

At first a reference model of control laws was implemented and provided by H/S flight control software engineers. This version was implemented in Basic on an IBM PC to serve as the test case generator for the unit tests and the integration tests. Criteria of open loop testing and closed loop testing were used, respectively. Due to the wide numerical discrepancies between this version and the other six versions, a larger tolerance level was chosen.

Later in the acceptance test, this reference model proved to be less reliable (several faults were found) and less efficient, since the PC was quite slow in numerical computations and I/O operations. It was necessary to replace it with a more reliable and efficient testing procedure for a large volume of test data. For this procedure, the outputs of the six versions were voted and the majority results were used as the reference points to generate test data during the acceptance tests. This was also the test phase that programmers were required to submit their programs to the coordinating team and wait for the test results. A finer tolerance level was used based on the observation that less discrepancies were expected if programs would compute the results right. An exception had to be made for the prolog program due to the lack of accuracy in its internal representation of real numbers.

As to the numbers of test cases performed in each phase, the detailed information is presented in the following three tables, Table 3, Table 4, and Table 5.

name of the module	number of test cases
baro altitude filter	7
radio altitude filter	11
glideslope filter	7
mode logic	11
altitude hold mode outerloop	10
glideslope mode outerloop	12
flare mode outerloop	15
innerloop	23
command monitor	5
display module	32
total	**133**
total frames	about **1330***

* There were roughly ten frames per each test case.

Table 3: Test Data in Unit Test Phase

id	testing time	involved modes	wind turbulence
data.1	12 sec	AHD mode only	no wind turbulence
data.2	12 sec	AHD-GSCD-GSTD	no wind turbulence
data.3	12 sec	AHD mode only	average wind turbulence
data.4	12 sec	AHD-GSCD-GSTD	average wind turbulence
total frames		**960 frames***	

* Each second has 20 frames of execution.

Table 4: Test Data in Integration Test Phase

id	time	involved modes	turbulence	other test
data.1	100 sec	AHD-GSCD-GSTD	no	no
data.2	180 sec	AHD-GSCD-GSTD-FD-TD	no	no
data.3	100 sec	AHD-GSCD-GSTD	average	no
data.4	180 sec	AHD-GSCD-GSTD-FD-TD	average	no
data.5	100 sec	AHD-GSCD-GSTD	maximum	no
data.6	180 sec	AHD-GSCD-GSTD-FD-TD	maximum	no
data.7	30 sec	AHD mode only	maximum	recovery
data.8	22 sec	AHD-GSCD-GSTD-FD-TD	maximum	recovery
data.9	30 sec	AHD mode only	maximum	display
total frames		**18440 frames***		

* Each second has 20 frames of execution.

Table 5: Test Data in Acceptance Test Phase

During the above testing phases, numerous errors were found. The audit of errors in both the specification and the programs, together with their descriptions and classifications, is discussed in later sections.

4. The Evolution of the Specification

In this section we will briefly describe how the specification (also called Software Requirements Document) for the application problem was developed. We will also cover the changes made to the specification during the course of the experiment. Finally, some remarks and criticism from members of the programming teams, as well as some other observations are reported.

4.1 Process of Writing the Specification

The efforts to develop a specification that is suitable to be used by the programming teams started early in 1987. Since it was clear from the beginning that programming the complete autopilot would be too complex and too large a task for a twelve week programming experiment, the first major task was to find a subset of the autopilot that, when programmed, would result in a program of "reasonable" size. "Reasonable" was informally defined as "as large as possible while still being manageable within twelve weeks". It was decided that the "pitch" function of the autopilot with some added display functions should be programmed.

In the next step, representatives from H/S extracted the information needed for the experiment from their original Demonstrator specification and provided it in a "System Description Document", which was subsequently reviewed by the members of the coordinating team. The goal was to understand the problem as thoroughly as possible, in order to avoid as many ambiguities as possible and to provide a clear specification. Many meetings of the coordinating team with representatives from H/S were devoted to clarify various aspects, partly on a very detailed level.

To write the specification that was given to the programmers, the UCLA coordinating team followed the principle of supplying only minimal (i.e., only absolutely necessary) information to the programmers, so as not to unwillingly bias the programmers' design decisions and overly restrict the potential design diversity. The diagrams describing the major system functions were taken directly from the System Description Document, while the explanatory text was shortened and made more concise.

The original System Description Document specified "test points", i.e., selected intermediate values of each major system function which had to be provided as outputs for additional error checking. A further enhancement to the specification was the introduction of *cross-check points* [Aviz85a] and a *recovery point* [Tso87]. Seven cross-check points are used to cross-check the results of the major system functions (e.g. Complementary Filters, Mode Logic, Outer Loop, Inner Loop, etc.) with the results of the other versions before they are used in any further computation. They have to be executed in a certain predetermined order, but again great care was taken not to overly restrict the possible choices of computation sequence. One recovery point is used to recover a failed version by supplying it with a set of new internal state variables that are obtained from the other versions by the Community Error Recovery technique [Tso87].

In an appendix, the symbols used in the graphical representations of the system functions were explained, and it was explained how to deal with feed-back loops that appeared in the charts. In addition, the coordinating team imposed the requirement that two input routines and eight so-called "vote routines" be inserted at well defined points of the computation sequence. The purpose of the input routines was to facilitate the reading of sensor data for each channel. The purpose of the vote routines was to allow cross-checking and comparison of different versions' outputs of major system functions, as well as of some selected intermediate results (test points). A second appendix defined the syntax of all these vote routines.

We have noted that a small number of errors in the original specification could lead to numerous ambiguous and contradictory addenda in the form of question and answer pairs [Kell86]. To prevent the confusion, the coordinating team at UCLA was very careful about message replying in order to minimize the broadcast information. Since there was always a quick response from H/S flight control engineers when the question needed to be forwarded to them, the turn-around time for the programmers to receive their answers was very short.

During software generation, many errors and ambiguities in the specification (including the electronic communications) were revealed. All questions by the programming teams were handled according to the communication protocol described before. Throughout the program development phase, the specification has been maintained as clear and precise as possible. More details about the questions that programmers came up with, as well as the changes to the specification they triggered, can be found later in this section.

The specification has now been restored to a single document, a document that has benefited from the scrutiny of more than 16 motivated programmers and researchers. This current version of the specification is provided in Appendix I. The changes of the specification, compared with its original version given to the programmers, are marked in bold faces.

4.2 Questions and Answers

The so-called "Question and Answer" messages conveyed information that the coordinating team judged to be important for all teams. Therefore, these messages were broadcast to all programming teams. Specification updates and changes were always announced with messages of this type, but not all Question and Answers contained specification updates. The following is a short summary about the "Question and Answer" messages.

All in all, there were 40 Questions and Answers. (There was another one that did not deal with specification related issues and is therefore not counted here.) In addition, 3 Questions and Answers required a follow-up message, to provide further clarification or to correct errors in the original message.

With respect to the time these messages were sent, we found that the majority of them (34) were sent during the Design and Coding phases of the experiment, and that 6 were sent after the coding phase.

Another interesting aspect is to examine the reason why these messages were sent. Most of them (30) were sent in response to a question from one or more programming teams. The answer to such questions was broadcast if there was an indication of a common misunderstanding of some teams, if the question revealed a problem in the specification and the answer thus contained a specification update, or if the answer was considered to be important to other teams for some other reason. 6 messages were initiated by the coordinating team, because some mistake was discovered. 5 of these latter messages were a result of the Design Review, and were broadcast to all teams because some common misinterpretation of the specification were found then. 3 Questions and Answers were sent because of specification updates or changes from H/S, made after the experiment had already begun. Finally, one specification update was necessary after an error had been detected during testing.

The number of Questions and Answers, grouped by their content, is shown in Table 6.

typos and readability problems	6
Inner Loop	6
Mode Logic	5
Vote Routines	4
Initialization	3
switch SW3 in G/S Control Law	3
general questions on basic operations	3
miscellaneous	rest

Table 6: Distribution of Broadcast Questions and Answers

Note that the numbers do not necessarily add up to 40 because some messages might have been counted more that once. For instance, if a Question and Answer was concerned with the initialization of the Inner Loop, then it is counted in the category "Initialization", as well as in "Inner Loop".

4.3 Specification Changes

In this section we give a list of all specification changes made during the twelve weeks of the experiment. Only the correction of typos is ignored. These changes can be audited by either reviewing the 40 Question and Answer messages, or by comparing the version of the specification at the end of the experiment with the first version. In the following, changes are reported in the order in which they are found in the specification, not in chronological order.

4.3.1 Changes in Chapter 2 (System Overview)

In the section on the major computation sequence the words "with path integrator" have been deleted in item 6) because they did not convey any information at this point, and led to some confusion. The mistake here was to copy directly from H/S's System Description Document.

4.3.2 Changes in Chapter 3 (Complementary Filters)

In the section on the Radio Altitude Complementary Filter, the initialization of integrator I6 was specified twice, and inconsistently. This was solved by deleting the words "and I6" in the first sentence of the initialization part. The reason for this error is a typo during editing that was not found by subsequent proof-reading.

In Figure 3.3, the Glide Slope Deviation Complementary Filter, the drawing note 2) had to be changed from "see Fig. 5.3.3-2" to "see page 13". The old version of the drawing note related to a figure in H/S's original System Description Document. Instead of using that figure the information was given in textual form in the specification. This was simply an oversight when taking the figures from the System Description Document and putting them into the specification.

4.3.3 Changes in Chapter 4 (Mode Logic)

A lot of changes had to be made in the Mode Logic. First of all, the sentence "The initial values for GSCD, GSTD, FD, and TD are all 'false'" was added at the end of section 4.3. It was decided to add this sentence for clarity although it is stated in the introduction that the initial mode is Altitude Hold Mode, therefore the variables for all other modes have to be 'false' initially.

In figure 4.1 of the Mode Logic, three changes were made: First, the sign of the input FPDC1 was changed from '-' to '+', indicating that it should be added to FPEC1 instead of being subtracted from. Then, additional inputs were added to the last AND-gates before the outputs AHD, GSCD, and GSTD, to ensure all Mode Logic variables are properly reset, even if a mode should happen to be skipped (i.e. not entered at all). These two changes were made because of specification updates received from H/S. Last, a statement was added to the drawing note 2), saying that zero should be regarded as a positive number in this case. This was done to ensure that comparing the sign of magnitude of two numbers was a defined operation even if one (or both) of these numbers happened to be zero.

In figure 4.2 of the Mode Logic, the drawing note 1) was changed from "initialize to input" to "initialize to zero". This was a specification change from H/S.

4.3.4 Changes in Chapter 5 (Altitude Hold Mode Control Law)

A paragraph was added to the section "Processing of the Inner Loop" to explain the special case of a feedback loop overlapping with another computation path. Rather detailed instructions were given on how to deal with and how to compute this special case. The reason for this change was that it was not easily possible to apply the general explanation about how to determine the computation sequence (Appendix A) to this case.

Some changes were made to figure 5.1, Altitude Hold Control Law. The gain constant above summer SU10 was changed to $\frac{1}{16}$, the gain constant to the left of summer SU5 was changed to 0.05, and the input "Pitch Attitude" was fed to summer SU3 (with a negative sign) instead as to summer SU6. These three changes were due to specification updates from H/S. Then, the line leading from the output of limiter LM1 to the determination of the condition for switch SW2 was moved to the right, so that it originates on top of the rate limiter LR1. This was done in order to clarify the fact that the history of rate limiter LR1 has to be used to determine the condition for SW2. (Note: The same change, for the same reasons, was made in figures 6.1 and 7.1, Glide Slope Capture and Track and Flare Control Laws, resp.) Finally, the drawing note 1) was removed from integrator I1, in order to eliminate an inconsistency between the figure and the text – where the text turned out to be correct.

4.3.5 Changes in Chapter 6 (Glide Slope Capture and Track Control Law)

A paragraph was added in the section "Processing of the Outer Loop" to explain the condition for switch SW3. This was necessary because the notation used was not self-explanatory. In figure 6.1, Outer Loop of G/S Capture and Track Control Law, the condition for switch SW3 was changed from "GSCD + 0.5 SEC" to "(GSCD + GSTD) + 0.5 SEC" to account for the possibility that G/S Track Mode is entered immediately after Altitude Hold Mode. Furthermore, the labels "A" and "B" were added to the appropriate inputs of divider D6.

As for the Inner Loop of the G/S Capture and Track Control Law, the gain constant above summer SU10 was changed to $\frac{1}{16}$, and the drawing note 1) was deleted from the rate limiters LR1 and LR2 to ensure that the rate limiters are only initialized when the Altitude Hold mode is entered. These changes were due to specification updates from H/S. Furthermore, changes in the corresponding text (in the part on the differences to the Altitude Hold Inner Loop and on initialization) had to be made to stay consistent with the changes made to figures 5.1 and 6.1. Last, the formulation "... resulting value of $\Theta_C(T) = \Theta_C(T-1)$ ..." was altered to "... resulting value $\Theta_C(T)$ approximately equals $\Theta_C(T-1)$ ..." because H/S found that the exact equality could not be guaranteed in every case. (Note: Similar changes to the text were also made in the description of the Flare Control Law Inner Loop.)

4.3.6 Changes in Chapter 7 (Flare Control Law)

In figure 7.1, the following changes were made to the Inner Loop of the Flare Control Law: the gain constant above summer SU10 was changed to $\frac{1}{16}$, and the drawing note 1) was removed from the rate limiters LR1 and LR2 (for the same reasons as before). Furthermore, changes in the corresponding text (in the part on the differences to the Altitude Hold Inner Loop and on initialization) had to be made to stay consistent with the changes made to figures 5.1 and 7.1.

4.3.7 Changes in Chapter 8 (Command Monitors)

In figure 8.1, Command Monitor, "≥" was replaced by "<" in the condition for function F13, because the output should be "true" if no error was detected, "false" otherwise. This error was due to a misunderstanding of what the actual meaning of the Command monitor output was.

4.3.8 Changes in Chapter 9 (Displays)

In the description of the displays "eight-bit field" was replaced by "seven-bit field" to remove an inconsistency between text and the corresponding figure. Furthermore, in the section on special requirements, the sentence "If the voter alters their values, the altered values must be left in the variables" was deleted because in the case of display no meaningful voting can be done, and the vote routine is just used to output the results of the Display routines.

4.3.9 Changes in Appendix A (System Diagram Symbols)

In Appendix A8, "Function", two changes were made: The word "filter" in the last sentence of the first paragraph was replaced by "function" because here we wanted to talk about functions in general, whereas a filter is just a special function. In the second paragraph, the sentence "The input X(T-1) is set to zero during the initialization of a linear filter" was added because this turned out to be generally true; so we decided to state it only once.

In Appendix A15, "Computation Procedures", the following paragraph was inserted as the first one: "Each integrator in the diagram should complete the integration process before the output is computed, so that the output includes the most recent input." Also, Appendix A15.1 and the following example were updated to meet this principle. Last, wrong constants "32.2" instead of "32" had been used in this example.

4.3.10 Changes in Appendix B (Parameter List Definitions of External Routines)

In Appendix B4, the test points "tp_9" and "tp_11" were added to the parameters of the routine VOTEMODE.

A forth note was added at the end of Appendix B, in order to make clear what should be done if some test point or a state variable is undefined: "Whenever some parameters in a voting routine are not defined or are not up-to-date because the corresponding part of the computation was not performed in the current frame, default values of 0.0 should be passed to that voting routine."

4.4 Other Questions and Clarifications

Answers to other questions that were not considered important for all the teams were sent only to the team that asked the question. The same applied for inquiries for more background information or for general discussions. Table 7 shows the total number of such questions asked by each team. Only questions related to specification issues are counted here. This number is not necessarily equal to the number of messages sent because sometimes more that one question was asked (and answered) in the same message, or sometimes it took more than one message to answer a question satisfactorily. One message per team was initiated by the coordinating team to provide some individual feedback from the Design Review. This message is disregarded in the subsequent discussion.

team	# questions raised by each team	# questions raised by coordinating team
ADA	20	1
C	12	1
MODULA-2	10	1
PASCAL	9	1
PROLOG	18	1
T	15	1
Total	86	6

Table 7: Number of Questions and Answers Replied Individually

The ADA team had many questions concerning the Mode Logic. Most of these, however, were just to confirm some poorly legible text on the figures pertaining to the Mode Logic. Some confusion was due to the fact that figure 4.2 was just a copy of part of figure 6.1, which led to some name conflicts, although it was stated in the specification that these two figures should be regarded as being independent from each other. Some questions related to the Inner Loop, specifically to the initialization of the Inner Loop state variables and to switch

SW2. Other specific questions were asked regarding limiter LM9 in the Flare Control Law, which is a special case because it does not have an upper bound, and to switch SW3 in the Glide Slope Capture and Track Control Law. Another concern was the use of the vote routines provided by the coordinating team. The rest of the questions was about miscellaneous topics.

One of C team's concerns was poorly legible text in the Mode Logic and Flare Control Law figures. Two of their three questions on the Command Monitor asked for a background explanation of some statements in the text of the specification. The question of "figure 4.2 versus figure 6.1" also came up. Specific questions were asked about limiter LM12 and function F9 in the Glide Slope Deviation Complementary Filter. The former is a special case because it uses dynamic upper and lower bounds, the latter was given in a graphical representation and was not so easy to read. One question was concerned with the initialization of a Flare Control Law variable because an intermediate result had to be used which was not available at the entry point of the Flare Control Law. The rest of the questions was about miscellaneous topics.

Most of the questions of the MODULA-2 team addressed the Mode Logic. Two of them were concerned with poorly legible text in the figures. The name conflict of test points between figures 4.2 and 6.1 (mentioned above) prompted another one. The other questions were due to uncertainties of how to compute (parts of) the Mode Logic algorithm. Most of the other questions were general questions on the use and implementation of the vote routines. Another questions was raised about the initialization of the rate limiters LR1 and LR2 in the Inner Loop.

The PASCAL team misunderstood at first the algorithm to compute a Linear Filter. Poorly legible text in some figures (Mode Logic, Flare Control Law) was again a problem. The use of default values for the vote routines (Mode Logic, Inner Loop) when a function is not computed also caused some questions. Initialization of state variables (Inner Loop, Command Monitor) was also one of their concerns.

The PROLOG team had many general questions on the correct way to handle feedback loops in the control laws and other functions. The way to compute the Mode Logic was unclear and the question of "figure 4.2 versus figure 6.1" also came up. The use of default values for vote routines (Mode Logic, Inner Loop) again raised some questions. Most of the other questions were general ones, specifically about vote routines.

The T team asked many general questions about the overall control loop, initialization, vote routines and test points, mode selection, and recovery. Poorly legible text in some figures (Radio Altitude, Glide Slope Deviation Complementary Filter, and Mode Logic) was once more a problem. The correct way to compute the Mode Logic was a major concern. Other questions related to the initialization and the correct way to compute the Inner Loop.

In summary, Table 8 gives the number of questions asked by each team for each major subfunction (or part of the specification).

Major Function	ADA	C	MODULA-2	PASCAL	PROLOG	T
Main Program	3	1	0	0	3	4
Complementary Filters	4	2	0	0	1	1
Mode Logic	7	2	5	2	4	2
Alt. Hold Outer Loop	0	0	0	1*	0	0
Glide Slope Outer Loop	1	0	0	1*	0	0
Flare Outer Loop	1	2	0	2*	0	0
Inner Loop	3	0	1	2	0	2
Command Monitor	0	3	0	1	0	0
Display	1	1	0	0	0	0
General, other	0	1	4	2	10	6
Total	20	12	10	9	18	15

*: There was one message related to the Outer Loop in general, so it was counted in all three rows.

Table 8: Number of Questions Answered by Each Team for Each Subfunction

Table 9 distinguishes the questions according to their content, i.e. if the questions asked about (some aspect of) initialization, about readability of text, the handling of feedback loops, general background information, etc.

Question Contents	ADA	C	MODULA-2	PASCAL	PROLOG	T
Initialization	2	1	1	2	1	2
Interpretation of Diagram, Algorithm	5	5	1	1	4	5
Handling of Feedback Loops	1	0	0	0	3	0
Main Program computation	2	0	0	0	2	0
Vote Routines	2	1	4	0	4	3
Readability	3	2	2	2	0	1
Background Information	0	2	0	0	0	1
Other	5	1	2	4	4	3
Total	20	12	10	9	18	15

Table 9: Distribution of Questions According to Their Content

4.5 Summary and Observations

Every team had difficulty reading some of the text in the figures (only some of the figures were criticized). Ironically, most of the trouble occurred with machine-typed numbers (or text) that became blurred or were reduced in size by frequent photocopying. In general, handwritten numbers, however, were big enough to be easily discernible . This problem occurred because the figures were not redrawn, but copies from H/S's System Description Document were used. The motivation behind this decision was convenience and the fear of introducing errors (typos) during the the process of regenerating the figures. What should have been done was a "figure walk-through" in order to ensure that all numbers were clearly readable. Several teams (ADA, PASCAL, PROLOG) complained about the readability of numbers in their Final Reports.

Another major drawback was that the part of the specification that tried to give a general rule on how to handle feedback loops was not very precise and also not exactly correct in the first version of the specification. Quite a few explanations and specification updates were necessary to remedy the situation. It was especially disturbing because this rule is central to develop almost all the algorithms of the problem and thus caused some delay in the Design Phase. Some teams even had to re-develop some of their algorithms according to new rules. The problem most likely was due to the fact that the coordinating team at the time of writing the specification was already fairly familiar with the application and did not notice this poorly defined guideline. Again, several teams (PASCAL, PROLOG) mentioned that problem in their Final Reports.

The Mode Logic presented a big problem, too. The algorithm was specified as an interconnection of logical AND and OR gates. However, had it been a piece of hardware, a race condition would have occurred. Stated in terms of software, one output variable was defined as a function of itself. Originally, H/S was reluctant to change the specification because all diagrams used were "certified". In the end, H/S and the coordinating team agreed on a correct way to compute the Mode Logic, and the programming teams were simply told how to implement it. Nevertheless, this procedure probably caused all teams to develop the algorithm in a very similar manner – thus reducing some opportunity for diversity.

In two cases, the problem was that the graphic language used did not have the power to express a complicated condition correctly. This occurred with the conditions for switch SW2 in the Inner Loop and for switch SW3 in the Outer Loop of the Glide Slope Capture and Track Control Law.

To determine the position of switch SW2 the rate of a certain signal had to be evaluated. As the past value of that signal, the history of the rate limiter LR1 was supposed to be used. However, there was no means of depicting this interrelationship, and many teams just used the signal value itself instead of its rate.

Switch SW3 was to be closed 0.5 seconds after the system entered Glide Slope mode, and had to be open before that time. This was represented as "(GSCD + GSTD) + 0.5 sec", which led to some confusion – probably because the first "+" represented the logical OR and the second "+" was meant to say "after the preceding logical condition". Another problem was that it was not clear whether the switch should remain closed after the specified time, or not. In both cases, additional paragraphs were added to the specification to define these special cases.

When trying to determine the cause of committing the errors found during the test phases, one must come to the conclusion that there are at least three more errors in the specification (apart from typos):

A) In the Mode Logic (pp. 18-21), it is not specified that the variables FPEC1 and FPDC1 have to be set to zero when the system is not in Altitude Hold mode. This omission occurred because originally it had been decided that it should be up to the programmers whether to compute FPEC1 and FPDC1 during modes other than Altitude Hold or not. But later it turned out that a uniform decision has to be required to avoid unnecessary error signals from the cross-check routines. (For example, if one version always computes updated values of FPEC1 and FPDC1 and another version stops computing these values after Altitude Hold mode, these two versions will – most likely – disagree at the cross-check point that checks FPEC1 and FPDC1.)

B) On page 30, the method for transitioning from Altitude Hold mode to Glide Slope Capture and Track mode is not described correctly, because it is possible that the system goes from Altitude Hold directly to Glide Slope Track mode. Thus the transition should be described as follows:

(1) Compute the states of GSCD, Glide Slope Capture Discrete, and GSTD, Glide Slope Track Discrete.

(2) If GSCD = 0 and GSTD = 0, proceed with Altitude Hold mode computation.

(3) If GSCD = 1 or GSTD = 1

(... rest as before ...)

C) On pages 30 and 35 it is not specified that the input of integrator I1 in the Inner Loop must be re-initialized with zero upon the mode transitions from Altitude Hold to Glide Slope Capture and Track and from Glide Slope Capture and Track to Flare. This omission is due to a deletion during a specification update.

In general, many participants in the experiment felt that the specification was very detailed, strict, and pretty much partitioned the problem. While this saved some development time, it was considered unlikely on the other hand that the resulting programs could be in any significant way "different". This observation is certainly correct; it is due to the nature of the application. Partitioning according to function seemed very natural, and so each function was described in a chapter of the specification. The sequence of computing these functions was also almost completely determined by the data dependencies between them. And since every function was small enough so that it did not need further partitioning, each function became essentially a software module. We feel that not much could have been done about it, or how does one unpartition a specification? It remains to be evaluated how much diversity there nevertheless is between different programs.

5.2 Faults Detected during Program Development

A total of 82 faults was found and reported during program development. The following four tables present the distribution of these faults in the six versions under different categories. Detailed description of all these faults is shown in Appendix III.

Table 11 shows the fault distribution in each system function. The total adds up to more than 82 since all the modules affected by one fault are counted. An asterisk indicates such a case.

Classification of faults according to fault types is shown in Table 12. This category considers the following type of faults: (1) typographical; (2) error of omission (missing code); (3) unnecessary implementation (which was deleted); (4) incorrect algorithm; (5) specification misinterpretation; and (6) specification ambiguity. "Incorrect algorithm" is the most frequent fault type, which includes miscomputation, logic fault, initialization fault, and boundary fault.

Metric	ADA	C	MODULA-2	PASCAL	PROLOG	T
LINES	2253	1378	1521	2234	1733	1575
STMTS	1031	746	546	491	1257	1089
LN-CM	1517	861	953	1288	1374	1263
OBJS	85.6k	83.7k	51.9k	37.5k	N.A.	N.A.
MODS	36	26	37	48	77	44
STM/M	29	25	15	10	16	25
CALLS	97	68	65	93	81	87
LIBS	2	2	2	10	7	N.A.
LCALL	3	9	6	12	61	N.A.
GBVAR	139	141	91	81	90	97
LCVAR	117	197	132	127	209	251
CONST	68	21	18	16	N.A.	N.A.
BINDE	74	114	78	118	74	86

N.A. = not applicable

Table 10: Software Metrics for the Six Programs

	ADA	C	MODULA-2	PASCAL	PROLOG	T	Total
Main Program	1	2	0	0	7	6*	16
BACF	1	0	1	0	2*	3*	7
RACF	0	0	0	0	1*	1*	2
GSCF	1	1	1	4	7	2*	16
Mode Logic	1	4	0	0	1*	2*	8
Alt. Hold Outer Loop	0	0	0	1*	0	0	1
Glide Slope Outer Loop	0	1	0	0	0	0	1
Flare Outer Loop	1	2	0	1	2*	1	7
Inner Loop	1	3	0	4*	4*	2	14
Command Monitor	0	0	0	0	1	2	3
Display	0	0	1	3	1	1	6
General, other	0	0	1	0	5*	4	10
Total	6	13	4	13	31	24	91

*: This fault affected more than one subfunction.

Table 11: Fault Distribution by Subfunctions

	ADA	C	MODULA-2	PASCAL	PROLOG	T	Total
Typo	0	1	0	0	9	0	10
Omission	1	3	0	0	8	5	17
Unnecessary	1	0	0	1	0	2	4
Incorrect Algorithm	3	5	2	7	9	13	39
Spec. Misinterpretation	1	3	1	4	0	1	10
Spec. Ambiguity	0	1	0	0	0	0	1
Other	0	0	1	0	0	0	1
Total	6	13	4	12	26	21	82

Table 12: Fault Classification by Fault Types

Table 13 shows during which phases of testing the faults were detected. A finer granularity for classification of the Integration Testing and Acceptance Testing phases is presented in Table 14, where each error is associated with the test data detecting it. The item of "other" indicates those errors independent of the test data, but nevertheless found in these phases (e.g., by code inspection).

	ADA	C	MODULA-2	PASCAL	PROLOG	T	Total
Coding/Unit Testing	2	4	4	10	15	7	42
Integration Testing	2	5	0	2	7	4	20
Acceptance Testing	2	4	0	0	4	10	20
Total	6	13	4	12	26	21	82

Table 13: Fault Classification by Phases

	ADA	C	MODULA-2	PASCAL	PROLOG	T	Total
Integration/data.1	1	3	0	2	1	1	8
Integration/data.2	1	2	0	0	0	0	3
Integration/data.3	0	0	0	0	0	0	0
Integration/data.4	0	0	0	0	0	0	0
Acceptance/data.1	0	1	0	0	0	0	1
Acceptance/data.2	2	2	0	0	1	2	7
Acceptance/data.3	0	0	0	0	0	1	1
Acceptance/data.4	0	0	0	0	1	0	1
Acceptance/data.5	0	0	0	0	0	0	0
Acceptance/data.6	0	0	0	0	0	0	0
Acceptance/data.7	0	0	0	0	0	3	3
Acceptance/data.8	0	1	0	0	1	0	2
Acceptance/data.9	0	0	0	0	0	0	0
Subtotal	4	9	0	2	4	7	26
Other	0	0	0	0	7	7	14
Total	4	9	0	2	11	14	40

Table 14: Error Classification by Test Data

Finally, Table 15 shows the classification of faults according to the categories of "requirements fault" and "structural fault" (see section 2.4).

	ADA	C	MODULA-2	PASCAL	PROLOG	T	Total
Requirements	5	12	3	11	21	19	71
Structural	1	1	1	1	5	2	11
Total	6	13	4	12	26	21	82

Table 15: Fault Classification: Requirements Faults vs. Structural Faults

5.3 Additional Observations

All cross-check and recovery point routines were written in the C programming language, and therefore five of the six programs had the additional problem of interfacing to another language. The Prolog and the T team had the most severe problems. The Prolog team had to modify the Prolog interpreter; the solution of the T team was to convert all parameters to ASCII strings, pass them to a C routine, convert them back into numbers, do the cross-checking, convert the results into strings, and pass them back to the T functions.

Three compiler or interpreter bugs were found during program development: the Ada compiler did not support nested generic packages (which resulted in a design change to avoid using this feature). With the Modula-2 compiler the expression "i+i" had to be used as an array index instead of "2*i" to achieve the desired result. This fault is classified as the type "other" in Table 12. The T interpreter had a problem with its garbage collection which resulted in uncompleted long test runs. This problem delayed the T program's passing of the acceptance test for over a month.

In addition, we experienced a computing environment change during the experiment. This did cost some time, but finally all teams were moved to the new Sun workstations. Only the Modula-2 team had to continue to use the original VAX computers, due to their compiler not being available on the Sun.

It is interesting to note that there was *only one* incidence of an identical fault, committed by two teams, ADA and MODULA-2. In both cases the fault was discovered during unit testing. The fault was the following: the output of an integrator in the Barometric Altitude Complementary Filter must be limited by 65,536. Both teams mistook the comma after the 1000's place for a decimal point and used the constant 65.536. We are not sure whether to classify that fault as a typo or a specification misinterpretation. Although we think that this particular number is easily readable, the example still shows that it is dangerous to provide hand-written numbers in a specification.

In conclusion, we believe that the number of faults found indicates that all six programs were quite thoroughly tested before they were accepted.

6. Testing and Evaluation After Acceptance of the Versions

Requirements-based stress testing and structural analysis are the two employed approaches for the evaluation of the six programs. The efforts of finding more faults (requirements-based or structure-based) and the search for evidence of structural diversity among these programs have been the major concerns.

For the purpose of industrial-standard validation and verification, a Model Definition Document was supplied by H/S to provide mathematical models for functions within the landing/approach control loop, but external to the control laws defined in the System Description Document. These models were programmed by the UCLA coordinating team to provide a suitable control problem for the experiment. Two program versions of the aircraft models, one in C and the other in Pascal, were independently generated. They were rather short programs of about 100 lines of code. Nevertheless, "back-to-back" testing between these two versions effectively revealed a bug in one of them. These versions were later certified by H/S personnel. Generation of input data and interpretation of the results were also performed and suggested by H/S experts.

Based on these tools, the UCLA coordinating team has been conducting H/S approved "Level 2" stress testing for months since the software generation phase was completed in early September 1987. The major strategy in this requirements-based testing is so-called "dynamic closed-loop" tests, which have the purpose of verifying performance, detecting any tendency towards dynamic mistracking between the different program versions, and exposing requirements faults not caught in static testing. In practice, the 3 channels of diverse software each compute a surface command to guide a simulated aircraft along its flight path. To ensure that significant command errors could be detected, random wind turbulences of different levels are superimposed. The individual commands are recorded and compared for discrepancies which could indicate the presence of faults.

The configuration of the flight simulation system (shown in Figure 3) consists of three lanes of control law computation, three command monitors, a servo control, an airplane model, and a turbulence generator.

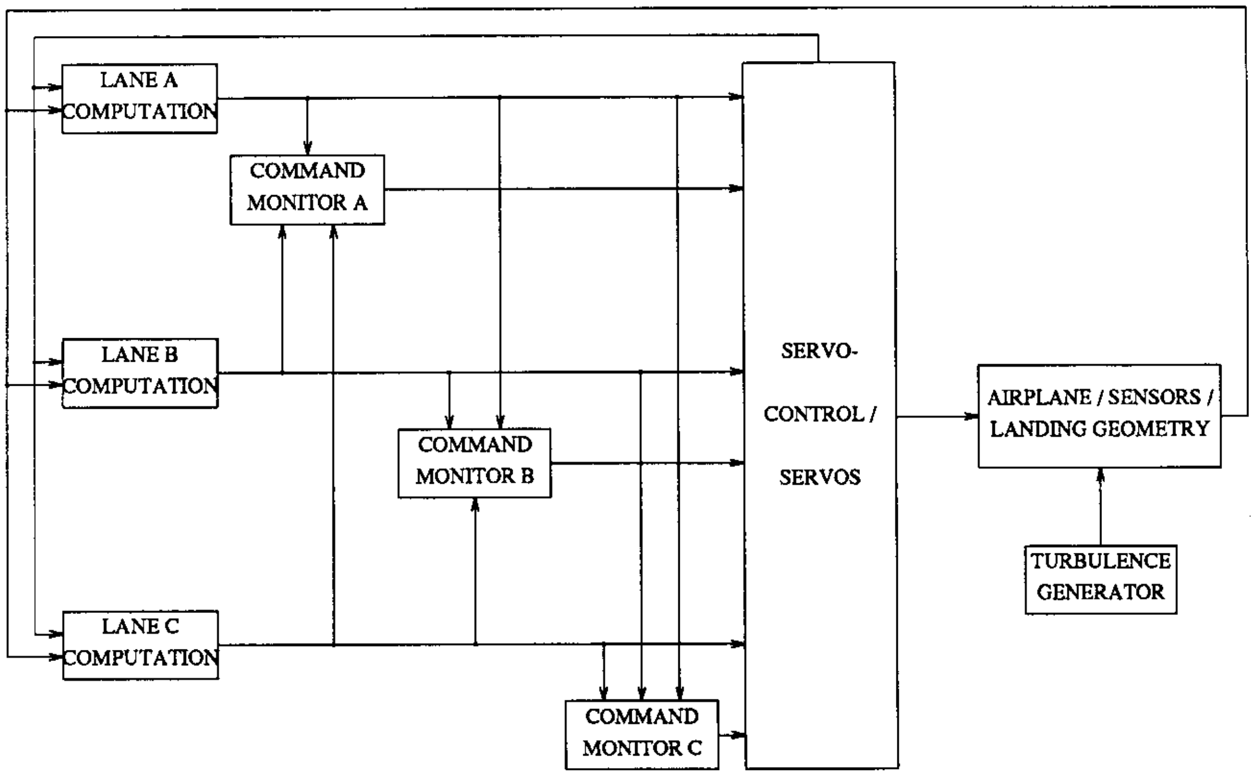


Figure 3: 3-Channel Flight Simulation Configuration

The *lane computations* and the *command monitors* are the redundant software versions generated by the six UCLA programming teams. Each lane of independent computation monitors the other two lanes. However, no single lane can make the decision as to whether another lane is faulty. A separate servo control logic function is required to make that decision, based on the monitor states provided by all the lanes. This control logic is based on a strategy that ignores the elevator command from a lane when that lane is judged failed by both of the other lanes, and these lanes are judged valid.

The airplane is a mathematical model that computes the response of the airplane to an elevator command in terms of attitude, attitude rate, flight path, altitude, altitude rate, and vertical acceleration. In a real aircraft these values would be directly measured by sensors. The landing geometry model describes the deviation from the glide slope beam center as a function of aircraft position relative to the end of the runway. Moreover, in order to provide a set of inputs to the airplane model which create large error magnitudes, and thereby force off-nominal software operating conditions, turbulence in the form of vertical wind gusts is introduced.

One run of flight simulation is characterized by the following five initial values: (1) initial altitude (about 1500 feet); (2) initial distance (about 52800 feet); (3) initial nose up relative to velocity (range from 0 to 10 degrees); (4) initial pitch attitude (range from -15 to 15 degrees); and (5) vertical velocity for the wind turbulence (0 to several ft/sec). One simulation consists of 5000 time frame computations of 50 msec/frame, for a total landing time of 250 seconds.

For the purpose of efficiency, a testing procedure equivalent to Figure 3 was used (approved by H/S): first, each lane by itself guided the airplane for a complete landing; second, the whole history of the flight simulation was recorded; and finally, the flight profiles of all versions were compared and analyzed to observe discrepancies and determine faults. In this manner, over 1000 flight simulations (over 5,000,000 time frames) have been exercised on the six software versions generated from this project.

In addition to the flight simulations, a structural analysis also was carried out. The six versions were compared to find the differences in structure and implementation that resulted from the application of the N-version programming methodology [Schu87]. An additional benefit of this analysis was that it necessitated a thorough code inspection, during which some additional faults that were not caught by any tests were detected.

7. Results of Testing and Evaluation

7.1 Disagreements Detected by Flight Simulations

So far, four disagreements at the Inner Loop cross-check point have been detected during the flight simulations. Due to the additional information provided by the test points, it was relatively easy to determine the faulty part of the code in each case. The C version experienced two disagreements. The first one resulted in the detection of two faults, namely initialization with a wrong value (an intermediate value of the present time frame computation was used instead of a result of the previous time frame computation), and the introduction and use of an unnecessary state variable. This latter fault is related to the "underground variables" discussed in the next section; the only difference is that in this case the fault caused a disagreement. This fault is traceable to an ambiguity in the specification: the graphical language used was not powerful enough to express the exact semantics of the required operation. The third fault discovered in the C version is the too frequent initialization of a state variable (it is re-initialized at every pitch mode change, while it should be initialized only once at the entry of Altitude Hold mode). In this case, the team did not follow a specification update that was made very late in the programming process (during integration testing).

Two disagreements were traced to an identical fault; they occurred in the Prolog and T versions. Both teams made the same design decision to update a state variable of the Inner Loop twice during one computation of the Inner Loop. This fault is due to the same specification ambiguity as mentioned above, but in addition these teams did not pay attention to a broadcast clarification that addressed exactly that problem. Although similar in nature, the two versions disagreed in slightly different ways from the other versions.

It is noteworthy that all observed disagreements were very small, and further experiments showed that the versions with these discrepancies are always able to achieve proper

Touchdown. Furthermore, all these faults are specification related. It is interesting to note that the Inner Loop was the program part that was most thoroughly tested during all test phases.

7.2 Faults Found During Inspection of Code

The following faults were detected during the code inspection performed as part of the structural analysis (see section 5):

One *requirements fault* was found in the Display, where rounding to 5 significant digits was not done correctly. The error occurs only when rounding overflow (e.g., 6 or more subsequent 9's) changes the decimal point position. This special case was not triggered by any of the acceptance test or flight simulation data. Other teams, however, had discovered the same kind of fault during unit testing. Therefore one explanation might be that this team did not perform the unit test sufficiently carefully.

The other six faults were three types of structural faults, discovered in the C, Modula-2, Pascal, Prolog, and T versions. They and their possible impacts are discussed next.

One fault was Type 1, as described next. Normally, the boundaries within which the output of certain functions (integrator, rate limiter, and magnitude limiter) had to be limited was a finite constant. There were a few cases (in the Inner Loop and the Command Monitor), however, where the bound was either $+\infty$ or $-\infty$. To implement these special cases, the C version used the arbitrarily chosen values +99999.0 or -99999.0 and passed them as parameters to the subprogram that implements the functions mentioned above. This is a structural fault because an unintended (unspecified) function (i.e. the limiting of an output value) is performed if this value exceeds the arbitrarily chosen values. In this application, however, this might not be a problem since the output of the Inner Loop (elevator command) will be further limited to ± 15 degrees. Similarly, the Command Monitor will indicate a disagreement between two versions long before this structural fault has any effect.

Type 2 faults are more serious. They are caused by the introduction of new, unspecified state variables which we call "underground variables", since they are neither checked nor corrected in any cross-check or recovery point. This may lead to an inconsistent state which is impossible to recover from. An example follows: the C team decided to move the computation of some parameters for the Glide Slope Deviation Complementary Filter outside of this Filter. Unfortunately, this computation depends on some other, state dependent computations in this Filter. These latter computations were re-implemented outside the Glide Slope Deviation Complementary Filter which also led to a duplication of their state variables. Therefore, a new design rule for multi-version software must be stated as "Do not introduce any 'underground' variables". Note that this rule is irrelevant if only cross-check points are used, since these do not attempt to recover the internal state of the version. Only one Type 2 fault was uncovered.

Type 3 faults occurred when the C, Modula-2, Prolog, and T teams used the output of the Mode Logic in some further (but different!) computations before it was voted upon. This was in violation of a rule stated in the specification, explicitly forbidding that. If the Mode Logic output is corrected by the Decision Function, an fault of this kind could lead to a situation where the Mode Logic output is correct, but the variables dependent on this output are not, since they were computed using the old, uncorrected values of the Mode Logic output. Then an inconsistent state between different variables of the version might exist which could be impossible to recover from. Apparently, more programmer training is necessary to prevent these types of mistake since the reason for this fault is obviously a misunderstanding or unawareness of some of the multi-version software design rules. Although this might seem a dangerous possibility of introducing common faults, faults of this kind are easily checked for. Thus they can be eliminated by the acceptance test. We conclude that the acceptance test should always check for compliance with all the N-version software design rules specified.

The six discovered structural faults that are described above are uncorrelated, and thus will be tolerated by the multi-version software approach.

7.3 Assessment of Structural Diversity

A fundamental first step in assessing the diversity that is present in a set of versions must be an assessment of the *potential for diversity* (PFD) that is indicated by a given specification. Some reasonable evidence that *meaningful diversity* can occur is needed in order to justify the effort of multi-version programming. Here we exclude the "pseudo-diversity" that can be attained by rearranging code, using simple substitutions of identities, etc. It is introduced too late in the programming process to be effective, and is likely to replicate and camouflage already existing faults.

After the PFD assessment, a decision must be made whether certain diversity shall be "enforced", i.e., specified; examples would be a requirement to use different algorithms [Chen78], several versions of the specification [Kell83], different compilers, programming languages, etc. The alternative is to depend on the isolation between programmers and on the differences in their backgrounds and approaches to the problem as the means to get diversity. This is the "random" approach to the attainment of diversity.

It is our position that the minimal requirement must be (1) the isolation of programming efforts, and (2) "enforced" diversity that is needed to avoid predictable causes of common faults, such as compiler bugs and other defects that could exist in a shared support environment.

In the present investigation the only additional choice of "enforced" diversity is the use of six different programming languages. One of our goals is to evaluate the effectiveness of this choice in attaining meaningful diversity between the six versions that originated from one specification. A summary of the observations follows.

Program Module	PFD	Observed Diversity
Main Program	good	level of detail implemented, information handling, organization of state variable initialization, placement of calls to vote routines
Radio Altitude Complementary Filter	poor	grouping, sequence
Barometric Altitude Complementary Filter	medium	grouping, sequence
Glide Slope Deviation Complementary Filter	medium	grouping, sequence, time-dependent computation
Mode Logic	good	constants, sequence, algorithm
Altitude Hold Control Law, Outer Loop	poor	constants, grouping, sequence
Glide Slope Capture and Track Control Law, Outer Loop	good	constants, grouping, sequence, time-dependent computation
Flare Control Law, Outer Loop	good	constants, grouping, sequence
Inner Loop	poor	constants, grouping, sequence, organization
Command Monitor	poor	grouping, algorithm, organization
Mode Display	poor	algorithm
Fault Display	poor	algorithm
Signal Display	medium	algorithm
Primitive Operations	poor	choice, organization

Table 16: Potential for and Observed Diversity

The "PFD" column of Table 16 gives our assessment of the extent of diversity (structural differences) that may be expected for each program module. A module has "poor" potential for diversity if it is either so small and simple, or else if its computation sequence (in terms of primitive operations) is so well-defined by data dependencies, that there is little room for diversity in implementation and organizational aspects. In the modules with "good" potential for diversity, many (between 5 and 10) independent computation paths exist which could be traversed in any order. In the case of the Main Program the sequence of the major system functions is determined by data dependencies (cf. Figure 1); here the PFD lies in the organizational aspects. Modules with "medium" PFD are estimated to lie somewhere between these two limiting cases. It must be noted that the PFD assessment is somewhat subjective; the factors used in the assessment include the specification of each program module as well as the observed structural differences.

The column "Observed Diversity" of Table 16 lists the attributes in which structural diversity actually was observed between two or more of the six versions. Further explanations and comments on this column follow.

The first notable difference between the Main Programs is the level of detail implemented there. The Ada version is one extreme example; it deals with all the organizational details, such as initialization of state variables, or determination of which function to perform at a given instant, in the Main Program. This leads to a calling hierarchy which is exactly one level deep, if some auxiliary subprograms and the calls to primitive operations are ignored. The T version is similar in the sense that all the system functions are called directly by the Main Program. However, most of the organization (especially initialization of state variables) is done locally by these system functions. The other versions (C, Modula-2, Pascal) generally show a two-level calling hierarchy, i.e., they define relatively general subprograms like "Filter Module", "Mode Logic", or "Altitude Hold Control Law", and deal with the organization of the appropriate system functions locally. Nevertheless, there are some differences between these latter versions, too. For instance, the C and Modula-2 versions organize the Control Laws into three different Control Laws (one for each pitch mode), each consisting of an Outer and an Inner Loop. The Pascal version, on the other hand, divides the Control Laws into an Outer and an Inner Loop, where the Outer Loop consists of three different Outer Loop procedures. Finally, the C and Modula-2 versions differ also in the organization of their Filter Module, or their Mode Logic. The Prolog version is a special case. It has a rather large and complex calling hierarchy because the language is such that IF-statements have to be implemented by function calls.

Another important difference that was noted is the strategy chosen to handle information, i.e., state, interface, and output variables. Solutions range from extensive parameter passing (Pascal) to the exclusive use of global variables (C, Prolog). We note that this choice was unavoidable for the Prolog version because of the language properties. The other versions use solutions between these two extremes, by trying to define as many variables as possible locally. The choices are partly programming language dependent, e.g., dependent on the availability of local static variables. A related aspect is the organization of state variable initialization: the two basic solutions are initialization by the Main Program, or initialization within each program module.

The third aspect of diversity in the Main Program concerns the placement of vote routines: either all vote routines are called in the Main Program, or they are called in the system function whose result they check. The recovery point routine, however, is always called by the Main Program.

The notation "constants" in Table 16 indicates that some teams chose to simplify the computation by manually evaluating some expressions consisting of constants only. "Grouping" refers to the fact that different teams chose different ways of combining primitive operations into statements of their programming language. "Sequence" denotes that some versions use a different computation sequence (in terms of primitive operations) to implement a system function than others. Sometimes the differences are very minor, for instance in the Outer Loop of the Altitude Hold Control Law or in the Inner Loop.

"Time-dependent computation" means that this system function contains an algorithm that is dependent on real time. In both cases, we observe much variety among the strategies chosen (1) to keep track of real time; and (2) to guard against effects of limited precision of real number representation. (Note: Real time was simulated in this application.)

"Algorithm" indicates that different versions use different algorithms to implement a certain system function. These differences are mostly minor ones; only in the Signal Display more interesting differences can be found, both in the structure of the algorithm and in implementation details.

"Organization" refers mainly to the fact that some versions chose to implement a certain subprogram as a procedure (results are returned via parameter passing), while other versions used a function (a RETURN statement or similar construct is used). In the case of the Inner Loop, slightly different requirements existed for different pitch modes. A variety of solutions to cope with these has been found.

Primitive operations are integrators, linear filters, magnitude limiters, and rate limiters. The algorithms for these operations were exactly specified, however, different choices of which primitive operations to implement as subprograms have been made, mainly whether the integrators include limits on the magnitude of the output value (as is required in most cases), or not. Only the Prolog version implemented a "switch" subprogram; this choice has clearly been influenced by programming language properties – all other versions just use IF-statements. The T version defined only a subprogram for magnitude limiting, all other primitive operations are implemented directly in each system function. The Prolog version uses procedures to implement these operations, all other versions use functions. Lastly, the Ada functions also do the state update, in the case of state dependent primitive operations; all other versions have to do this within each system function.

Due to space constraints, no examples could be given here. These and more details can be found in [Schu87]. In conclusion, we note that both the PFD assessment and the search for meaningful structural differences were based on individual judgements of the investigators and are somewhat subjective. However, it is evident that (1) aspects of meaningful diversity can be identified; and (2) diversity in programming languages definitely motivates structural diversity between the versions. We hope that our modest first steps will stimulate further investigations into the problems of qualitative and quantitative assessment of meaningful diversity in a set of program versions.

7.4 Observations from the Diversity Assessment

In general, it can be said that more diversity was observed in the aspects whose method of implementing was not explicitly stated in the specification, such as the Signal Display, the organization of different Inner Loop algorithms (depending on the pitch mode), the organization of state variable initialization, or the implementation of time-dependent computations. Furthermore, not all the design choices outlined above can be made independently. For instance, whether a primitive operation is defined as a function or a procedure determines if it can be combined with other operations in a single statement, or not. Similarly, if the update of state variables is performed as part of the primitive operation the upper levels do not have to be concerned with this. As a last example, if the state variables of a system function are defined as local static variables, then they cannot be initialized by the Main Program.

Two factors that limit actual diversity have been observed in the course of this assessment. One of them is that programmers obviously tend to follow a "natural" sequence, even when coding independent computations that could be performed in any order. The observation made was that algorithms specified by figures were generally implemented by following the corresponding figure from top to bottom. In this case the "natural" order was given by the normal way to read a piece of paper, i.e. from left to right and from top to bottom. Only when enforced by data dependencies, a different order was chosen, e.g. from bottom to top. It can be safely assumed that the same phenomenon would occur if the specification was stated in another form than graphical; especially this is true for a textual description. The latter can be exemplified by the Display Module: Only one team chose the order of computation Fault Display, Mode Display, Signal Display; all other teams chose the order Mode Display, Fault Display, Signal Display which was also the order used in the specification. That means that if there is a number of independent computations that could be performed in any order there exist some permutations of these computations that are more likely to be chosen than other permutations, due to human, psychological factors.

The Outer Loops of the Glide Slope Capture and Track and the Flare Control Law, and the Mode Logic were affected the most; their good potential diversity was not exploited as much as expected and possible, due to this phenomenon. In retrospect, a second reason for this lack of diversity is that we have concluded that the logic part of the Mode Logic was overspecified. A description of the conditions that have to be met to enter the next pitch mode would have been more appropriate than the logic diagram which biased the programmers too much towards using identical or very similar algorithms.

One possible solution to the "natural" sequence problem is to provide different specifications to individual teams. They could either be required to follow a specific unique computation sequence, or the order of presenting the independent computations could be different in each specification while still having each team decide which sequence to follow. The problem of this approach is the possibility of introducing additional faults into the specification, i.e., more faults than would have been made in a single specification, unless the process of generating different versions of a specification can be proven to be correct.

The H/S concept of "test points" is the second factor that tends to limit diversity. Their purpose is to output and compare not only the final result of the major subfunctions, but also some intermediate results. However, that restricted the programmers on their choices of which primitive operations to combine (efficiently!) into one programming language statement. In effect, the intermediate values to be computed were chosen for them. These restrictions are rather unnecessary and can easily be removed. An additional benefit is that output and the use of vote routines would become simpler. On the other hand, the test points proved very beneficial in version debugging. A way to preserve this useful feature is to add test points only during the testing and debugging phase, and to remove them afterwards. Each team should be free to choose its own test points; in addition, the program development coordinator can request specific test points if it is intended to compare the results of two or more different versions.

7. Conclusions

The major conclusions of this study are:

(1) The UCLA paradigm for systematic generation of multiple-version software is sufficiently complete and stable for application in industrial environments.

(2) The use of different programming languages has supported very effective inter-team isolation, since different support environments were used. It also has promoted the appearance of diversity in versions that began with a common specification.

(3) Identical faults in two versions were very rare. Only one identical pair existed in the 82 faults removed from the six versions before acceptance - and it was due to a comma being misread as a period. During post-acceptance testing and inspection, five faults were uncovered by testing. One pair again was identical, and this fault was due to failure to properly incorporate a clarification to a specification ambiguity. Six more faults were discovered by code inspection, all unrelated and different.

(4) The "Type 3" structural faults (section 6.2) are due to disregard of clearly stated multi-version software design rules. They are potentially identical and therefore dangerous. This is also true of the Type 2 "underground variable" fault. Very strict verification that design rules were followed must be a part of the acceptance test.

(5) The order of computations that is implied by the specification has a strong influence on the programmers' choice, even if other alternatives exist. This is especially true of graphical specifications used in this effort. "Test points" given in the specification also tend to limit diversity. There is a need to develop effective means to minimize these diversity-limiting factors.

(6) The original specification, as received from H/S, contained too much information on implementation issues, which would tend to limit diversity. Our concentrated effort to reduce the specification as much as possible to the "what", removing the "how", paid off by encouraging diversity.

We also note that we found only two identical faults that cause similar errors, described in (3) above. This is very different from previously published results by Knight and Leveson [Knig86]. Upon reviewing that reference, we conclude that there are several significant differences: the previous problem had limited potential for diversity, the programming process was rather informally formulated, testing was limited, and the acceptance test was totally inadequate according to industrial standards that we have followed. For this reason, our conjecture is that a rigorous application of the paradigm described in this paper would have led to the elimination of most faults described in [Knig86] before acceptance of the programs.

References

[Aviz82] A. Avižienis, "Design Diversity - The Challenge for the Eighties," in *Digest of 12th Annual International Symposium on Fault-Tolerant Computing,* Santa Monica, California: June 1982, pp. 44-45.

[Aviz84] A. Avižienis and J.P.J. Kelly, "Fault-Tolerance by Design Diversity: Concepts and Experiments," *Computer,* Vol. 17, No. 8, August 1984, pp. 67-80.

[Aviz85a] A. Avižienis, P. Gunningberg, J.P.J. Kelly, L. Strigini, P.J. Traverse, K.S. Tso, and U. Voges, "The UCLA DEDIX System: A Distributed Testbed for Multiple-Version Software," in *Digest of 15th Annual International Symposium on Fault-Tolerant Computing,* Ann Arbor, Michigan: June 1985, pp. 126-134.

[Aviz85b] A. Avižienis, "The N-Version Approach to Fault-Tolerant Software," *IEEE Transactions on Software Engineering,* Vol. SE-11, No. 12, December 1985, pp. 1491-1501.

[Bish86] P.G. Bishop, D.G. Esp, M. Barnes, P. Humphreys, G. Dahll, and J. Lahti, "PODS - A Project of Diverse Software," *IEEE Transactions on Software Engineering,* Vol. SE-12, No. 9, September 1986, pp. 929-940.

[Chen78] L. Chen and A. Avižienis, "N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation," in *Digest of 8th Annual International Symposium on Fault-Tolerant Computing,* Toulouse, France: June 1978, pp. 3-9.

[Gmei79] L. Gmeiner and U. Voges, "Software Diversity in Reactor Protection Systems: An Experiment," *Proceedings IFAC Workshop SAFECOMP'79,* May 1979, pp. 75-79.

[Kell83] J.P.J. Kelly and A. Avižienis, "A Specification Oriented Multi-Version Software Experiment," in *Digest of 13th Annual International Symposium on Fault-Tolerant Computing,* Milan, Italy: June 1983, pp. 121-126.

[Kell86] J.P.J. Kelly, A. Avižienis, B.T. Ulery, B.J. Swain, R.T. Lyu, A.T. Tai, and K.S. Tso, "Multi-Version Software Development," in *Proceedings IFAC Workshop SAFECOMP'86,* Sarlat, France: October 1986, pp. 43-49.

[Knig86] J.C. Knight and N.G. Leveson, "An Experimental Evaluation of the Assumption of Independence in Multiversion Programming," *IEEE Transactions on Software Engineering,* Vol. SE-12, No. 1, January 1986, pp. 96-109.

[Li87] H. F. Li and W. K. Cheung, ''An Empirical Study of Software Metrics,'' *IEEE Transactions on Software Engineering,* Vol. SE-13, No. 6, June 1987, pp. 697-708.

[RTCA85] RTCA, Radio Technical Commission for Aeronautics, ''Software Considerations in Airborne Systems and Equipment Certification,'' Technical Report DO-178A, Washington, D.C., March 1985. Order from: RTCA Secretariat, One McPherson Square, 1425 K Street, N.W., Suite 500, Washington, DC 20005.

[Schu87] W. Schuetz, ''Diversity in N-Version Software: An Analysis of Six Programs,'' Master Thesis, UCLA Computer Science Department, Los Angeles, CA, November 1987.

[Trea82] J. J. Treacy, ''Certification of Digital Avionics: A Review of Recent FAA Experience,'' in *Aerospace Congress and Exposition,* Anaheim, California: October 1982, pp. 3-7.

[Tso87] K.S. Tso and A. Avižienis, ''Community Error Recovery in N-Version Software: A Design Study with Experimentation,'' in *Digest of 17th Annual International Symposium on Fault-Tolerant Computing,* Pittsburgh, Pennsylvania: July 1987, pp. 127-133.

[Will83] J. F. Williams, L. J. Yount, and J. B. Flannigan, ''Advanced Autopilot Flight Director System Computer Architecture for Boeing 737-300 Aircraft,'' in *Proceedings Fifth Digital Avionics Systems Conference,* Seattle, WA: November 1983.

APPENDIX I. Current Version of the Software Specification

UCLA / HONEYWELL JOINT PROJECT

FCS DESIGN UTILIZING N-VERSION PROCESSING

SOFTWARE REQUIREMENTS DOCUMENT

FOR A

FLIGHT CONTROL COMPUTER

September–4–1987

Version 1.3

TABLE OF CONTENTS

page

LIST OF FIGURES

Pitch modes entered by the autopilot-airplane combination, in the landing process, are Altitude Hold, Glideslope Capture and Track, Flare, and Touchdown. Mode entry and exit is determined by the mode logic equations, which use filtered airplane sensor data to switch the controlling equations at the correct point in the trajectory.

Flight begins with the initialization of the system in the Altitude Hold mode, at a point approximately ten miles from the airport. Initial altitude is about 1500 feet, initial speed 120 knots (200 feet per second), with zero flight path angle. Responding to turbulence-induced errors in attitude and altitude with automatic elevator control motion, the aircraft maintains the reference altitude until the edge of the glideslope beam is reached.

If the capture conditions are met, the airplane enters the Capture and Track mode and begins a pitching motion to acquire the beam center. A short time after capture, the track mode is engaged to reduce any static displacement towards zero.

The airplane maintains a constant speed along the glideslope until an altitude of about 50 feet is reached. Flare logic equations determine the precise altitude at which the Flare mode is entered. In response to the Flare control law, the vehicle is forced along a path which targets a vertical speed rate of two feet per second at touchdown.

Upon entering Touchdown mode (altitude less than 10 feet), the automatic portion of the landing is complete and the system is automatically disengaged by the control mode logic. This completes the automatic landing flight phase.

1.2 Organization of this Document

After the definition of some terms that will be used in the rest of the document, chapter 2 gives a general, high-level view of the subsystems that your software consists of, and of how they interact. It also informs about the special features that have to be included in the software in order to achieve fault tolerance and to get experimental data on the performance of the software.

Chapters 3 through 7 describe the function of the various subsystems, i.e. the **Complementary** Filters, the Mode Logic, the Altitude Hold control law, the Glideslope Capture and Track control law, and the Flare control law. These subsystems take care of the lane command computation.

Additionally, you have to provide a Command Monitor, to compare your lane command with the ones computed by the other lanes, and a Display Module, to display the status of the flight control system on the pilot's and the flight engineer's consoles. These functions will be discussed in chapters 8 and 9, respectively.

Since the main information about the algorithms is given in terms of diagrams, the Appendix A contains an explanation of all symbols used in these diagrams. You will also find there examples on how diagrams have to be read and understood, and on how they can be transformed into a program. Parameter list definitions of external routines are provided in the Appendix B.

CHAPTER 2

SYSTEM OVERVIEW

You are required to write a parameterless routine named "AUTOLAND", for the management of automatic landing of an aircraft. Figure 2 shows the submodules of the lane command computation and data flow diagram.

In this document, the following terms will be used: A "frame" is defined as one pass of execution of the FCC. The notation X(T) refers to the value of X in the current computation frame, while X(T-1) refers to that in the previous computation frame. Δt refers to the length of a frame, which must be defined as a constant named DELT and set to the value 0.050 (second). This time limit is required to the hardware design. In this software experiment, you do not have to be concerned with this time requirement. However, efficiency of your software is always a good practice.

2.1 Major Computation Sequence

This section provides general requirements for the sequence of computations. Computations of control law and monitoring functions within the frame interval are to be performed in the following sequence:

1) Call SENSORINPUT to receive airplane sensor data from DEDIX.

2) Compute data estimates from complementary filters (Chapter 3), and call VOTEFILTER1, VOTEFILTER2 properly to cross check output values of complementary filters.

3) Determine mode from mode logic (Chapter 4), and call VOTEMODE to cross check output values of mode logic.

4) If model valid discrete (MODV) is false, terminate your program. This lane is closed.

5) If the mode is TOUCHDOWN, set lane command to 0.0 and call VOTEINNER, then go to 9) to perform monitor function and display module.

6) Initialize capture/track or flare control law **(deletion)** if mode has changed.

7) Compute the outer loop from altitude hold, capture/track, or flare control laws (Chapters 5, 6 or 7). Then call VOTEOUTER to cross check output values of outer loop.

8) Compute lane command from inner loop of altitude hold, capture/track, or flare control laws (Chapters 5, 6 or 7), and call VOTEINNER to cross check output values of inner loop.

9) Call LANEINPUT to receive the lane commands from the other lanes via DEDIX.

10) Perform the command monitor functions (Chapter 8). Then call VOTEMONITOR to output the monitor results and get the recovery command (via the variable RECOVERY).

11) Compute display outputs (Chapter 9), and call VOTEDISPLAY to cross check output values of display.

12) If RECOVERY is true, call VOTESTATES to recover the internal states of the FCC.

13) If the mode is TOUCHDOWN, terminate your program. This completes the execution of the flight control. Otherwise, go to 1) for the computation of the next frame.

2.2 Fault-Tolerant Mechanisms

DEDIX serves as a supervisory system which continually operates to receive data from your program, perform the appropriate data comparison and returns the corrected value. Three types of fault-tolerant mechanism are used in this experiment for error detection and recovery:

- Test Point – a point indicated by a hexagonal symbol to observe the computed data. When you reach a test point symbol specified in a diagram, the signal value (usually a real number) at that point must be stored in order to be used in the subsequent cross-check point.

- Cross-Check Point – a point to cross check values with other lanes. Seven cross-check points are specified: VOTEFILTER1, VOTEFILTER2, VOTEMODE, VOTEINNER, VOTEOUTER, VOTEMONITOR, VOTEDISPLAY. They will be preformed from time to time.

- Recovery Point – a point to recover the failed version.
  One recovery point is used at the end of each computation: VOTESTATES. It is performed only if failure occurs in the FCC and the recovery command is issued.

The major difference between recovery point and cross-check point is that all the state variables (which keep the history of the program) of your program are specified in the recovery point, thus no other internal variables of your program can be used in next frame's computation.

Complete definitions of the parameter list of the above external routines are provided in Appendix B.

2.3 Special Requirements

In order to achieve high degrees of modularity and design diversity, you are required to decompose each major function (described in the following chapters) into at least two (or more) modules. Monitor function (Chapter 8) is the only exception that you can implement it as a single module.

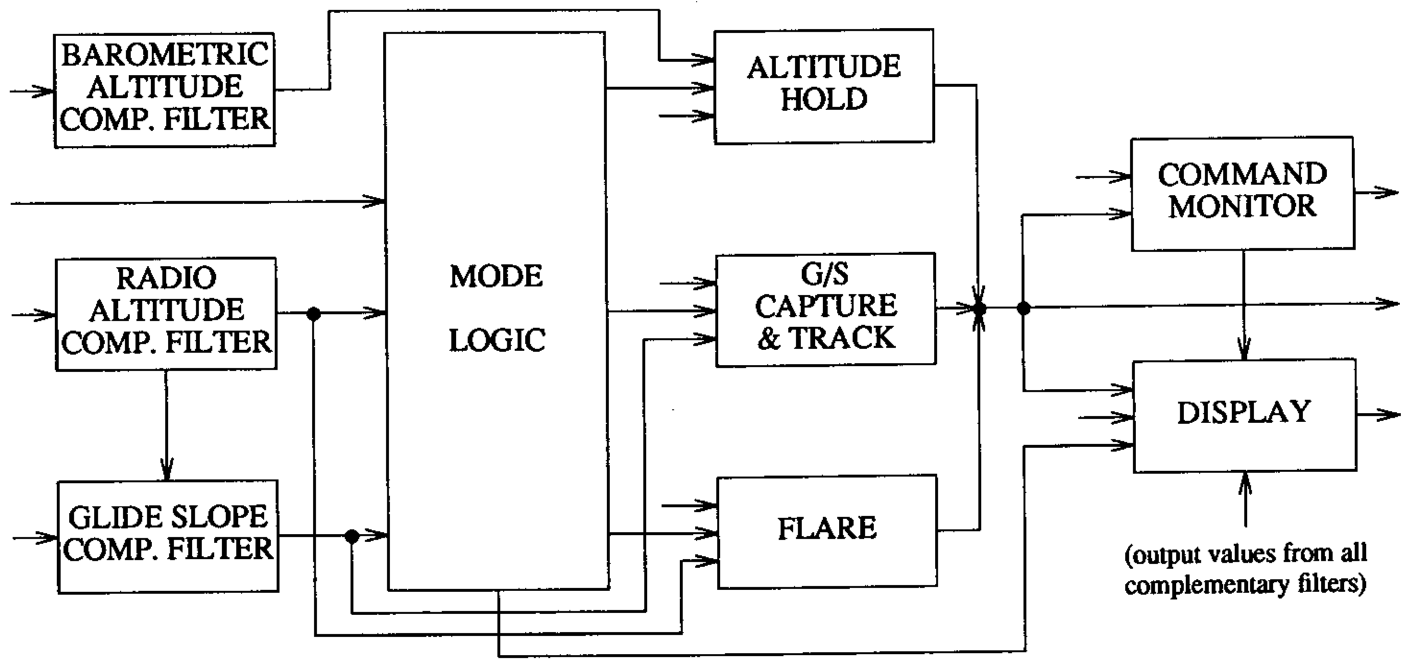

BAROMETRIC ALTITUDE COMP. FILTER
RADIO ALTITUDE COMP. FILTER
GLIDE SLOPE COMP. FILTER
MODE LOGIC
ALTITUDE HOLD
G/S CAPTURE & TRACK
FLARE
COMMAND MONITOR
DISPLAY
(output values from all complementary filters)

Figure 2: Submodules of the Lane Command Computation and Data Flow

CHAPTER 3

COMPLEMENTARY FILTERS

The complementary filters for barometric and radio altitude provide an estimate of true altitude from various altitude related signals. The Barometric Altitude Complementary Filter provides the altitude reference for the Altitude Hold Mode, and the Radio Altitude Complementary Filter provides the altitude reference for the Flare Mode.

The complementary filter for the glide slope mode provides estimates for beam error and radio altitude in the Glide Slope Capture and Track mode.

All filters are initialized when the altitude hold mode is entered.

Which filters have to be computed depends on the mode the system is in. At this point in the computation sequence it was not yet possible to compute the new mode variables (AHD, GSCD, GSTD, FD, and TD) for the current frame. This is done later, by the Mode Logic (cf. chapter 4). Therefore, to determine the mode at this point, use the values of the mode variables that were computed in the previous frame.

3.1 Barometric Altitude Complementary Filter

The long-term accuracy of the barometric altitude is complemented with the superior short-term accuracy of vertical acceleration and inertial altitude rate.

3.1.1 Inputs

Description	Units	Range	Resolution
Altitude (H)	ft	+2500.0/0.0	0.10
Altitude Rate (HR)	ft/sec	+/- 50.0	0.01
Vertical Acceleration (VA)	ft/sec/sec	+/- 32.2	0.001

3.1.2 Outputs

Description	Units	Range	Resolution
Baro Altitude Estimate (BAE)	ft	+2500.0/0.0	0.10

3.1.3 Processing

Filter processing is shown in Figure 3.1. Since the output of this complementary filter is used only in the Altitude Hold Control Law, processing is necessary only while the system is in Altitude Hold mode. However, to facilitate cross checking (cf. chapter 2) and display (cf. chapter 9), a default value of 0.0 must be assigned to the output variable BAE whenever the filter function is not computed. The same applies to the values of the test-points inside this filter.

Upon entering Altitude Hold mode, the following initializations have to be performed:

1) The input values of the integrators I2, I3, and I4, as well as the output value of integrator I4, are initialized by zero.

2) The output value of integrator I2 is initialized by the current input HR.

3) The output value of integrator I3 is initialized by the current input H.

3.2 Radio Altitude Complementary Filter

The long-term accuracy of the radio altitude is combined with the superior short-term accuracy of vertical acceleration.

3.2.1 Inputs

Description	Units	Range	Resolution
Radio Altitude (RA)	ft	+2500.0/0.0	0.10
Vertical Acceleration (VA)	ft/sec/sec	+/- 32.2	0.001

3.2.2 Outputs

Description	Units	Range	Resolution
Radio Altitude Estimate (RAE)	ft	+2500.0/0.0	0.10
Radio Altitude Rate Est (RARE)	ft/sec	+/- 30.0	0.01

3.2.3 Processing

Filter processing is shown in Figure 3.2. Since the output of the Radio Altitude Complementary Filter is used in the Glide Slope Complementary Filter, the Mode Logic, and the Flare Control Law, it has to be computed during all flight modes.

Upon entering Altitude Hold mode, the following initializations have to be performed:

1) Both the input and output value of integrator I5 **(deletion)**, the input value of integrator I6, and both the input and output value of filter F8, are initialized by zero.

2) The output value of integrator I6 is initialized by the current input RA.

3.3 Glide Slope Complementary Filter

Basic shaping and filtering of inertial and radio beam signals to generate an improved estimate of the airplane position relative to the beam is provided.

3.3.1 Inputs

Description	Units	Range	Resolution
Radio Altitude Estimate (RAE)	ft	+2500.0/0.0	0.10
Vertical Acceleration (VA)	ft/sec/sec	+/- 32.2	0.001
Glide Slope Deviation (GSD)	deg	+2.0/-1.0	0.001

3.3.2 Outputs

Description	Units	Range	Resolution
Rad Alt G/S Filter (RAGSF)	ft	+2500.0/0.0	0.10
G/S Rate Error (GSRE)	ft/sec	+/- 30.0	0.01
G/S Deviation Est (GSDE)	ft	+/- 600.0	0.10
G/S Error Lambda (GSEL)	ft	+/- 600.0	0.10
G/S Dev Rate Limited (GSDRL)	deg	+1.0/-2.0	0.001

3.3.3 Processing

Filter processing is shown in Figure 3.3. The first stage generates a rate-limited beam error from the primary glide slope sensor. The rate limited error is combined with vertical acceleration to provide the glide slope displacement and rate signals utilized in the Glide Slope Capture and Track mode. A third part of the filter generates a smoothed radio altitude signal.

Since the output of the Glide Slope Complementary Filter is used in the Mode Logic (cf. chapter 4) to determine the transition from Altitude Hold mode to Glide Slope Capture and Track mode as well as for the Glide Slope Capture and Track Control Law, it must be computed while the system is in Altitude Hold and in Glide Slope Capture and Track modes. It need not be computed during Flare mode. However, to facilitate cross checking (cf. chapter 2) and display (cf. chapter 9), a default value of 0.0 must be assigned to all its output variables (RAGSF, GSRE, GSDE, GSEL, and GSDRL) and to all the values of test-points inside this filter, whenever the filter function is not computed.

Upon entering Altitude Hold mode, the following initializations have to be performed:

1) The input values of all integrators (I7, I8, I9, I10), the input value of filter F10, and the output values of integrators I7 and I9 are initialized by zero.

2) The output of integrator I8 is initialized by the output variable G/S Error Lambda (GSEL).

3) The output of integrator I10 is initialized by the current input G/S Deviation (GSD).

4) The output of filter F10 is initialized by the current input Radio Altitude Estimate (RAE).

Note that GSEL can be computed by using I10 and F10 only; the result is then used to initialize I8.

The magnitude limiter LM12 is a special case that needs to be explained: The arrow entering the box on the left side and leaving it on the right side represents the quantity to be limited, as usual. The arrow entering the limiter symbol from the bottom is meant to denote variable (dynamic) upper and lower limits. The absolute magnitude of that signal is to be taken as the upper limit, while the lower limit is the negative value of the upper limit.

The following values for the constants K_0, K_2, and K_3 (cf. Figure 3.3) are to be used:

1) For 10 seconds after Altitude Hold initialization and before G/S Capture and Track mode:

$$K_0 = 1;\ K_2 = \frac{1}{4};\ K_3 = 0$$

2) After the initial 10 seconds and before G/S Capture and Track mode:

$$K_0 = 1;\ K_2 = \frac{1}{8};\ K_3 = \frac{1}{1024} + \frac{200\text{ft}}{820{*}\text{RAGSF(T)}}$$

3) During G/S Capture and Track mode:

$$K_0 = \frac{200\text{ft}}{\text{RAGSF(T)}} K_X;\ K_2 = \frac{200\text{ft}}{\text{RAGSF(T)}} \frac{K_X}{16.5};\ K_3 = \frac{200\text{ft}}{\text{RAGSF(T)}} \frac{1}{820}$$

where $\frac{200\text{ft}}{\text{RAGSF(T)}}$ is limited to $\frac{1}{5.5}$ minimum

and $K_X = \frac{\text{RAE(T)}-50\text{ft}}{50\text{ft}}$ is limited to minimum 0 and maximum 1

Note that you can compute the output value RAGSF without using the gains K_0, K_2, and K_3. The current value of RAGSF can then be used to determine the current value of these "constants".

3.4 Special Requirements

As soon as the output variables of the Radio Altitude Complementary Filter have been computed, and before they are used in any further computation, they and the values of all the test points inside this filter must be passed to the voter routine VOTEFILTER1. This routine may change the values you computed. If it does, computations must proceed with the new values.

As soon as the output variables of the other two complementary filters have been computed, and before they are used in any further computation, they and the values of all the test points inside the filters must be passed to the voter routine VOTEFILTER2. This routine may change the values you computed. If it does, computations must proceed with the new values.

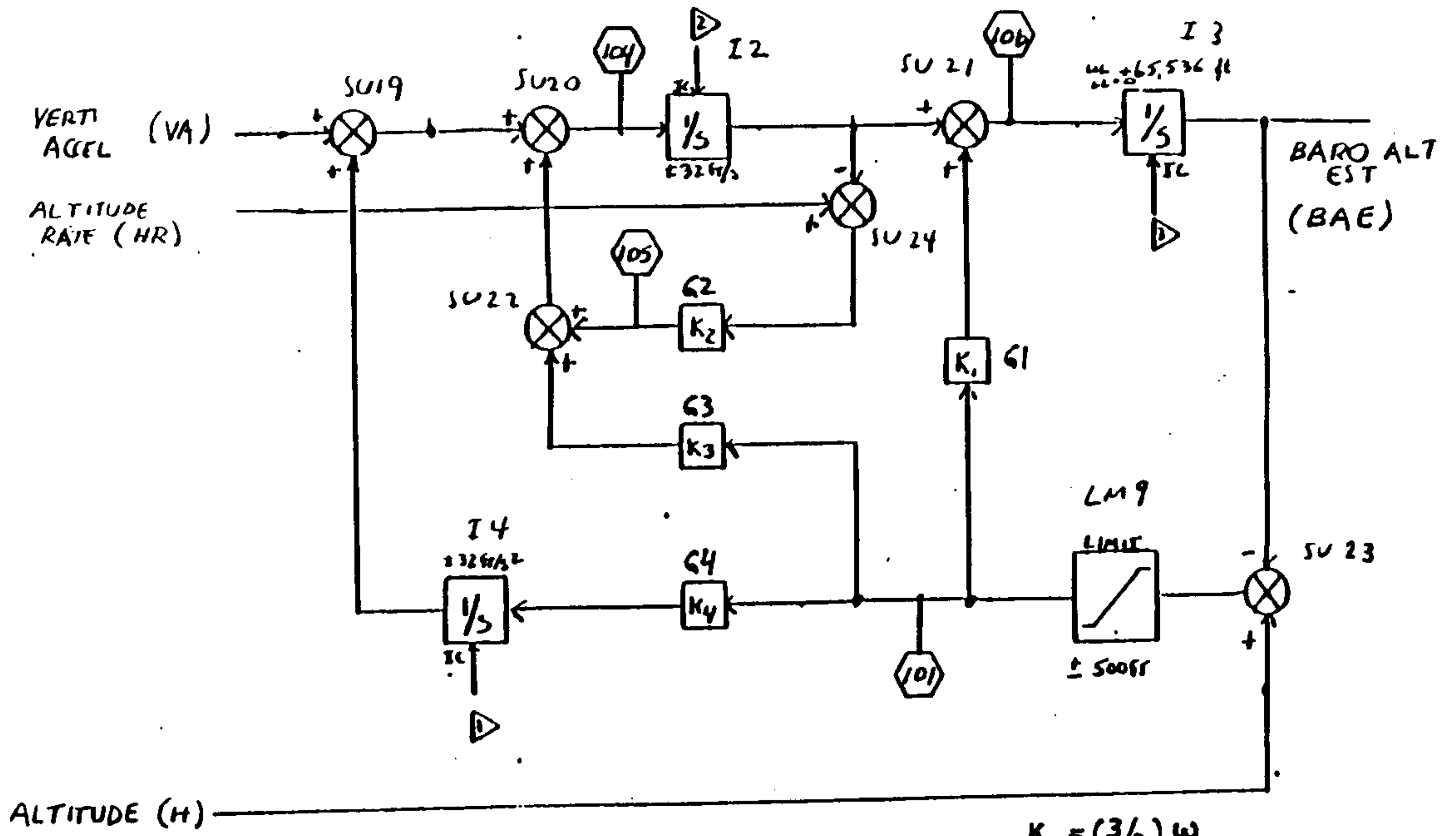


FIGURE 3.1

BAROMETRIC ALTITUDE COMPLEMENTARY FILTER

1 - IC TO ZERO @ MODV

2 - IC TO $\dot{h}$ @ MODV ($\dot{h}$ = HR)

3 - IC TO h @ MODV (h = H)

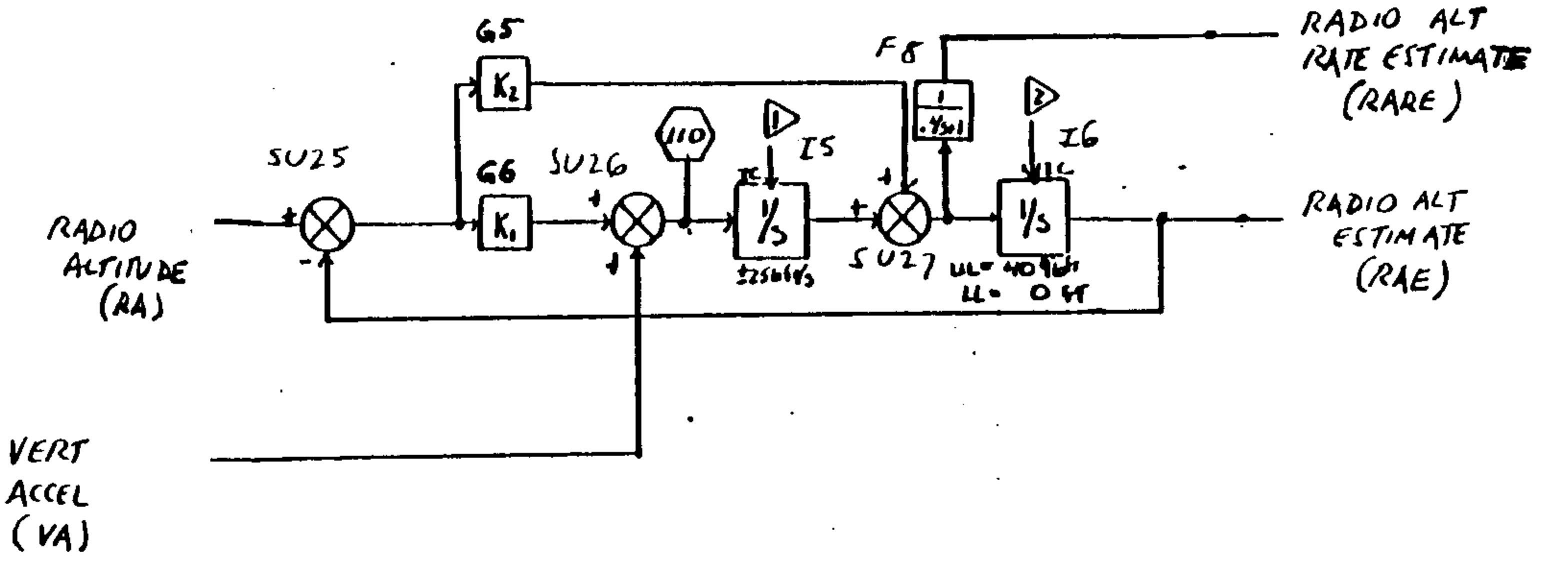
$K_1 = (3/2)\,\omega$

$K_2 = (3/2)\,\omega$

$K_3 = (3/4)\,\omega^2$

$K_4 = \omega^3$

$\omega = .1$ RAD/SEC



$K_1 = .25$
$K_2 = 1.00$

▷1 IC TO φ·φ @ MODV

▷2 IC TO h_R @ MODV (h_R = RA)

FIGURE 3.2
RADIO ALTITUDE
COMPLEMENTARY FILTER

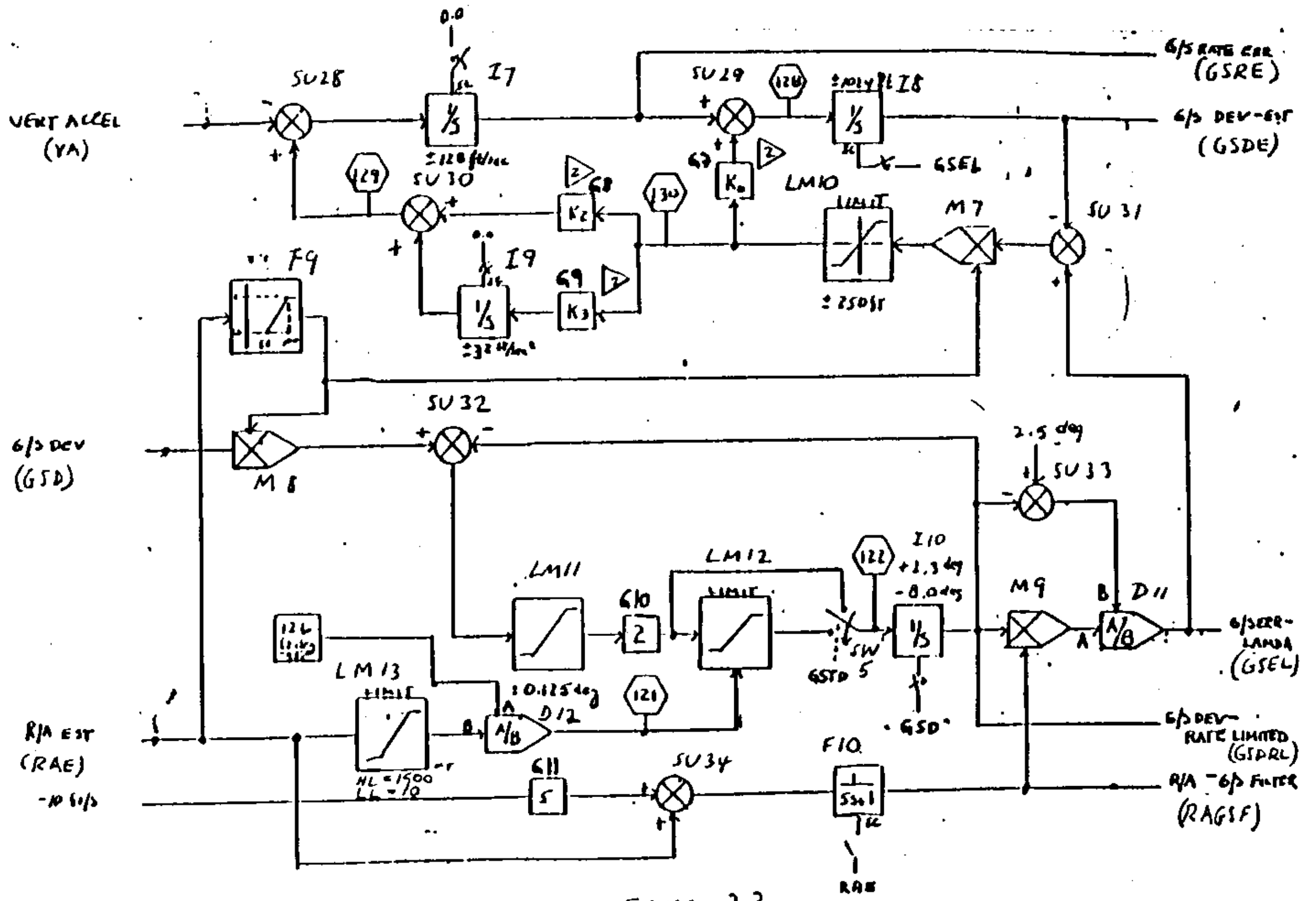


FIGURE 3.3
GLIDESLOPE DEVIATION
COMPLEMENTARY FILTER

NOTE 1 - ALL IC OCCUR
AT MODV

2 SEE PAGE 13

CHAPTER 4

MODE LOGIC

This set of logic statements controls the transitions from one mode to the next, as a function of the model status and the magnitude of certain system variables. Mode logic provides the mode status for the flight control system. Using data from the complementary filters, the points in the landing trajectory at which mode transfer takes place are defined in terms of the five states: Altitude Hold, Glideslope Capture, Glideslope Track, Flare and Touchdown.

4.1 Inputs

Description	Units	Range	Resolution
Model Valid (MODV)	disc	+1/0	–
G/S Capture Discrete (GSCD)	disc	+1/0	–
G/S Deviation (GSD)	deg	+2.0/-1.0	0.01
G/S Deviation Estimate (GSDE)	ft	+/- 600.0	0.10
G/S Rate Error (GSRE)	ft/sec	+/- 30.0	0.01
Rad Alt Estimate (RAE)	ft	+2500/0.0	0.10
Rad Alt Rate Estimate (RARE)	ft/sec	+/- 30.0	0.01
Rad Alt G/S Filter (RAGSF)	ft	+2500/0.0	0.01
G/S Dev Rate Limited (GSDRL)	deg	+2.0/-1.0	0.001
Vertical Acceleration (VA)	ft/sec/sec	+/-32.2	0.001
Pitch Attitude Rate (PAR)	deg/sec	+/-128.0	0.01

4.2 Outputs

Description	Units	Range	Resolution
Alt Hold Discrete (AHD)	disc	+1/0	-
G/S Capture Discrete (GSCD)	disc	+1/0	-
G/S Track Discrete (GSTD)	disc	+1/0	-
Flare Discrete (FD)	disc	+1/0	-
Touchdown Discrete (TD)	disc	+1/0	-

4.3 Processing

Mode logic functions, in AND and OR gate format, are shown in Figure 4.1. Figure 4.2 provides algorithms required to compute the quantities FPEC1 and FPDC1 during the Altitude Hold Mode. These two quantities are developed from certain of the algorithms included in the Glideslope Capture and Track control Law, Figure 6.1. However, there is no data dependency between these two figures. The logic provides the following functions:

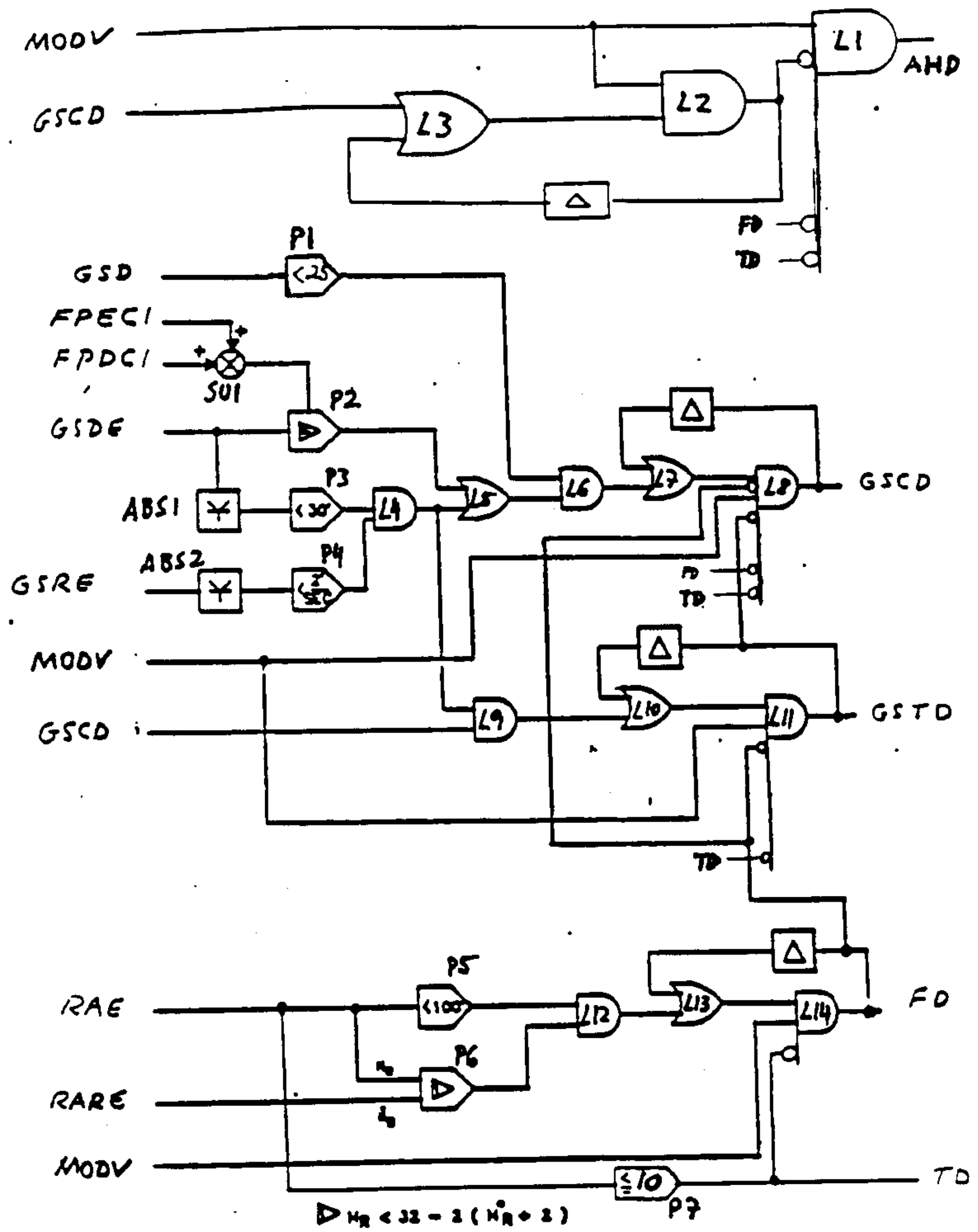
1) Ensures that the model portions of the experiment are operational throughout the run as long as the model valid variable MODV is true.

2) Provides the clues to transition from the initial state (altitude hold mode) to subsequent states in accordance with the magnitude or sign of critical variables.

The "delta" wrap-around (Δ) shown in the figure denotes a one-frame delay, and provides a latching function. **The initial values for GSCD, GSTD, FD, and TD are all "false".**

4.4 Special Requirements

As soon as the mode logic output variables have been computed, and before they are used as a basis for any further flight control computations, they must be passed to the voter routine VOTEMODE. This routine may change the value you computed. If it does, computations should proceed with the changed value.



▷ $H_R < 32 - 2(\dot{H}_R + 2)$

▷ DET IF THE G/S DEV & FLT PATH CMDS HAVE OPPOSITE SIGNS OF MAGNITUDE (0.0 COUNTS AS POSITIVE)

PITCH MODE LOGIC

FIGURE 4.1

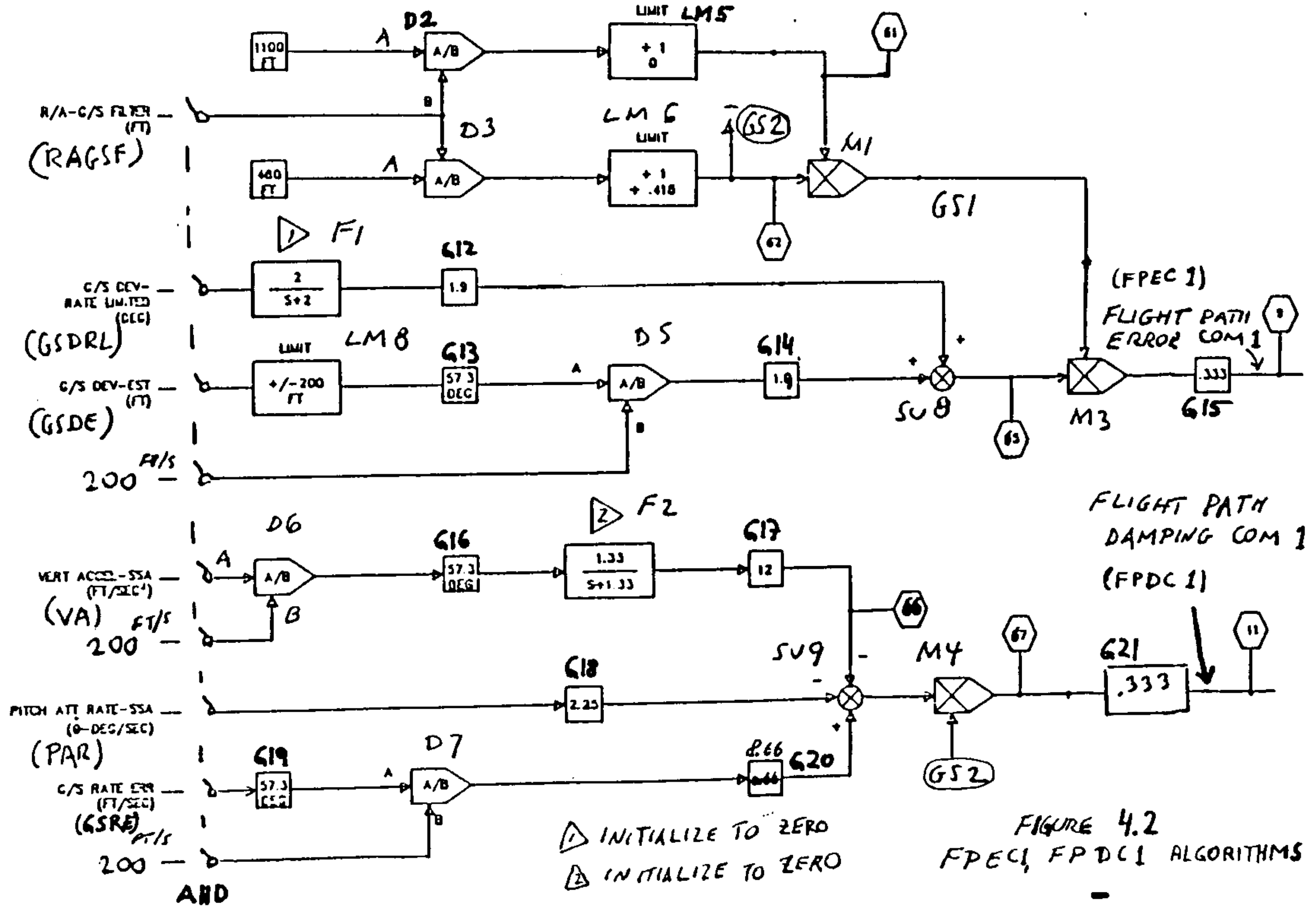


FIGURE 4.2
FPEC1, FPDC1 ALGORITHMS

CHAPTER 5
ALTITUDE HOLD MODE CONTROL LAW

This control law generates the pitch commands required to hold the aircraft at a reference altitude. The mode is entered when the aircraft is in level flight at an altitude of about 1500 feet. It is the initial mode.

The control law has two parts which are called "outer loop" and "inner loop", respectively. The outer loop is the first part of the control law, while the inner loop is the second part of the control law. The signals Theta Command Integral, Flight Path Error Command, and Flight Path Damping Command (i.e. the values at the test-points 8, 9, and 11) are the interface between the inner and the outer loop (refer to Figure 5.1). Note that the inner loop is very similar for all three control laws; therefore it should be implemented only once.

5.1 Outer Loop

5.1.1 Inputs

Description	Units	Range	Resolution
Altitude Hold Discrete (AHD)	disc	1/0	–
Baro Altitude Estimate (BAE)	ft	+2500/0.0	0.01
Altitude Reference (HREF)	ft	+2500/0.0	0.01
Flight Path (FP)	deg	+/- 15.0	0.01
Equalization (EQ)	deg	+/- 5.0	0.01

5.1.2 Outputs

Description	Units	Range	Resolution
Theta Command Integral (THCI)	deg/sec	+/- 10.0	0.01
Flight Path Error Command (FPEC)	deg	+/- 20.0	0.01
Flight Path Damping Command (FPDC)	deg	+/- 20.0	0.01

5.1.3 Processing of the Outer Loop

Control law functions are shown in Figure 5.1. The algorithm computes the intermediate values Theta Command Integral, Flight Path Error Command, and Flight Path Damping Command. Any variation in these controlled quantities, due to external disturbances, is reduced by the feedback action. The main feedback quantities are the displacement, the rate, and the integral of the altitude error.

Although the variable Altitude Reference (HREF) is a true input to the Altitude Hold Control Law, it does not change while an automatic landing is performed. (It is selected by the pilot by his control panel.) Therefore, it will not be input at the beginning of each computation frame. Instead, you are required to set it to the value of the input variable BAE at the instant when the Altitude Hold mode is entered (cf. switch SW1, Figure 5.1).

There are no filters, integrators, or rate limiters that need to be initialized.

5.2 Inner Loop

5.2.1 Inputs

Description	Units	Range	Resolution
Theta Command Integral (THCI)	deg/sec	+/- 10.0	0.01
Flight Path Error Command (FPEC)	deg	+/- 20.0	0.01
Flight Path Damping Command (FPDC)	deg	+/- 20.0	0.01
Pitch Attitude (PA)	deg	+/- 15.0	0.01
Pitch Attitude Rate (PAR)	deg/sec	+/- 128.0	0.01

5.2.2 Outputs

Description	Units	Range	Resolution
Lane Command (LC)	deg	+/- 15.0	0.01

5.2.3 Processing of the Inner Loop

Control law functions are shown in Figure 5.1. The algorithm computes elevator commands due to changes in the input variables. Any variation in these controlled quantities, due to external disturbances, is reduced by the feedback action. The main feedback quantities are the displacement, the rate, and the integral of the altitude error.

A number of signal limiters are required, as shown in Figure 5.1, to prevent a large error from overdriving the following stages. In addition, the design includes provisions to limit the input to the path integrator I1 when limiting output conditions are reached. If the output exceeds a specified rate limit, the input contribution to the integrator from the path command is set to zero. If the output exceeds a displacement limit, the excess is fed back to the integrator to reduce the input command. This helps ensure prompt control system response to input signal changes, by minimizing error buildup before the integrator.

Upon entering Altitude Hold mode, both the input and the output value of integrator I1 have to be initialized by zero. The output values of the rate limiters LR1 and LR2 must be initialized with their current inputs (i.e. a rate of 0.0 is assumed in that case).

It is important that the (intermediate) value Theta Command (Θ_C), i.e. the signal at test-point 3, be saved as a state variable because it will be used upon transition to Glide Slope Capture and Track mode.

A general requirement is that the final output of a computation (in this case LC) should depend on the most current input values (see Appendix A.15). A little complication

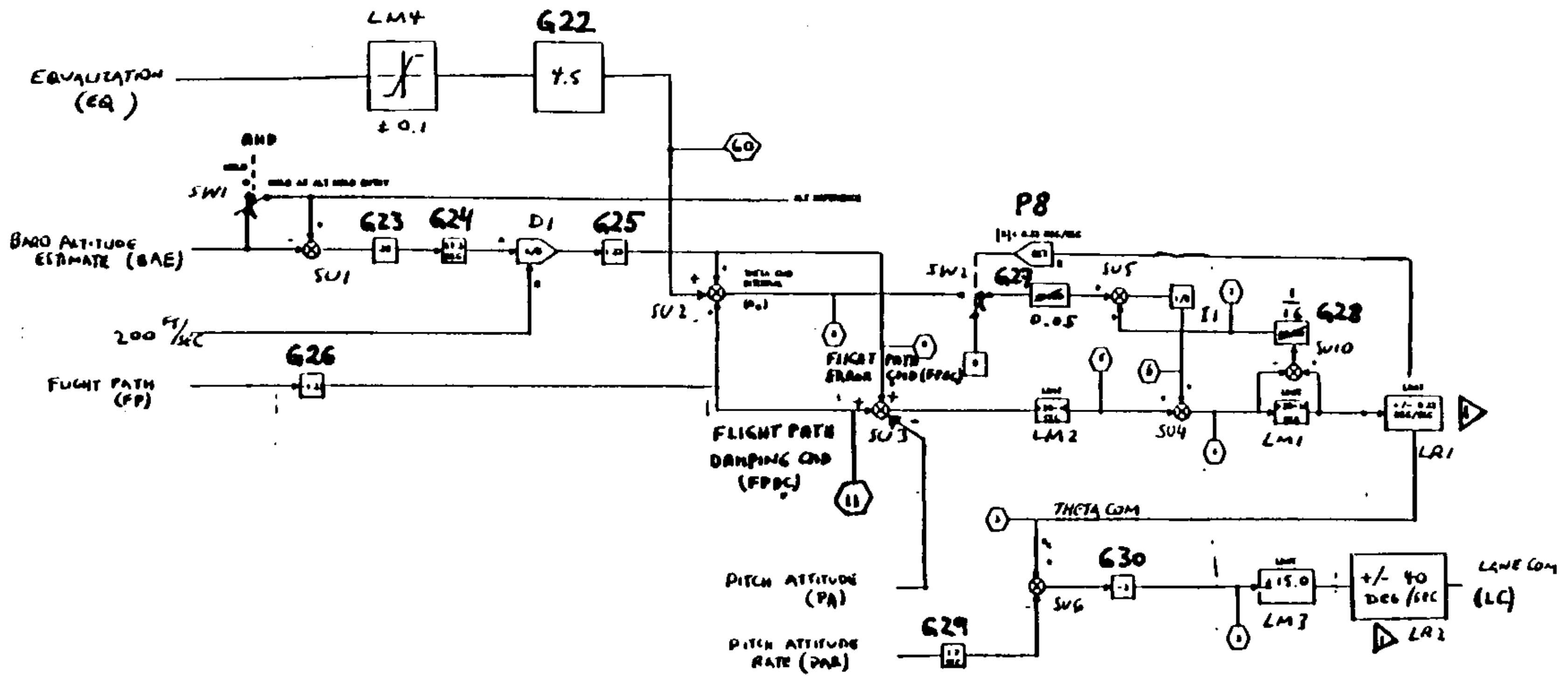
arises in the case of the inner loop because the computation path from integrator I1 to the final output LC overlaps with the feedback loop of I1. The way to resolve this problem is as follows: First, use the old output of I1 to compute the feedback value for I1, i.e. the input to summer SU5 coming from testpoint 7, and to determine the position of switch SW2. Then you can compute the new input value of I1, and thus do the integration. Now, use the new integrator output to compute THETA sub C, and subsequently the final output LC. Note that you compute summer SU4 and limiter LM1 twice, once with the old integrator output to get the feedback value, and once with the new integrator output to get the final output. Also, the testpoints 4 and 6 are encountered twice; the values that should be passed to the voter routine VOTEINNER are the values that were computed using the new integrator output.

5.3 Special Requirements

As soon as the outer loop output variables have been computed, and before they are used in any further computation, they and the values of all the test points inside the outer loop must be passed to the voter routine VOTEOUTER. This routine may change the values you computed. If it does, computations must proceed with the new values.

If the control law is not computed (Glide Slope Capture and Track mode, Flare mode), the values of the test points in its Outer Loop must be set to 0.0 for the voter routine VOTEOUTER.

As soon as the inner loop output variables have been computed, and before they are used in any further computation, they and the values of all the test points inside the inner loop must be passed to the voter routine VOTEINNER. This routine may change the values you computed. If it does, computations must proceed with the new values.



INITIALIZE TO INPUT

TEST POINT

FIGURE 5.1
ALT HOLD CONTROL LAW

CHAPTER 6

GLIDE SLOPE CAPTURE AND TRACK CONTROL LAW

This control law generates the pitch guidance commands required to maneuver the aircraft to capture and track the glide slope beam upon receipt of the capture discrete from the capture mode logic. The control law is designed to acquire the glide slope beam and hold the airplane on the beam in preparation for flare and touchdown.

The control law has two parts which are called "outer loop" and "inner loop", respectively. The outer loop is the first part of the control law, while the inner loop is the second part of the control law. The signals Theta Command Integral, Flight Path Error Command, and Flight Path Damping Command (i.e. the values at the test-points 8, 9, and 11) are the interface between the inner and the outer loop (refer to Figure 6.1). Note that the inner loop is very similar for all three control laws; therefore it should be implemented only once.

6.1 Outer Loop

6.1.1 Inputs

Description	Units	Range	Resolution
G/S Capture Discrete (GSCD)	disc	1/0	-
G/S Track Discrete (GSTD)	disc	1/0	-
Rad Alt G/S Filter (RAGSF)	ft	+2500/0.0	0.10
G/S Rate Error (GSRE)	ft/sec	+/- 30.0	0.01
G/S Error Lambda (GSEL)	ft	+/- 600.0	0.10
G/S Dev Rate Limited (GSDRL)	deg	+2.0/-1.0	0.001
G/S Dev Estimate (GSDE)	ft	+/- 600.0	0.10
Vertical Acceleration (VA)	ft/sec/sec	+/- 32.2	0.01
Pitch Attitude Rate (PAR)	deg/sec	+/- 128.0	0.01
Equalization (EQ)	deg	+/- 5.0	0.01

6.1.2 Outputs

Description	Units	Range	Resolution
Theta Command Integral (THCI)	deg/sec	+/- 10.0	0.01
Flight Path Error Command (FPEC)	deg	+/- 20.0	0.01
Flight Path Damping Command (FPDC)	deg	+/- 20.0	0.01

6.1.3 Processing of the Outer Loop

Capture and track control law functions are shown in Figure 6.1. The algorithms consist of a blend of the following feedback quantities: displacement, rate, acceleration, and integral of the beam error.

It should be noted that no control law effect results from a transition from capture to track mode. Acquisition of the beam is initiated when the conditions for capture or track are met. Track mode annunciation indicates that the beam error is reduced below that required for capture.

In general, the same comments as made for the Altitude Hold control law are applicable here.

The switch SW3 is a special case, which requires some extra information. The condition (GSCD + GSTD) + 0.5 SEC has to be understood as follows: Switch SW3 is closed 0.5 seconds after the expression (GSCD OR GSTD) became 'true' for the first time, i.e. 0.5 seconds after entering the Glide Slope Capture and Track mode. For example, if the frame length is 0.05 seconds, SW3 would be closed during the eleventh frame, counting the first frame as that in which (GSCD + GSTD) = 1 for the first time, i.e. in the eleventh frame of the Glide Slope Capture and Track mode computation. The effect is to force the airplane to rotate nose-down to acquire and hold the beam. While the switch is open, the input to SU11 from SW3 is zero.

Upon entering the Glideslope Capture and Track mode, some initializations have to be performed:

1) The input values of the filters F1 and F2 are initialized by zero.

2) The output value of filter F1 is initialized with its current input.

3) The output value of filter F2 is initialized by zero.

6.2 Inner Loop

Refer to section 5.2 for a description of the inner loop. There are **three** differences that have to be observed:

The first difference is that the input variable PA is not used at summer SU3 (refer to Figure 6.1).

The second difference is that the gain constant immediately before summer SU5 has a different value.

The **third** difference concerns the initialization of the output value of integrator I1, for the following reason:

As described in Section 4, Mode Logic, the outputs from the complementary filters determine the mode state. The mode changes generally result in replacing current complementary filter signals and control laws with changed signals and control laws. If these changes are made without consideration of the possible effect on surface (elevator) position, undesirable surface transients may result. In order to ensure a smooth transition from one mode state to another, the path integrator is initialized, prior to the mode change, to absorb the lane command difference due to the state change. The method for transitioning from the Altitude Hold mode to the Capture and Track mode is as follows:

(1) Compute the state of GSCD, Glide Slope Capture Discrete.

(2) If GSCD = 0, proceed with Altitude Hold mode computation.

(3) If GSCD = 1

- Close the sensor and complementary filter signal switches to Glideslope Capture and Track control laws.

- Recall $\Theta_C(T-1)$ from the previous frame of Altitude Hold mode.

- Use the present input values FPEC(T) and FPDC(T) to initialize the output value of integrator I1 with: $\Theta_C(T-1)$ - [FPEC(T) + FPDC(T)], so that the resulting value of $\Theta_C(T)$ **approximately equals** $\Theta_C(T-1)$ in the first Glideslope Capture and Track computation frame. **(deletion)**

Again, it is important that the (intermediate) value Theta Command (Θ_C), i.e. the signal at test-point 3, be saved as a state variable because it will be used upon transition to Flare mode.

6.3 Special Requirements

As soon as the outer loop output variables have been computed, and before they are used in any further computation, they and the values of all the test points inside the outer loop must be passed to the voter routine VOTEOUTER. This routine may change the values you computed. If it does, computations must proceed with the new values.

If the control law is not computed (Altitude Hold mode, Flare mode), the values of the test points in its Outer Loop must be set to 0.0 for the voter routine VOTEOUTER.

As soon as the inner loop output variables have been computed, and before they are used in any further computation, they and the values of all the test points inside the inner loop must be passed to the voter routine VOTEINNER. This routine may change the values you computed. If it does, computations must proceed with the new values.

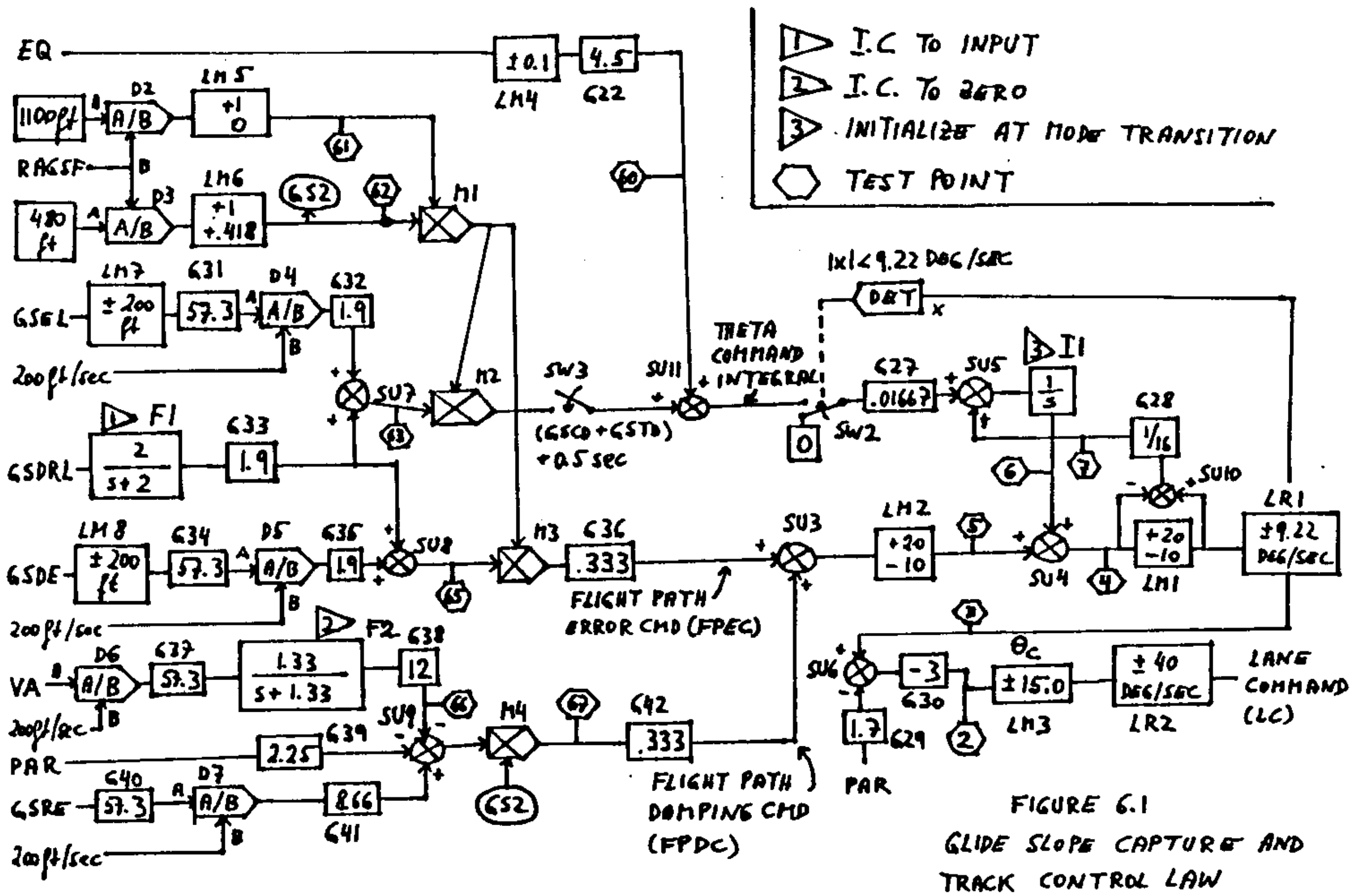


FIGURE 6.1
GLIDE SLOPE CAPTURE AND
TRACK CONTROL LAW

CHAPTER 7

FLARE CONTROL LAW

This control law generates the pitch guidance commands required to flare the aircraft, upon generation of the flare discrete by the flare mode logic. The flare control law is designed to bring the airplane to a suitable vertical speed at touchdown (landing).

The control law has two parts which are called "outer loop" and "inner loop", respectively. The outer loop is the first part of the control law, while the inner loop is the second part of the control law. The signals Theta Command Integral, Flight Path Error Command, and Flight Path Damping Command (i.e. the values at the test-points 8, 9, and 11) are the interface between the inner and the outer loop (refer to Figure 7.1). Note that the inner loop is very similar for all three control laws; therefore it should be implemented only once.

7.1 Outer Loop

7.1.1 Inputs

Description	Units	Range	Resolution
Flare Discrete (FD)	disc	1/0	–
Rad Alt Estimate (RAE)	ft	+2500/0.0	0.10
Rad Alt Rate Est (RARE)	ft/sec	+/- 20.0	0.01
Pitch Attitude Rate (PAR)	deg/sec	+/- 128.0	0.01
Flight Path (FP)	deg	+/- 15.0	0.01
Vertical Acceleration (VA)	ft/sec/sec	+/- 32.2	0.01
Equalization (EQ)	deg	+/- 5.0	0.01

7.1.2 Outputs

Description	Units	Range	Resolution
Theta Command Integral (THCI)	deg/sec	+/- 10.0	0.01
Flight Path Error Command (FPEC)	deg	+/- 20.0	0.01
Flight Path Damping Command (FPDC)	deg	+/- 20.0	0.01

7.1.3 Processing of the Outer Loop

Flare control law functions are shown in Figure 7.1. Essentially, the flight path is compared to a reference path and the error (difference) applied to a displacement and integral functions. Altitude rate and vertical acceleration are added as stabilizing terms. Any variation in the path variables, due to turbulence, is reduced by the pitch rate feed back action.

In general, the same comments as made for the Altitude Hold control law are applicable here.

Upon entering the Flare mode, the following initializations have to be performed:

1) The input value of the filter F6 and F7 are initialized by zero.

2) The output value of filter F6 is initialized with its current input, i.e. the current signal value after summer SU16.

3) The output value of filter F7 is initialized by zero.

7.2 Inner Loop

Refer to section 5.2 for a description of the inner loop. There are three differences that have to be observed:

The first difference is that the input variable PA is not used at summer **SU3** (refer to Figure 7.1).

The second difference is that the gain constant immediately before summer SU5 has a different value.

The third difference concerns the initialization of the output value of integrator I1. Essentially the same method for transitioning from Glide Slope Capture and Track mode to Flare mode as before (section 6.2) is used:

(1) Compute the state of FD, Flare Discrete.

(2) If FD = 0, proceed with Glide Slope Capture and Track mode computation.

(3) If FD = 1

- Close the sensor and complementary filter signal switches to Flare control laws.
- Recall $\Theta_C(T-1)$ from the previous frame of Glide Slope Capture and Track mode.
- Use the present input values FPEC(T) and FPDC(T) to initialize the output value of integrator I1 with: $\Theta_C(T-1)$ - [FPEC(T) + FPDC(T)], so that the resulting value of $\Theta_C(T)$ **approximately equals** $\Theta_C(T-1)$ in the first Flare computation frame. **(deletion)**

7.3 Special Requirements

As soon as the outer loop output variables have been computed, and before they are used in any further computation, they and the values of all the test points inside the outer loop must be passed to the voter routine VOTEOUTER. This routine may change the values you computed. If it does, computations must proceed with the new values.

If the control law is not computed (Altitude Hold mode, Glide Slope Capture and Track mode), the values of the test points in its Outer Loop must be set to 0.0 for the voter routine VOTEOUTER.

As soon as the inner loop output variables have been computed, and before they are used in any further computation, they and the values of all the test points inside the inner loop must be passed to the voter routine VOTEINNER. This routine may change the values you computed. If it does, computations must proceed with the new values.

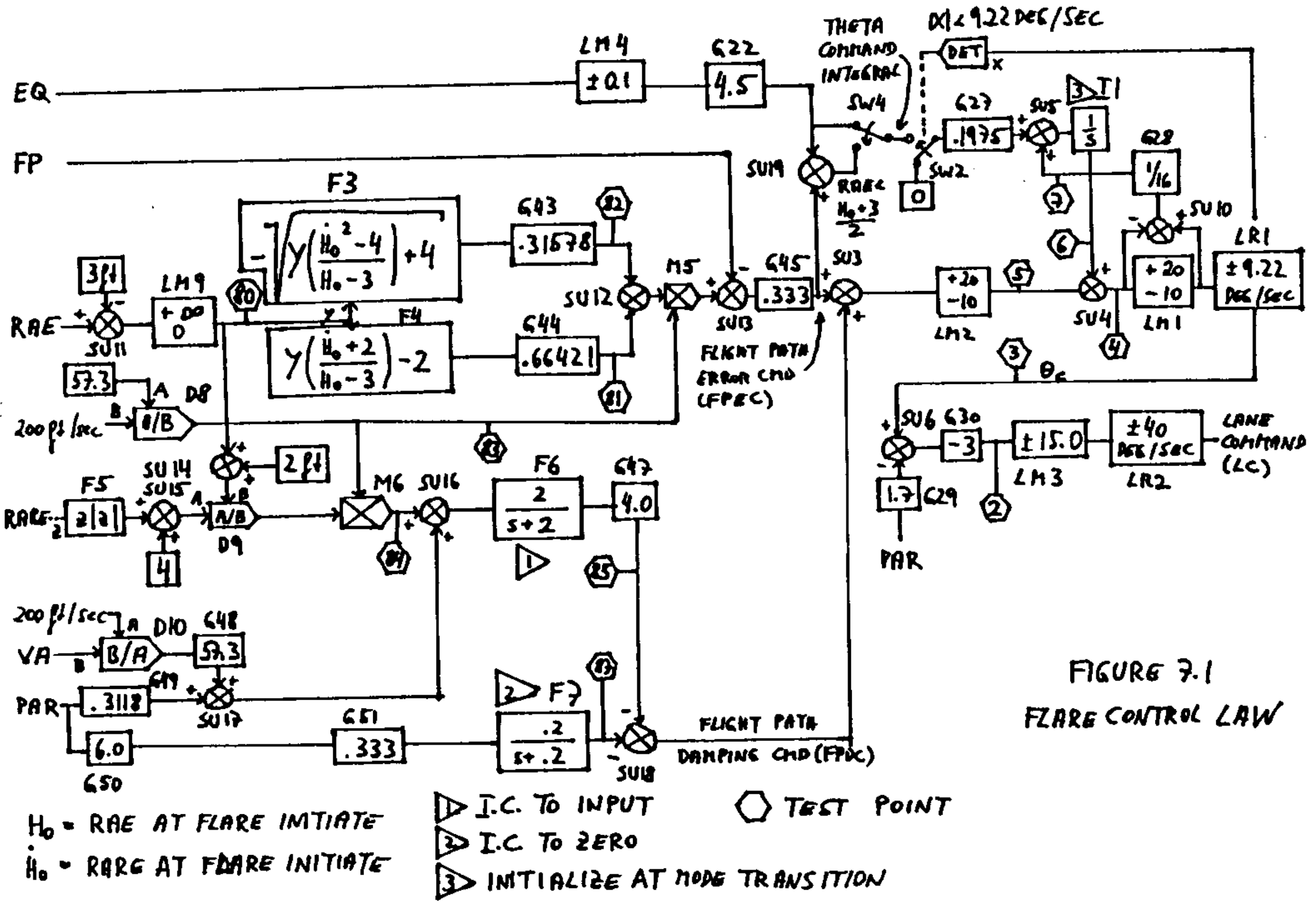


FIGURE 7.1
FLARE CONTROL LAW

CHAPTER 8

COMMAND MONITORS

Command Monitors, by comparing each locally computed lane command with the two others, provide the basic fault detection function in critical flight control systems. Two command monitors are included in each computing lane. One of these monitors compares the elevator command of its own lane (LCA, i.e, LC of your program) with that of one adjacent lane (LCB, to be provided), and the second performs the identical task with the other lane (LCC, to be provided). If the magnitude of the difference between the two inputs lies within a designated band (threshold), no action is taken and the detector output is MXY = 1. If the difference exceeds the threshold, the difference is integrated. A lane miscompare state occurs if the integrated difference exceeds a fixed number of degree-seconds. In that case the miscompare state (MXY = 0) is annunciated to the DEDIX supervisor program. DEDIX will base upon the overall opinions from all three lanes to issue the recovery command.

8.1 Inputs

Description	Units	Range	Resolution
Lane A Command (LCA)	deg	+/- 15.0	0.01
Lane B Command (LCB)	deg	+/- 15.0	0.01
Lane C Command (LCC)	deg	+/- 15.0	0.01

8.2 Outputs

Description	Units	Range	Resolution
Monitor State B by A (MAB)	disc	+1/0	-
Monitor State C by A (MAC)	disc	+1/0	-

8.3 Processing

Monitor processing is shown in Figure 8.1. The monitor provides an output when the absolute value of the error (difference between two lane commands) exceeds a threshold ΔM. The error is integrated and a fault declared if the integrator output exceeds a value specified in Table 1 in Figure 8.1.

The integrated error is proportional to the physical response that the airplane would have if the surface were moved an amount equivalent to the integrated error. Stated another way, the error usually presents a disturbance of very short duration relative to the airplane time response, and the airplane response is then dependent only on the integral of the error, not the shape. Therefore, a monitor based on error integral (time-magnitude monitor), rather than error, is an effective way to establish fault tolerance levels.

When the error drops below the threshold ΔM, a negative signal of magnitude ΔM/2 is applied to the integrator. This feature is designed to ensure that the monitor can recover from temporary faults. The integrator lower limit is zero, so that the application of the negative input cannot drive the integrator to negative values.

4.4 Special Requirements

When the command monitor output variables have been computed, they must be passed to the external routine VOTEMONITOR. This routine will return the input variable RECOVERY to indicate the recovery command.

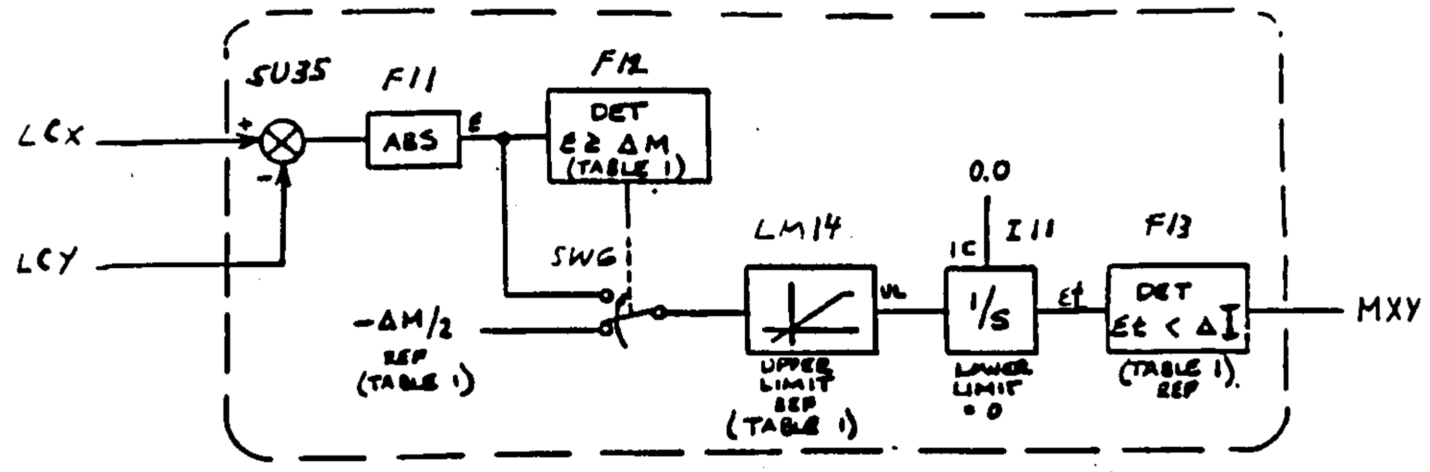


TABLE 1

ΔM = 1.0 DEG
ΔI = 4.0 DEG-SEC
UPPER LIMIT = 6 DEG

FIGURE 8.1

COMMAND MONITOR

CHAPTER 9
DISPLAYS

Three display panels will provide continuous information to the crew on autopilot modes, fault status, and signal levels for each lane. The *Mode Panel* is located in the cockpit below the glare-shield and contains one mode display consisting of ten five-by-seven dot matrix grids. The *Fault Status Panel* is located in the Flight Engineer's station and contains two fault displays each consisting of four seven-segment elements. The *Signal Panel* is located at the flight test engineer's station and consists of a single display which can provide the value (sign and magnitude) of any one of sixteen signals from each lane. The signal is selectable from push button switches located on the panel. The display consists of five seven-segment elements for the signals.

Figure 9.1-1~3 shows the layout of the three panels.

9.1 Inputs

Most of the input variables to the display module are those that have been computed inside the FCC. The only external input variable is the following:

Description	Units	Range	Resolution
Signal Display Indicator (SIGIN)	–	1/16	–

9.2 Outputs

Description	Units	Range	Resolution
Mode Word 1 (MWORD1)	–	$0/2^{16}$-1	–
Mode Word 2 (MWORD2)	–	$0/2^{16}$-1	–
Mode Word 3 (MWORD3)	–	$0/2^{16}$-1	–
Mode Word 4 (MWORD4)	–	$0/2^{16}$-1	–
Mode Word 5 (MWORD5)	–	$0/2^{16}$-1	–
MAB Fault Status Word 1 (ABFWORD1)	–	$0/2^{14}$-1	–
MAB Fault Status Word 2 (ABFWORD2)	–	$0/2^{14}$-1	–
MAC Fault Status Word 1 (ACFWORD1)	–	$0/2^{14}$-1	–
MAC Fault Status Word 2 (ACFWORD2)	–	$0/2^{14}$-1	–
Signal Word 1 (SWORD1)	–	$0/2^{14}$-1	–
Signal Word 2 (SWORD2)	–	$0/2^{14}$-1	–
Signal Word 3 (SWORD3)	–	$0/2^{15}$-1	–

9.3 Processing

9.3.1 Display Description

The content of the *Mode Display* will be a string of characters denoting the mode mnemonic; a typical mode display is shown in Figure 9.2-1. Table 9.1 provides the relation between the character symbol and the ASCII hexadecimal code. Table 9.2 defines the modes to be displayed.

The content of each *Fault Display* will depend upon the faults detected by the lane comparators located in each lane. If no fault is detected, the display will read "PASS". Otherwise, the display will correspond to a code as shown in Table 9.3. A typical fault display is shown in Figure 9.2-2.

Figure 9.2-3 shows a seven-segment digit. The segments are denoted "A" -"F" in a clockwise direction starting at the top with the center bar being segment "G". Each digit maps to seven bits, one for each segment of the display, as shown in Table 9.4. A segment of a digit is turned on by setting its corresponding bit in a seven-bit field of an output word to zero (the digits use negative logic). The mapping from segments to symbols is as shown in Table 9.5.

The content of the *Signal Display* will be a symbol corresponding to the selected display signal which is a number consisting of two sign indicators, six decimal point indicators and five seven-segment digits; a typical readout is shown in Figure 9.2-4. Table 9.6 provides the relation between the components and the function appearing in the number display. Table 9.7 describes the data types to be displayed as indicated by the SIGIN indicator.

9.3.2 Word Descriptions

Assume that the display driver that would be a part of the final program extracts the necessary data from the word format and ignores the unneeded bits. An 0 in a particular bit position indicates that bit should be set to zero. "(" and ")" mark the bounds of a digit field.

9.3.2.1 Mode Words

The mode display is driven by a five word output as shown in Figure 9.3-1. Digits D_1 through D_{10}, coded in hexadecimal format, represent a maximum of 10 characters. D_1 represents the first letter of the mode; remaining letters will be transmitted in sequence.

9.3.2.2 Fault Status Words

Each fault status display is driven by two two-word outputs as shown in Figure 9.3-2. Words 1 and 2 provide display data for 4 seven-segment elements which correspond to the fault status. D_1 represents the first letter of the fault status; remaining letters will be transmitted in sequence. One of these outputs concerns with the fault status of monitor AB, while the other one concerns with the fault status of monitor AC.

9.3.2.3 Signal Data Words

Each signal data display is driven by a three-word output as shown in Figure 9.3-3. Words 1 through 3 provide display data for 5 seven-segment elements. The signal is to be transmitted upon receipt of the signal indicator code from the flight test engineer given in Table 9.7. The signal display format is shown as follows:

Signed Decimal:	Signed fixed point numbers ranging from -99999. to -.00001, .00000, and +.00001 to +99999. The value being displayed is to be rounded to 5 significant digits during conversion. Values in the open interval (-0.000005, 0.000005) are displayed as ".00000" . Values greater than +99999. are displayed as "+99999.", and similarly values less than -99999. are displayed as "-99999." .

The indicator bars used for sign indication (S_1 and S_2) are turned on by setting a zero in the appropriate bit of Word 3 of the display control variable. The decimal point indicators P_1 through P_6 use positive logic and are turned on by setting a one in the appropriate bit of Word 3 of the display control variable.

9.4 Special Requirements

The computed values of the display control words must be passed to the voting routine VOTEDISPLAY as soon as they have been calculated. **(deletion)**

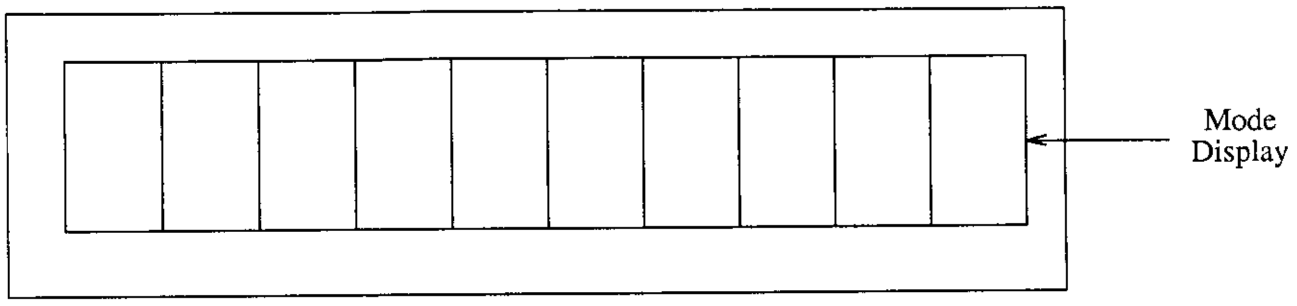


Figure 9.1-1: Mode Display Panel

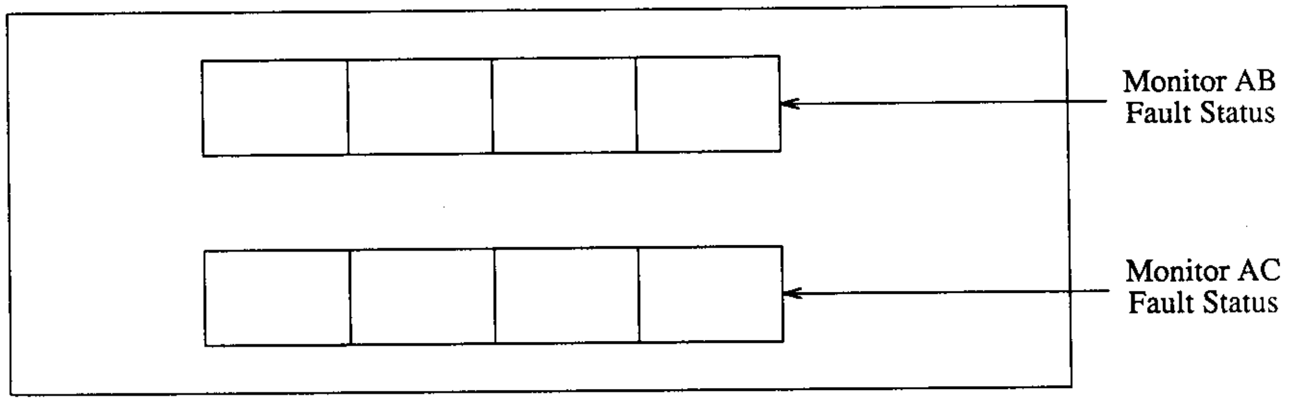


Figure 9.1-2: Fault Status Display Panel

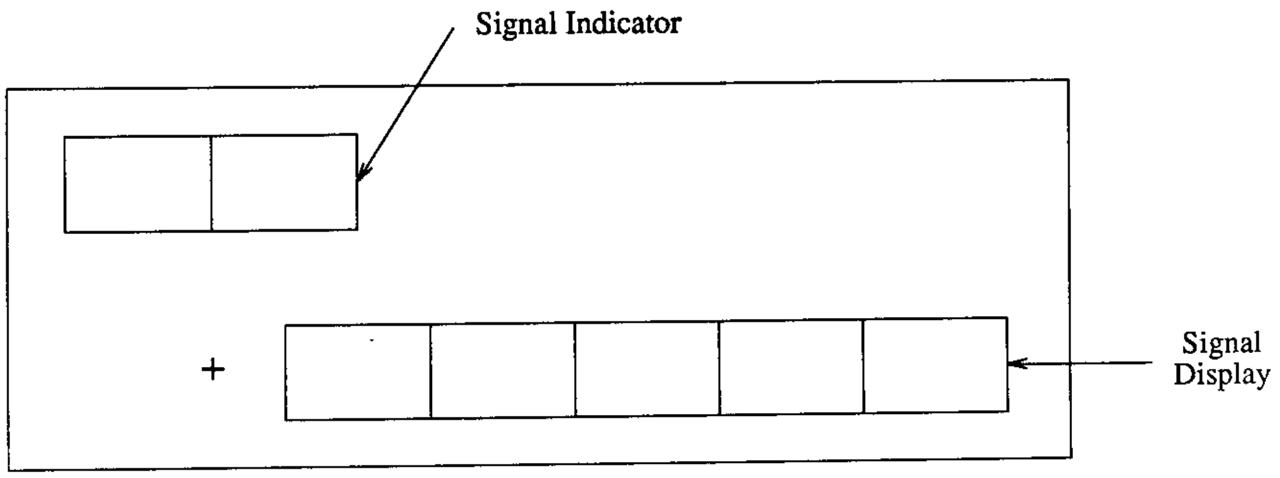


Figure 9.1-3: Signal Display Panel

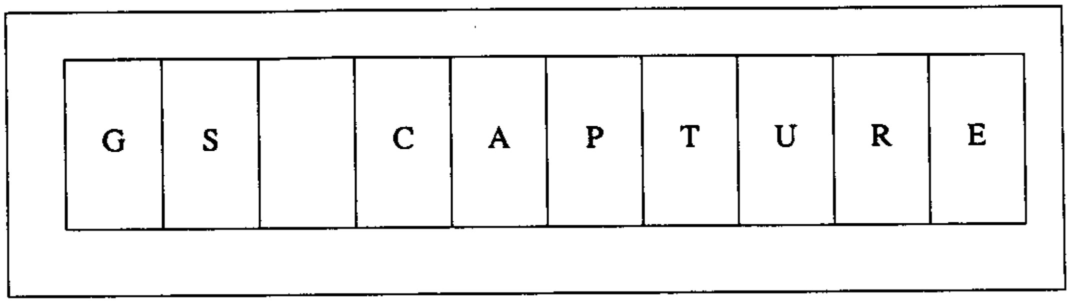


Figure 9.2-1: Typical Mode Display

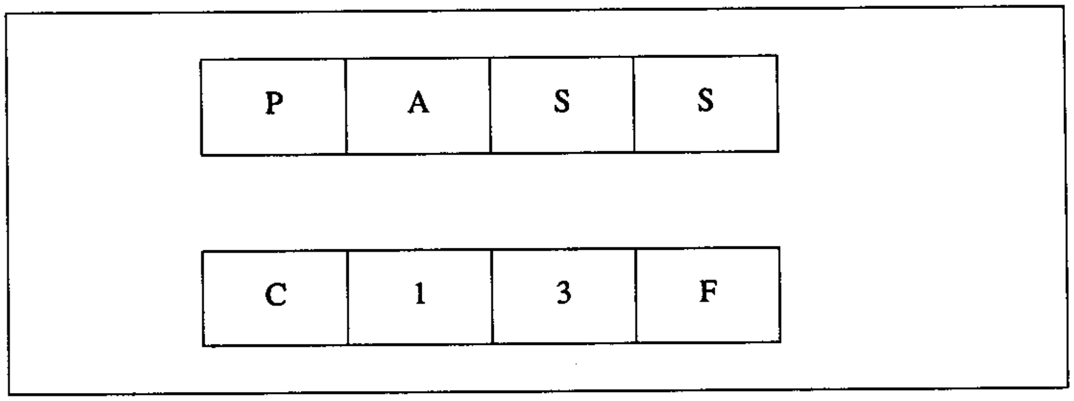


Figure 9.2-2: Typical Fault Display

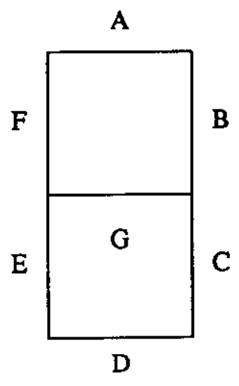


Figure 9.2-3: Seven-Segment Element

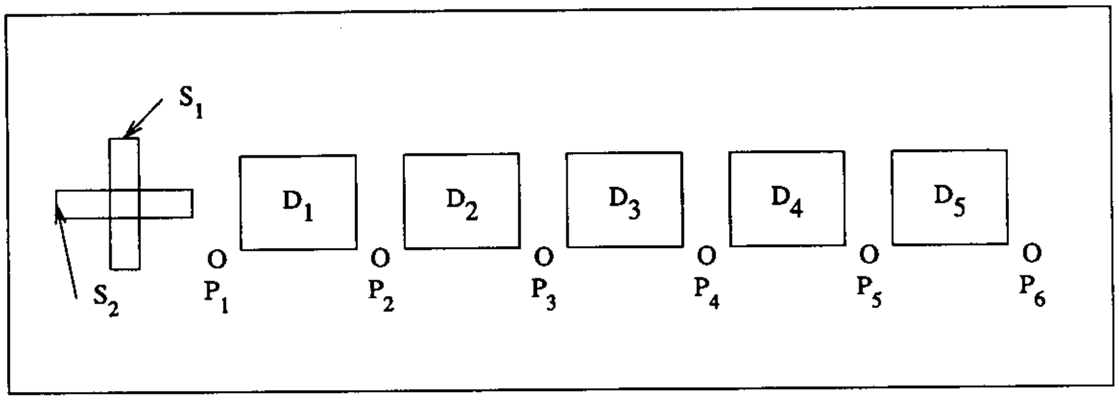


Figure 9.2-4: Typical Signal Panel Readout

Word 1

MSB															LSB
(			D_1				)	(			D_2				)
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Word 2

MSB															LSB
(			D_3				)	(			D_4				)
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Word 3

MSB															LSB
(			D_5				)	(			D_6				)
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Word 4

MSB															LSB
(			D_7				)	(			D_8				)
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Word 5

MSB															LSB
(			D_9				)	(			D_{10}				)
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 9.3-1: Mode Display Data Words

Word 1

MSB															LSB
0	0	(			D_1			)	(			D_2			)
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Word 2

MSB															LSB
0	0	(			D_3			)	(			D_4			)
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 9.3-2: Fault Status Display Data Words

Word 1

MSB															LSB
0	0	(			D_1			)	(			D_2			)
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Word 2

MSB															LSB
0	0	(			D_3			)	(			D_4			)
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Word 3

MSB															LSB
0	S_1	S_2	P_1	P_2	P_3	P_4	P_5	P_6	(			D_5			)
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 9.3-3: Signal Display Data Words

Character	ASCII (hexadecimal)
A	41
B	42
C	43
D	44
E	45
F	46
G	47
H	48
I	49
J	4A
K	4B
L	4C
M	4D
N	4E
O	4F
P	50
Q	51
R	52
S	53
T	54
U	55
V	56
W	57
X	58
Y	59
Z	5A
BLANK	20

Table 9.1: Relation Between Characters and ASCII Code

	Mode Status	Symbols to Be Displayed									
o	Altitude Hold Mode	A	L	T		H	O	L	D		
o	Guide Slope Capture Mode	G	S		C	A	P	T	U	R	E
o	Guide Slope Track Mode	G	S		T	R	A	C	K		
o	Flare Mode	F	L	A	R	E					
o	Touchdown	T	O	U	C	H	D	O	W	N	

Table 9.2: Displayed Modes

Comparator Status	Related Output	Display
Monitor AB Fail	ABFWORD1, ABFWORD2	C12F
Monitor AC Fail	ACFWORD1, ACFWORD2	C13F

Table 9.3: Fault Status Conditions

	MSB						LSB
Segment:	A	B	C	D	E	F	G
Bit:	6	5	4	3	2	1	0

Table 9.4: Segment-To-Bit Mapping

Symbol	Segments
0	ABCDEF
1	BC
2	ABDEG
3	ABCDG
4	BCFG
5	ACDFG
6	ACDEFG
7	ABC
8	ABCDEFG
9	ABCFG
A	ABCEFG
B	CDEFG
C	ADEF
D	BCDEG
E	ADEFG
F	AEFG
H	BCEFG
I	EF
N	ABCEF
P	ABEFG
S	ACDFG
Blank	none

Table 9.5: Symbol to Segment Mapping

Component	Function
S_1	Vertical Bar.
S_2	Horizontal Bar. S_2 is used as a minus sign for displaying negative numbers. S_1 and S_2 together form a plus sign for positive numbers.
P_1	Decimal Points. These allow a display range of .00001 up to 99999.
P_2	
P_3	
P_4	
P_5	
P_6	
D_1	Digits of the display. D_1 is the most significant digit, D_5 is the least.
D_2	
D_3	
D_4	
D_5	

Table 9.6: Signal Display Relation of Components to Function

Value of SIGIN	Signal to Display	Variable Name
1	Baro Altitude Estimate	BAE
2	Pitch Attitude	PA
3	Pitch Attitude Rate	PAR
4	Flight Path	FP
5	Altitude	H
6	Altitude Rate	HR
7	Vertical Acceleration	VA
8	Rad Alt G/S Filter	RAGSF
9	G/S Rate Error	GSRE
10	G/S Error Lambda	GSEL
11	G/S Dev Rate Limited	GSDRL
12	G/S Dev Estimate	GSDE
13	Rad Alt Estimate	RAE
14	Rad Alt Rate Est	RARE
15	G/S Deviation	GSD
16	Lane Command	LC

Table 9.7: The 16 Signal Data Types

APPENDIX A
SYSTEM DIAGRAM SYMBOLS

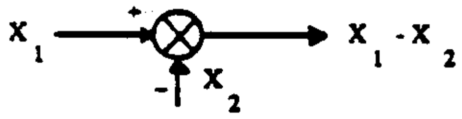
A1 Scope

This appendix provides guidelines for interpreting requirements stated in the form of system block diagrams of the type included in this document. These diagrams are a typical feature of current specifications for flight control systems. Signal flow, elementary functions and computation procedures are covered in the following discussion. References are to Figure 3.1, Barometric Altitude Complementary Filter, from Chapter 3.

A2 Signal Flow $\longrightarrow$

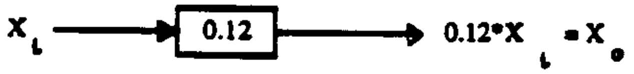
Signal flow, denoted by a line with an arrow, shows the time sequence of signal processing. The signal VA and the signal from I4 flow into summer SU19. The operation performed by SU19 is completed before an output flow to SU20 takes place. Similarly, the operation performed by SU20 is completed before an output flow to I2 can take place. However, all operations included in the entire diagram will be completed in a single frame of computation.

A3 Summers

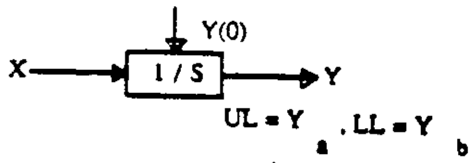


Summers are symbolized by a circle with intersecting lines and designated by the expression SU19, SU20, etc. A summer acts to provide as an output the sum or difference of its inputs in accordance with the sign associated with its input signal flow arrows. The sum of its inputs is outputted by SU19, but SU23 outputs H minus BAE. When two or more summers are connected by signal flow lines without intervening functions, the inputs may be combined as a single summer. Thus, SU19, SU20, and SU22 could be treated as a single summing junction.

A4 Gains



Gains are symbolized by blocks containing numbers or mnemonics which represent the desired gain, the amount by which the incoming signal is to be multiplied. K2 represents a gain of 0.15, which is to be applied to the output of SU24.



A5 Path Integrators

Path integrators are symbolized by the expression 1/s within a block, and designated by the expression I2, etc. A path integrator acts to provide the time integral of an input quantity, so that its output consists of a previous value (integral of the previous frame) plus the integral of the input generated in the current frame. Initialization of an integrator consists of setting the output to a desired initial value before proceeding with the calculation of the integral of the input.

It may be desirable to use a common digital algorithm for integration, in order to assure signal matching in all versions. If trapezoidal integration is used, the following algorithm applies. The input X(T-1) is set to zero during the initialization of a trapezoidal integrator.

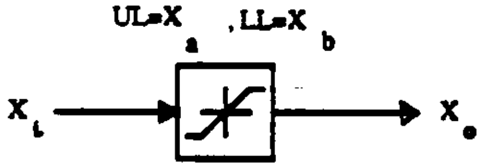
$$Y(T) = Y(T-1) + 0.5 * \Delta t * (X(T-1) + X(T))$$

$$Y(T-1) = Y(T)$$

$$X(T-1) = X(T)$$

where Y(T) is integrator output at end of computation,
Y(T−1) is integrator output at beginning of computation,
X(T) is integrator input during the current computation frame,
X(T−1) is integrator input during the previous computation frame,
Δt is computation frame time interval.

When two limiting values are specified outside of an integrator, the integrator output is limited by the Lower and Upper bounds before it is output and stored as the history for the next computation.

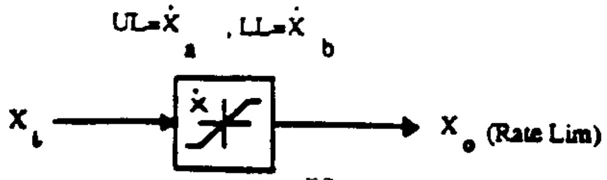


A6 Magnitude Limiter

Magnitude limiters are usually symbolized by a diagonal line within a block having one or both ends bent to show the nature of the limiting. The limiting value is often provided by notation outside or inside the block, such as +/- 500, UL 500/LL -200, +500/-200, etc. The designation for a magnitude limiter is LM9, etc.

Magnitude limiting calls for restricting the output from a given input to the defined limits, and no more. The signal within the limit region is transmitted without change (unity gain).

A7 Rate Limiter

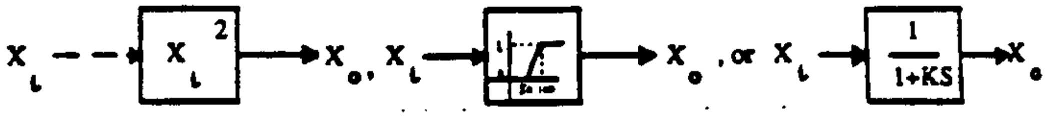


Rate limiters are similar to magnitude limiters, except that the rate of change of the output is limited rather than the magnitude. The designation for a rate limiter is LR. Rate of a variable is computed by dividing the difference of the variable in two successive frames by the frame duration:

$$\dot{X} = (X(T) - X(T-1)) / \Delta t$$

Magnitude limiting is performed on $\dot{X}$. If $\dot{X}$ violates the specified limits, then X(T) has to be recomputed such that the new value of $\dot{X}$ will be equal to either the upper or the lower bound, depending on the type of the violation. X(T) has then to be retained as the history for the next frame computation. Rate limiters are always initialized to the input quantity.

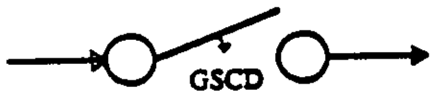
A8 Function



Functions are indicated in blocks which describe (by formula or by graphs) either a nonlinear operation or a linear transfer function filter. Linear filters are frequency sensitive and therefore contain functions of s, showing that integration is required. The designation for a **function** is F.

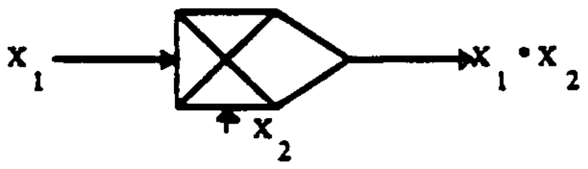
Linear filters to be mechanized in the experiment are first-order lag type, meaning that a single integration is to be performed. A procedure for generating a computing algorithm for this filter is suggested in Figure A.1. **The input X(T-1) is set to zero during the initialization of a linear filter.**

A9 Switch



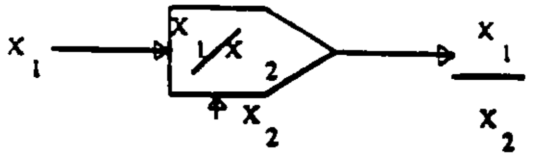
Switching functions are symbolized by a two-position switch whose blade position is determined by a logic statement. The logic statement is normally placed next to the contact to which it applies. A truth of the statement means the switch is to be closed. The designation for a switch is SW.

A10 Multiplier



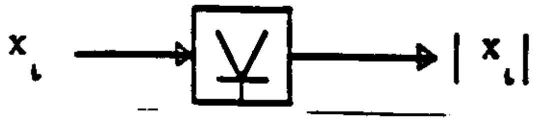
Multipliers are normally symbolized by a block containing cross- diagonals indicating time-variable multiplication. While the diagrams show this function, the present design usually requires only constant multiplication. The designation for a multiplier is M.

A11 Divider



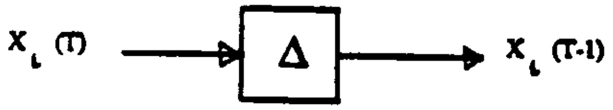
Dividers are normally symbolized by a block containing a single diagonal indicating time variable division. Designation for a divider is D.

A12 Absolute Value



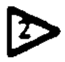
Absolute value is indicated by a "V" imposed on a Cartesian coordinate pair within a block. The input quantity is transformed to a magnitude independent of sign.

A13 Frame Delay



Frame delay is indicated by a triangle symbol within a block. The input discrete is delayed one frame. Application is usually to ensure latching of a new mode state for the condition that the enabling discrete appears once.

A14 Drawing Note



A note to the system drawing is indicated by a number within a triangle symbol.

A15 Computation Procedures

Each integrator in the diagram should complete the integration process before the output is computed, so that the output includes the most recent input data.

Figure A.2, Barometric Altitude Complementary Filter, is a modified version of Figure 3.1 from Chapter 3. It provides the basis for an example of a procedure for computing BAE in terms of VA, HR and H.

A15.1 Computation Sequence

a. Label each interior signal point with a convenient symbol, such as X1, X2, ... X14, as shown.

b. Define the integrator initial conditions, based on mode entry.

c. Write the system summing point and gain equations associated with integrator initial conditions. Generally best to work backwards from last integrator in loop -- in this case, I3.

d. Write integrator equations in an order such that each integrator receives the input corresponding to a previous integration (compute **X2, X5, then X9** last.)

e. Assign the integration past value terms in preparation for the next pass.

Example:

Initial Conditions:

```
X5o = HR(0)          X3o = 0.0
X9o = H(0)           X8o = 0.0
X2o = 0.0            X12o = 0.0
```

Program:

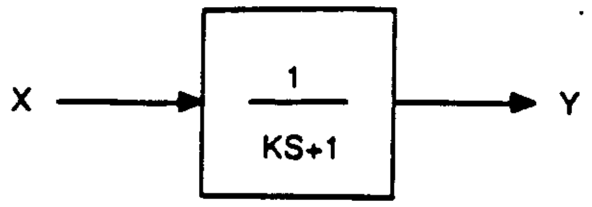
```
    X10 = H-X9o
    X6 = HR-X5o

    If -500<=X10<=500 then X11 = X10
    If -500>X10 then X11 = -500
    If X10>500 then X11 = 500

    X7 = K1*X11
    X12 = K4*X11
    X2 = X2o +0.5*T(X12+X12o)
{Limit function similar to that for X5}
    X1 = VA+X2

    X13 = K3*X11
    X14 = K2*X6
    X4 = X13+X14
    X3 = X1+X4

    X5 = X5o +0.5*T(X3+X3o)
    If X5<-32 then X5 = -32
If X5>32 then X5 = 32
X8 = X5+X7

X9 = X9o +0.5*T(X8+X8o)
{Limit function similar to that for X5}
    BAE = X9

    X5o = X5
    X3o = X3
    X9o = X9
```

```
X8o = X8
X2o = X2
X12o = X12
Return
```

Note: 1) The above example is for your reference only. You are encouraged to come up with a different computation sequence as long as it is correct.

2) The effort to save the value of each test point is not shown in this example.

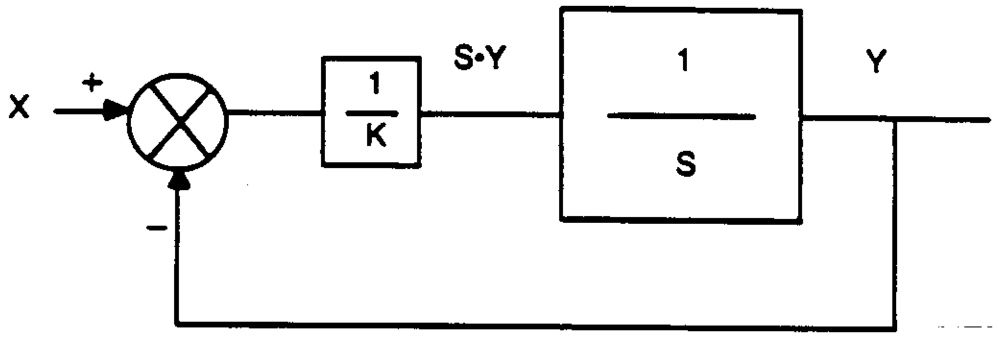
Original Filter Diagram :



$$\frac{Y}{X} = \frac{1}{KS+1}$$

$$S \cdot Y = (1/K)(X-Y)$$

Equivalent Diagram :



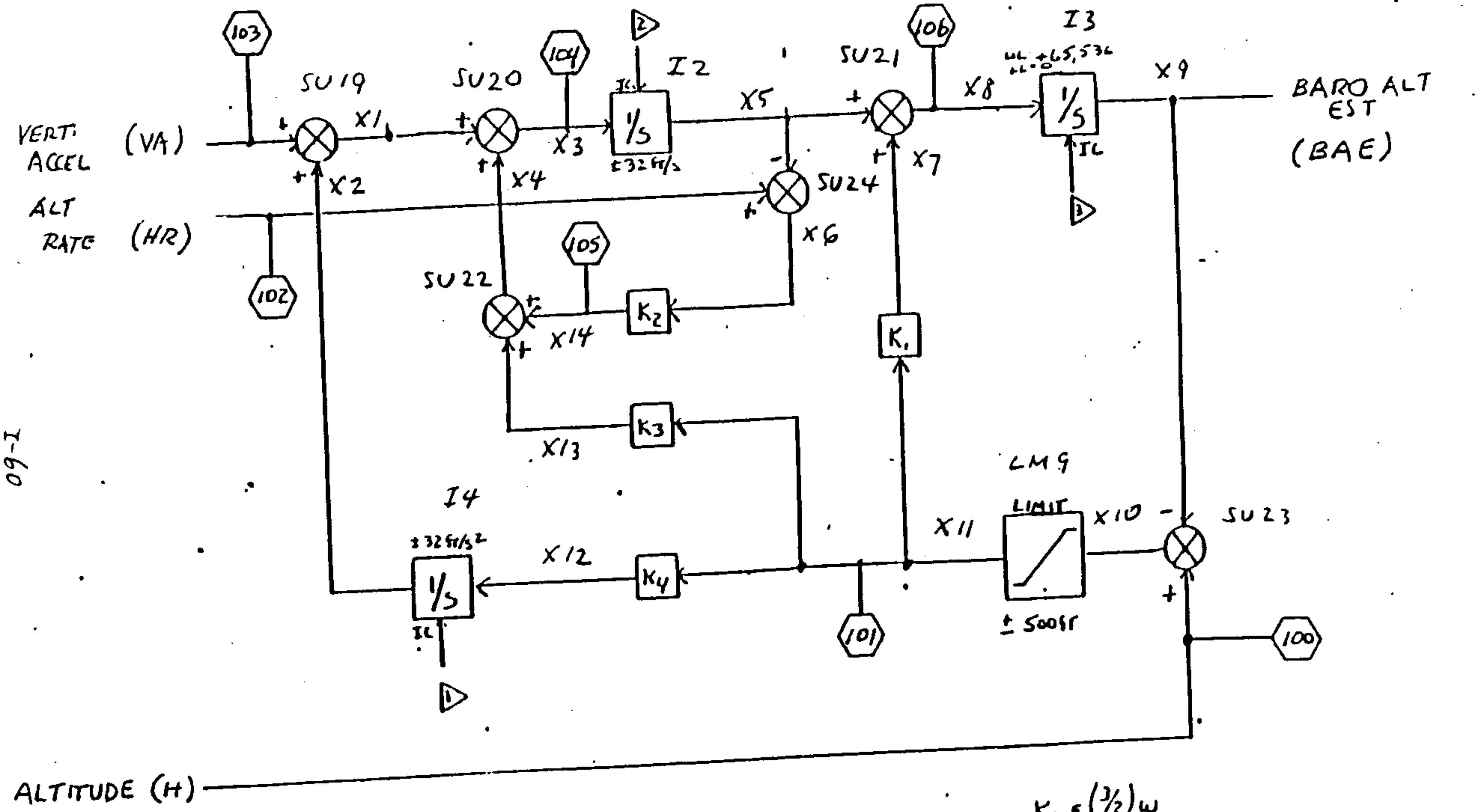
Computing Algorithm :

$$Y_T = Y_{T-1} + \frac{\Delta t}{2K}\left[(X-Y)_T + (X-Y)_{T-1}\right]$$

$$Y_{T-1} = Y_T$$

$$(X-Y)_{T-1} = (X-Y)_T$$

FIGURE A.1 Filter Algorithm



1 - IC TO ZERO @ MODV

2 - IC TO $\dot{h}$ @ MODV ; $\dot{h}$ = HR

3 - IC TO h @ MODV , h = H

FIGURE **A.2**

BAROMETRIC ALTITUDE COMPLEMENTARY FILTER

$K_1 = (3/2)\omega$
$K_2 = (3/2)\omega$
$K_3 = (3/4)\omega^2$
$K_4 = \omega^3$
$\omega = .1$ RAD/SEC

APPENDIX B
PARAMETER LIST DEFINITIONS OF EXTERNAL ROUTINES

B1 SENSORINPUT Routine

```
SENSORINPUT (
            var H : real;                       { Altitude }
            var HR : real;                      { Altitude Rate }
            var VA : real;                      { Vertical Acceleration }
            var RA : real;                      { Radio Altitude }
            var GSD : real;                     { Glide Slope Deviation }
            var MODV : Boolean;                 { Model Valid }
            var PA : real;                      { Pitch Attitude }
            var PAR : real;                     { Pitch Attitude Rate }
            var FP : real;                      { Flight Path }
            var EQ : real;                      { Equalization }
            var SIGIN : integer;                { Signal Display Indicator }
            );
```

B2 VOTEFILTER1 Routine

```
VOTEFILTER1 (
            var RAE : real;                     { Radio Altitude Estimate }
            var RARE : real;                    { Radio Altitude Rate Estimate }
            tp_110: real;                       { internal test point }
            );
```

B3 VOTEFILTER2 Routine

```
VOTEFILTER2 (
            var BAE : real;                     { Baro Altitude Estimate }
            var RAGSF : real;                   { Radio Altitude G/S Filter }
            var GSRE : real;                    { G/S Rate Error }
            var GSDE : real;                    { G/S Deviation Estimation }
            var GSEL : real;                    { G/S Error Lambda }
            var GSDRL : real;                   { G/S Deviation Rate Limited }
            tp_101, tp_104, tp_105, tp_106: real; { test points of Baro Altitude Filter }
```

```
        tp_121, tp_122, tp_128, tp_129, tp_130: real; { test points of G/S Filter }
        );
```

B4 VOTEMODE Routine

```
VOTEMODE (
        var AHD : Boolean;                    { Altitude Hold Discrete }
        var GSCD : Boolean;                   { G/S Capture  Discrete }
        var GSTD : Boolean;                   { G/S Track Discrete }
        var FD : Boolean;                     { Flare Discrete }
        var TD : Boolean;                     { Touchdown Discrete }
        tp_9, tp_11, tp_61, tp_62, tp_65, tp_66, tp_67: real; { test points }
        );
```

B5 VOTEOUTER Routine

```
VOTEOUTER (
        var THCI : real;                      { Theta Command Integral }
        var FPEC : real;                      { Flight Path Error Command }
        var FPDC : real;                      { Flight Path Damping Command }
        tp_60: real;                          { common test point for all control laws }
        tp_61, tp_62, tp_63, tp_65, tp_66, tp_67: real; { G/S mode test points }
        tp_80, tp_81, tp_82, tp_83, tp_84, tp_85, tp_87: real; { Flare mode test points }
        );
```

B6 VOTEINNER Routine

```
VOTEINNER (
        var LC : real;                        { Lane Command }
        tp_2, tp_3, tp_4, tp_5, tp_6, tp_7: real; { Inner Loop test points }
        );
```

B7 LANEINPUT Routine

```
LANEINPUT (
        var LCB : real;                       { Lane B Command }
        var LCC : real;                       { Lane C Command }
        );
```

B8 VOTEMONITOR Routine

```
VOTEMONITOR (
            MAB : Boolean;                  { Monitor State B by A }
            MAC : Boolean;                  { Monitor State C by A }
            var RECOVERY : Boolean;         { Recovery Command from Dedix }
            );
```

B9 VOTEDISPLAY Routine

```
VOTEDISPLAY (
            MWORD1 : integer;               { Mode Word 1 }
            MWORD2 : integer;               { Mode Word 2 }
            MWORD3 : integer;               { Mode Word 3 }
            MWORD4 : integer;               { Mode Word 4 }
            MWORD5 : integer;               { Mode Word 5 }
            ABFWORD1 : integer;             { MAB Fault Status Word 1 }
            ABFWORD2 : integer;             { MAB Fault Status Word 2 }
            ACFWORD1 : integer;             { MAC Fault Status Word 1 }
            ACFWORD2 : integer;             { MAC Fault Status Word 2 }
            SWORD1 : integer;               { Signal Word 1 }
            SWORD2 : integer;               { Signal Word 2 }
            SWORD3 : integer;               { Signal Word 3 }
            );
```

B10 VOTESTATES Routine

```
VOTESTATES (
            { input of I2 in Figure 3.1 }   { Barometric Altitude Complementary Filter }
            { output of I2 in Figure 3.1 }
            { input of I3 in Figure 3.1 }
            { output of I3 in Figure 3.1 }
            { input of I4 in Figure 3.1 }
            { output of I4 in Figure 3.1 }
            { input of I5 in Figure 3.2 }   { Radio Altitude Complementary Filter }
            { output of I5 in Figure 3.2 }
            { input of I6 in Figure 3.2 }
            { output of I6 in Figure 3.2 }
            { input of F8 in Figure 3.2 }
            { output of F8 in Figure 3.2 }
            { input of I7 in Figure 3.3 }   { Glide Slope Complementary Filter }
            { output of I7 in Figure 3.3 }
            { input of I8 in Figure 3.3 }
{ output of I8 in Figure 3.3 }
{ input of I9 in Figure 3.3 }
{ output of I9 in Figure 3.3 }
{ input of I10 in Figure 3.3 }
{ output of I10 in Figure 3.3 }
{ input of F10 in Figure 3.3 }
{ output of F10 in Figure 3.3 }
{ input of F1 in Figure 4.2 }         { Mode Logic }
{ output of F1 in Figure 4.2 }
{ input of F2 in Figure 4.2 }
{ output of F2 in Figure 4.2 }
{ input of F1 in Figure 6.1 }         { outer loop of G/S Capture and Track Mode }
{ output of F1 in Figure 6.1 }
{ input of F2 in Figure 6.1 }
{ output of F2 in Figure 6.1 }
{ input of F6 in Figure 7.1 }         { outer loop of Flare Mode }
{ output of F6 in Figure 7.1 }
{ input of F7 in Figure 7.1 }
{ output of F7 in Figure 7.1 }
{ input of I1 in Figure 5~7.1 }       { inner loop }
{ output of I1 in Figure 5~7.1 }
{ output of LR1 in Figure 5~7.1 }
{ output of LR2 in Figure 5~7.1 }
{ input of I11 in Figure 8 }          { Command Monitor AB }
{ output of I11 in Figure 8 }
{ input of I11 in Figure 8 }          { Command Monitor AC }
{ output of I11 in Figure 8 }
);
```

Note: 1) Boolean type variables should be represented by an integer in which "1" means true and "0" means false, and nothing else.

2) Parameter names in VOTESTATES routine contain those internal states of your program and are therefore not defined here. Whenever some state variables are not defined or are not up-to-date because the corresponding part of the computation was not performed in the current frame, default values of 0.0 should be passed to VOTESTATES.

3) The names of the test point variables ("tp_<id>") are generic and primarily define the *required sequence* in the parameter list of the voter routines; you can use any name you like.

4) **Whenever some parameters in a voting routine are not defined or are not up-to-date because the corresponding part of the computation was not performed in the current frame, default values of 0.0 should be passed to that voting routine.**

APPENDIX II. Forms and Guidelines Used in the Experiment

UCLA / HONEYWELL JOINT PROJECT

FCS DESIGN UTILIZING N-VERSION PROCESSING

FORMS AND GUIDELINES

June–15–1987

to

September–4–1987

The forms and guidelines broadcast in the experiments are listed as follows:

1. UCLA/Sperry Software Fault Tolerance Project Application Form (*April 1987*)
2. UCLA/Sperry Software Fault Tolerance Project Intention Form (*May 5, 1987*)
3. Rules and Guidelines For Programmers (*June 15, 1987*)
4. Design Document Outline (*June 15, 1987*)
5. Weekly Progress Report (*June 19, 1987*)
6. Design Walkthrough Keypoints (*July 6, 1987*)
7. Design Walkthrough Checklist (*July 6, 1987*)
8. Design Walkthrough Report (*July 13, 1987*)
9. Code Development Plan & Unit Testing Plan (*July 14, 1987*)
10. How to Use Code Update Sheets & Code Update Report (*July 27, 1987*)
11. On-line Program Milestones (*July 29, 1987*)
12. Coding/Testing Meeting Guidelines (*Aug 9, 1987*)
13. Final Meeting Guidelines (*September 2, 1987*)
14. Fault-Tolerance Project Post-Experiment Questionnaire (*September 4, 1987*)

UCLA/SPERRY Software Fault Tolerance Project Application Form *(April 1987)*

NAME ______________________________ Login-id ____________ Phone __________(o) DATE ______________
Last First __________(h)

A. EDUCATION BACKGROUND

1. Please list your degree(s) awarded, and degree being sought:

School	Department	Degree	from	to

2. How many hours of computer science courses have you taken? graduate: ______________
undergraduate: ______________

3. If employed at UCLA, describe position: ______________________________________

4. Please list graduate courses you have taken (with grade) or are taking in the spring quarter:

Course number or title	Quarter	Year	Grade

5. Please list any other training (e.g., industry sponsored seminars) in computer science you have had.

____
____
____

B. GENERAL SOFTWARE ENGINEERING EXPERIENCE

1. How many years (or months) of full-time programming experience have you had? ______________

2. How many years (or months) of part-time programming experience have you had? ______________

3. How many years (or months) of experience have you had performing the following functions?

preparing design description	______________
programming	______________
testing programs	______________

4. Please evaluate your knowledge of the following programming languages:

Language	Fluency (0:novice, 5:expert)	Longest lines of code written	Preference order (1:best, 7:worst)
Ada			
C			
Fortran			
Lisp			
Modula-2			
Pascal			
Prolog			

Any other programming languages you are familiar with? (please identify fluency)

____

____

____

5. Please evaluate your knowledge of the Unix system:

System	Fluency (0:novice, 5:expert)	Years of experience
Unix		

C. SPECIFIC SOFTWARE ENGINEERING EXPERIENCE

In the space below, please describe the software development projects that you have participated in. Attach additional pages if necessary.

Including the following in your description:

1. Project Title

2. Project Size (i.e. lines of code, no. of modules, no. of participants)

3. Project Duration

4. Development Environment (i.e. programming language, operating system, command language, development techniques)

5. Your Responsibilities (i.e. full-time or part-time involvement, percent of different activities performed)

UCLA/SPERRY Software Fault Tolerance Project *(May 5, 1987)*

NAME ______________________________ Login-id ____________ Phone ____________(o)
Last First ____________(h)

1. Do you have an office right now? yes ____________ no ____________

   if yes, where is your office? ____________________

2. Are you available during the whole summer? (from 6/15 to 9/4) yes ________ no ________

   if no, when are you not available? ______________________________

3. Please indicate the time which is NOT good for you to have a weekly meeting this quarter:

Time	*8-9*	*9-10*	*10-11*	*11-12*	*12-1*	*1-2*	*2-3*	*3-4*	*4-5*	*5-6*
Monday										
Tuesday										
Wednesday										
Thursday										
Friday										

4. Are you interested in this project?

yes, very ______ yes ______ not sure ______ not quite ______ not at all ______

5. Would you be willing to take the position if you get the offer?

definitely ________ possibly ________ not sure _______ no ________

Rules and Guidelines For Programmers

UCLA/Sperry Summer Project

June 15, 1987

1 Work Environment

1.1 First Day

On the first day you will be given slide presentations on the goals of the experiment, the phases of the experiment, and an overview of N-Version Programming. You will also be given all written training materials, empty log books, the application specifications you are to design, code, and test, and all other information you should need.

On this day, you will need to review all written materials and familiarize yourself with the schedule and rules. If necessary, read about and use any Unix tools you are not comfortable with. You should log onto your Unix account to make sure it is set up, see if you have any mail, and create the first-level subdirectory "Timesheets". This directory will hold all your online timesheets, that you will create daily with the "tsheet" program. (Note : The timesheets will not be used for payroll or accounting purposes, they will be used to determine how much time "really" was needed to develop the required program.)

1.2 Your Unix account

Each student has a Unix account. Each team has a different Unix group id, so each team's online work will be read and write protected from all others, yet members of one team will be able to freely read and write in each other's directories and files. Do not change protections on any work-related files.

A "calendar" file will be provided to you by mail, please put it into your home directory. It causes mail reminders of due dates, etc. to automatically be mailed to your account on the appropriate days. You may append to the calendar file as you wish, but do not remove the file.

1.3 Time and Time sheets

You must work a minimum of 40 hours a weeks. Any time over 40 hours a week is optional; you will not be paid for overtime, but you are expected to meet all deadlines and complete this project. No strict work schedule will be enforced; you may work any combination of hours that adds up to 40 per week, but you must coordinate with your teammate so both of your schedules coincide as much as necessary.

You will log your time every day you work. An online log has been provided for you. The system will remind you every afternoon if you are logged in to fill out your time log. If you do not fill out a log for any one day, you will receive a mail message the next morning asking you if you forgot to fill it out. Ignore it if you did not work the previous day, otherwise fill it out promptly. (The command to run the timesheet program is "tsheet".) The timesheet files made by this program will be stored in your home directory unless you create the directory "Timesheets" (note the upper case "T") as a first level subdirectory.

1.4 Communication Policy

1. General
   Communications are restricted during this experiment, as follows. You may openly communicate with your teammate on all aspects of the experiment. However, you may not discuss any aspect of your work in this job with members of other teams. Any work-related communication between you and the coordinating team is to be conducted via Unix mail. We require this so in the event of an error or ambiguity in the specifications, or some other significant event, all team members may be sent a copy of the mailed question and its answer.

   Every day you work you must log onto your Unix account. In this way you will receive in a timely manner all mail concerning answers to questions, any updates in the specifications, calendar reminders of important due dates, and you will be able to fill out your daily time sheets.

2. Administrative Problems and Questions
   All administrative or personnel problems and questions should be directed to the Professor or the Administrative Analyst, through the Unix mail system.

3. Site-Specific Problems and Questions
   All site-specific questions should be directed to the coordinating team. An exception is requests for maintenance of terminals, printers, etc. In any case, you should notify CECS and the coordinating team by mail if at any time your work is hindered by nonfunctioning hardware.

4. Technical Problems and Questions
   Direct all technical questions to the coordinating team through Unix mail. All valid technical questions and their answers will be broadcast to all programmers.

5. Responses to your Questions
   The coordinating team members will read their Unix mail once in the early

morning and again in the late evening (Monday through Friday guaranteed), to keep up with all work-related mail. (You should do the same.)

2 Phases of the experiment

2.1 Design of the Experiment .... (6/15—7/10)

Four weeks are allowed for this stage. A design document outline has been provided, and tells you what information is required in your preliminary and final design documents. At the end, you will have developed a design document, conducted a walkthrough of the design , made any changes found necessary by the walkthrough, and neatly formatted and printed the final design document.

Deliverables at this stage are daily time sheets and weekly progress reports (about 1 page summarizing the weeks activities, to be turned in every Friday afternoon). On Friday 7/10, a preliminary design document will be due. It will be used as the basis for a design walkthrough to be conducted during the following week (tentatively on Wednesday, 7/15 and Thursday, 7/16). After that, you will have to submit a final design document which includes the design walkthrough results.

2.2 Coding/Unit Testing .... (7/13—8/7)

Approximately four weeks are allowed for this stage. (A total of six weeks are allowed for the Coding/Unit testing and Validation Testing phases; each phase need not take you exactly the time alotted, but the total time allowed will not exceed six weeks.) Code and unit test responsibilities should be divided evenly between team members, and an order of development must be worked out. At the end of the first week, you will turn in a code development plan and a test plan that is consistent with the development plan. At approximately the end of the four-week period, you will have coded the design, conducted a code walkthrough and prepared a walkthrough report, and made any changes found necessary by the walkthrough. Please remember to use the RCS revision control to maintain your software development.

Deliverables of this stage are daily time sheets, weekly progress reports at the end of each week , a code development plan (including a list of who will code/unit test which parts) , a test plan description, and the compilable source programs.

2.3 Validation Testing .... (8/10—8/21)

Approximately two weeks are allowed for this stage also, as explained above.Several test cases will be provided to you in the beginning of of this phase. In this time you must complete your test harness program and test the program. During the testing you will keep a validation test log, detailing inputs, outputs, and program failures.

Deliverables in this phase include daily time sheets, weekly progress reports, and a completed validation test log.

2.4 Acceptance Testing .... (8/24—9/4)

Two weeks are allowed for this stage. You will hand in your program which will be run in a test harness, and will be subjected to a given number of input sets. Once your program passes all tests, you have completed your summer work obligation, and any remaining time before the 2-week work period is over becomes paid vacation time. If your procedure does not pass the first time it is subjected to acceptance testing, it will be returned to you with the input cases that it failed on, and you will be required to debug and reset your procedure. Then you once again submit it for acceptance testing. This process continues until your procedure passes an acceptance test.

Deliverables in this phase are daily time sheets, weekly progress reports, an acceptance test log (only if your procedure fails an acceptance test), final program, and a post-experiment questionnaire.

Design Document Outline

1. **Title page**
   (a) Title
   (b) Design version number
   (c) Authors
   (d) Team id
   (e) Date
2. **Table of Contents**
   (a) Shows chapters, sections, subsections
3. **Introduction (Chapter 1)**
   (a) Structural diagram showing abstraction layers, and the abstract data types in those layers.
   (b) All layers and abstract types named and labeled.
   (c) Brief (couple of paragraphs) describing design.
4. **Global Information (Chapter 2)**
   (a) List all global data types.
   (b) List all global variables.
   (c) List all global functions, procedures.
   (d) Brief comments describing the above.
   (e) Pseudocode of main program.
5. **Non-Global Information (Chapter 3–N)**
   (a) Chapter for each major function (there should be at least two modules per function), containing:
      i. Subsystem structure chart showing relationships of modules to that major function, and the computation sequence among modules.
      ii. Section for each procedure, containing:
         A. comment briefly describing its function
         B. module heading and interface to other modules
         C. all local types, variables, subprocedures, sub-functions defined and commented
         D. Pseudocode (combination of English and control structures) defining its computation algorithm and computation sequence.
6. **Summary of Design Walkthrough Results (The last Chapter, for final version only)**
   (a) For all problems found, brief comment on them and describe the solution.
   (b) Attach a design walkthrough sheet for each change.
7. **Index, glossary optional**

```
Please turn in a hardcopy (1-2 pages) of the weekly progress report.
The format should be something like:

-----------------------------------------------------------------

                UCLA/Sperry Joint Project

                 Weekly Progress Report

                      (team id)

                     (your names)

                   week #  (date)


(the text goes here)
...
...
...
...
...


-----------------------------------------------------------------
```

Design Walkthrough Keypoints

You will meet with experimenters for just one hour. The entire design walkthrough is expected to take much longer. You will conduct the second and subsequent hours privately.

Obviously, the entire detailed design cannot be reviewed in a formal way at a design walkthrough in any reasonable time. Instead, the presenters are asked to perform the following tasks:

1. Present the software development plan.

2. Present the program hierarchical functional definition and design architecture using data flow diagrams, flowcharts and explanatory narrative as appropriate.

3. Show that the program definition and design architecture are technically feasible and compatible and responsive to the software functional requirements.

4. Present implementation testing criteria, plans and procedures, and show that these are adequate.

5. Provide evidence that the design is complete: all modules are defined, and have specific and definitive interfaces.

6. Provide evidence that all technical requirements have been satisfied.

7. Report on status.

8. Answer the design walkthrough checklist questions on next pages.

July 6, 1987 at UCLA

Design Walkthrough Checklist

1. Is the design complete with respect to the specification? Have you considered all normal cases as well as boundary conditions?

2. Can you trace both from your design to the specification and from the specification to your design? Has anything been left out? Are all functions in the external specification reflected in the program? Has anything been included that should not be? Does the program do more than that stated in the external specification?

3. What is the global control flow?

4. Are all global data structures consistent?

5. Are all specified input variables used? Are all required output variables produced?

6. Is the design of your program modular? Examine the following features:

   - simple interfaces
   - small modules
   - low connectivity
   - top-down design
   - information hiding

7. Are all these modules well defined?

8. Is the decomposition complete? That is, can the logic of each module be easily visualized?

9. Do you use all the modules, data structures and variables? Are any modules, data structures or variables missing?

10. Is the algorithm for each module completely specified?

11. Are there multiple modules in the program that seem to perform the same function?

12. Is each module described by its function rather than by its context or internal logic? Has each function been stated accurately?

13. Are all interfaces precisely defined and consistently invoked? Is there any unnecessary redundancy in any interface?

14. Are the interface data to each module consistent with the definition of the module's function?

15. What is the implementability of the whole design? That is, is there any aspect of the design that is precluded by the programming language to be used?

16. Is the design obscure? Does it contain anything that might be easily misunderstood?

17. Have any unstated assumptions been made?

July 6, 1987 at UCLA

Design Walkthrough Report

Function (Module): ______________________ Date: ________________________

Change number: ______

From: (Identify boundaries of text in document where change will be made)

To: (Describe the change to be made)

Change type: ____________

Types are:

1: typo	6: spec ambiguity
2: error of omission	7: update to match new specs
3: unnecessary-deleted	8: efficiency increase
4: incorrect algorithm	9: other (explain: ________________)
5: spec misinterpretation	

Detected by: programming team _______ coordinating team _____

The cause of the fault:

____

The following information are required for these two documents which are due next Monday (7/20).

Code Development Plan (1 page):

* the hardware and software environment
* the language and the compiler version to be used
* partition and allocation of the coding duty
* tentative schedule

Unit Testing Plan (1-2 page):

* methods to be used on particular modules
  (walkthroughs, inspections, static analysis, dynamic tests,
  formal verification, etc.)
* test completion criteria
* available testing tools
* tentative schedule

-coordinating team

How to Use Code Update Sheets

1. General

The purpose of this document is twofold. First, it is a means of logging all results of your code walkthrough, so you may easily divide between you and your teammate the work to correct your software following the walkthrough. Secondly, it will serve to mark all the changes in your code during the testing phases as your program evolves.

2. Instructions for Completing

Use one sheet for each logical code change to be made. Fill out the Change number, Date, Module and Procedure sections on each sheet. The term "Procedure" is indicative here; use the corresponding unit in the language you are using.

In the ''From:'' section, identify the exact location of the code to be changed. For example, you may need to change Module X, Procedure Y, lines 2 through 4. If one fault causes more than one contiguous portion of code to be changed, clearly identify all portions on one change sheet.

In the ''To:'' section, you may either give a general description what the change will be or you may specify the exact code substitution (for simple changes).

Be sure to identify the type of change by its type code. If a change is of type 4 (incorrect algorithm), classify it furthermore as miscomputation (e.g., value miscomputed), logic fault (e.g., control flow fault, missing code), initialization fault or boundary fault (e.g., value out of bound, divide by zero). If a change does not fit one of the categories provided, identify it as type "other" and give explanation.

Finally, indicate during which phase the fault is corrected and give explanation about its cause (similar to the Design Walkthrough Report). Code update reports should be turned in each week (on Monday, together with the weekly progress report) for all the code updates of previous week.

Code Update Report

Team: ____________ Change number: _______ Date: ____________________
Function (Module): ________________________ Procedure: _______________
File name: __________________ RCS id: before change: _____ after change: _____
No. of corresponding Design Walkthrough Report (if appropriate): _______

From: (Identify boundaries of code to be changed/deleted, or where to be added)

To: (English, pseudocode, or language description of correction to be made)

Change type (mark as many as appropriate): ________

1: typo	6: spec ambiguity (on page _____)
2: error of omission	7: software reqs doc. update (question-answer No.____)
3: unnecessary-deleted	8: efficiency: (time__ storage__)
4: incorrect algorithm	9: clarity: (change structure__ add comments__)
5: spec misinterpretation (on page ____)	10: other (explain: ____________________________)

(option) What is it if type 4: miscomputation____ logic fault____ initialization fault____ boundary fault____
(option) Error due to peculiarities of language____ or of hardware/operating system____
(option) Error introduced by previous change (No.____): simply deleted____ further change____

Detected during: coding____ unit test____ integration test____ acceptance test____

Detected by: reading the code____ your own test data____ the coordinators' test data (test data id: ________)

Time spent in the change, including fault location (approximately):____________

Cause of the fault, or other comments:

____

____

(Use additional sheets if needed)

We plan to archive "snapshots" of your programs when certain milestones are reached: we are asking you to turn in these snapshots, i.e., when the corresponding milestone is reached, you should proceed as follows.

For each milestone, we will provide a directory, (the figure below shows the tree structure for all these directories) and you shall put into it a consistent copy of all the pertinent files, and mail us a message telling that the milestone has been reached and the copy has been put there.

The directory structure is as follows:
under zeus:/s/s1/dedix/Sperry
there are directories called "ucla.ada", "ucla.c", etc.

Each one of these contains the following directories:

```
ucla.* -------------------------------- AUTOLAND --------- 1.complete
                |                                   |
                |                                   +----- 2.tested
                |                                   |
                |                                   +----- 3.submitted
                |                                   |
                |                                   +----- 4.final
                |                                   |
                |                                   ------ harness
                |
                +---------------------- commonit --------- 1.complete
                |                                   |
                |                                   +----- 2.tested
                |                                   |
                |                                   +----- harness
                |
                +---------------------- display
                |
                +---------------------- filter.ba
                |
                +---------------------- filter.gs
                |
                +---------------------- filter.ra    ........
                |
                +---------------------- inner
                |
                +---------------------- modelog
                |
                +---------------------- outer.ah
                |
                +---------------------- outer.f
                |
                ----------------------- outer.gs
```

The directories pertaining to the modules contain subdirectories for two milestone snapshots; the one called AUTOLAND for four. Each also contains a directory called "harness", for the tests instruments you used.

By "all pertinent files" we mean:

- there must be a file, called 'SDDrefer', telling which are the numbers of the Design Walkthrough Report and the Code Change Report that describe the last change made before the milestone, i.e. all changes made before that are reflected in the snapshot, all changes after that are not.

- there must be a series of RCS files, sufficient for 'making' the objects, as explained in the next item, in the state they are at the time of the milestone. If this directive reaches you when you are already past the milestone, make sure your makefile extracts the RCS release corresponding to when the milestone was reached.

- there must be a makefile, containing commands to make the relevant object code ('all' for the whole autolanding package; filter.ba for the barometric complementary filter; filter.ra for the radio altitude complementary filter; etc.....); when one invokes "make" from within this directory, it must be able to execute without accessing anything outside the directory itself. In the makefile, all commands that check out a file from RCS must contain explicit revision numbers.

In this way, we can have a consistent copy to look at, and you can keep modifying your own files without concurrency problems.

The milestones are:

1) for each individual function (corresponding to the SRD):

   - first time that any given function is complete and ready for Unit Testing

   - first time that any given function passes Unit Testing with all the test data provided by the coordinators;

2) for the complete programs:

   - first time that it all compiles together;

   - first time it passes your own integration testing;

   - first time it is ready for acceptance testing;

   - first time it passes acceptance testing with all the acceptance test data (this is the last milestone!).

3) for the test harnesses, we only require that they are sufficient to test all the other programs you submit. We should be able to validate that your milestones indeed pass the required test by these test harnesses.

Dear Fellows:

8/12 (Wed) and 8/13 (Thur) will be the time of our scheduled meeting.
There will be three teams for each day, tentative schedule as follows:

Wednesday (8/12)	Thursday (8/13)
	Modula (9:00 --- 10:30 am)
C (10:00 --- 11:30 am)	Prolog (10:45 --- 12:15 pm)
T (1:00 --- 2:30 pm)	Ada (1:30 --- 3:00 pm)
Pascal (3:00 --- 4:30 pm)	Discussion with Sperry (3:30 --- 5:00 pm)

Again, if have any problems, please reply to us ASAP.
It is anticipated that each team should prepare slides and give
a detailed (of course within the time allotted) presentation about
what's been going on since the last walkthrough meeting.
However, since testing is the most important issue right now,
you don't have to spent too much time (say, less than half of a day)
in the preparation for the presentation.

There will be three topics to be presented: 1. Progress Report,
2. Testing Experience, 3. Program Statistics. Each team has one
hour for the presentation and half of an hour for the discussion.

Here is the suggested framework for each item:

Topic 1 (Progress Report) should discuss current status of each module,
including difficulties, problems encountered, and time spent.

Topic 2 (Testing Experience) should include number of bugs found in
each module, sufficiency or insufficiency of the test data, and
those tools that have been used.

Topic 3 (Program Statistics) refers to the following list of the
"suggested" items. Items 1,3,4 are required (can be easily done by "wc",
"size", "grep", "ls -ls", ...). We would be appreciated if you can
come up with others, or more.

1. LINES	lines of code count (including comments)
2. STMTS	statements count ** STMTS counts executable statements such as assigements, control, I/O, and arithmetic statement functions. **
3. LN-CM	total number of lines excluding comments
4. OBJS	size of object code
5. MODS	counts programming modules: subroutines, functions, procedures, etc

```
6. STM/M         mean number of statements per module
7. CALLS         number of calls to programming modules (MODS)
8. LIBS          counts library functions
9. LCALL         number of calls to library functions (LIBS)
10.GBVAR         number of global variables
11.LCVAR         number of local variables
12.CONST         number of constants
13.BINDE         number of binary decision
                 e.g.,
                       if (a>b)
                          then ...  <=== this counts as one
                          else ...

                       if (a>b)
                          then ...  <=== this counts as one
                       ** no else part **

                       if (a>b)
                          then ...
                       else if (b>c)
                          then ...  <=== this counts as three
                       else if (c>d)
                          then ...      (hint:           /\        )
                       else ......                      /  \
                                                           /\
                                                          /  \
                                                             /\
                                                            /  \

                       ** for the decision part of "do loop",
                          "while loop", or what you have,
                          count as one **
                       ** If you have a case statement with,
                          e.g., four outcomes, then BINDE=3,
                          i.e., in order to calculate BINDE,
                          try to simulate a case statement by
                          if then ... else if ...... **

                 ** Present tree structure of 13 by extra slides if necessary.

14.COMSQ         computation sequence of those symbols in each diagram.

                 eg. F5 -> SU15 -> D9 -> M6 -> SU17 -> ...

                 ** Present 14 by extra slides if necessary.
```

Another note: The statistics you present should be summarized in tables, for each module. (Some of the above items, e.g., COMSQ, might not fit into a table. You can show them on separate slides.) Besides that, we recommend that you show them according to your levels of abstraction, e.g., if you have under AUTOLAND the following modules: filter.ba, filter.gs, filter.ra, inner, ..., display, you can have the following table summarizing your first level of abstraction:

	main	filter.ba	filter.gs		total (AUTOLAND)
item1	35	96	128		1403
item2					
item3					

```
+-----+------+-----------+-----------+-------------------+------------+
|item4|      |           |           |                   |            |
+-----+------+-----------+-----------+-------------------+------------+
|     |      |           |           |                   |            |

...

The last note: Please make two copies of your slides and give to us right
before the presentation so that we can have a clearer picture of your work.
Thanks and Good Luck.

-coordinating team
```

Dear Fellows,

September 4th is the end of our project. There will be, as promised, an all-get-together party on this date, and we would like each team to prepare an half hour presentation. Of course, half an hour will never be enough for each team, but basically, we would like each team to address the following topics:

1) your whole idea and experience of the project,
2) major difficulties encountered,
3) your testing experience, including major bugs encountered.
4) programming language experience,
5) characteristics of the programming language explored by your program,
6) program statistics,
7) subjective observation of this experiment,
8) your recommendation of how to evaluate these programs.

You can also express your viewpoints of the specification and the communication protocol used and how to improve the future exercise if you would have to do it again, and further topics that you can think of.

Before 5:00 pm of September 3rd, please submit a pretty printed final report which covers all the above topics that are related to this project. We will collect them all and make copies for all the team members. Every team will be able to know what other teams have been doing, and most of all, all of us can benefit from your insight and experience.

We will hold our party at the Archive beginning from 10:00 am and after that will be an all-get-together dinner (place not decided yet), temporary schedule as follows:

10:00 am --- 1:30 pm	1:30 --- 8:00 pm
Introductory Material (10:00 --- 11:00 am)	Prolog (1:30 --- 2:00 pm)
Pascal (11:00 --- 11:30 am)	Ada (2:00 --- 2:30 pm)
C (11:30 --- 12:00 am)	Coffee Break (2:30 --- 3:00 pm)
Lunch (12:00 --- 1:00 pm)	T (3:00 --- 3:30 pm)
Modula-2 (1:00 --- 1:30 pm)	Conclusions (4:00 --- 5:00 pm)

```
                                 |        Dinner Party         |
                                 |     (5:00 --- 8:00 pm)      |
                                 |                             |
                                 +-----------------------------+


p.s. Please ask Jackie for slides.
p.s. About the program statistics, please refer to the mail we sent
     you about the last progress report meeting for items.  Please
     prepare for all the 14 items in your final report. You don't need
     to mention everything in the presentation, though.  Following is
     an excerpt from the previous mail about the program statistics:

1. LINES         lines of code count (including comments)
2. STMTS         statements count
                 ** STMTS counts executable statements such as
                 assignments, control, I/O, and arithmetic statement
                 functions. **
3. LN-CM         total number of lines excluding comments
4. OBJS          size of object code
5. MODS          counts programming modules: subroutines, functions,
                 procedures, etc
6. STM/M         mean number of statements per module
7. CALLS         number of calls to programming modules (MODS)
8. LIBS          counts library functions
9. LCALL         number of calls to library functions (LIBS)
10.GBVAR         number of global variables
11.LCVAR         number of local variables
12.CONST         number of constants
13.BINDE         number of binary decision
                 e.g.,
                        if (a>b)
                           then ...  <=== this counts as one
                           else ...

                        if (a>b)
                           then ...  <=== this counts as one
                        ** no else part **

                        if (a>b)
                           then ...
                        else if (b>c)
                           then ...  <=== this counts as three
                        else if (c>d)
                           then ...       (hint:           /\        )
                        else ......                       /  \
                                                            /\
                                                           /  \
                                                             /\
                                                            /  \

                        ** for the decision part of "do loop",
                           "while loop", or what you have,
                           count as one **
                        ** If you have a case statement with,
                           e.g., four outcomes, then BINDE=3,
                           i.e., in order to calculate BINDE,
                           try to simulate a case statement by
                           if then ... else if ...... **


14.COMSQ         computation sequence of those symbols in each diagram.

                 eg. F5 -> SU15 -> D9 -> M6 -> SU17 -> ...
```

Another note: The statistics should be summarized in tables
for each module. (Some of the above items, e.g., COMSQ, might not
fit into a table.) We recommend that you show them according to your
levels of abstraction, e.g., if you have under AUTOLAND the following
modules: filter.ba, filter.gs, filter.ra, inner, ..., display, you can
have the following table summarizing your first level of abstraction:

	main	filter.ba	filter.gs		total (AUTOLAND)
item1	35	96	128		1403
item2					
item3					
item4					

...

-coordinating team

F-T Software Project Post-Experiment Questionnaire (*Part 1*)

September, 1987

(1) How hard was the program-writing effort? 1(simple) -- 5(very difficult)

(2) How many hours on the average did you spend each working day for this project?

(3) How do you think you and your teammate compared as far as skill level goes? (Only consider skills needed for all phases of your work this summer.) Use the scale : 1(almost equal skill levels) -- 5(extremely different skill levels)

(4) Estimate the percentages of the total work you and your teammate did.

You:
Teammate:
(total = 100%)

(5) Rate your record-keeping on a scale of 1(extremely inaccurate) -- 5(extremely accurate).

(6) Did you have or notice any application-dependent conversations across team boundaries? If so, about how many times, and concerning what?

F-T Software Project Post-Experiment Questionnaire (*Part* 2)

Name: ______________________ Team: ___________ Date: _____________

(1) How difficult was it to use your language for the "autoland" application? 1(simple) -- 5(very difficult)

(2) For each phase of this experiment please comment on how well the amount of work was matched to the time allowed for completion of that phase.

phase	alotted time	time spent	your recommendation
design	4 weeks		
coding	3 weeks		
unit test	1 week		
integration test	2 weeks		
acceptance test	2 weeks		

(3) Did you use any references in the course of the summer? If so, please specify titles and type of information referenced.

(4) What, if anything, would you do differently if you were designing a similar experiment in the future?

(5) How much interest do you presently have in research of this type? Scale: 1 = very little, to 5 = very high.

Fault-tolerant software: 1 2 3 4 5

Fault-tolerant systems: 1 2 3 4 5

APPENDIX III. Summary of All Faults Found

UCLA / HONEYWELL JOINT PROJECT

FCS DESIGN UTILIZING N-VERSION PROCESSING

AUDIT OF ALL UNCOVERED FAULTS

June–15–1987

to

February–19–1988

With the help of the change documentation submitted by each team at the end of each test phase (Coding and Unit Test, Integration Test, and Acceptance Test), the following summary of faults was produced. Changes that were not made due to a fault are ignored.

Starting with a particular id, each fault is described, the location and the type of the fault is given (e.g. typo, omission, unnecessary, incorrect algorithm, specification misinterpretation/ambiguity etc.), and the method of detection is mentioned, followed by special remarks, if any. Reference to the corresponding Design Walkthrough Report(s) (DWR) and/or Code Update Report(s) (CUR) is given. Finally, a classification of the fault into requirements or structural fault (unintended function) is attempted.

III.1 ADA Version

III.1.1 Faults detected during Coding and Unit Test

a1. Some unnecessary statements in the initialization of the Inner Loop (CUR #1)

Type: unnecessary. Detected by: reading the code

Classification: structural fault

a2. Wrong constant (wrong place of the decimal point) for the upper limit in the declaration of an integrator in the specification part of the Complementary Filter Module (CUR #3)

Type: typo / *spec misinterpretation* / due to poor readability. Detected by: test data data.2, data.13

Classification: requirements fault

The other changes were due to floating point representation problems (CUR #2), compiler peculiarities (CUR #4), and a specification update (CUR #5).

III.1.2 Faults detected during Integration Test

a3. There were some unnecessary local variables in the main program which interfered with global (imported) variables (CUR #6).

Type: incorrect algorithm. Detected by: test data data.1

Classification: requirements fault

a4. Two initialization statements were omitted when initializing the Glide Slope Complementary Filter (CUR #8).

Type: incorrect algorithm (due to lack of communication with team mate). Detected by: test data data.2

Classification: requirements fault

The other changes were due to a specification update (CUR #7) and to adjust to the C interface (CUR #9).

III.1.3 Faults detected during Acceptance Test

a5. In the outer loop of the Flare Control Law, some test points which are not used in Flare Mode were not reset to the specified default value (CUR #10).

Type: omission. Detected by: test data data.2

Classification: requirements fault

a6. A new version of the Mode Logic was written because a numeric error exception resulted from execution (CUR #11).

Type: incorrect algorithm. Detected by: test data data.2

Classification: requirements fault

III.1.4 Faults detected after Acceptance Test

No faults were detected.

III.2 C Version

III.2.1 Faults detected during Coding and Unit Test

c1. A wrong constant was used in the Mode Logic (CUR #4).

Type: specification misinterpretation (due to poor readability of Fig. 4.2). Detected by: reading the code

Classification: requirements fault

c2. OR-gates in Mode Logic were implemented wrong (CUR #5).

Type: incorrect algorithm. Detected by: team's own test data

Classification: requirements fault

c3. In the Inner Loop, another state variable for the determination of the condition for switch SW2 was added (CUR #9).

Type: incorrect algorithm. Detected by: test data data.18.

Classification: requirements fault

Note: As it was detected during operational testing, this change resulted in an incorrect algorithm.

c4. Part of the Outer Loop of the Flare Control Law was not implemented correctly ($\frac{200}{VA}$ was used instead of $\frac{VA}{200}$) (CUR #10).

Type: incorrect algorithm. Detected by: reading the code and the coordinators' test data.

Classification: requirements fault

The other changes were made in order to confirm to the C interface (CUR #1), to obtain a clear structure of the program (CUR #2, 3), or were due to a specification update (CUR #6, 7, 8).

III.2.2 Faults detected during Integration Test

c5. In the Outer Loop of the Glide Slope Control Law, a comparison statement of "time >= 0.5" had to be changed to "time > 0.45", due to the lack of infinite precision in real number representation (CUR #11).

Type: due to peculiarities of real number representation. Detected by: test data data.1

Classification: structural fault of the C compiler

c6. In the Inner Loop, initialization of the integrator I1 was not done upon the transition from Altitude Hold to Glide Slope Track Mode (CUR #12).

Type: incorrect algorithm (due to specification ambiguity). Detected by: test data data.2

Classification: requirements fault

c7. In the Inner Loop, the state of the rate limiters LR1 and LR2 was not saved correctly (CUR #13, 14).

Type: incorrect algorithm (count as only one fault here) Detected by: test data data.2

Classification: requirements fault

c8. In the main program, the test points of the Outer Loops were not properly reset to the specified default value (CUR #15).

Type: omission. Detected by: test data data.1

Classification: requirements fault

c9. In the Mode Logic, the values FPEC1 and FPDC1 were not reset to zero when not in Altitude Hold mode (CUR #16).

Type: spec misinterpretation or ambiguity. Detected by: test data data.1

Classification: requirements fault

III.2.3 Faults detected during Acceptance Test

c10. Some functions used in the Outer Loop of the Flare Control Law were not declared to return double precision values (CUR #17).

Type: omission. Detected by: test data data.2

Classification: requirements fault

c11. The constant K3 of the Glide Slope Complementary Filter was computed incorrectly while in Altitude Hold mode. It was assumed that the factor $\frac{200}{\text{RAGSF}}$ is always limited to $\frac{1}{5.5}$ minimum, but this applies during Glide Slope Capture and Track modes only

(CUR #18).

Type: spec misinterpretation. Detected by: test data data.1

Classification: requirements fault

c12. In the Mode Logic, a wrong variable was used in an expression (CUR #19).

Type: typo. Detected by: test data data.8

Classification: requirements fault

c13. In the main program, all the parameters of the routine "VOTESTATES" were forgotten (CUR #20).

Type: omission. Detected by: test data data.2

Classification: requirements fault

III.2.4 Faults detected after Acceptance Test

c14. In the Inner Loop, initialization of integrator I1 should be in the beginning of the module (CUR #21).

Type: incorrect algorithm. Detected by: flight simulation test data.

Classification: requirements fault

c15. Introduction of an underground variable "rr_old" which made the internal states inconsistent (CUR #22).

Type: specification ambiguity. Detected by: flight simulation test data.

Classification: structure fault (which happened to be detected by requirements test)

c16. LR1 and LR2 of the Inner Loop should be initialized only once upon entering AHD mode instead of every mode change (CUR #23, #24 and #25).

Type: incorrect algorithm. Detected by: flight simulation test data.

Classification: requirements fault

c17. Using +/- 99999.0 instead of +/- ∞ in the Inner Loop and the Command Monitor.

Type: incorrect algorithm. Detected by: code inspection.

Classification: structure fault

c18. Output of Mode Logic was used in some further computations before it was voted upon.

Type: incorrect algorithm. Detected by: code inspection.

Classification: structure fault

III.3 MODULA-2 Version

III.3.1 Faults detected during Coding and Unit Test

m1. "2*i" as index to an array did not work; "i+i" had to be used instead (CUR #3).

Type: compiler fault Detected by: team's own test data

Classification: structural fault of the compiler

m2. In the Barometric Altitude Complementary Filter, the wrong constant (wrong position of the decimal point) was used for the upper limit of the output of integrator I3 (CUR #4).

Type: typo / *spec misinterpretation* / due to poor readability. Detected by: test data data.2, data.13

Classification: requirements fault

Note: same as ADA team!

m3. A wrong computation sequence was used in the Glide Slope Complementary Filter: The constants K0, K2, and K3 were computed without first computing the value of the variable RAGSF (DWR #18; CUR #5).

Type: incorrect algorithm. Detected by: test data data.1

Classification: requirements fault

m4. The Signal Display Algorithm (procedure todigit) could not handle a boundary case (DWR #21; CUR #9).

Type: incorrect algorithm, partly due to language peculiarities (floating point operations for LONGREAL variables). Detected by: test data sig_words

Classification: requirements fault

The other changes were due to specification updates (DWR #17; CUR #2, 6) and to compiler peculiarities (too many LONGREAL parameters in a procedure call – DWR #19, 20; CUR #7, 8 – and too complicated an expression in a RETURN statement – CUR #1).

III.3.2 Faults detected during Integration Test

No faults were detected.

Changes were made because of an specification update (DWR #22; CUR #10), and the interface to "VOTESTATES" had to be changed because the compiler could not handle so many parameters (see above) (CUR #11, 12).

III.3.3 Faults detected during Acceptance Test

No faults were detected.

III.3.4 Faults detected after Acceptance Test

m5. Output of Mode Logic was used in some further computations before it was voted upon.

Type: incorrect algorithm. Detected by: code inspection.

Classification: structure fault

III.4 PASCAL Version

III.4.1 Faults detected during Coding and Unit Test

p1. The Signal Display did not round correctly (DWR #32, CUR #1).

Type: incorrect algorithm. Detected by: reading the code

Classification: requirements fault

p2. A wrong constant was used in the Fault-word Display (CUR #2).

Type: incorrect algorithm. Detected by: test data fault_words

Classification: requirements fault

p3. In the Signal Display, zero was displayed as "+.00000" instead of as ".00000" (CUR #3).

Type: spec misinterpretation / incorrect algorithm. Detected by: test data sig_words

Classification: requirements fault

p4. The constants of the Glide Slope Complementary Filter were not computed correctly in all system modes (CUR #4).

Type: spec misinterpretation. Detected by: test data data.18

Classification: requirements fault

p5. The output of integrator I8 in the Glide Slope Complementary Filter was not initialized correctly (CUR #5, 6).

Type: spec misinterpretation. Detected by: test data data.11

Classification: requirements fault

p6. The initialization of the variable HREF in the Outer Loop of the Altitude Hold Control Law was incorrect (CUR #7, 8, 10).

Type: incorrect algorithm. Detected by: reading the code

Classification: requirements fault

p7. The initialization of Inner Loop variables was incorrect (CUR #9).

Type: incorrect algorithm. Detected by: reading the code

Classification: requirements fault

p8. Logic Fault in the evaluation of the condition for switch SW2 in the Inner Loop (DWR #36; CUR #19).

Type: incorrect algorithm. Detected by: test data data.20

Classification: requirements fault

p9. In the Outer Loop of the Flare Control Law, the function F4 was evaluated incorrectly (wrong setting of parenthesis) (DWR #34; CUR #13).

Type: typo / spec misinterpretation. Detected by: test data data.11

Classification: requirements fault

p10. Redundant computation of test point 7 in the Inner Loop (DWR #37; CUR #20).

Type: unnecessary. Detected by: test data data.20

Classification: requirements fault

Other changes were made in response to specification updates in the Mode Logic (CUR #11, 21), to aid in debugging (CUR #14 - 16, 18), and to remove a compiler warning message (CUR #12, 17).

III.4.2 Faults detected during Integration Test

p11. The variable RAGSF in the Glide Slope Complementary Filter was not initialized correctly (CUR #26, 27).

Type: incorrect algorithm. Detected by: test data data.1

Classification: requirements fault

p12. The integration of the integrator I8 in the Glide Slope Complementary Filter was not performed during the very first computation frame (CUR #26).

Type: incorrect algorithm. Detected by: test data data.1

Classification: requirements fault

Other changes were made to be compatible with the C interface and to remove debugging facilities introduced before Unit testing.

III.4.3 Faults detected during Acceptance Test

No faults were detected; some changes were made for the sake of compatibility with the C interface.

III.4.4 Faults detected after Acceptance Test

No faults were detected.

III.5 PROLOG Version

III.5.1 Faults detected during Coding and Unit Test

pg1. The names of some Mode Logic variables (Fig. 4.2) conflicted with variable names in the Flare Control Law (DWR #24).

Type: incorrect algorithm. Detected during coding

Classification: requirements fault

pg2. In the Inner Loop, the variable names THETA_C and THCI were confused (DWR #29; CUR #15).

Type: incorrect algorithm. Detected during coding

Classification: requirements fault

pg3. Integrator I8 in the Glide Slope Complementary Filter was not initialized correctly (DWR #30; CUR #7).

Type: incorrect algorithm / spec misinterpretation. Detected by: test data filter.gs/data.9

Classification: requirements fault

pg4. A comma was misplaced in the Barometric Altitude Complementary Filter (CUR #2).

Type: typo. Detected during coding

Classification: requirements fault

pg5. Initialization of the Inner Loop was forgotten in the main program (DWR #32).

Type: omission. Detected during coding

Classification: requirements fault

pg6. The order of parameters in the definition of the integrator function was inconsistent with its use (CUR #3).

Type: typo / incorrect algorithm. Detected by: test data filter.ba/data.1

Classification: requirements fault

pg7. The order of parameters in the definition of the linear-filter function was inconsistent with its use (CUR #5).

Type: typo / incorrect algorithm. Detected by: test data filter.ba/data.1

Classification: requirements fault

pg8. A wrong global variable was used in the Glide Slope Complementary Filter (CUR # 8).

Type: typo. Detected by: test data filter.gs/data.9

Classification: requirements fault

pg9. The constant K3 in the Glide Slope Complementary Filter was computed incorrectly (wrong magnitude limitation when not in Glide Slope Capture or Track modes) (CUR #9).

Type: incorrect algorithm. Detected by: test data filter.gs/data.18

Classification: requirements fault

pg10. In the main program, the initialization of the variable "first_ahd" was forgotten (CUR #11).

Type: omission. Detected by: team's own test data

Classification: requirements fault

pg11. Missing declaration of THETA_C as a global variable (CUR #14).

Type: omission. Detected by: reading the code

Classification: structural fault

pg12. In the Inner Loop, a wrong algorithm was used (CUR #16).

Type: typo / incorrect algorithm. Detected by: test data inner/data.6

Classification: requirements fault

pg13. In the Inner Loop, SU4 and LM1 are computed twice, but no new variable names were used for the second computation (CUR #17).

Type: omission, due to language peculiarities (one cannot reassign a value to a bound variable). Detected by: test data inner/data.6

Classification: structural fault due to the language

pg14. Wrong constants were used in the Outer Loop of the Flare Control Law (CUR #49).

Type: typo (readability problems). Detected by: coordinators' test data

Classification: requirements fault

pg15. The value of SIGIN was not input to the Display procedure, to determine which signal should be displayed (CUR #54).

Type: omission. Detected by: coordinators' test data

Classification: requirements fault

The other changes were due to specification updates (DWR #23; CUR #53), efforts to increase the efficiency (DWR #25, 27, 28; CUR #1, 4, 6, 51), were made to add debugging facilities (CUR #10, 12, 13, 18), and to add an "absolute value" routine to the Prolog interpreter (CUR #58).

CUR #55 is questionable: it seems that this update was actually not done (comparison with code).

III.5.2 Faults detected during Integration Test

pg16. Wrong initialization of Filter F10 in the Glide Slope Complementary Filter (DWR #34, 35; CUR #19, 20, 33, 34, 35).

Type: typo / incorrect algorithm. Detected by: test data data.1

Classification: requirements fault (count as only one fault)

pg17. A wrong function name was used in the definition of "do_ahd" (main program) (CUR #23).

Type: typo

Classification: requirements fault

pg18. Incorrect organization of the command monitors (global variables) (CUR #24).

Type: incorrect algorithm. Detected by: reading the code

Classification: requirements fault

pg19. Syntax error in a comment in the main program (CUR #28).

Type: typo. Detected by: Prolog interpreter

Classification: requirements fault

pg20. Inconsistencies between the global database and the Barometric Altitude Complementary Filter, the Radio Altitude Complementary Filter, the Inner Loop, and the routine "VOTESTATES". Also, a "retract" statement was forgotten in the initialization of the Barometric Altitude Complementary Filter (CUR #29, 30, 37, 44).

Type: omission. Detected by: reading the code

Classification: structural fault

pg21. A state variable of the Glide Slope Complementary Filter was not entered into the global database (CUR #31, 32).

Type: omission. Detected by: reading the code

Classification: structural fault

pg22. A wrong variable name was used in the "VOTEFILTER2" function in the interface (CUR #39).

Type: typo.

Classification: requirements fault

The other changes were due to adding/deleting debugging facilities (CUR #21, 22, 25, 26, 38, 42), to efforts to increase efficiency and/or clarity (CUR # 27, 43, 56, 58), to numerical difficulties (CUR #36), and to the installation of the Display procedure in the Prolog interpreter (CUR #40, 41). CUR #33, 34 are duplications of CUR #19, 20, respectively.

III.5.3 Faults detected during Acceptance Test

pg23. Fault in "do_gscf" function: VOTEFILTER2 must be called for every frame computation (DWR #33; CUR #47).

Type: incorrect algorithm. Detected by: test data data.4

Classification: requirements fault

pg24. The condition for initializing the Outer Loop of the Flare Control Law was wrong (CUR #45).

Type: incorrect algorithm. Detected by: test data data.2

Classification: requirements fault

pg25. The function name "set_gsp" was misspelled in the main program (CUR #46).

Type: typo. Detected by: reading the code

Classification: requirements fault

pg26. In the definition of "callVotestates" for the C interface, new variable names had to be introduced for the return values (CUR #60).

Type: omission, due to language peculiarities (one cannot reassign a value to a bound variable). Detected by: test data data.8

Classification: structural fault due to the language

III.5.4 Faults detected after Acceptance Test

pg27. A state variable of the Inner Loop was updated twice during one computation (CUR #61).

Type: specification ambiguity. Detected by: flight simulation test data.

Classification: requirements fault

pg28. Rounding errors in the Display module (CUR #62).

Type: incorrect algorithm. Detected by: code inspection.

Classification: requirements fault

pg29. Output of Mode Logic was used in some further computations before it was voted upon.

Type: incorrect algorithm. Detected by: code inspection.

Classification: structure fault

III.6 T Version

III.6.1 Faults detected during Coding and Unit Test

t1. No magnitude limitation was performed on the output of integrator I2 in the Barometric Altitude Complementary Filter (CUR #1).

Type: omission. Detected by: test data data.11

Classification: requirements fault

t2. The output of integrator I2 in the Barometric Altitude Complementary Filter was initialized by zero, instead of by HR (DWR #10; CUR #2).

Type: omission. Detected by: test data data.13

Classification: requirements fault

t3. The output of integrator I8 in the Glide Slope Deviation Complementary Filter was initialized by zero, but it should be initialized by the current value of the output variable GSEL (DWR #8, 9; CUR #3).

Type: The team claimed the fault was due to a spec ambiguity, but we feel that it could be a spec misinterpretation or a oversight.

Classification: requirements fault

t4. Several changes in the Mode Logic (CUR #5).

Type: wrong algorithm, typos, omissions. Detected by: test data data.1, data.6

Classification: requirements fault

t5. In all the Complementary Filters, the variables of the current computation state were initialized, instead of the variables of the previous computation state (CUR #6).

Type: incorrect algorithm. Detected by: test data data.1

Classification: requirements fault

t6. Incorrect computation of the output of the Command Monitor, "smaller" and "smaller or equal" was confused (CUR #7).

Type: incorrect algorithm. Detected by: reading the code

Classification: requirements fault

t7. Incorrect rounding in the Signal Display function (CUR #8).

Type: incorrect algorithm. Detected by: test data

Classification: requirements fault

Other changes were due to specification updates concerning the Mode Logic (DWR #11; CUR #4).

III.6.2 Faults detected during Integration Test

t8. In the Inner Loop, the rate limiters and switch SW2 were implemented incorrectly (CUR #9).

Type: incorrect algorithm. Detected by: test data data.1

Classification: requirements fault

t9. The global variable definitions in the main program were incomplete (CUR #11). Type: omission. Detected by: reading the code

Classification: requirements fault

t10. The overall organization of the initialization (main program, Mode Logic) had to be fixed, so that all functions would get properly initialized. Originally, a flag indicating a mode change was erroneously reset by the Mode Logic (CUR #13, 14).

Type: incorrect algorithm. Detected by: reading the code

Classification: requirements fault

t11. The arguments standing for global variables were deleted from the functions VOTEMODE and VOTEFILTER1. Only test point variables are passed as arguments

(CUR #15, 16).

Type: unnecessary code. Detected by: reading the code

Classification: structural fault

The other changes were made to add comments, and to correct the interface between C and T (CUR #10). To make the program more clear, arguments were explicitly added to the VOTEFILTER2 function (CUR #12) – this could also indicate a fault, however.

III.6.3 Faults detected during Acceptance Test

t12. In the VOTESTATES function, some variables were omitted from the format statements, and some variables were not assigned their value as returned by VOTESTATES (CUR #19, 20).

Type: omission. Detected by: reading the code, test data data.7

Classification: requirements fault

t13. In the VOTESTATES function, some wrong variable names were used when assigning the return value (CUR #21).

Type: typo, incorrect algorithm, due to inconsistent naming conventions among team mates. Detected by: test data data.7

Classification: requirements fault

t14. The transition from Glide Slope Capture to Glide Slope Track mode was considered a mode change requiring reinitialization of the Inner Loop (CUR #22).

Type: incorrect algorithm. Detected by: test data data.3

Classification: requirements fault

t15. In the interface to the C routines, the parameters of VOTESTATES were called by value instead of by reference (CUR #23).

Type: omission, incorrect algorithm. Detected by: test data data.7

Classification: requirements fault

t16. The current value of RAE was used in function F3 in the Flare Outer Loop, instead of the value of RAE at Flare initiate (CUR #24).

Type: incorrect algorithm, possibly caused by typo or omission. Detected by: test data data.2

Classification: requirements fault

t17. A parameter was omitted in the definition of VOTEOUTER (CUR #25).

Type: omission. Detected by: test data data.2

Classification: requirements fault

t18. A global variable was initialized twice in the Main Program (CUR #27).

Type: unnecessary statement. Detected by: code reading

Classification: structural fault (but did not cause any error)

t19. The routine LANEINPUT, the Command Monitors, and the Display are not called when the mode is Touchdown.

Type: incorrect algorithm. Detected by: code reading

Classification: requirements fault

t20. The Command Monitors are re-initialized at every mode change.

Type: incorrect algorithm. Detected by: code reading

Classification: requirements fault

t21. The rate limiters LR1 and LR2 in the Inner Loop are re-initialized at every mode change.

Type: incorrect algorithm. Detected by: code reading

Classification: requirements fault

III.6.4 Faults detected after Acceptance Test

t22. A state variable of the Inner Loop was updated twice during one computation (CUR #28).

Type: specification ambiguity. Detected by: flight simulation test data.

Classification: requirements fault

Note: same as PROLOG team!

t23. Output of Mode Logic was used in some further computations before it was voted upon.

Type: incorrect algorithm. Detected by: code inspection.

Classification: structure fault

The other changes were made in order to increase storage efficiency (CUR #17), to increase the size of the buffer in which arguments are passed from the C routines to the T functions, just to be on the safe side (CUR #18), and to make the structure of the Mode Logic more clear (CUR #26).