Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

# DIVERSITY IN N-VERSION SOFTWARE: AN ANALYSIS OF SIX PROGRAMS

Werner Schuetz

October 1988
CSD-880078

UNIVERSITY OF CALIFORNIA

Los Angeles

# Diversity in N-Version Software:
# An Analysis of Six Programs

A thesis submitted in partial satisfaction of the requirements for the degree Master of Science in Computer Science

by

Werner Schuetz

1987

The thesis of Werner Schuetz is approved.

[illegible]

Milos D. Ercegovac

[illegible]

David A. Rennels

[illegible]

Algirdas Avižienis, Committee Chair

University of California, Los Angeles

1987

TABLE OF CONTENTS

## LIST OF FIGURES

ACKNOWLEDGEMENTS

The research described in this thesis was supported jointly by the Sperry Commercial Flight Systems Division of Honeywell, Inc., Phoenix, Arizona, and by the Microelectronics Innovation and Computer Research Opportunities (MICRO) program of the State of California.

I would like to thank my committee, Dr. Avižienis, Dr. Ercegovac, and Dr. Rennels, for serving on my committee. They were all extremely helpful under tight time pressure.

A special thanks to Dr. Avižienis, director of the UCLA Dependable Computing and Fault-Tolerant Systems Laboratory, who enabled me to come to UCLA at all.

Thanks go also to the members of the fault tolerance research group, Michael R. Lyu, Mark K. Joseph, Johnny J. Chen, and Chi S. Wu, for their support and stimulating technical discussions.

ABSTRACT OF THE THESIS

Diversity in N-Version Software:

An Analysis of Six Programs

by

Werner Schuetz

Master of Science in Computer Science

University of California, Los Angeles, 1987

Professor Algirdas Avižienis, Chair

N-Version software is an approach to software fault tolerance that uses several redundant, but independently developed, versions of the software to mask errors in individual versions. This thesis examines and analyzes the structural differences found in six redundant programs that were generated in the course of a N-Version software development experiment during the summer of 1987 at UCLA. The application programmed in this experiment was a subset of the functions of a Flight Control Computer for a commercial airliner. The thesis offers an empirical evaluation of the effectiveness of the N-Version software methodology in producing redundant programs with different structures.

# CHAPTER 1

# INTRODUCTION

## 1.1 Purpose and Goals of this Research

The multi-version software approach to fault-tolerant software systems involves the development of functionally redundant, yet independently developed software components. These components are executed concurrently under a supervisory system that uses a decision algorithm based on consensus to determine the final result [Aviz84, Aviz85a].

In all discussions and publications up to date, "Design Diversity" is in effect referred to as a *method*: If two or more functionally equivalent programs are written by independent programmers or programming teams, perhaps in addition using different variations of the specification, different software development environments, different programming languages, or different software tools – as the case might be – then the programs resulting from this effort are called "diverse". This method has been criticized for not being as effective as desired, even for not being effective enough to be practical at all [Knig86].

As far as can be ascertained, no one has ever tried to analyze and compare different programs produced by this approach to find out which and how many *differences* can be identified and generated by the Design Diversity methodology. This is exactly what is done in this study. Here, "diversity" refers to the structural differences found when analyzing and comparing different programs produced by the Design Diversity methodology. "Structural differences" means differences with respect to aspects that can be determined by examination and inspection of the (static) code of the programs, for example the modularization, the algorithms, and the definition and use of data structures. There are also other indicators of diversity, such as differences in execution time, memory requirements, or the type of errors found during testing. These are not examined in this study because they appear to be only indirect indications of diversity that are dependent on the approach used by the individual programmer (and thus on the structural aspects, e.g. the algorithms) and on the software development tools (especially the compilers) used.

Two of UCLA's long-range research goals [Aviz87a] are: 1) to develop methods to qualitatively and quantitatively assess the amount of diversity present in a set of program versions, in order to be able to determine the effects of different variations of the Design Diversity Paradigm under development; and 2) to explore the benefits and effects of "random" versus "enforced" diversity. "Enforced" diversity is obtained by imposing requirements to use different algorithms, data structures, specifications, software development tools, programming languages, etc. "Random" diversity, on the other hand, relies on the isolation between programming teams and on their different backgrounds and approaches to the problem.

The immediate goal of this study is to gain empirical evidence, and thus some confidence, that the application of the Design Diversity methodology results in programs that are different in some aspects. It is also a first step towards attacking the research goals mentioned above.

## 1.2 The Methodology Used in this Study

The information and the data used for this study were: 1) the specification of the application problem from the UCLA/Honeywell Experiment (see below); 2) the source code of the six programs that have been developed in the course of this experiment; and 3) (to a lesser degree) the Design Documents produced by the programming teams after the Design phase of the experiment.

The specification was used to determine the *potential for diversity* (see Chapter 5) that is present in each part of the application problem. That determination is basically an estimation of the maximum possible amount of diversity between the six programs which is based on the characteristics of the algorithm specified for each part of the problem. In addition, this estimate was checked against the amount of diversity actually found, but this only confirmed the original estimate.

To find as many differences between the six programs as possible, a two-step approach was used. In the first step, the code of each program was inspected and examined in order to get a general idea of what differences were present, and in which part of the program. The result was the impression that a huge number of differences could be listed, if one was only willing to go sufficiently detailed into the source code. Therefore, the next step was to categorize and order all the differences observed into a number of classes or aspects. Finally, the source code of each program was inspected again with respect to each aspect identified before, and an attempt was made to summarize the differences found in each class and to try to abstract as much as possible from the source code.

The rest of this thesis is organized as follows: The next sections of the introduction give some background information about the UCLA/Honeywell Experiment and its application problem, respectively. The next three chapters contain a description of the differences found between the six programs. This material has been organized into a description of organizational and structural (high-level) differences, a description of diversity in implementation and computation sequence (lower-level), and a description of various other aspects which could not be categorized easily in one of the two categories above. A reader who is not interested in these very detailed descriptions might want to skip these three chapters and proceed directly to Chapter 5. There, a general summary of the results, as well as a few observations and conclusions are presented. Finally, an Appendix contains reproductions of the figures used in the specification to define most of the algorithms of the application. This material can be used to verify some of the observations and conclusions (especially those related to the material described in Chapter 3).

## 1.3 The UCLA/Honeywell Experiment

UCLA and the Sperry Commercial Flight Systems Division of Honeywell, Inc., Phoenix, AZ, are conducting a joint investigation of Flight Control System Design utilizing N-version diverse software. The project was designed to evaluate the contribution of diverse program versions to fault-tolerant software systems in a realistic avionics application. The application is a digital Flight Control System for future commercial airliners. The objectives of this project include: 1) to conduct studies and experiments related as closely as is practical to the industrial environment in terms of procedures and types of problems; 2) to develop a practical and effective set of ground rules for multi-version software development for an industrial environment; 3) to estimate the effectiveness of multi-version software in an industrial environment; and 4) to continue developing a multi-version software design paradigm. Because of these objectives, the problem chosen had to be as large as practical, and it had to be a "real-world" problem (thus the choice of the Flight Control System); furthermore specifications and software development activities employed were as similar as possible to the ones used in an industrial environment.

The project started in November 1986 with the selection of an appropriate subset of the application to be programmed and the development of a specification for this subset. During the summer of 1987, 12 graduate students (six programming teams) were employed to design, code, and document six software versions in the programming languages ADA®, C, MODULA-2, PASCAL, PROLOG, and T (a dialect of LISP). The software development was structured into a design phase (4 weeks), a coding and unit test phase (4 weeks), an integration test phase (2 weeks), and an acceptance test phase (2 weeks).

A coordinating team, consisting of three researchers at the UCLA Dependable Computing and Fault-Tolerant Systems Laboratory, supervised the six programming teams and monitored their progress. The programmers and the coordinating team communicated via a formal tool, i.e. electronic mail, to avoid the ambiguity of oral communication, to control the type and amount of information communicated, and to preserve a record for later inspection. The answer to each question or other problem was only sent to the programming team that asked that question, with the help of Sperry flight control experts, if necessary. Only answers that lead to an update or a clarification of the specification were broadcast to all programming teams.

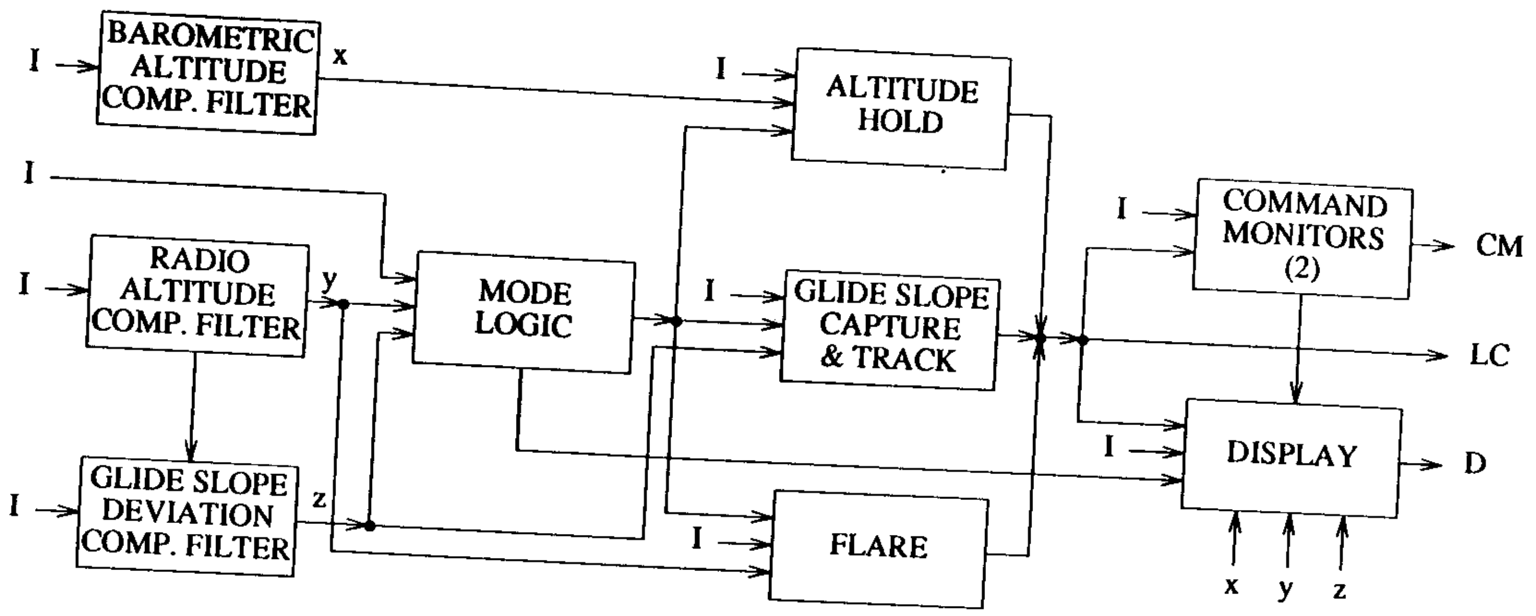
The twelve week programming effort lead to six programs of an average size of 1739 lines of code, ranging from 1378 to 2253. More details about the objectives, organization, execution, and evaluation results of this project can be found in two technical reports [Aviz87a, Aviz87b].

ADA® is a registered trademark of the US Government AJPO.

## 1.4 The Application Problem

The programming task of the experiment was to write software for the Flight Control Computer (FCC) whose function is to control and guide the aircraft along the desired flight path. More specifically, the programmers were asked to provide the software for the automatic landing phase (i.e. the last phase) of the flight. The problem has been simplified insofar as only the vertical motion of the airplane is considered; this is called the *pitch control*. It can be assumed that the vertical motion can be controlled by a single surface of the airplane, the *elevator*. Thus, the main purpose of the software is to compute and output commands to set the position of the elevator. This final output is called *elevator command* or *lane command*, since each program will be but one of three computation lanes, each performing the same computation simultaneously. The major system functions of the pitch control and its data flow are shown in Figure 1. For more details, the reader should refer to the Specification Document [Lyu87].

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 (AHD), Glide Slope Capture (GSCD), Glide Slope Track (GSTD), Flare (FD), and Touchdown (TD).


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

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

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. 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 miscompare state (according to the algorithm used) 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.

It was assumed that conceptually three of the programs would be executed in a real FCC. Therefore, fault tolerance support mechanisms had to be specified in addition to the application itself. These are:

1) ***Test Points*** were used to output some selected intermediate values of each major subfunction. Originally, Honeywell specified them for their own purposes, especially error detection by version comparison. Most of them were kept in the specification for the experiment, although some of them were replaced by Cross-Check Points. For simplicity, they were output by using the

subsequent cross-check point routine; however, the Test Points were not cross-checked. There are between 0 and 13 Test Points per major system function.

2) *Cross-Check Points* [Aviz85b] are used to cross-check the results of the major system functions of the FCC (e.g. Complementary Filters, Mode Logic, Outer Loop, Inner Loop, etc.) with the results of the other lanes before they are used in any further computation. They have to be executed in a certain predetermined order, but great care was taken not to overly restrict the possible choices of computation sequence. Seven cross-check points were specified: VOTEFILTER1, VOTEFILTER2, VOTEMODE, VOTEOUTER, VOTEINNER, VOTEMONITOR, and VOTEDISPLAY.

3) *Recovery Points* [Tso87] are used to recover a failed program by supplying it with a set of new, correct internal state variables. One recovery point was specified: VOTESTATES.

In addition, two input routines were specified that were supplied by the coordinating team. SENSORINPUT simulates reading all the airplane sensor input data, LANEINPUT inputs the lane commands of the other two versions so that they can be used in the Command Monitor.

# CHAPTER 2

# THE ORGANIZATION AND OVERALL STRUCTURE OF THE PROGRAMS

In this chapter the modularization of the six programs, the overall organization, and the interface between the main program and its submodules will be compared.

## 2.1 Introduction and General Observations

The first fact to observe is that all teams identified practically the same modules. This was partly due to the nature of the problem and partly due to the organization of the specification which was written as a functional partitioning of the problem.

In the languages that explicitly support modularization constructs, the partitioning was made as shown in Table 1:

| Packages in ADA | Modules in MODULA-2 |
|---|---|
| Main Program | AutolandModule (Main Program) |
| Complementary Filters | Radio Alt. Comp. Filter |
| | Barometric Alt. and Glide Slope Dev. Comp. Filters |
| Mode Logic | Mode Logic |
| Altitude Hold Control Law | Altitude Hold Control Law |
| Glide Slope Capture and Track Control Law | Glide Slope Capture and Track Control Law |
| Flare Control Law | Flare Control Law |
| Inner Loop | Inner Loop |
| Command Monitor AB | Command Monitors |
| Command Monitor AC | |
| Displays | Displays |
| Common_Pool | Operator_Module |

Table 1: Modularization in the ADA and MODULA-2 Programs

The last item in each of the above lists defines functions typically needed in more than one of the other modules, such as Integrators, Magnitude Limiters, Rate Limiters, and Linear Filters. No further comparisons are made here because only two languages had the opportunity to use explicit modularization constructs.

To have a common reference point for the following discussion we will consider the programs to consist of the following global functions (or modules): Main Program, Complementary Filters, Mode Logic, Control Laws (each consisting of Outer Loop and Inner Loop), Command Monitors, and Displays.

Some specific observations are needed here. The C program has a main program that consists only of a call to a C-function "autoland". The PROLOG program uses a PROLOG-function "autoframe" that does all the computations for one time frame and is repeatedly called by the main program. Similarly, a T-function "fly-airplane" is executed in each time frame which in turn uses "control-plane" to do the Control Law, Command Monitor, and Display computations. In these cases, we consider the main program plus these auxiliary functions to be the "Main Program" module.

In the following discussion, we will refer to a subprogram as a "procedure" if its result(s) are returned by parameter passing or by setting of global variables, and it will be referred to as a "function" if its result is returned via a "return" statement or a similar construct. This distinction will be made regardless of whether a particular programming language actually supports the distinction between procedures and functions, or not. Those that do not support the distinction are C, PROLOG, and T. The terms "procedure" and "function" are used according to the definition above throughout the remainder of this thesis.

## 2.2 The Main Program

In general, the Main Program consists of some initialization statements and/or procedures and of a loop which performs the actual computations until the aircraft touches the ground. One iteration of the loop corresponds to the computations performed within one time frame of 0.05 seconds. The T program only implements this loop by using tail recursion of "fly-airplane"; all other programs use an appropriate loop construct of their programming language.

Since the sequence of modules is practically unanimously determined by the data dependencies of the problem, all teams developed basically the same algorithm. It is also stated rather explicitly in the specification.

The following is a reproduction of the corresponding part of the specification:

---

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.

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

3) Determine mode from mode logic, 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 if mode has changed.

7) Compute the outer loop from altitude hold, capture/track, or flare control laws. 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, and call VOTEINNER to cross check output values of inner loop.

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

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

11) Compute display outputs, 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.

---

The following variations can be observed:

The ADA and PROLOG programs call SENSORINPUT for the first time outside the main loop. The motivation for that is obviously that some input values are required to perform certain initializations. Thus this scheme enables the programs to do as much initialization as possible before entering the main loop. The other programs do more initialization inside the main loop, and generally use a boolean flag to determine if it should be performed in a given iteration of the loop or not. To make up for this, the ADA program calls SENSORINPUT again at the end of the main loop, but only if there will be another iteration of the loop (i.e. the mode is not yet Touchdown). The PROLOG program uses a similar scheme, but calls SENSORINPUT at the beginning of the main loop, after checking that the current computation frame is not the first one.

All teams moved step 5) below step 8) (see above) while making the Control Law computation conditional on the fact that Touchdown mode is not yet reached. Thus, in effect, step 5) became a "Touchdown Control Law", and this change resulted in a cleaner program structure.

The main differences in the Main Program modules are found in the level of detail implemented there. This will be discussed in the subsequent sections.

## 2.3 The Organization of the Complementary Filter Module and its Interface with the Main Program

The Complementary Filter module consists of three individual filters, the Radio Altitude Complementary Filter, the Barometric Altitude Complementary Filter, and the Glide Slope Deviation Complementary Filter.

The Radio Altitude Complementary Filter is required to be computed during all flight modes, the Barometric Altitude Complementary Filter only during Altitude Hold mode, and the Glide Slope Deviation Complementary Filter during both Altitude Hold and Glide Slope Capture and Track modes. Consequently, and because there is a data dependency between the Radio Altitude and the Glide Slope Deviation Complementary Filters (the latter uses an output value of the former), it has been decided that the vote routine VOTEFILTER1 should check the output of the Radio Altitude Complementary Filter, while VOTEFILTER2 should check both Barometric Altitude and Glide Slope Deviation Complementary Filters.

The following Table 2 shows the overall organization of the Complementary Filter module:

| | |
|---|---|
| ADA | Three individual procedures for each Complementary Filter, called directly by the Main Program. |
| C | One procedure that internally calls three individual procedures for each Complementary Filter. |
| MODULA-2 | The procedure for the Radio Alt. Comp. Filter is called directly by the Main Program. Another procedure organizes the calls to the other two Compl. Filters. |
| PASCAL | One procedure that internally calls three individual procedures for each Complementary Filter. |
| PROLOG | Three individual procedures for each Complementary Filter, called directly by the Main Program. |
| T | Three individual procedures for each Complementary Filter, called directly by the Main Program. |

Table 2: Overall Organization of the Complementary Filters

Table 3 gives the sequence in which the three Complementary Filters are computed (an entry 1 means that this procedure or function is called first, etc.):

| | ADA | C | MODULA-2 | PASCAL | PROLOG | T |
|---|---|---|---|---|---|---|
| Radio Alt. C. F. | 1 | 1 | 1 | 1 | 2 | 2 |
| VOTEFILTER1 | 2 | 2 | 2 | 2 | 3 | 3 |
| Baro. Alt. C. F. | 3 | 3 | 3 | 3 | 1 | 1 |
| G/S Dev. C. F. | 4 | 4 | 4 | 4 | 4 | 4 |
| VOTEFILTER2 | 5 | 5 | 5 | 5 | 5 | 5 |

Table 3: Computation Sequence of the Complementary Filters

Another interesting difference is found in the way the programs decide whether to call the Barometric Altitude and the Glide Slope Deviation Complementary Filters in a given time frame or not. Two approaches are found: ADA, MODULA-2, PROLOG, and T decide on a "individual filter first"- basis, i.e. they make the call to a given Complementary Filter conditional on the flight mode(s) during which that filter is required to be computed. The condition for the Glide Slope Deviation Complementary Filter is either "system is not in Flare mode" (ADA) or "system is either in Altitude Hold, Glide Slope Capture, or Glide Slope Track mode" (MODULA-2, PROLOG, T). C and PASCAL decide on a "flight mode first"-basis, i.e. for each possible flight mode, the computations to be performed are determined and the appropriate calls made. The first approach results in two consecutive IF-statements, whereas the latter results in nested IF-statements.

## 2.4 The Organization of the Mode Logic Module and its Interface with the Main Program

The Mode Logic module consists of two parts: the *logic part* which determines the current flight mode and the *flight-path part* which computes two intermediate values (FPDC1 and FPEC1) that are input to the logic part, but that are needed only while the system is in Altitude Hold mode. For layout reasons, two different figures were used in the specification which seems to have caused some teams to treat these two parts quite independently from each other.

The following Table 4 shows the overall organization of the Mode Logic Module:

| | |
|---|---|
| ADA | Procedure "Mode" is called by the Main Program. It uses two functions "FPDC1" and "FPEC1" that compute and return the intermediate values mentioned above. They are only called when the system is in Altitude Hold mode. |
| C | Procedure "mode_logic" is called by the Main Program. It uses another procedure "flight_path" that computes the two intermediate values when the system is in Altitude Hold mode, and sets these values to zero otherwise. |
| MODULA-2 | Procedure "mdlgc" is called by the Main Program. It uses two functions "fpdc1" and "fpec1" that compute the corresponding intermediate value when the system is in Altitude Hold mode, and sets these values to zero otherwise. |
| PASCAL | Procedure "COMPUTEMODELOGIC" is called by the Main Program. It uses the procedures "COMPUTEFP" and "COMPUTEMODESTATE" for the two parts of the Mode Logic module. "COMPUTEFP" computes the two intermediate values and is called only when the system is in Altitude Hold mode. (Otherwise a procedure "SETDEFAULTS" sets them to zero.) |
| PROLOG | The Main Program first calls "fp_proc", if the mode is Altitude Hold, to compute the two intermediate values. Then, "ml_proc" is called for the processing of the logic part. |
| T | The Main Program calls the procedure "mode_logic", which in turn calls "fp_error" and "fp_damping" if the mode is Altitude Hold to get the two intermediate values. Then it calls "pitch_mode_logic" to compute the final output. |

Table 4: Overall Organization of the Mode Logic

## 2.5 The Organization of the Control Laws and their Interface with the Main Program

The three control laws compute the final and most important output of the Flight Control Computer, the Lane Command, in Altitude Hold, Glide Slope Capture and Track, and Flare mode respectively. Parts of these three control laws are (almost) identical to each other; this part is referred to as the "Inner Loop". The specification gave a hint that this part could be implemented only once; all teams adhered to that suggestion. The parts of the control laws that differ from each other are called "Outer Loop".

Table 5 shows the overall organization of the Control Laws:

| | |
|---|---|
| ADA | The Main Program calls a procedure to compute the appropriate Outer Loop (depending on the flight mode), then calls a function that computes the Inner Loop and returns the Lane Command. |
| C | The Main Program calls a procedure to compute the appropriate control law (depending on the flight mode). Internally, calls to procedures to compute the Outer and the Inner Loop are made. |
| MODULA-2 | The Main Program calls a procedure to compute the appropriate control law (depending on the flight mode). Internally, calls to procedures to compute the Outer and the Inner Loop are made. |
| PASCAL | The Main Program first calls the procedure "OUTERLOOPPROC" which internally decides which one of three control law computation procedures to call (depending on the flight mode), then calls the procedure "INNERLOOPPROC" to compute the Inner Loop. |
| PROLOG | The Main Program calls the procedure "do_outer" (a misleading name!), that first calls the appropriate Outer Loop procedure (depending on the flight mode), then the Inner Loop computation procedure. |
| T | Depending on the flight mode, the Main Program calls a procedure to compute the appropriate Outer Loop, then calls the Inner Loop computation procedure. |

Table 5: Overall Control Law Organization

## 2.6 The Organization of the Command Monitors and their Interface with the Main Program

Each program was required to implement two Command Monitors to compare its Lane Command with that computed by each of the other two lanes (a configuration of three lanes per FCC was assumed). The result of each Command Monitor is a boolean variable, indicating the outcome of the comparison.

The following Table 6 shows the overall organization of the Command Monitors:

| | |
|---|---|
| ADA | The Main Program calls a function that returns the required result for each Command Monitor. This function is implemented twice (once for each Command Monitor) because different state variables are used internally. |
| C | The Main Program calls a "monitor" procedure which does all the necessary computations for both Command Monitors (duplicated code). |
| MODULA-2 | The Main Program calls a procedure for Command Monitor processing. Internally, a procedure that computes one Command Monitor is called twice. |
| PASCAL | The Main Program calls a procedure for Command Monitor processing. Internally, a procedure that computes one Command Monitor is called twice. |
| PROLOG | The Main Program calls a procedure for Command Monitor processing. Internally, a procedure that computes one Command Monitor is called twice. |
| T | The Main Program calls a "monitor" procedure which does all the necessary computations for both Command Monitors (duplicated code). |

Table 6: Overall Organization of the Command Monitor Module

Note: "Duplicated code" means that the Monitor procedure consists of two parts for the implementation of one Command Monitor each. These parts are identical except for the names of the state variables used.

## 2.7 The Organization of the Display Module and its Interface with the Main Program

The Display Module is specified to consist of three parts, the Mode Display which displays the current flight mode, the Fault Display which displays the compare/miscompare states as computed by the Command Monitors, and the Signal Display which displays the value of one of sixteen signals, selected by one of the parameters input by SENSORINPUT. The information has to be encoded in different ways to drive the hardware of a (fictitious) display panel.

Table 7 shows the overall organization of the Display Module:

| | |
|---|---|
| ADA | For each Display part, the Main Program calls a different procedure. |
| C | The Main Program calls a procedure "display" that internally calls one procedure per Display part. |
| MODULA-2 | The Main Program calls a procedure "display" that internally calls one procedure per Display part. |
| PASCAL | The Main Program calls a procedure "DISPLAYPROC" that internally calls one procedure each for the Mode Display and the Signal Display. For the Fault Display, it twice calls a procedure that displays the result of one Command Monitor. |
| PROLOG | The Main Program calls a procedure "dspl_proc" that in turn calls "Display_signals" to do the whole Display module. The latter uses "calcnum" to do part of the Signal Display. |
| T | The Main Program calls a procedure "disp" that processes all three parts of the Display Module. |

Table 7: Overall Organization of the Display Module

Note: The PROLOG implementation of the Display module is somewhat special because bit manipulations are extremely tedious in PROLOG. Therefore, "Display_signals" was written in the C language, put into the PROLOG interpreter, and called in conjunction with VOTEDISPLAY. "Dspl_proc" itself only selects the current flight mode and the correct signal value to be displayed by the Signal Display and puts them in global variables to be passed to "Display_signals".

Five teams chose the following order of computation: Mode Display, Fault Display, and Signal Display. Only the PROLOG program computes the Fault Display first, then the Mode Display, and the Signal Display last.

Another interesting variation depends on when the correct signal to be displayed by the Signal Display is selected. If this is not done in the Main Program, then all possible signals have to be communicated to the Display module. Different strategies can be observed here, too. The following Table 8 summarizes these two aspects:

| | signal selected in | signals passed by |
|---|---|---|
| ADA | Main Program | N/A |
| C | Signal Display | global variables |
| MODULA-2 | Signal Display | collected in a record and passed by parameter to "display"; used as global variables by "signaldisplay". |
| PASCAL | Display | parameter |
| PROLOG | Display | global variables |
| T | Display | parameter |

Table 8: Other Variations of the Signal Display

## 2.8 Summary

In this section, the calling hierarchy of subprograms for each of the six programs is given in Figures 2 – 7. It is felt that these will supply an overview of the differences in structure and organization of the programs. Indentation in the following figures corresponds to levels of the hierarchy of calls, *not* the hierarchy of subprogram definition or declaration. Additional explanation in C-style comments is added if necessary. The Main Program level has been omitted from the figures. Note that the number of subprograms of the PROLOG program is about twice the number of subprograms in other programs; this is due to the fact that basically for each IF-statement a new subprogram had to be written in PROLOG.

```
SENSORINPUT                          /* supplied by coordinating team */
Initialize_Radio_Altitude
Initialize_Baro_Altitude
Initialize_Glide_Slope
Initialize_Mode_Logic
Initialize_Command_Monitor_AB
Initialize_Command_Monitor_AC
Radio_Altitude                         /* Complementary Filter */
VOTEFILTER1                          /* supplied by coordinating team */
Baro_Altitude                          /* Complementary Filter */
Glide_Slope                            /* Complementary Filter */
VOTEFILTER2                          /* supplied by coordinating team */
Mode Logic
    FPDC1             /* function to compute intermediate value FPDC1 */
    FPEC1             /* function to compute intermediate value FPEC1 */
VOTEMODE                             /* supplied by coordinating team */
Initialize_Altitude_Hold_Outer
Altitude_Hold_Law_Outer
Initialize_Glide_Slope_Outer
Glide_Slope_Law_Outer
Initialize_Flare_Outer
Flare_Law_Outer
VOTEOUTER                            /* supplied by coordinating team */
Initialize_Altitude_Hold_Inner
Altitude_Hold_Law_Inner                /* function Inner was renamed here */
Initialize_Glide_Slope_Inner
Glide_Slope_Law_Inner                  /* function Inner was renamed here */
Initialize_Flare_Inner
Flare_Law_Inner                      /* function Inner was renamed here */
VOTEINNER                             /* supplied by coordinating team */
LANEINPUT                             /* supplied by coordinating team */
Command_Monitor_AB
Command_Monitor_AC
VOTEMONITOR                            /* supplied by coordinating team */
Mode Display
Fault Display
Signal Display
VOTEDISPLAY                           /* supplied by coordinating team */
VOTESTATES                            /* supplied by coordinating team */
```

Figure 2: Structure of the ADA Program

```
autoland
    clear_tp   /* sets the test point variables of the Outer Loops to zero */
    SENSORINPUT                        /* supplied by coordinating team */
    init_main      /* all initializations at the start of Alt. Hold mode */
    filter_module
        radio_filter
        VOTEFILTER1                    /* supplied by coordinating team */
        gs_filter_k      /* to compute constants of Glide Slope Filter */
        barometric_filter
        barometric_housekeeping /* reset test points of Baro. Alt. Filter */
        gs_filter
            fn_limit                   /* function to compute F9 */
        gs_housekeeping  /* reset test points of Glide Slope Comp. Filter */
        VOTEFILTER2                    /* supplied by coordinating team */
    Mode Logic
        flight_path      /* computes intermediate values FPDC1 and FPEC1 */
    init_law                   /* initialization of Inner Loop */
    ahd_law                    /* Altitude Hold Control Law */
        ahd_outer_loop
        VOTEOUTER                      /* supplied by coordinating team */
        inner_loop
        VOTEINNER                      /* supplied by coordinating team */
    gsct_law         /* Glide Slope Capture and Track Control Law */
        gsct_outer_loop
        VOTEOUTER                      /* supplied by coordinating team */
        inner_loop
        VOTEINNER                      /* supplied by coordinating team */
    flare_law                          /* Flare Control Law */
        flare_outer_loop
            f_3                /* to compute function F3 */
            f_4                /* and F4             */
        VOTEOUTER                      /* supplied by coordinating team */
        inner_loop
        VOTEINNER                      /* supplied by coordinating team */
    VOTEINNER                          /* supplied by coordinating team */
    LANEINPUT                          /* supplied by coordinating team */
    Command Monitors                   /* supplied by coordinating team */
        VOTEMONITOR                    /* supplied by coordinating team */
    Display
        Mode Display
        Fault Display
        Signal Display
            d_seven /* to convert a single digit into the required code */
        VOTEDISPLAY                    /* supplied by coordinating team */
    VOTESTATES                         /* supplied by coordinating team */
```

Figure 3: Structure of the C Program

```
SENSORINPUT                          /* supplied by coordinating team */
raf                                  /* Radio Alt. Comp. Filter */
    VOTEFILTER1                      /* supplied by coordinating team */
bagsf                /* Baro. Alt. and Glide Slope Dev. Comp. Filters */
    baf                              /* Baro. Alt. Comp. Filter */
    gsf                              /* Glide Slope Dev. Comp. Filter */
        kcmp         /* to compute constants of Glide Slope Filter */
    VOTEFILTER2                      /* supplied by coordinating team */
Mode Logic
    fpdc1                /* to compute intermediate value FPDC1 */
    fpec1                /* to compute intermediate value FPEC1 */
    pgscd      /* to compute the output GSCD (this part is done twice) */
    VOTEMODE                         /* supplied by coordinating team */
ahc                                  /* Alt. Hold Control Law */
    Outer Loop
        VOTEOUTER                    /* supplied by coordinating team */
    Inner Loop
        VOTEINNER                    /* supplied by coordinating team */
gsctc              /* Glide Slope Capture and Track Control Law */
    Outer Loop
        VOTEOUTER                    /* supplied by coordinating team */
    Inner Loop
        VOTEINNER                    /* supplied by coordinating team */
fc                                   /* Flare Control Law */
    Outer Loop
        VOTEOUTER                    /* supplied by coordinating team */
    Inner Loop
        VOTEINNER                    /* supplied by coordinating team */
VOTEINNER                            /* supplied by coordinating team */
LANEINPUT                            /* supplied by coordinating team */
Command Monitors
    pmxy                 /* to compute one Command Monitor */
    VOTEMONITOR                      /* supplied by coordinating team */
Display
    Mode Display
    Fault Display
    Signal Display
        todigit      /* breaks positive number into single digits */
    VOTEDISPLAY                      /* supplied by coordinating team */
xvotestates
    VOTESTATES                       /* supplied by coordinating team */
```

Figure 4: Structure of the MODULA-2 Program

```
Main_Initialize          /* to initialize timer, Mode Logic variables */
SENSORINPUT                         /* supplied by coordinating team */
AHD_Initialize       /* all initializations at the start of Alt. Hold mode */
Timer_Update                 /* to keep track of the elapsed real time */
Filter Module
    racf                         /* Radio Alt. Comp. Filter */
    VOTEFILTER1                        /* supplied by coordinating team */
    bacf                         /* Baro. Alt. Comp. Filter */
    Reset_Bacf         /* reset test points of Baro. Alt. Comp. Filter */
    gsdcf                     /* Glide Slope Dev. Comp. Filter */
        lin_fun                    /* function to compute F9 */
    Reset_Gsdcf        /* reset test points of Glide Slope Comp. Filter */
    VOTEFILTER2                        /* supplied by coordinating team */
Mode Logic
    Compute_FP         /* compute FPDC1 and FPEC1 when in Alt. Hold mode */
    Set_Defaults  /* set FPDC1 and FPEC1 to zero when not in Alt. Hold mode */
    Compute_Mode_State                 /* compute the Mode Logic output */
    VOTEMODE                         /* supplied by coordinating team */
Mode_Update            /* to encode result of Mode Logic in Integer values */
Outer_Flare_Init
Outer_GSCTC_Init
Outer Loop
    T_Points_Reset               /* reset test points to default values */
    AH_Outer
    GS_Outer
    Flare_Outer
    VOTEOUTER                          /* supplied by coordinating team */
Inner_Loop_Init
Inner_Loop
    VOTEINNER                         /* supplied by coordinating team */
Reset_LC                /* to set default values for Touchdown mode */
    VOTEINNER                         /* supplied by coordinating team */
LANEINPUT                             /* supplied by coordinating team */
Command Monitors
    Fault_Detect                /* to compute one Command Monitor */
    VOTEMONITOR                         /* supplied by coordinating team */
Display
    Mode Display
    Set_Fault_Word          /* displays result of one Command Monitor   */
                   /* is called twice to compute Fault Display */
    Signal Display
        convert        /* to encode a single digit in the required code */
    VOTEDISPLAY                         /* supplied by coordinating team */
VOTESTATES                            /* supplied by coordinating team */
```

Figure 5: Structure of the PASCAL Program

```
init_system
    SENSORINPUT                     /* supplied by coordinating team */
    bacf_init              /* Baro. Alt. Comp. Filter initialization */
    racf_init              /* Radio Alt. Comp. Filter initialization */
    gscf_init              /* Glide Slope Comp. Filter initialization */
    ml_init                        /* Mode Logic initialization */
    inner_init                     /* Inner Loop initialization */
    cm_init                   /* Command Monitor initialization */
autoframe                  /* all computations for one time frame */
    check_first            /* determine if it is the first frame */
        SENSORINPUT                 /* supplied by coordinating team */
    increment_time         /* to keep track of the elapsed real time */
    do_bacf        /* call Baro. Alt. Comp. Filter if mode is Alt. Hold */
        bacf_proc                  /* Baro. Alt. Comp. Filter */
    racf_proc                      /* Radio Alt. Comp. Filter */
    VOTEFILTER1                    /* supplied by coordinating team */
    do_gscf            /* call G/S Comp. Filter if mode is Alt. Hold  */
                   /* or Glide Slope Capture or Glide Slope Track */
        gscf_proc              /* Glide Slope Dev. Comp. Filter */
            init_F10           /* initialize linear filter F10 */
            determine                   /* compute constants */
                min_5_pt_5              /* limit output to 5.5 */
            fcn_F9                      /* compute function F9 */
            init_I8              /* initialize integrator I8 */
    VOTEFILTER2                    /* supplied by coordinating team */
    do_first_AHD  /* determines if this is the first Alt. Hold mode frame */
        fp_init    /* initialize the flight path part of the Mode Logic */
    do_fp                      /* determines if mode is Alt. Hold */
        fp_proc        /* compute FPDC1 and FPEC1 of Mode Logic */
    ml_proc                        /* compute Mode Logic output */
        deter_td                      /* determine Touchdown */
        deter_fd                         /* determine Flare */
        pml_t1              /* computes some intermediate result */
        pml_t3              /* computes some intermediate result */
        deter_gscd            /* determine Glide Slope Capture */
        deter_gstd             /* determine Glide Slope Track */
        update_gscd              /* reset GSCD if GSTD changes */
        deter_ahd                     /* determine Alt. Hold */
    VOTEMODE                       /* supplied by coordinating team */
    do_modv              /* halts the program if Model Valid is false */
    do_into_fd             /* determine transition into Flare mode */
        fd_init                    /* initialize Flare mode */
    do_gs_bool           /* determine transition to Glide Slope mode */
    do_into_gs               /* if transition to Glide Slope mode */
        gs_init              /* initialize Glide Slope mode      */
```

Figure 6: Structure of the PROLOG Program

```
(autoframe)    /* this is only repeated to give a reference for indentation */
    Control Laws
        do_fd                                   /* if in Flare mode */
            fd_proc                             /* Flare Outer Loop */
                limit_LM9          /* computes output of limiter LM9 */
                do_sw4_bool        /* determines state of switch SW4 */
                abs         /* procedure to compute absolute value */
                do_init_F6         /* initialize linear filter F6 */
        do_gs                              /* if in Glide Slope mode */
            gs_proc                        /* Glide Slope Outer Loop */
                determine_sw3      /* compute output of switch SW3 */
                    sw3_open   /* determine if switch SW3 is open */
        do_ahd                                /* if in Alt. Hold mode */
            ah_proc                           /* Alt. Hold Outer Loop */
        VOTEOUTER                             /* supplied by coordinating team */
        do_inner_bool  /* determine if Inner Loop must be reinitialized */
        do_inner_init        /* if Inner Loop must be reinitialized */
            set_gsp          /* set old value of Glide Slope mode */
            inner_init                 /* initialize Inner Loop */
        inner_proc                          /* compute Inner Loop */
            init_LR1              /* initialize rate limiter LR1 */
            abs_LM1          /* determines condition for switch SW2 */
            init_LR2              /* initialize rate limiter LR2 */
        do_ahdp_bool           /* compute old value of Alt. Hold mode */
    VOTEINNER                            /* supplied by coordinating team */
    LANEINPUT                            /* supplied by coordinating team */
    Command Monitors
        cm_proc                   /* computes one Command Monitor */
            abs          /* procedure to compute absolute value */
            do_sw6_bool           /* compute condition for switch SW6 */
            limit_LM14            /* compute output of limiter LM14 */
            limit_I11         /* limit the output of integrator I11 */
            do_F13               /* compute output of function F13 */
    VOTEMONITOR                              /* supplied by coordinating team */
    Display
        set_sigmode      /* encode current mode as an integer, store it */
        set_sigdat       /* select signal to be displayed, store it */
    callVotedisplay                             /* interface routine */
        Display_Signals        /* C routine to do the bit manipulation */
            calcnum      /* break the signal value into an integer, */
                      /* a decimal position, and a sign indicator */
        VOTEDISPLAY                       /* supplied by coordinating team */
    do_Recovery                  /* check if recovery is necessary */
        VOTESTATES                      /* supplied by coordinating team */
    check_TD                        /* halt if mode is Touchdown */
```

Figure 6: Structure of the PROLOG Program (continued)

```
fly-airplane
    SENSORINPUT                        /* supplied by coordinating team */
    bacf         /* initialization and call of Baro. Alt. Comp. Filter */
        b_alt_comp_filt                /* Baro. Alt. Comp. Filter */
    racf         /* initialization and call of Radio Alt. Comp. Filter */
        r_alt_comp_filt                /* Radio Alt. Comp. Filter */
        VOTEFILTER1                    /* supplied by coordinating team */
    gscf                  /* constant computation, initialization
                          and call of Glide Slope Dev. Comp. Filter */
        g_slope_comp_filt              /* Glide Slope Dev. Comp. Filter */
    VOTEFILTER2                        /* supplied by coordinating team */
    mode_logic
        fp_error                       /* compute FPEC1 */
        fp_damping                     /* compute FPDC1 */
        pitch_mode_logic               /* compute the Mode Logic output */
            abs                 /* self-defined absolute value
                                that counts zero as positive */
        VOTEMODE                       /* supplied by coordinating team */
    control-plane
        VOTEINNER                      /* supplied by coordinating team */
        AHDouterloop
            VOTEOUTER                  /* supplied by coordinating team */
        GSouterloop
            VOTEOUTER                  /* supplied by coordinating team */
        FDouterloop
            VOTEOUTER                  /* supplied by coordinating team */
        innerloop
            VOTEINNER                  /* supplied by coordinating team */
        LANEINPUT                      /* supplied by coordinating team */
        Monitor                 /* for the two Command Monitors */
            VOTEMONITOR                /* supplied by coordinating team */
        disp                           /* Display Module */
            convert        /* to encode a single digit in the code
                           required by the Signal Display  */
            VOTEDISPLAY                /* supplied by coordinating team */
        VOTESTATES                     /* supplied by coordinating team */
fly-airplane                           /* tail recursion */
```

Figure 7: Structure of the T Program

# CHAPTER 3

# THE IMPLEMENTATION OF THE MAJOR SYSTEM FUNCTIONS

In this chapter it will be examined how different teams translated the algorithms (most of which were specified graphically – as shown in the Appendix) into programs. Most interesting are the differences in the computation sequence with respect to the primitive operations that are defined in the graphic specification. To denote these operations, the same labels as in the specification are used; each label consists of a combination of one or two letters and a number. The letters denote the type of the primitive operation, as shown in Table 9:

| | |
|---|---|
| SU | summer |
| M | multiplier |
| D | divider |
| SW | switch |
| G | gain (constant multiplication factor) |
| I | integrator |
| LM | magnitude limiter |
| LR | rate limiter |
| F | function (either a nonlinear function or a linear filter) |
| ABS | absolute value |
| P | predicate (returns a boolean value) |
| L | logic function (AND, OR) |

Table 9: Primitive Operations

Tables 10 – 18 give the computation sequence as a list of primitive operations. This list has to be read as usual, i.e., from left to right. Furthermore, it is indicated which primitive operations are combined into one statement of the programming language in question by enclosing these operations in round parentheses. In this case, operators that have equal precedence in the examined programming language are assumed to be computed from left to right. Operations that are not enclosed in parentheses are implemented as a single statement. In some cases, the sequence of primitive operations cannot be determined (mostly because two or more constants have been combined into a single constant); then the labels of the corresponding primitive operations are written in a column. Please refer to the Appendix for a reproduction of the figures used in the specification.

The Display algorithms were not described graphically, but by text. For these, any differences in the implementation, the data structures used, and any other differences will be listed and described.

The last subsection compares some aspects of the implementation of the primitive operations themselves.

## 3.1 Barometric Altitude Complementary Filter

This function is defined in figure 3.1 of the specification (see Appendix, page 97).

| | |
|---|---|
| ADA | (SU23, LM9); (SU24, G2); (G4, I4, SU19, G3, SU22, SU20); (I2, G1, SU21); I3 |
| C | SU23; LM9; G4; I4; (SU24, G2); (G3, SU22); (SU19, SU20); I2; (G1, SU21); I3 |
| MODULA-2 | (SU23, LM9); G4; I4; (SU24, G2); (SU19, G3, SU22, SU20); I2; (G1, SU21); I3 |
| PASCAL | (SU23, LM9); G4; I4; (SU24, G2); (G3, $\overset{\text{SU19}}{\underset{\text{SU20}}{\text{SU22}}}$); I2; (G1, SU21); I3 |
| PROLOG | SU23; (SU24, G2); LM9; G4; I4; (G3, $\overset{\text{SU19}}{\underset{\text{SU20}}{\text{SU22}}}$); I2; (G1, SU21); I3 |
| T | (SU24, G2); (SU23, LM9); G4; (G3, SU22); I4; $\substack{\text{SU19,}\\ \text{SU20}}$; I2; (G1, SU21); I3 |

Table 10: Barometric Altitude Complementary Filter Computation Sequence

## 3.2 Radio Altitude Complementary Filter

This function is defined in figure 3.2 of the specification (see Appendix, page 98).

| | |
|---|---|
| ADA | SU25; (G6, SU26); (G5, I5, SU27); F8; I6 |
| C | SU25; (G6, SU26); I5; (G5, SU27); I6; F8 |
| MODULA-2 | SU25; (G6, SU26); I5; (G5, SU27); I6; F8 |
| PASCAL | SU25; (G6, SU26); I5; (G5, SU27); I6; F8 |
| PROLOG | SU25; (G6, SU26); I5; SU27; I6; F8 |
| T | SU25; (G6, SU26); I5; SU27; F8; I6 |

Table 11: Radio Altitude Complementary Filter Computation Sequence

Note: The PROLOG and the T programs omit the multiplication by gain constant G5 since it is 1.00.

## 3.3 Glide Slope Deviation Complementary Filter

This function is defined in figure 3.3 of the specification (see Appendix, page 99).

| | |
|---|---|
| ADA | (G11, SU34); F10; compute K0, K2, K3; (LM13, D12); F9; (M8, SU32); (LM11, G10); (SW5, LM12); I10; (M9, SU33, D11); SU31; (M7, LM10); (G9, I9); (G8, SU30); (SU28, I7); (G7, SU29); I8 |
| C | $\overset{\text{G11'}}{\text{SU34'}}$ ; F10'; compute K0, K2, K3; F9; (M8, SU32); LM11; G10; (LM13, D12); LM12; SW5; I10; $\overset{\text{G11}}{\text{SU34}}$; F10; (M9, SU33, D11); (SU31, M7); LM10; G9; I9; (G8, SU30); SU28; I7; (G7, SU29); I8 |
| MODULA-2 | $\overset{\text{G11}}{\text{SU34}}$; F10; compute K0, K2, K3; F9; (M8, SU32, LM11, G10); (LM13, D12); (SW5, LM12); I10; (M9, SU33, D11); (SU31, M7, LM10); G9; I9; (G8, SU30); SU28; I7; (G7, SU29); I8 |
| PASCAL | $\overset{\text{G11}}{\text{SU34}}$; F10; compute K0, K2, K3; F9; (M8, SU32); (LM11, G10); LM13; D12; (SW5, LM12); I10; (M9, SU33, D11); (SU31, M7); LM10; G9; I9; (G8, SU30); SU28; I7; (G7, SU29); I8 |
| PROLOG | $\overset{\text{G11}}{\text{SU34}}$; F10; compute K0, K2, K3; LM13; D12; F9; (M8, SU32); LM11; G10; LM12; SW5; I10; (M9, SU33, D11); (SU31, M7); LM10; G7; G8; G9; I9; SU30; SU28; I7; SU29; I8 |
| T | $\overset{\text{G11'}}{\text{SU34'}}$ ; F10'; compute K0, K2, K3; F9; (LM13, D12); $\overset{\text{G11}}{\text{SU34}}$; (M8, SU32, LM11); F10; (G10, LM12); SW5; I10; (M9, SU33, D11); SU31; (M7, LM10); G9; I9; (G8, SU30); SU28; I7; (G7, SU29); I8 |

Table 12: Glide Slope Deviation Complementary Filter Computation Sequence

Note: The C and the T programs compute the constants K0, K2, and K3 outside and independently of the Complementary Filter itself. Due to data dependencies, they have to replicate (i.e., implement twice) summer SU34 and the linear filter F10 because their outputs are needed to compute the constants. The C program maintains the state of the replicated F10 and of the one used in the Complementary Filter itself separately; the T program uses the same state variables to compute the constants and in the Complementary Filter itself. In Table 12 above, a prime (') was added to the labels of these replicated primitive operations.

## 3.4 Mode Logic

This function is defined in figures 4.1 and 4.2 of the specification (see Appendix, pages 100 and 101).

| | |
|---|---|
| ADA | in FLARE mode: P7; L14<br>in GST mode: P7; P5; P6; L12; L13; L14; L11<br>in GSC mode: P7; P5; P6; L12; L13; L14; (ABS1, P3); (ABS2, P4); L4; L9; L10; L11; L8<br>in AH mode: P7; P5; P6; L12; L13; L14; P1; (D2, LM5); (D3, LM6); (F1, G12, LM8, G13, D5, G14, SU8); (M1, M3, G15); (D6, G16, F2, G17); (G19, D7, G20, G18, SU9); M4; G21; SU1; P2; (ABS1, P3); (ABS2, P4); L4; L5; L6; L7; L8; L9; L10; L11; L8; L3; L2; L1 |
| C | P5; P6; L12; P7; L13; L14; P1; D2; D3; LM5; LM6; M1; F1; G12; LM8; G13; (D5, G14); SU8; (G15, M3); (D6, G16); F2; G17; G18; (G19, D7, G20); (SU9, M4); G21; SU1; ABS1; P3; P2; ABS2; P4; L4; L5; L6; L7; L8; L3; L2; L1; L9; L10; L11; L8 |
| MODULA-2 | (D2, LM5); (D3, LM6); F1; (G12, LM8, G13, D5, G14, SU8); (M1, M3, G15); (D6, G16); F2; G17; (G19, D7); (G20, G18, SU9, M4); G21; P7; (P5, P6, L12); (L13, L14); SU1; P2; (ABS1, P3, ABS2, P4, L4); (P1, L5, L6); (L7, L8); (L9, L10, L11); SU1; P2; (ABS1, P3, ABS2, P4, L4); (P1, L5, L6); (L7, L8); (L3, L2); L1 |

Table 13: Mode Logic Computation Sequence

| | |
|---|---|
| PASCAL | (D2, LM5); (D3, LM6); (LM8, G13, D5); F1; (SU8, $\substack{G12\\and\\G14}$); (G15, M1, M3); (D6, G16); F2; G17; (G19, D7, G20, G18, SU9, M4); G21; P7; (P5, P6, L12, L13, L14); (ABS1, P3, ABS2, P4, L4); SU1; P2; (L5, P1, L6, L7, L8); (L9, L10, L11); L8; (L3, L2); L1 |
| PROLOG | D2; LM5; D3; LM6; F1; LM8; (G12, $\substack{G13\\D5\\G14}$, SU8); D6; G16; F2; G17; ($\substack{G19\\D7\\G20}$, G18, SU9); M4; G21; (G15, M1, M3); P7; P5; P6; L12; $\substack{L13,\\L14}$; $\substack{ABS1,\\P3}$; $\substack{ABS2,\\P4}$; L4; SU1; P2; P1; $\substack{L5\\L6\\L7\\L8}$; $\substack{L9\\L10\\L11}$; L8; $\substack{L3\\L2\\L1}$ |
| T | (D2, LM5); F1; (G12, LM8, $\substack{G13\\D5\\G14}$, SU8); (D3, LM6); (G15, M1, M3); $\substack{D6\\G16}$; F2; G17; ($\substack{G19\\D7\\G20}$, G18, SU9, M4); G21; P7; (P5, P6, L12, L13, L14); (SU1, P2, ABS2, P4, ABS1, P3, L4, L5, P1, L6, L7, L8); (ABS2, P4, ABS1, P3, L4,L9, L10, L11); L8; (L3, L2); L1 |

Table 13: Mode Logic Computation Sequence (continued)

Note: Instead of multiplying each term of a sum with the same gain constant, the PASCAL program moves the multiplication after the summation. Thus, G12 and G14 are performed simultaneously.

## 3.5 Altitude Hold Control Law, Outer Loop

This function is defined in figure 5.1 of the specification (see Appendix, page 102).

| | |
|---|---|
| ADA | (LM4, G22); (SU1, G23, G24); (D1, G25); G26; SU2 |
| C | (LM4, G22); (SU1, G24, G23, D1, G25); G26; SU2 |
| MODULA-2 | LM4; G22; (SU1, $\begin{array}{c}G23\\G24\\D1\\G25\end{array}$); G26; SU2 |
| PASCAL | (LM4, G22); (SU1, G24, G23, D1, G25); G26; SU2 |
| PROLOG | LM4; SU1; $\begin{array}{c}G23\\G24\\D1\\G25\end{array}$; G26; G22; SU2 |
| T | (LM4, G22); SU1; (G24, G23, G25, D1); G26; SU2 |

Table 14: Altitude Hold Control Law, Outer Loop Computation Sequence

## 3.6 Glide Slope Capture and Track Control Law, Outer Loop

This function is defined in figure 6.1 of the specification (see Appendix, page 103).

| | |
|---|---|
| ADA | (LM4, G22); (D2, LM5); (D3, LM6); M1; (F1, G33); (LM7, G31, D4, G32, SU7); (LM8, G34, D5, G35, SU8); (D6, G37, F2, G38); (G40, D7, G41, G39, SU9, M4); (SW3, M2, SU11); (M3, G36); G42 |
| C | (LM4, G22); D2; LM5; D3; LM6; M1; (LM7, G31); (D4, G32); F1; G33; SU7; M2; (LM8, G34, G35, D5); SU8; (G36, M3); (D6, G37); F2; G38; G39; (G40, D7, G41); (SU9, M4); G42; (SW3, SU11) |
| MODULA-2 | LM4; G22; D2; LM5; D3; LM6; M1; LM7; $\begin{matrix}G31\\D4\end{matrix}$; F1; G33; (G32, SU7); M2; (SW3, SU11); LM8; $\begin{matrix}G34\\D5\end{matrix}$; (G35, SU8); (M3, G36); $\begin{matrix}D6\\G37\end{matrix}$; F2; G38; ( $\begin{matrix}G40\\D7\\G41\end{matrix}$ , G39, SU9); M4; G42 |
| PASCAL | (D6, G37); F1; F2; (LM4, G22); (D2, LM5); (D3, LM6); G33; (LM7, G31, D4); (G32, SU7); M1; (SW3, M2, SU11); (LM8, G34, D5); (G35, SU8); (G36, M3); G38; (G40, D7); (G41, G39, SU9, M4); G42 |
| PROLOG | LM4; D2; LM5; D3; LM6; M1; LM7; F1; G33; ( $\begin{matrix}G31\\D4\\G32\end{matrix}$ , SU7); M2; SW3; LM8; ( $\begin{matrix}G34\\D5\\G35\end{matrix}$ , SU8); D6; G37; F2; G38; ( $\begin{matrix}G40\\D7\\G41\end{matrix}$ , G39, SU9); M4; G22; SU11; (G36, M3); G42 |
| T | (LM4, G22); (D2, LM5); (D3, LM6); (LM7, G31, G32, D4); F1; (LM8, G34, G35, D5); (G33, SU8); (G33, SU7); (G36, M1, M3); (SW3, M1, M2, SU11); (D6, G37); F2; G38; (D7, G40, G41, G39, SU9, M4); G42 |

Table 15: Glide Slope Capture and Track Control Law, Outer Loop Computation Sequence

## 3.7 Flare Control Law, Outer Loop

This function is defined in figure 7.1 of the specification (see Appendix, page 104).

| | |
|---|---|
| ADA | (LM4, G22); (SU11, LM9); (F4, G44); (F3, G43); D8; (F5, SU15, SU14, G46, D9, M6); (G49, D10, G48, SU17, SU16); (F6, G47); (G50, G51, F7); (SU12, M5, SU13, G45); (SW4, SU19); SU18 |
| C | (LM4, G22); (SU11, LM9); D8; (F3, G43); (F4, G44); (SU12, M5); (SU13, G45); (SW4, SU19); (F5, SU15); (SU14, G46, D9); M6; (G49, D10, G48, SU17); SU16; F6; G47; (G51, G50); F7; SU18 |
| MODULA-2 | LM4; G22; (SU11, LM9); (F3, G43); (F4, G44); "D8"; (SU12, M5, SU13, G45); (SW4, SU19); (F5, SU15); (SU14, G46, $\substack{D9\\M6}$); ($\substack{D10\\G48}$, G49, SU17); SU16; F6; G47; $\substack{G50\\G51}$; F7; SU18 |
| PASCAL | SU11; LM9; (SU14, G46); (F5, SU15); D8; (D9, M6); (G49, D10, G48, SU17); SU16; (G50, G51); F6; F7; G47; SU18; F3; F4; G43; G44; (SU12, M5, SU13, G45); (LM4, G22); (SW4, SU19) |
| PROLOG | LM4; SU11; LM9; F3; F4; (G44, G43, SU12); D8; M5; SU13; SW4$^{(1)}$; (G22, G45, SU19); SW4$^{(2)}$; SU14; F5; SU15; (G46, D9); M6; D10; (G48, G49, SU17); SU16; F6; (G50, G51); F7; (G47, SU18); G45; G22; G44; G43; G47 |
| T | (LM4, G22); (SU11, LM9); (F3, G43); (F4, G44); D8; (F5, SU15, SU14, G46, D9, M6); (D10, G48, G49, SU17, SU16); F6; G47; (G50, G51); F7; SU18; (SU12, M5, SU13, G45); (SW4, SU19) |

Table 16: Flare Control Law, Outer Loop Computation Sequence

Notes: In the MODULA-2 program, the division D8 is not really performed, but the two constant inputs were divided manually and just an assignment to a "test point" variable is made. Hence the notation "D8". In the PROLOG program, the computation of switch SW4 is split into two steps: In the first step it is determined whether the switch is open or closed, in the second step the appropriate output value is selected, based on that condition. At the end of the PROLOG program some multiplications are repeated to get the required test point values; thus a few gains appear twice in the list above.

## 3.8 Inner Loop

This function is defined in figures 5.1, 6.1, and 7.1 of the specification (see Appendix, pages 102, 103, and 104).

| | |
|---|---|
| ADA | (SU3, LM2); SU4; LM1; (SU10, G28); (P8, SW2, G27, SU5, I1); SU4; LM1; LR1; (G29, SU6, G30); (LM3, LR2) |
| C | SU3; LM2; SU4; LM1; SU10; G28; P8; SW2; (G27, SU5); I1; SU4; LM1; LR1; (G29, SU6); G30; LM3; LR2 |
| MODULA-2 | (SU3, LM2); SU4; LM1; (SU10, G28); (P8, SW2, G27, SU5); I1; SU4; LM1; LR1; (G29, SU6, G30); LM3; LR2 |
| PASCAL | (SU3, LM2); SU4; LM1; (SU10, G28); P8; (SW2, G27, SU5); I1; SU4; LM1; LR1; (G29, SU6); G30; LM3; LR2 |
| PROLOG | SU3; LM2; SU4; LM1; SU10; P8; SW2; (G27, G28, SU5); I1; SU4; LM1; LR1; (G29, SU6); G30; LM3; LR2 |
| T | (SU3, LM2); SU4; LM1; (SU10, G28); P8; (SW2, G27, SU5); I1; SU4; LM1; LR1; (G29, SU6); G30; LM3; LR2 |

Table 17: Inner Loop Computation Sequence

## 3.9 Command Monitor

This function is defined in figure 8.1 of the specification (see Appendix, page 105).

| | |
|---|---|
| ADA | (SU35, F11); (F12, SW6); LM14; I11; F13 |
| C | (SU35, F11); (F12, SW6); LM14; I11; F13 |
| MODULA-2 | (SU35, F11); (F12, SW6); LM14; I11; F13 |
| PASCAL | (SU35, F11); (F12, SW6); LM14; I11; F13 |
| PROLOG | SU35; F11; F12; SW6; LM14; I11; F13 |
| T | (SU35, F11); (F12, SW6); LM14; I11; F13 |

Table 18: Command Monitor Computation Sequence

Note: There are also some differences in the way the operation F13 is handled. The MODULA-2, PASCAL, PROLOG, and T programs compute a boolean variable which is assigned to the output variable. The C program does the same but reverses the condition in F13, thus using "$\geq$" instead of "<". In the ADA program, the Command Monitor is declared as a function and a RETURN statement is used.

## 3.10 Mode Display

This function is described by text and figures in chapter 9 of the specification. It displays a string of characters that indicate the flight mode. ASCII code is used to represent characters. The PROLOG team used the C programming language to implement that function. Table 19 gives an outline of the algorithm used by each team.

| | |
|---|---|
| ADA | Upon initialization, all five possible outputs are "precomputed" and stored in a static array variable. Constants are used for the ASCII code. Depending on the current mode, the correct output is selected and copied to the parameters of the procedure. |
| C | Constants are used for the ASCII code. Depending on the current mode, the output is computed and assigned to the global variables reserved for the output of the Mode Display. |
| MODULA-2 | Depending on the current mode, the correct output string is selected and assigned to an intermediate variable. The string is then converted to ASCII (using the 'ORD' function) and assigned to the output variables. |
| PASCAL | Depending on the current mode, the appropriate values are assigned to the output parameters. These values have been computed by hand (in the decimal system), for every combination of characters needed. |
| PROLOG | Depending on the current mode, the appropriate values are assigned to the output parameters. These values have been computed by hand, using hexadecimal notation, for every combination of characters needed. |
| T | Depending on the current mode, the appropriate values are assigned to the output parameters. These values have been computed by hand (in the decimal system), for every combination of characters needed. |

Table 19: Mode Display Algorithm

## 3.11 Fault Display

This function is described by text and figures in chapter 9 of the specification. For each Command Monitor, four characters denoting its status are displayed. Bit-mapping to seven-segment display elements has to be performed. The PROLOG team used the C programming language to implement that function.

In principle, all programs use the same algorithm: depending on the result of each Command Monitor, the code for the correct string is assigned to the output variables. The only difference is found in the way this code is computed: for the MODULA-2, PASCAL, PROLOG, and T programs, the codes for all possible two-character combinations have been computed by hand, using hexadecimal constants in the case of PROLOG, and using decimal constants otherwise. The ADA and C program use constants for a single character display. In the C program the two-character combinations are computed every time the Fault Display is called, whereas in the ADA program all possible combinations are computed by the compiler (since in ADA expressions are allowed as constants).

## 3.12 Signal Display

This function is described by text and figures in chapter 9 of the specification. One of 16 possible signals, i.e., inputs or (intermediate) results of the current frame computation, is selected by one of the values input by the SENSORINPUT routine. This value is rounded to 5 decimal digits and displayed. Bit-mapping to seven-segment display elements has to be performed. In addition, the sign and the position of the decimal point have to be encoded. The PROLOG team used the C programming language to implement that function.

In the following, we list the main steps of the algorithm for each program. The purpose is to give an overview of the structure of the algorithms and to provide some implementation details.

### 3.12.1 ADA

1) select the signal to be displayed;

2) limit its value to the displayable range (-99999.0, +99999.0);

3) determine the sign, set the value to zero if its magnitude is smaller than .000005, and use its absolute value from now on;

4) encode the sign in an auxiliary variable;

5) determine the decimal point position and multiply the value by a power of 10 such that the 5 digits to be displayed are to the left of the decimal point;

6) round the value to an integer;

7) adjust the value and the decimal point position in case rounding gave a value larger than 99999;

8) encode the decimal point position in the same auxiliary variable that holds the code for the sign;

9) extract the 5 single digits and store them in an auxiliary array;

10) encode each digit;

11) and put all digits, the sign, and the decimal point position into the output variables in the correct order.

### 3.12.2 C

1) select the signal to be displayed;

2) limit its value to the displayable range (-99999.0, +99999.0);

3) if the magnitude is too small, encode ".00000" in the output variables;

4) otherwise, encode the sign in the appropriate output variable;

5) use absolute value from now on;

6) add 0.5 to the 5th most significant digit (rounding can be done now as truncation);

7) determine the decimal point position;

8) split the value into an integer part and a fraction;

9) get as many single digits as possible from the integer part, get the rest of the digits from the fraction part of the signal value, and store them in an auxiliary array;

10) encode each digit and put it into its place in its output variable;

11) and encode the decimal point position in its output variable.

### 3.12.3 MODULA-2

1) initialize an auxiliary array for the 5 single digits with all zeroes;

2) select the signal to be displayed;

3) if the signal magnitude is too small encode "no sign" and the leftmost decimal point position in an auxiliary variable;

4) otherwise

   a) encode the sign in this auxiliary variable;

   b) use the absolute value from now on;

   c) if the value is too large set all 5 digits to nine and encode the rightmost decimal point position in the auxiliary variable;

   d) otherwise

      - determine the decimal point position while taking into account future rounding errors, and multiply the value by a power of 10 such that the 5 digits to be displayed are to the left of the decimal point;

      - round the value to an integer;

      - encode the decimal point position into the auxiliary variable;

      - extract the 5 single digits and store them in the auxiliary array;

5) and encode each digit, and put it into its place in its output variable. Put sign and decimal point position into the correct output variable.

### 3.12.4 PASCAL

1) select the signal to be displayed;

2) compute the absolute value and limit it to the displayable range: 0 or (.00001, +99999.0);

3) determine the sign and encode it in the appropriate output variable;

4) determine the decimal point position;

5) multiply the value by a power of 10 such that the 5 digits to be displayed are to the left of the decimal point;

6) round the value to an integer;

7) adjust the value and the decimal point position in case rounding gave a value larger than 99999;

8) encode the decimal point position in the appropriate output variable;

9) extract the 5 single digits and store them in an auxiliary array;

10) encode each digit;

11) and put all digits into the output variables in the correct order.

### 3.12.5 PROLOG

1) select the signal to be displayed;

2) determine the (integer) number to be displayed, the decimal point position, and the sign for the three special cases:

    a) the magnitude is too small;

    b) the value is too big (> 99999.0);

    c) the value is too small (< -99999.0);

3) for the normal case:

    a) determine the sign;

    b) determine the decimal point position and multiply the value by a power of 10 such that the 5 digits to be displayed are to the left of the decimal point;

   c) round the value to an integer and take care of any rounding error;

4) encode the sign in the appropriate output variable;

5) encode the decimal point position in the appropriate output variable;

6) extract each of the 5 digits, encode it, and put it in the correct output variable.

### 3.12.6 T

1) select the signal to be displayed;

2) compute the absolute value of the signal;

3) use the absolute value to determine the decimal point position, encode it in an auxiliary variable, add 0.5 to the 5th most significant digit, and multiply the signal value by a power of 10 such that the 5 digits to be displayed are to the left of the decimal point. In addition, if the absolute value is too small set the signal value to zero;

4) limit the signal value to the displayable range (-99999.0, +99999.0);

5) truncate the signal value to an integer;

6) encode the sign in an auxiliary variable;

7) compute the absolute value of the signal and use that from now on;

8) extract the 5 single digits and store them in auxiliary variables;

9) and encode each digit, and put it into its place in its output variable. Put sign and decimal point position into the correct output variable.

### 3.12.7 Additional Observations

Tables 20 – 22 compare some other aspects of the Signal Display which were not mentioned in the survey above.

|  | sign | DPP* | digits |
|---|---|---|---|
| ADA | binary constants | binary constants for each of the 6 possible positions | binary constants |
| C | bit shifting | bit shifting | hexadecimal constants |
| MODULA-2 | decimal constants | partly decimal constants, partly bit shifting | decimal constants |
| PASCAL | decimal constants | decimal constants | decimal constants |
| PROLOG | hexadecimal constants | hexadecimal constants for each of the 6 possible positions | hexadecimal constants, different for digits that have to be put in the most significant byte and in the least significant byte of an output variable |
| T | bit shifting | bit shifting | decimal constants |

* DPP = Decimal Point Position

Table 20: Implementation of Encoding

|  |  |
|---|---|
| ADA | nested IF-statements |
| C | nested IF-statements |
| MODULA-2 | nested IF-statements |
| PASCAL | computed using logarithmic function |
| PROLOG | "while" loop |
| T | "CASE"-statement |

Table 21: Determination of the Decimal Point Position

| | |
|---|---|
| ADA | least significant digit first |
| C | least significant digit first for the integer part, most significant digit first for the fraction part of the signal value |
| MODULA-2 | least significant digit first |
| PASCAL | most significant digit first |
| PROLOG | least significant digit first |
| T | most significant digit first |

Table 22: Sequence of Determining each Single Digit

## 3.13 The Implementation of the Primitive Operations

Most of the programs defined subprograms (either "procedures" or "functions") to implement some of the primitive operations which are used in more than one major system function. Typically, the more complex of these operations were chosen to be implemented by a subprogram. The following Table 23 shows the choices different teams made:

| | |
|---|---|
| ADA | magnitude limiter, rate limiter, linear filter, integrator without bounds, integrator with upper and lower bound |
| C | magnitude limiter, rate limiter, linear filter, integrator with upper and lower bound |
| MODULA-2 | magnitude limiter, rate limiter, linear filter, integrator without bounds |
| PASCAL | magnitude limiter with lower bound only or with upper and lower bound, rate limiter, linear filter, integrator with no bounds, lower bound only, or upper and lower bound |
| PROLOG | magnitude limiter, rate limiter, linear filter, integrator without bounds, switch |
| T | magnitude limiter |

Table 23: Choices of Primitive Operations

The T program is the only one that implements linear filters, rate limiters, and integrators without the help of a subprogram. The magnitude limiter is used to implement the rate limiters and the integrators with bounds.

"Integrator without bounds" means that the subprogram integrates, but does not do any magnitude limiting on the integration result, as is required in most cases. "Integrator with upper and lower bound" means that the subprogram is designed to integrate first, then perform magnitude limiting on the integration result according to the upper and lower bound specified. As can be seen from the table above, different teams chose different variations as "their" primitive operation. The ADA team implemented both variations as independent subprograms, whereas the PASCAL team integrated all variations into one subprogram and used an additional parameter to decide which variation to use at a given invocation. In addition, they also implemented the case where only a lower bound is specified, both for the magnitude limiter and for the integrator. All other teams just chose one of the above variations as "their" primitive operation. The teams that chose "integrator without bounds" (MODULA-2, PROLOG) implemented integrators with bounds by calling the magnitude limiter after the integration.

The PROLOG program implements the "switch" operation as a subprogram. This choice has clearly been influenced by the programming language properties – all other programs just use IF-statements.

The algorithms for these primitive operations are rather simple and have been specified very clearly. So only very few differences can be observed here. However, the ADA program is the only one that uses the magnitude limiter to implement the rate limiter and the integrator with bounds. Then, the C program is the only one to use two parameters to define a particular linear filter; all other programs just use one.

The PROLOG program only implements these subprograms as procedures, where the result is returned in a parameter. All other programs use functions. Input parameters are (with the exception of the ADA program) the input value, the state variable(s) (for state dependent computations, i.e. integrators, linear filters, and rate limiters), and bounds or filter constants (where appropriate). The ADA subprograms for state dependent computation only use the input value as an input parameter. This can be done because of the "generic definitions" in ADA: state variables can be stored locally, and bounds and filter constants must be only specified once, at the actual instantization of a single rate limiter, linear filter, or integrator. Therefore, the ADA functions are the only ones that also perform the state update; in all other programs this is done outside of these primitive operation routines. Thus, these ADA functions are actually functions with side-effects; in all other programs these functions are side-effect free.

Throughout the specification there are four special cases of integrators or magnitude limiters: Magnitude Limiter LM9 in the Outer Loop of the Flare Control Law has only a lower bound, Integrator I1 in the Inner Loop has no bounds, Magnitude

Limiter LM14 in the Command Monitor has only an upper bound, and Integrator I11 in the Command Monitor has only a lower bound. The following Tables 24 – 27 give a comparison of how different programs handle these special cases:

| | |
|---|---|
| ADA | current input used as upper bound |
| C | arbitrarily chosen large number (99999.0) used as upper bound |
| MODULA-2 | system constant MAXINT (converted to double precision) used as upper bound |
| PASCAL | independent implementation (no call to magnitude limiter) |
| PROLOG | independent implementation (no call to magnitude limiter) |
| T | independent implementation (no call to magnitude limiter) |

Table 24: Limiter LM9: Lower Bound Only

| | |
|---|---|
| ADA | primitive |
| C | arbitrarily chosen large and small numbers (+99999.0, -99999.0) used as bounds |
| MODULA-2 | primitive |
| PASCAL | primitive |
| PROLOG | primitive |
| T | N/A |

Table 25: Integrator I1: No Bounds

| | |
|---|---|
| ADA | system constant (smallest possible real number) used as lower bound |
| C | arbitrarily chosen small number (-99999.0) used as lower bound |
| MODULA-2 | independent implementation (no call to magnitude limiter) |
| PASCAL | independent implementation (no call to magnitude limiter) |
| PROLOG | independent implementation (no call to magnitude limiter) |
| T | independent implementation (no call to magnitude limiter) |

Table 26: Limiter LM14: Upper Bound Only

| | |
|---|---|
| ADA | system constant (largest possible real number) used as upper bound |
| C | arbitrarily chosen large number (99999.0) used as upper bound |
| MODULA-2 | independent implementation of output limiting (no call to magnitude limiter) |
| PASCAL | independent implementation (no call to integrator or magnitude limiter) |
| PROLOG | independent implementation of output limiting (no call to magnitude limiter) |
| T | independent implementation of output limiting (no call to magnitude limiter) |

Table 27: Integrator I11: Lower Bound Only

It is interesting to note that the PASCAL program actually *does not* use the variations of the magnitude limiter and the integrator that perform limiting on the lower bound only. LM9 and I11 are just these special cases. This is probably due to a communication problem between the two team members.

# CHAPTER 4

# COMPARISON OF VARIOUS OTHER ASPECTS

In this chapter, a number of aspects of the six programs that are not readily classified as being relevant to either the overall structure and organization or the computation sequence is examined. Some aspects, e.g., the use of variables or the use of voting routines, are more related to the overall organization, others, e.g., the initialization, the implementation of time-dependent computations, or the implementation of mode-dependent computations, are more related to the category of implementation details.

## 4.1 The Declaration and Use of Variables

The purpose of this subsection is to compare how variables were used in the different programs. This comparison is done in two ways: first, there will be an extensive survey of the programs on a "variable by variable" basis, then the information will be summarized on a "program by program" basis in order to try to expose the strategy used by each programming team.

In the following Tables 28 – 44, for each group of variables the following information is given: first, whether the variables in question are defined globally (i.e., in a way such that they are automatically known to all the subprograms of the Main Program) or locally (i.e., within one subprogram). With programming languages that explicitly support modularization constructs (ADA, MODULA-2) this distinction is a bit tricky: it is possible to define variables locally in a module, but to explicitly import or export them to other modules. Thus, it is possible to construct very specific name scopes for different kinds of variables. The notation "global" will be used in these cases. Then, it is indicated to which modules the variables in question are passed as parameters, and which modules use these variables as global variables. The notation $\rightarrow$ indicates a "chain" of parameter passing.

| | defined in | passed as parameter(s) to | used as global variable(s) by |
|---|---|---|---|
| ADA | Main Program | Comp. Filters, Mode Logic, Outer Loop procedures, Inner Loop | |
| C | global | | Comp. Filters, Mode Logic, Outer Loop procedures, Inner Loop |
| MODULA-2 | Main Program | Filter Module, Mode Logic, Control Laws, Display Module | Comp. Filters, Signal Display |
| PASCAL | Main Program | Filter Module → Comp. Filters, Mode Logic → subprograms thereof, Outer Loop → Outer Loop procedures, Inner Loop, Display Module | |
| PROLOG | global | | Comp. Filters, Mode Logic, Outer Loop procedures, Inner Loop, Display Module |
| T | global | Organizational harnesses of Comp. Filters → Comp. Filters, Mode Logic → subprograms thereof, Outer Loop procedures, Inner Loop, Display Module | |

Table 28: Parameters of SENSORINPUT

|  | defined in | passed as parameter(s) to | used as global variable(s) by |
|---|---|---|---|
| ADA | "global" |  | Main Program |
| C | global |  | Main Program, Comp. Filters |
| MODULA-2 | "global" |  | Main Program |
| PASCAL | Main Program | Filter Module → Comp. Filters |  |
| PROLOG | global |  | Main Program, Comp. Filters |
| T | global |  | Main Program, Organizational harnesses of Comp. Filters, Comp. Filters |

Table 29: State Variables of the Complementary Filters

|  | defined in | passed as parameter(s) to | used as global variable(s) by |
|---|---|---|---|
| ADA | "global" |  | Main Program |
| C | Filter Module | Comp. Filters |  |
| MODULA-2 | Filter Module |  | Comp. Filters |
| PASCAL | Filter Module | Comp. Filters |  |
| PROLOG | global |  | Main Program, Comp. Filters |
| T | global |  | Main Program, Comp. Filters, organizational harness of Radio Alt. Comp. Filter |

Table 30: Test Points of the Complementary Filters

|  | defined in | passed as parameter(s) to | used as global variable(s) by |
|---|---|---|---|
| ADA | Main Program | Comp. Filters, Mode Logic, Outer Loop procedures |  |
| C | global |  | Filter Module, Comp. Filters, Outer Loop procedures, Signal Display |
| MODULA-2 | Main Program | Filter Module, Mode Logic, Control Laws, Display Module | Comp. Filters, Outer Loop procedures, Signal Display |
| PASCAL | Main Program | Filter Module → Comp. Filters, Mode Logic, Outer Loop → Outer Loop procedures, Display Module |  |
| PROLOG | global |  | Main Program, Comp. Filters, Mode Logic, Outer Loop procedures, Display Module |
| T | global | Glide Slope Comp. Filter harness → Glide Slope Comp. Filter, Mode Logic → subprograms thereof, Outer Loop procedures, Inner Loop, Display Module | Organizational harnesses of Comp. Filters, Comp. Filters |

Table 31: Results of the Complementary Filters

|  | defined in | passed as parameter(s) to | used as global variable(s) by |
|---|---|---|---|
| ADA | "global" |  | Main Program |
| C | global |  | Main Program, Mode Logic |
| MODULA-2 | "global" |  | Main Program |
| PASCAL | Main Program | Mode Logic → subprograms thereof |  |
| PROLOG | global |  | Main Program, Mode Logic |
| T | global |  | Main Program, subprograms of the Mode Logic |

Table 32: State Variables of the Mode Logic

|  | defined in | passed as parameter(s) to | used as global variable(s) by |
|---|---|---|---|
| ADA | "global" |  | Main Program |
| C | Mode Logic | subprogram "flight path" |  |
| MODULA-2 | Mode Logic |  | subprograms of Mode Logic |
| PASCAL | Mode Logic | subprograms of Mode Logic |  |
| PROLOG | global |  | Main Program, Mode Logic |
| T | Mode Logic |  | subprograms of Mode Logic |

Table 33: Test Points of the Mode Logic

|  | defined in | passed as parameter(s) to | used as global variable(s) by |
|---|---|---|---|
| ADA | "global" |  | Main Program, Glide Slope Dev. Comp. Filter, Inner Loop, Mode Display |
| C | global |  | Main Program, Filter Module, Mode Logic, Inner Loop, Mode Display |
| MODULA-2 | Main Program | Filter Module, Mode Logic, Display Module | Glide Slope Dev. Comp. Filter, Mode Display |
| PASCAL | Main Program | Filter Module → Glide Slope Dev. Comp. Filter, Mode Logic, Display Module |  |
| PROLOG | global |  | Main Program, Glide Slope Dev. Comp. Filter, Mode Logic, Display Module |
| T | global | organizational harnesses of Comp. Filters → Comp. Filters, Altitude Hold Outer Loop, Glide Slope Capture and Track Outer Loop, Inner Loop, Display Module | Main Program, Mode Logic |

Table 34: Results of the Mode Logic

| | defined in | passed as parameter(s) to | used as global variable(s) by |
|---|---|---|---|
| ADA | "global" | | Main Program |
| C | global | | Main Program, Outer Loop procedures |
| MODULA-2 | "global" | | Main Program |
| PASCAL | Main Program | Outer Loop → Outer Loop procedures | |
| PROLOG | global | | Main Program, Outer Loop procedures |
| T | global | | Main Program, Outer Loop procedures |

Table 35: State Variables of the Outer Loop

| | defined in | passed as parameter(s) to | used as global variable(s) by |
|---|---|---|---|
| ADA | global | | Main Program, Outer Loop procedures |
| C | global | | Outer Loop procedures |
| MODULA-2 | each Outer Loop procedure | | |
| PASCAL | Outer Loop | Outer Loop procedures | |
| PROLOG | global | | Main Program, Outer Loop procedures |
| T | each Outer Loop procedure | | |

Table 36: Test Points of the Outer Loop

|  | defined in | passed as parameter(s) to | used as global variable(s) by |
|---|---|---|---|
| ADA | Main Program | Outer Loop procedures, Inner Loop |  |
| C | each Control Law | Outer Loop procedures, Inner Loop |  |
| MODULA-2 | each Control Law | Inner Loop | Outer Loop procedures |
| PASCAL | Main Program | Outer Loop → Outer Loop procedures, Inner Loop |  |
| PROLOG | global |  | Main Program, Outer Loop procedures, Inner Loop |
| T | global | Inner Loop | Outer Loop procedures |

Table 37: Results of the Outer Loop

|  | defined in | passed as parameter(s) to | used as global variable(s) by |
|---|---|---|---|
| ADA | "global" |  | Main Program |
| C | global |  | Main Program, Inner Loop |
| MODULA-2 | "global" |  | Main Program |
| PASCAL | Main Program | Inner Loop |  |
| PROLOG | global |  | Main Program, Inner Loop |
| T | global |  | Main Program, Inner Loop |

Table 38: State Variables of the Inner Loop

|  | defined in | passed as parameter(s) to | used as global variable(s) by |
|---|---|---|---|
| ADA | global |  | Main Program |
| C | Main Program | Control Laws → Inner Loop |  |
| MODULA-2 | Inner Loop |  |  |
| PASCAL | Inner Loop |  |  |
| PROLOG | global |  | Main Program, Inner Loop |
| T | Inner Loop |  |  |

Table 39: Test Points of the Inner Loop

|  | defined in | passed as parameter to | used as global variable by |
|---|---|---|---|
| ADA | Main Program | Command Monitors |  |
| C | global |  | Main Program, Inner Loop, Command Monitors, Signal Display |
| MODULA-2 | Main Program | Control Laws → Inner Loop, Command Monitors, Display Module | Signal Display |
| PASCAL | Main Program | Inner Loop, Monitor Module → Command Monitor, Display Module |  |
| PROLOG | global |  | Main Program, Inner Loop, Command Monitors, Display Module |
| T | global | Command Monitors, Display Module | Main Program, Inner Loop |

Table 40: Result of the Inner Loop

|  | defined in | passed as parameter(s) to | used as global variable(s) by |
|---|---|---|---|
| ADA | Main Program | Command Monitors |  |
| C | global, Main Program |  | Command Monitors |
| MODULA-2 | Main Program | Command Monitors |  |
| PASCAL | Main Program | Monitor Module → Command Monitor |  |
| PROLOG | global |  | Main Program, Command Monitors |
| T | global | Command Monitors | Main Program |

Table 41: Parameters of LANEINPUT

|  | defined in | passed as parameter(s) to | used as global variable(s) by |
|---|---|---|---|
| ADA | "global" |  | Main Program |
| C | global |  | Main Program, Command Monitors |
| MODULA-2 | "global" |  | Main Program |
| PASCAL | Main Program | Monitor Module → Command Monitors |  |
| PROLOG | global |  | Main Program, Command Monitors |
| T | global |  | Main Program, Command Monitors |

Table 42: State Variables of the Command Monitors

| | defined in | passed as parameter(s) to | used as global variable(s) by |
|---|---|---|---|
| ADA | Main Program | Fault Display | |
| C | global | | Command Monitors, Fault Display |
| MODULA-2 | Main Program | Command Monitors, Display Module | Fault Display |
| PASCAL | Main Program | Monitor Module $\rightarrow$ Command Monitors, Display Module $\rightarrow$ Fault Display | |
| PROLOG | global | | Main Program, Command Monitors |
| T | global | | Main Program, Command Monitors, Display Module |

Table 43: Results of the Command Monitors

| | defined in | passed as parameter(s) to | used as global variable(s) by |
|---|---|---|---|
| ADA | Main Program | Mode Display, Fault Display, Signal Display | |
| C | global | | Mode Display, Fault Display, Signal Display |
| MODULA-2 | Display Module | | Mode Display, Fault Display, Signal Display |
| PASCAL | Display Module | Mode Display, Fault Display, Signal Display | |
| PROLOG | global | | |
| T | Display Module | | |

Table 44: Results of the Display Module

Note: The variables for the Display results are not used in the PROLOG program because the Display was implemented in the C programming language as part of the

C-PROLOG interpreter.

In the remainder of this subsection, the information given above is summarized for each program:

**ADA:**

All variables that make up the interface between different Modules are defined in the Main Program, and are passed by parameter to the other Modules. An exception is the results of the Mode Logic which are defined in the declaration of the Mode Logic package; thus they are "global" variables. This obviously was done because these results are used by other Modules besides the Main Program, thus avoiding excessive parameter passing. All test point and state variables are defined in the declaration of their corresponding package. All packages are imported in the Main Program, thus these are "global" variables. The exception are the test points of the three Control Laws (Outer and Inner Loop): These variables are defined in a separate package, so that common test point variables can be re-used in different Control Laws. All other variables are local to the package they are defined in.

**C:**

All interface and state variables are declared global to the Main Program and are used as global variables by the other Modules; there is no parameter passing. There are two exceptions to that: first, the variables for the parameters of LANEIN-

PUT are defined twice, once global and once in the Main Program. Global variables are used in the Monitor Module, but the locally defined variables are set by LANEINPUT. This is a programming error, because the Monitor Module is required to use the values supplied by LANEINPUT. Currently, we are investigating why this error does not cause any failures during test executions. Secondly, the results of the Display Module are defined locally, and are used as global variables by its subroutines. Test point and other variables are defined locally to the subprogram where they are used. The test points for the Outer Loops of the three Control Laws, however, are defined as global variables, used by the Outer Loop procedures. The test points of the Inner Loop are defined locally to the Main Program and are passed as parameters to the Inner Loop.

**MODULA-2:**

The interface variables are defined in the Main Program and passed as parameters to all Modules that use them. Typically, there is only one step of parameter passing, i.e., all subprograms in a given Module use the parameters as global variables. Again, there are two exceptions: first, the interface variables between Outer and Inner Loop are defined locally in each Control Law module. They are passed as parameters to the Inner Loop and used as global variables by the Outer Loop. Secondly, the results of the Display Module are defined locally; its subprograms use them as global variables. All state variables are defined in the declaration part of each module and imported by the Main Program to be used with VOTESTATES. All test point and other variables are defined locally.

**PASCAL:**

All interface and state variables are defined in the Main Program and are passed as parameters to all subprograms that use them. Extensive parameter passing takes place; no global variables are used in this program. Here, the only exception is the results of the Display Module: these variables are defined locally and are passed as parameters to its subprograms. All test point and other variables are defined locally.

**PROLOG:**

All interface, state, and test point variables are defined globally, in the PROLOG database. Every subprogram accesses the variables it needs and updates them at the end if they were modified.

**T:**

All interface and state variables are defined to be global to the Main Program. State variables are used as global variables in the appropriate Modules and by the Main Program, for the routine VOTESTATES. Interface variables are passed as parameters to a Module if they are inputs, and are used as global variables if they are outputs. This is obviously due to the fact that there is no parameter passing by address in T. Only the output variables of the Display Module are declared locally since they are not used by any other Module. Test point variables are usually defined local to their Module. The test points of the Complementary Filters are an exception: these are defined global to the Main Program because VOTEFILTER2 is called in the Main Program; thus the test points of the Barometric Altitude and the Glide Slope Deviation Complementary Filters have to be passed to the Main Program.

## 4.2 Placement of Calls to Vote Routines

Vote Routines are the routines that were supplied by the coordinating team for cross-checking and recovery. There are 8 Vote Routines, seven of which implement one Cross-Check Point each, and one that implements the Recovery Point (cf. Chapter 1). The sequence and the syntax of the calls to these Vote Routines have been specified exactly. But the programming teams could still decide where to place these calls in their program.

Basically, two different approaches are found: The ADA and the PROLOG program called all Vote Routines in the Main Program, whereas with all other teams the vote routines are called in the appropriate function whose result they check. In the latter case, there are two exceptions: VOTESTATES is always called by the Main Program, and VOTEINNER is called by the Main Program in the case the mode is TOUCHDOWN. This was necessary because VOTESTATES uses information from all the Modules of the program, and because in the case of TOUCHDOWN the Inner Loop is not computed, thus VOTEINNER has to be called by the Main Program.

The T program uses a variation of the latter approach: VOTEFILTER2 is called in the Main Program because it checks the results and test points of two different Complementary Filters; therefore, it cannot be called within any single Complementary Filter.

In addition, some peculiarities can be observed. It turned out that the MODULA-2 Compiler could not handle the number of parameters of VOTESTATES (there are 42). Therefore, the MODULA-2 program puts all the parameters in a record, and passes this record to an auxiliary function XVOTESTATES (written in C) which in turn gets all the parameters from the record and calls VOTESTATES. The vote results are returned to the MODULA-2 program in the same manner. In the PROLOG program, the Vote Routines are called by subroutines of the Main Program that get the necessary values from the PROLOG database, call the Vote Routine, and store the return values back into the database. The Vote Routines are actually installed in the C-PROLOG interpreter.

## 4.3 Initialization

For the state dependent computations (rate limiter, integrator, linear filter), the state has to be initialized before the corresponding primitive operation is performed for the first time. This is a rather tricky part of the problem, not only because of the number of state variables, but also because different modules start to compute at different times. In addition, some states are not initialized by a constant, but either by an input value or some intermediate value of the corresponding modules. In the following paragraphs, the strategy of each team is briefly described.

In the ADA program, for each module that has internal states there is a corresponding initialization routine which is called by the Main Program. The initialization of the Complementary Filters (with one exception – see below), the Mode Logic, and the Command Monitors is done before the main loop of the Main Program, after the first call to SENSORINPUT. Initialization of the Control Laws is performed within the main loop; three boolean flags (AH_First, GS_First, FD_First) keep track of whether the corresponding Control Law has already been initialized or not. The initialization of the Mode Logic is organized in a rather peculiar way which is due to the fact that originally an output of the Glide Slope Deviation Complementary Filter is needed for the initialization. Thus, the Radio Altitude Complementary Filter has to be initialized and computed to provide one of the inputs to the Glide Slope Deviation Complementary Filter. Then, the latter can be initialized and computed which in turn enables the initialization of the Mode Logic. Now, however, the state of the Radio Altitude Complementary Filter and the Glide Slope Deviation Complementary Filter are not what they should be at the beginning of the first time frame, so these two functions have to be re-initialized. Despite a later specification change (which now requires only constants for the Mode Logic initialization), this complicated approach was not simplified. The initialization of integrator I8 in the Glide Slope Deviation Complementary Filter is done within its computation procedure, with the help of a local static variable "First". In the initialization procedures for the Outer Loop of the Flare Control Law and the Inner Loop the intermediate values needed for initialization are "precomputed", i.e. the necessary part of the algorithm is implemented. This applies to the output of linear filter F6 in the Outer Loop, and to the rate limiters LR1 and LR2 in the Inner Loop.

In the C program, an initialization procedure "init_main" is called by the Main Program after the first call to SENSORINPUT. It initializes all states whose initial value is defined at this point. Whenever the flight mode changes, "init_law" is called by the Main Program to set some flags that trigger re-initialization of the Inner Loop. All other initializations (these are in the Glide Slope Deviation Complementary Filter, the three Outer Loops, and the Inner Loop) are performed within the procedures that implement the corresponding functions. Global flags are used to keep track of whether initialization must be performed or not.

In the MODULA-2 program, the results of the Mode Logic are initialized at the beginning of the Main Program. All other initializations are performed within the module computation. A static variable "first" (local to each module) that itself is set in the initialization part of each module keeps track of whether initialization has to be done or not. Only for the Inner Loop, this flag is passed as parameter from the three Control Law procedures. Some state variables (Mode Logic, Command Monitor) are initialized directly in the initialization part of their corresponding module.

In the PASCAL program, mostly initialization routines are used to perform the initialization. The only exceptions are the output of the linear filter F10 and the output of the integrator I8, both in the Glide Slope Deviation Complementary Filter, which are initialized during their first computation, by using a flag "firstround" which is passed as a parameter from the Main Program. The global time, the results of the Mode Logic, and some other (auxiliary) state variables are initialized at the beginning of the Main Program. After the first call to SENSORINPUT, the procedure AHDINITIALIZE takes care of the Complementary Filters, the Mode Logic, the Command Monitors, and the Inner Loop for Altitude Hold mode. The Outer Loop of the Altitude Hold Control Law is initialized by the Main Program, after the Complementary Filter processing. If the flight mode has changed, OUTERFLAREINIT or OUTERGSCTCINIT, respectively, and INNERINIT are called by the Main Program. In all cases, intermediate values that are needed for initialization are obtained by re-implementing the corresponding part of the algorithm.

Similar to the ADA program, the PROLOG program has an initialization procedure for each of its computation procedures. Most of them (Complementary Filters, Mode Logic, Inner Loop, and Command Monitors) are called before the main loop of the Main Program, after the first call to SENSORINPUT. The Outer Loop of the Altitude Hold Control Law and part of the Mode Logic are initialized after the Complementary Filter processing. (Note: To initialize the Mode Logic at this point of the computation became unnecessary after a specification change – see the corresponding remarks in the paragraph about the ADA team.) The initialization of the Outer Loops of the Glide Slope Capture and Track and the Flare Control Law, and the re-initialization of the Inner Loop are performed by subroutines of the Main Program whenever the flight mode has been changed. All state variables which cannot be initialized in either of these ways (e.g. because they need intermediate values) are initialized within the corresponding computation procedure. These variables are the output of linear filter F10 and integrator I8 in the Glide Slope Deviation Complementary Filter, the output of linear filter F6 in the Outer Loop of the Flare Control Law, and the state of the rate limiters LR1 and LR2 in the Inner Loop. In these case, the corresponding initialization procedures set a global boolean initialization flag.

In the T program, the results of the Mode Logic are initialized at the beginning of the Main Program. All other initializations are performed within the appropriate Module. For the Complementary Filters, this is done within their organizational harnesses. For all other Modules, it is done at the beginning or (if that is not possible) during the Module computation, by using global boolean flags and/or the results of the Mode Logic.

## 4.4 Handling the Differences of the Inner Loop

Although the Inner Loop is basically the same in each flight mode, there are slight differences, depending on the flight mode. The input "Pitch Attitude" (PA) is only used in Altitude Hold mode, and the gain constant G27 (refer to the corresponding figures in the Appendix) has a different value in each of the three flight modes. Finally, the initialization of the output of integrator I1 is different in Altitude Hold mode than in the other two modes. Table 45 shows how these differences were implemented in each program.

|  |  |
|---|---|
| ADA | The global flight mode variables are used in the Inner Loop and its initialization procedure. |
| C | The global flight mode variables are used in the Inner Loop. |
| MODULA-2 | The Inner Loop is called from different Control Law procedures, depending on the flight mode. The correct gain constant and the correct value of PA (zero if it is not used) are passed as parameters. For correct initialization a flag indicating whether the system is in Altitude Hold mode or not is passed as parameter. |
| PASCAL | The correct gain constant and a boolean flag indicating whether PA should be used or not are passed as parameters. The gain constant is set by AHDINITIALIZE (for Altitude Hold mode) and by the Main Program (for the other modes); the flag is turned on by AHDINITIALIZE and turned off by INNERINIT. The initialization for Altitude Hold mode is performed by AHDINITIALIZE, the initialization for the other two modes is done by INNERINIT (which is only called when the mode has been changed). |
| PROLOG | The Inner Loop uses special global variables for the gain constant and the parameter PA. The correct gain value is set at the beginning of the program and by the initialization procedures for the G/S Capture and Track and the Flare mode, respectively. The Outer Loop of the Altitude Hold Control Law copies the current value of PA to the global variable used by the Inner Loop. This global variable is set to zero by the initialization procedures for the G/S Capture and Track and the Flare mode because PA is not used in these modes. |
| T | The global flight mode variables are used in the Inner Loop. |

Table 45: Implementation of Inner Loop Variations

## 4.5 Time-Dependent Computations

There are two references to the global time in the specification. The first one is in the Glide Slope Complementary Filter where the algorithm to compute its constants K0, K2, and K3 has to be changed after 10 seconds of Altitude Hold mode. The second is in the Outer Loop of the Glide Slope Capture and Track Control Law where switch SW3 should be closed 0.5 seconds after the Control Law has been entered. Following is an overview of how different programs implemented these requirements.

The ADA program uses a local static variable that counts the frames in Altitude Hold mode. The condition to detect that 10 seconds have not yet passed is: "FLOAT(frame) <= (10.0/DELT)" where DELT is the length of a computation frame. For switch SW3, a local static variable keeps track of the time since the Control Law was entered. The condition to close switch SW3 is: "timer >= 0.5 - 1E-4".

The C program uses a global counter to count time frames; it is initialized by zero and updated at the end of each computation frame. So, for the constant computation, the condition to detect that 10 seconds have passed is: "(time_count > 199.9)". (Note: 10 seconds correspond to 200 computation frames.) For switch SW3, a global variable is used that keeps track of the time spent in Glide Slide Capture or Glide Slide Track mode. The condition to close switch SW3 is: "(gscd || gstd) && (time >= 0.45)" where the expression "(gscd || gstd)" is redundant because it is always true when the Glide Slope Capture and Track Control Law is executed.

In the MODULA-2 program a local static variable counts the frames beginning in Altitude Hold mode. The condition to detect that 10 seconds have passed is: "ctr >= 200". For switch SW3, a local static variable keeps track of the time since the Control Law was entered. The condition to close switch SW3 is: "framecount >= 0.5".

The PASCAL Program uses a variable that counts the time frames from the beginning of the program. It is updated at the beginning of each time frame, thus the first frame is counted as "1", etc. The Main Program encodes the algorithm to compute the constants in an integer variable which is passed to the Glide Slope Deviation Complementary Filter. There, only a case selection has to be performed. The condition for detecting that 10 seconds have not yet passed is: "TIMER <= 200". For switch SW3, the time where SW3 should be closed is computed by "TIMER + 10" upon Glide Slope mode initialization. It is stored in a variable of the Main Program which is passed to the Outer Loop of the Glide Slope Capture and Track Control Law, along with the current TIMER. Thus, the condition to close SW3 is: "TIMER >= SW3CLOSETIME".

In the PROLOG program, a global variable keeps track of the elapsed time. It is updated at the beginning of each computation frame. The condition to detect that 10 seconds have passed is: "elapsed_time > 10 + DELTA/2". For switch SW3, the current time is stored in another global variable, "gs_timer", upon initialization of the Glide Slope mode. If the switch is open it is determined if it should stay open by "elapsed_time - gs_timer =< (0.45 + DELTA/2)". If it turns out that the switch should be closed a global variable "sw3_closed" is created in the PROLOG database whose presence indicates for all future computations that SW3 is closed.

The T program uses a local static variable "timer" to keep track of the time spent in Altitude Hold mode. The condition to detect that 10 seconds have not yet passed is: "(<= timer 10.0)". For switch SW3, a global variable counts the frames in Glide Slope Capture and Glide Slope Track modes. The condition to determine that switch SW3 should stay open is: "(< gscount 11)". (Note: 0.5 seconds correspond to

10 computation frames.)

## 4.6 Resetting of Test Points

The specification was augmented by UCLA researchers with the requirements that whenever a module (or a part of it) is not computed then default values of 0.0 should be passed to the Vote Routines for the results and the test point values of this module. This was done to ensure that comparison of these values did not result in false alarms in situations where these values are not significant. Most programs use a number of auxiliary procedures to reset the values of certain test point variables. More often than not, this resetting is done in every iteration (computation frame).

Only the PROLOG program resets unused variables consistently only once during the execution of the program. The resetting is done by the initialization routines that are called when the mode has changed. Variables that are not used during the new flight mode are set to zero. This strategy is possible because all variables are global in PROLOG. The Mode Logic test points are reset only once also by the ADA program, upon leaving Altitude Hold mode.

The MODULA-2 and the T teams were the only ones that realized that constants can be passed directly to the vote routines. Thus, in the Outer Loops "0.0" and "0", respectively, is passed to VOTEOUTER for all undefined test point variables. As a result, there is no need to reset any of the Outer Loop test point variables. The same approach is used for VOTEINNER when the mode is TOUCHDOWN.

# CHAPTER 5

# CONCLUSIONS

## 5.1 Summary of Results

In the three preceding chapters, an attempt was made to examine and analyze the structure of the six versions at different levels. In order to draw any meaningful conclusions about the ability of the N-version programming methodology to produce diverse versions, one first has to estimate the *potential for diversity* (PFD) that is indicated by a given specification. Thus, the PFD is an estimate of the amount of diversity that can be expected to occur between different versions.

The "PFD" column of the following Table 46 gives an 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 46 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.

| 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 46: Potential for and Observed Diversity

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 (cf. Figure 2 in chapter 2). 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 calls to 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 46 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.

## 5.2 Observations and Implications for the N-Version Design Principle

In general, it can be said that more diversity was observed in the parts of the program 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, then 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", i.e., implied 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, then there exist some orderings of these computations that are more likely to be chosen than other orderings, due to human, psychological factors.

The choice of an implied sequence affected the Outer Loops of the Glide Slope Capture and Track and the Flare Control Law, and the Mode Logic 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 implied "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 "implied sequence" problem is only one example of a more general problem, i.e., "*unintended implied messages*" in a specification. This term denotes any feature of the specification that does not reflect requirements of the application problem, and that somehow biases the programmers in their design decisions. An example is the specification of the Mode Logic in two separate figures. This caused all teams to partition the Mode Logic accordingly, and the PROLOG team even treated the two parts almost independent from each other. The reasons for presenting the requirements in this way are the convenience of figure layout, and the fact that one part of the Mode Logic is identical to another, completely unrelated algorithm. At present, no general method is known to detect and prevent such unintended implied messages. One possible approach might be the use of formal specification techniques. These are (hopefully) able to express exactly what is wanted. Furthermore, they are more difficult to understand and require more thought by the programmers; thus it may become more likely that one obtains diverse programs by relying on the programmers' different backgrounds and approaches to the problem. On the other hand, it is conceivable that some unintended implied messages might also be used to *enforce* diversity: include some subtle implied messages (these would then be *intended*) in the specification, and rely on that some teams "get" the message and some teams do not.

The 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 system functions, 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.

It is difficult to make a general statement on the impact of different programming languages on the diversity observed. Some single differences can easily be attributed to programming language properties and have been pointed out in the appropriate subsection of the thesis. The properties of the PROLOG language have been most influential, examples are the exclusive use of global variables, implementation of IF-statements as PROLOG-functions, and implementation of a "switch" primitive operation. The result was a program structure that was quite different from the other five programs. The properties of the T language resulted in only very few differences, because the programmers simulated the programming style of conventional languages with the constructs available in T. Not much truly "functional programming" has been observed. The other four languages are very similar with respect to instruction semantics. Some differences can be attributed to the presence of explicit modularization constructs in ADA and MODULA-2, such as certain aspects of using local versus global variables, or information hiding. The unavailability of local static variables and of the possibility to declare variables that are global to the Main Program led to the scheme of extensive parameter passing, that the PASCAL program employed. The C program, on the other hand did not use local static variables at all, although they are available in C. In conclusion, it is felt that in these four programming languages the programmers had more freedom to make decisions about their designs; only very few decisions were forced upon them by properties of their programming language.

## 5.3 Suggestions for Future Research

It is worthwhile stressing the fact that the particular application chosen for this experiment is specified very accurately and unanimously in many aspects, especially with respect to algorithms. Yet, many differences have been found between the six programs produced in the course of this experiment.

The first question now is, are these differences already sufficient for reliable system operation, i.e., maybe the observed restrictions in the diversity of the computation sequence (see the preceding subsection) do not really matter. If that is the case, then it would not be necessary to provide different specifications to different teams, or to take any other kind of measures to increase the probability of diverse computation sequences.

Furthermore, it is possible that this restriction of diverse computation sequences occurs only with applications similar to the one used in this experiment, where most algorithms were specified rather explicitly. In other applications, where the function of the program is stated only in a very general way, this problem might not occur. Then again, the additional effort of providing different specifications would not be necessary in these cases.

Last, but not least, it would be interesting to develop some kind of a "diversity metric" to be able to make statements about the degree of differences between two (or more) programs. One possibility is to determine a list of criteria or aspects in which programs can be different (such as the ones found in this study – in addition, standard software metrics can be used), and then rank each program on each of these criteria. For instance, one would have to determine that "Program A uses global variables to an extent of 35%, while program B uses global variables to an extent of 100%". The average difference of two programs in all criteria could then be used as a diversity indicator. However, there are several problems with this approach: first of all, not all criteria may be independent from each other. For example, if a team in this experiment decided to call the vote routines from the main program, then the test point variables had to be either global or made otherwise available to the main program. On the other hand, if they decided to call a vote routine from within the module whose output is checked by that vote routine, then it is possible to define the corresponding test point variables locally only. Second, the list of meaningful criteria depends on the application under examination, thus comparisons are only valid between programs that implement the same application problem unless some general catalog of criteria can be established. But then again, it is probably very difficult at least to get this catalog to be exhaustive, i.e., to contain all possible criteria by which two programs might be different. Lastly, not all criteria might be equally important, i.e., it might be more beneficial to system reliability if two programs differ with respect to criterion A than if they differ with respect to criterion B. Thus any research in program diversity should be done within the context of an evaluation of overall system reliability or reliability improvement, since we are not interested in software diversity per se, but only in the benefits software diversity has with respect to system dependability.

## REFERENCES

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

[Aviz85b] 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.

[Aviz87a] A. Avižienis, M. R. Lyu, and W. Schütz, "In Search of Effective Diversity: A Six-Language Study of Fault-Tolerant Flight Control Software," UCLA Computer Science Department, Los Angeles, CA, Tech. Rep. CSD-870060, November 1987.

[Aviz87b] A. Avižienis, M. R. Lyu, W. Schuetz, and J. J. Chen, "Multi-Version Software Development and Processing: A UCLA/Honeywell Joint Project for Flight Control System Design," Technical Report, UCLA Computer Science Department, Los Angeles, CA, December 1987.

[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.

[Lyu87] M. R. Lyu, W. Schuetz, and J. J. Chen, "FCS Design Utilizing N-Version Processing: Software Requirements Document for a Flight Control Computer," UCLA Computer Science Department, Internal Document, Version 1.3, Los Angeles, CA, September 4, 1987.

[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.

# APPENDIX A

# SYSTEM SPECIFICATION DIAGRAMS

This appendix contains reproductions of the diagrams that were used to specify the system functions in the Software Specification Document.

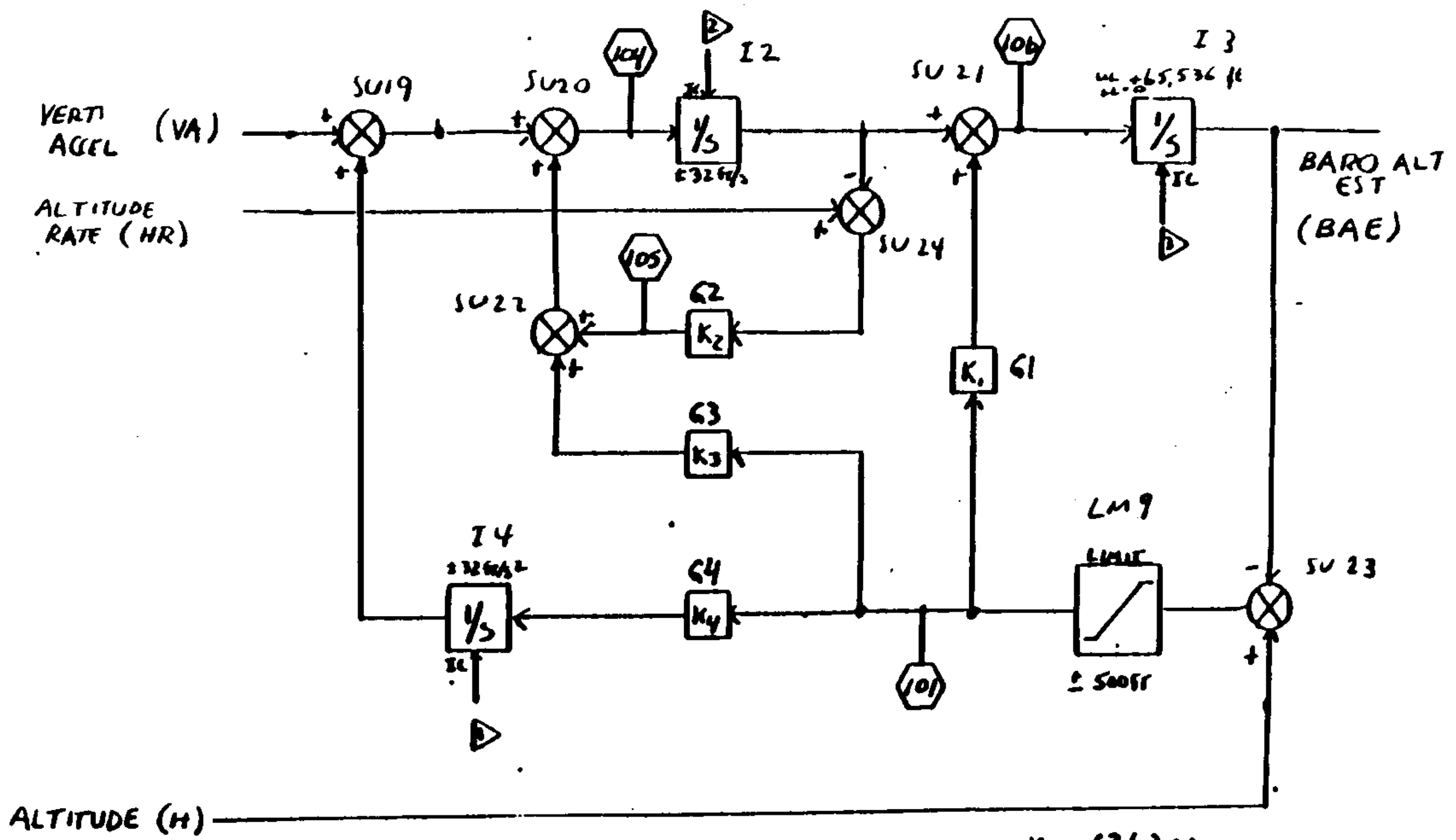


FIGURE 3.1

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)

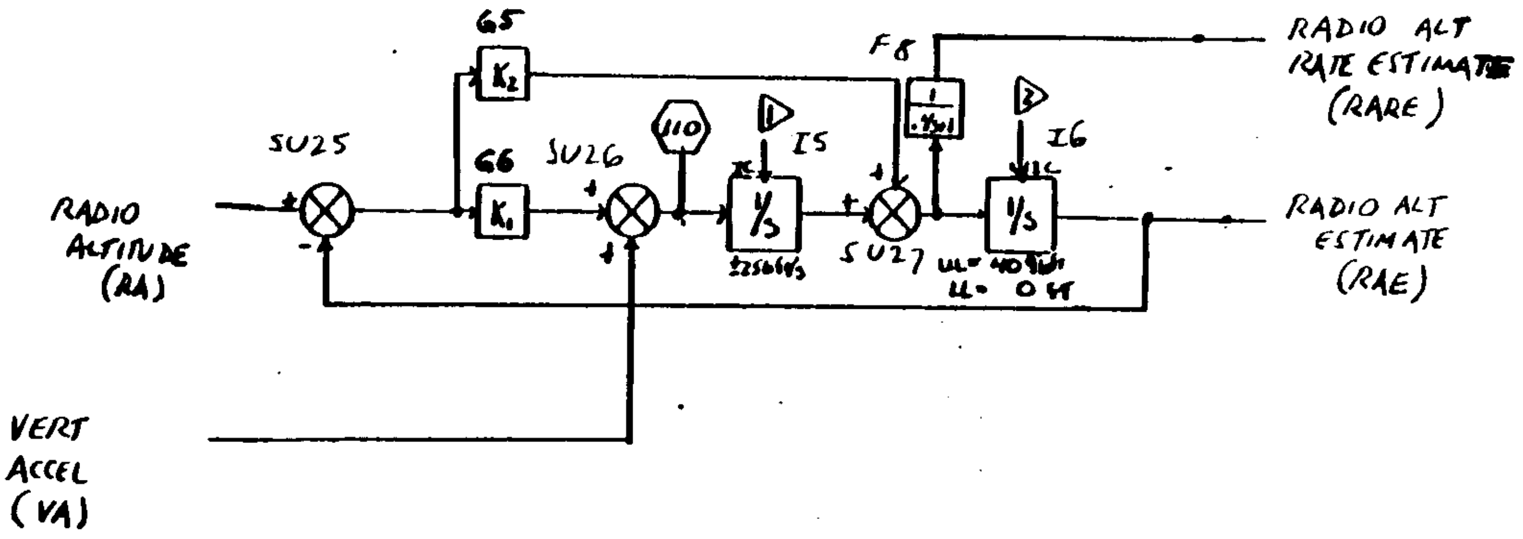
$K_1 = (3/2)\,\omega$

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

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

$K_4 = \omega^3$

$\omega = .1$ RAD/SEC

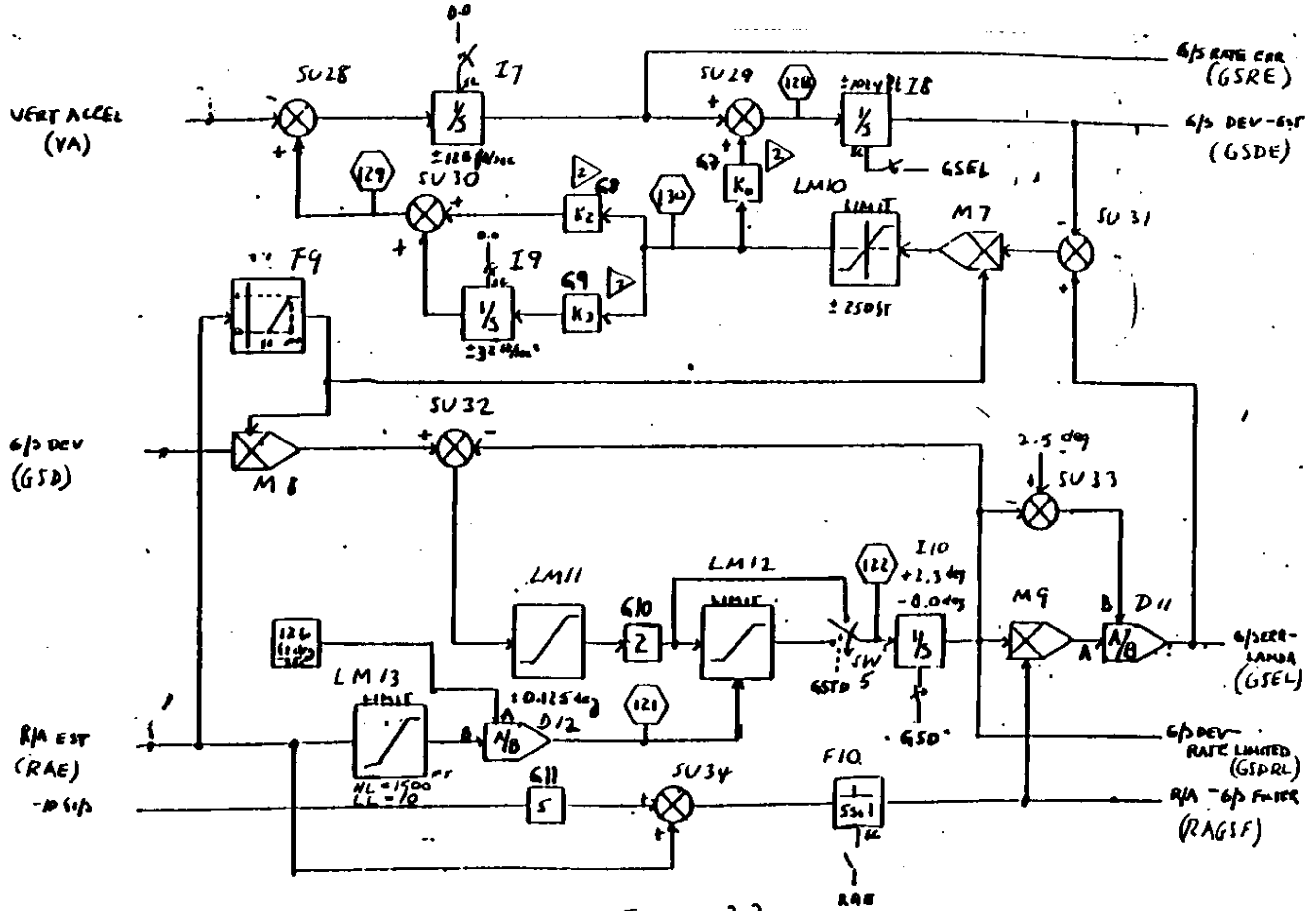


$K_1 = .25$
$K_2 = 1.00$

1 IC TO ∅.∅ @ MODV

2 IC TO $h_R$ @ MODV ($h_R$ = RA)

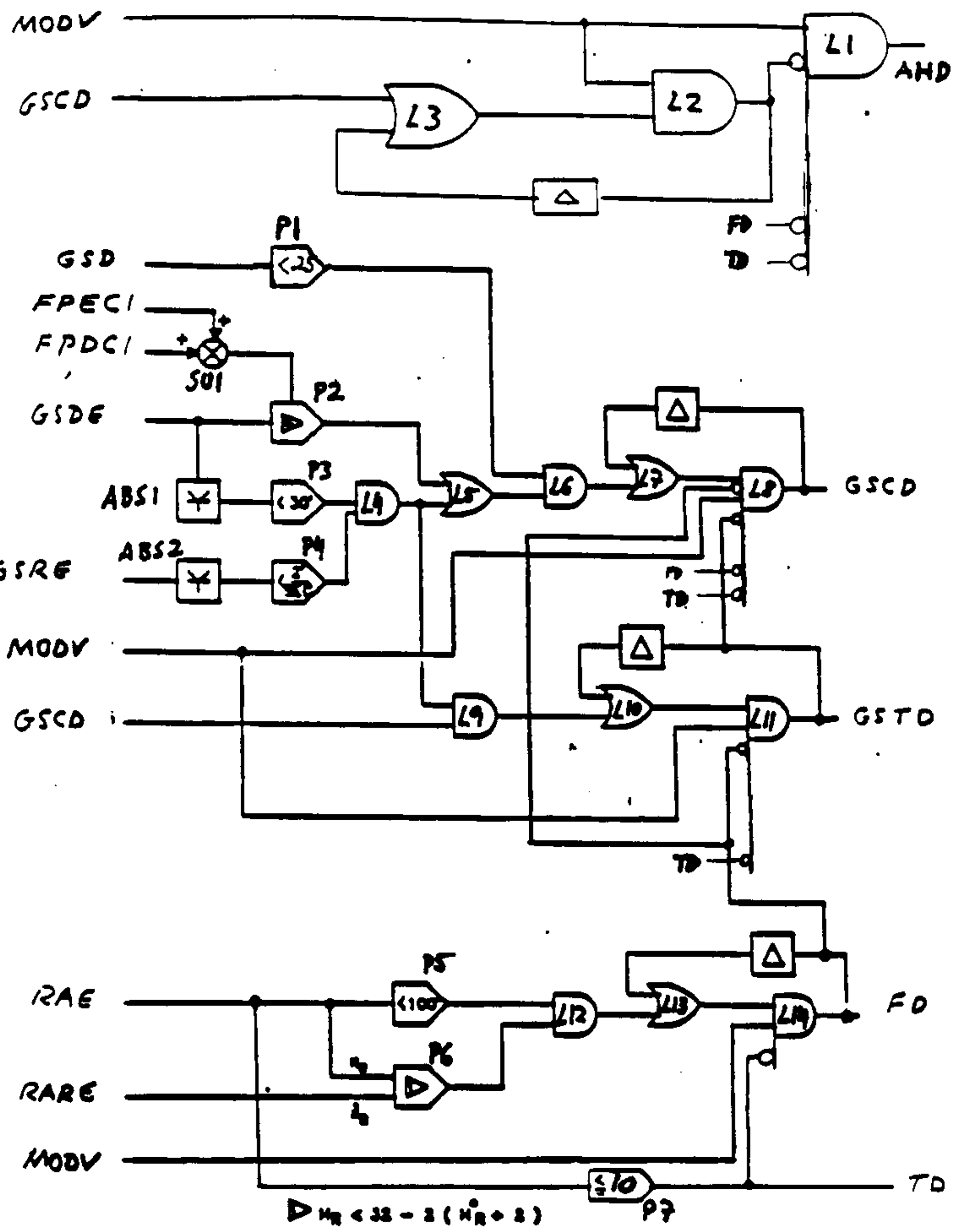
FIGURE 3.2
RADIO ALTITUDE
COMPLEMENTARY FILTER



Note 1 - All IC occur at MODV

⚠ See page 13

FIGURE 3.3
Glideslope Deviation Complementary Filter



▶ $H_R < 32 - 2(\dot{H}_R + 2)$

▶ 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

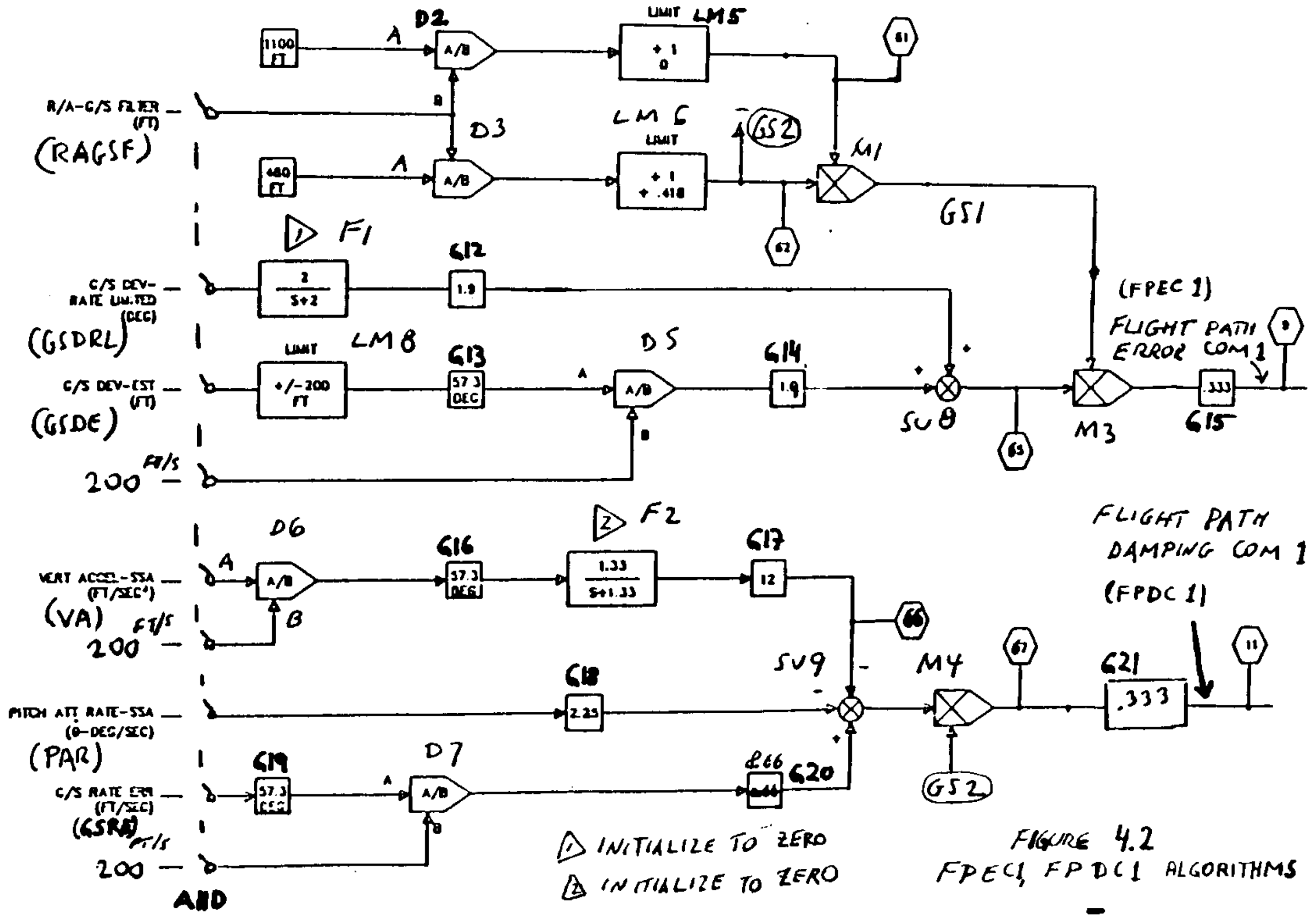
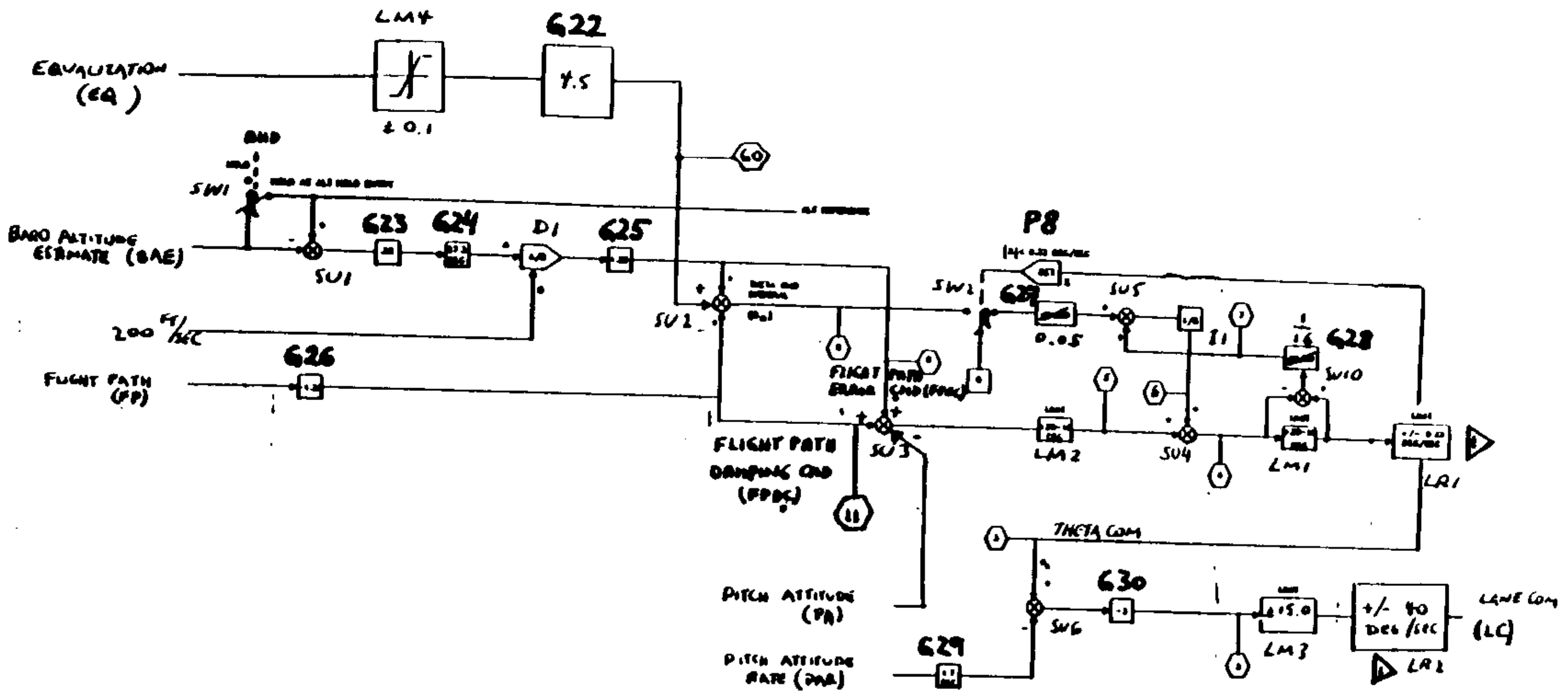


FIGURE 4.2
FPEC1, FPDC1 ALGORITHMS



▷ INITIALIZE TO INPUT

⬡ TEST POINT

FIGURE 5.1
ALT HOLD CONTROL LAW

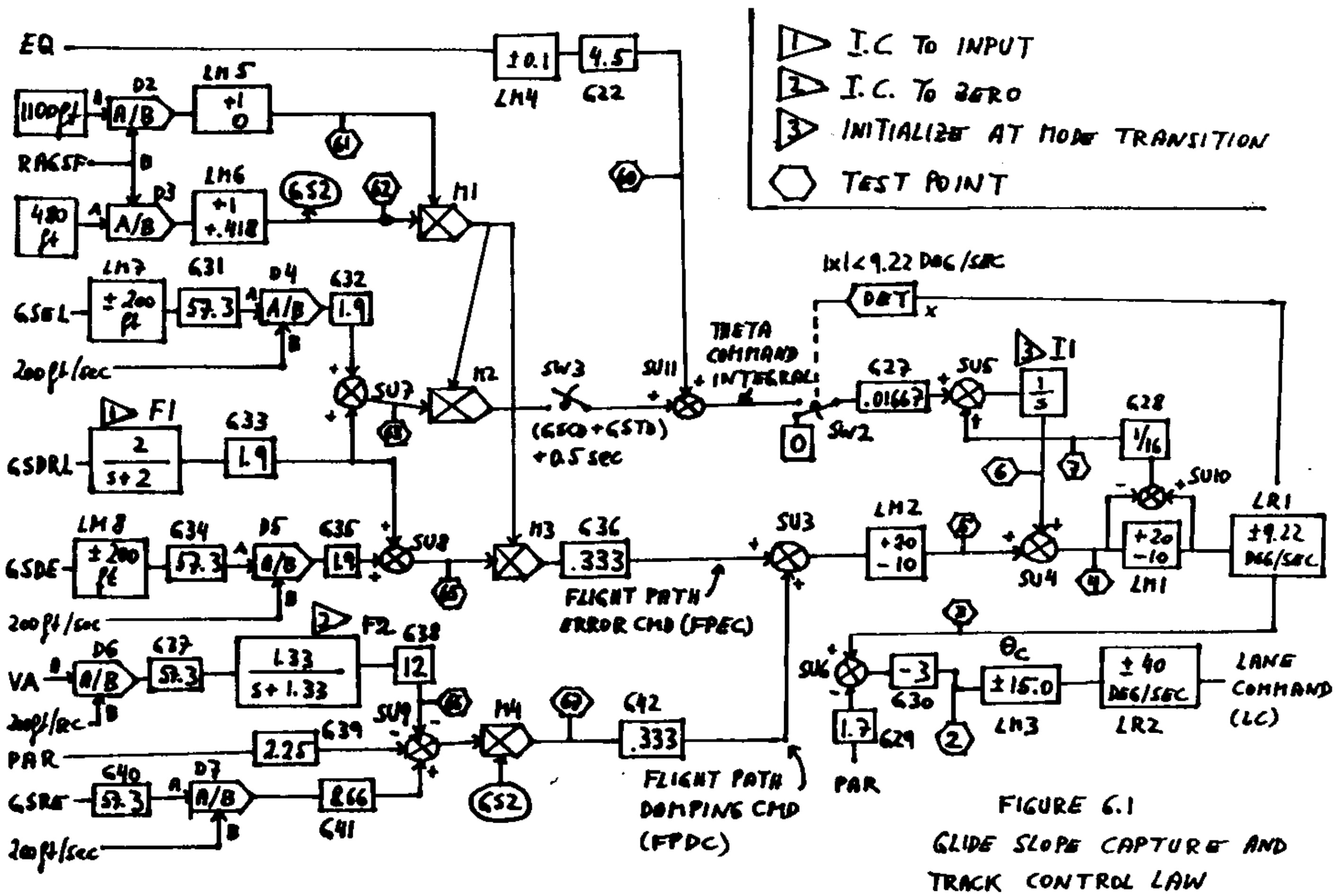


FIGURE 6.1
GLIDE SLOPE CAPTURE AND
TRACK CONTROL LAW

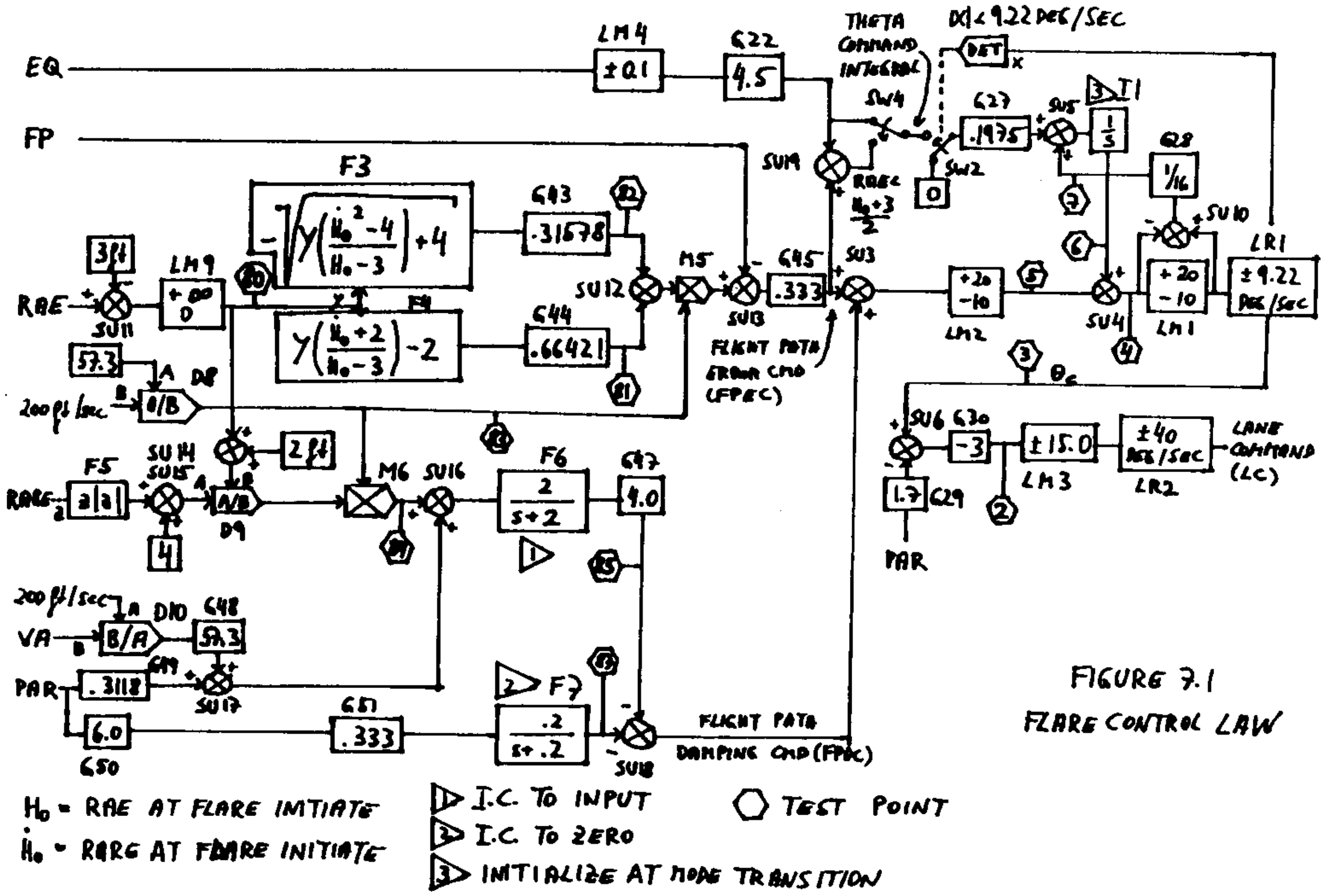


FIGURE 7.1
FLARE CONTROL LAW

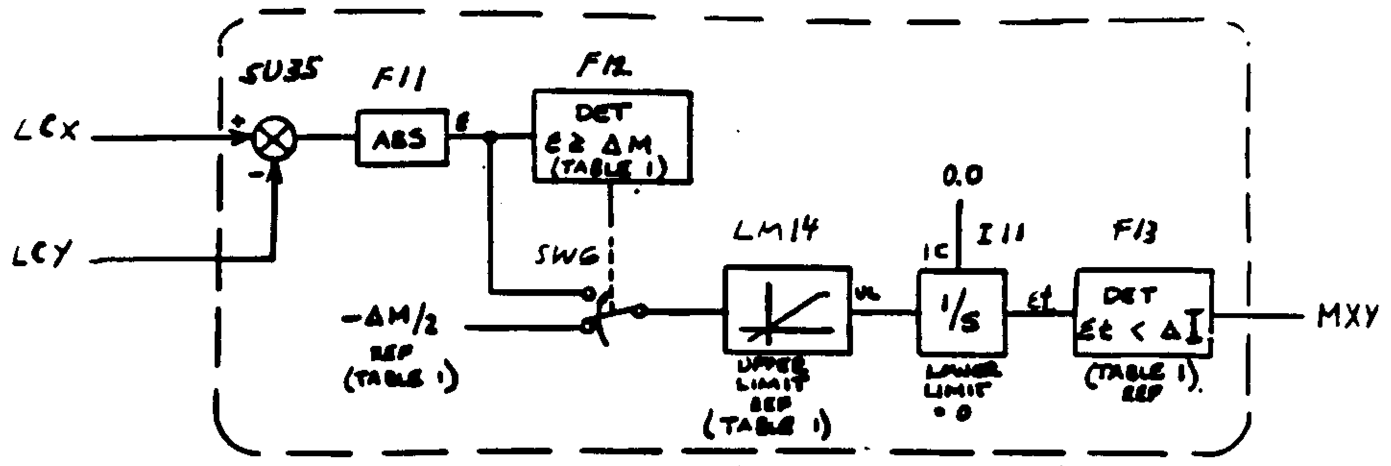


TABLE 1

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

FIGURE 8.1

COMMAND MONITOR