NASA/TM-2008-215551

Guidance and Control Software Project Data

Volume 2: Development Documents

Edited by
Kelly J. Hayhurst
Langley Research Center, Hampton, Virginia

December 2008

The NASA STI Program Office . . . in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA Scientific and Technical Information (STI) Program Office plays a key part in helping NASA maintain this important role.

The NASA STI Program Office is operated by Langley Research Center, the lead center for NASA's scientific and technical information. The NASA STI Program Office provides access to the NASA STI Database, the largest collection of aeronautical and space science STI in the world. The Program Office is also NASA's institutional mechanism for disseminating the results of its research and development activities. These results are published by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers, but having less stringent limitations on manuscript length and extent of graphic presentations.

- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.

- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.

- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services that complement the STI Program Office's diverse offerings include creating custom thesauri, building customized databases, organizing and publishing research results ... even providing videos.

For more information about the NASA STI Program Office, see the following:

- Access the NASA STI Program Home Page at ***http://www.sti.nasa.gov***

- E-mail your question via the Internet to help@sti.nasa.gov

- Fax your question to the NASA STI Help Desk at (301) 621-0134

- Phone the NASA STI Help Desk at (301) 621-0390

- Write to:
  NASA STI Help Desk
  NASA Center for AeroSpace Information
  7115 Standard Drive
  Hanover, MD 21076-1320

NASA/TM-2008-215551

Guidance and Control Software Project Data

Volume 2: Development Documents

Edited by
Kelly J. Hayhurst
Langley Research Center, Hampton, Virginia

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

December 2008

Available from:

NASA Center for AeroSpace Information (CASI)
7115 Standard Drive
Hanover, MD 21076-1320
(301) 621-0390

National Technical Information Service (NTIS)
5285 Port Royal Road
Springfield, VA 22161-2171
(703) 605-6000

Table of Contents

1 INTRODUCTION AND BACKGROUND ON SOFTWARE ERROR STUDIES 1

2 GUIDANCE AND CONTROL SOFTWARE APPLICATION 3

3 SOFTWARE LIFE CYCLE PROCESSES AND DOCUMENTATION 5

4 ROLE IN TRAINING 7

5 SUMMARY 7

6 REFERENCES 8

APPENDIX A: GUIDANCE AND CONTROL SOFTWARE DEVELOPMENT SPECIFICATION A-1

A.1 INTRODUCTION A-7

A.2 LEVELS 0 AND 1 SPECIFICATION A-24

A.3 LEVEL 2 SPECIFICATION A-33

A.4 LEVEL 3 FLOW DIAGRAMS AND C-SPECS A-37

A.5 P-SPECS FOR LEVELS 3 AND 4 .... A-44

AECLP -- AXIAL ENGINE CONTROL LAW PROCESSING (P-SPEC 2.3.1) A-44
ARSP -- ALTIMETER RADAR SENSOR PROCESSING (P-SPEC 2.1.2) A-50
ASP -- ACCELEROMETER SENSOR PROCESSING (P-SPEC 2.1.1 .... A-52
CP -- COMMUNICATIONS PROCESSING (P-SPEC 2.4) A-56
CRCP -- CHUTE RELEASE CONTROL PROCESSING (P-SPEC 2.3.3) A-61
GP -- GUIDANCE PROCESSING (P-SPEC 2.2) A-62
GSP -- GYROSCOPE SENSOR PROCESSING (P-SPEC 2.1.4) A-70
RECLP -- ROLL ENGINE CONTROL LAW PROCESSING (P-SPEC 2.3.2) A-72
TDLRSP -- TOUCH DOWN LANDING RADAR SENSOR PROCESSING (P-SPEC 2.1.3) A-74
TDSP -- TOUCH DOWN SENSOR PROCESSING (P-SPEC 2.1.6) A-78
TSP -- TEMPERATURE SENSOR PROCESSING (P-SPEC 2.1.5) A-79

A.6 DATA REQUIREMENTS DICTIONARY A-82

PART I. DATA ELEMENT DESCRIPTIONS A-82
PART II. CONTENTS OF DATA STORES A-96
PART III. CONTROL SIGNALS, DATA CONDITIONS, AND GROUP FLOWS A-100

A.7 BIBLIOGRAPHY A-117

APPENDIX B: DESIGN DESCRIPTION FOR THE PLUTO IMPLEMENTATION OF THE GUIDANCE AND CONTROL SOFTWARE B-1

B.1 INTRODUCTION TO PLUTO GCS DESIGN B-3

B.1.1 TOP-LEVEL DESCRIPTION B-3
B.1.2 DESIGN METHODOLOGY B-3
B.1.3 DESIGN SYNTAX SPECIFICATIONS B-4

B.2 DESIGN STRUCTURE B-4

B.2.1 HIGH-LEVEL SOFTWARE DESIGN B-4
B.2.2 DATA AND CONTROL FLOW B-6
B.2.3 MODULE DESCRIPTION B-6
B.2.4 PROCESS SCHEDULING B-16
B.2.5 DATA DICTIONARY B-16
B.2.6 DERIVED REQUIREMENTS B-16

B.3 REFERENCES B-16

B.4—TEAMWORK DESIGN B-17

APPENDIX C: SOURCE CODE FOR THE PLUTO IMPLEMENTATION OF THE GUIDANCE AND CONTROL SOFTWARE C-1

Abstract

The Guidance and Control Software (GCS) project was the last in a series of software reliability studies conducted at Langley Research Center between 1977 and 1994. The technical results of the GCS project were recorded after the experiment was completed. Some of the support documentation produced as part of the experiment, however, is serving an unexpected role far beyond its original project context. Some of the software used as part of the GCS project was developed to conform to the RTCA/DO-178B software standard, "Software Considerations in Airborne Systems and Equipment Certification," used in the civil aviation industry. That standard requires extensive documentation throughout the software development life cycle, including plans, software requirements, design and source code, verification cases and results, and configuration management and quality control data. The project documentation that includes this information is open for public scrutiny without the legal or safety implications associated with comparable data from an avionics manufacturer. This public availability has afforded an opportunity to use the GCS project documents for DO-178B training. This report provides a brief overview of the GCS project, describes the 4-volume set of documents and the role they are playing in training, and includes the development documents from the GCS project.

1 Introduction and Background on Software Error Studies

As the pervasiveness of computer systems has increased, so has the desire and obligation to establish the reliability of these systems. Reliability estimation and prediction are standard activities in many engineering projects. For the software aspects of computer systems, however, reliability estimation and prediction have been topics of dispute, especially for safety-critical systems. A primary challenge is how to accurately model the failure behavior of software such that numerical estimates of reliability have sufficient credibility for systems where the probability of failure needs to be quite small, such as in commercial avionics systems (ref. 1). A second challenge is how to gather sufficient data to make such estimates. Software reliability models are not used in the civil aviation industry, for example, because "currently available methods do not provide results in which confidence can be placed to the level required for this purpose." (ref. 2)

In an effort to develop methods to credibly assess the reliability of software for safety-critical avionics applications, Langley Research Center initiated a Software Error Studies program in 1977 (ref. 3). A major focus of those studies was on generating significant quantities of software failure data through controlled experimentation to better understand software failure processes. The intent of the Software Error Studies program was to incrementally increase complexity and realism in a series of experiments so that the final study would have statistically valid results, representative of actual software development processes.

The Software Error Studies program started with initial investigations by the Aerospace Corporation to define software reliability measures and data collection requirements (ref. 4-6).

Next, Boeing Computer Services (BCS) and the Research Triangle Institute (RTI) conducted several simple software experiments with aerospace applications including missile tracking, launch interception, spline function interpolation, Earth satellite calculation, and pitch axis control (refs. 7-11). The experiment design used in these studies generally involved a number of programmers (denoted *n*) who independently generated computer code from a given specification of the problem to produce *n* versions of a program. In these experiments, no particular software development standards or life-cycle models were followed. Because the problems were relatively small and simple, the versions were compared to a known error-free version of the program to obtain information on software errors.

Although the initial experiments were small and simplistic compared with real-world avionics development, they yielded some interesting results that have influenced software reliability modeling. The BCS and RTI studies showed widely varying error rates for faults. This finding refuted a common assumption in early software reliability growth models that faults produced errors at equal rates. These studies also provided evidence of fault interaction where one fault could mask potentially erroneous behavior from another fault, or where two or more faults together cause errors when alone they would not. (ref. 12) Additional investigations with *n*-version programs (ref. 13) found that points in the input space that cause an error can cluster and form "error crystals". Extrapolating this finding to aerospace applications, where input signals tend to be continuous in nature, the error crystals may manifest themselves as clusters of successive faults that could have unintended consequences. (ref. 14)

The last project in the Software Error Studies program was the Guidance and Control Software (GCS) project. It built on the previous experiments in two ways: (1) by requiring that the software specimens for the experiment be developed in compliance with current software development standards, and (2) by increasing the complexity of the application problem (ref. 15). At the time of the GCS project, the RTCA/DO-178B guidelines, "Software Considerations in Airborne Systems and Equipment Certification," (ref. 2) were the primary standard sanctioned by the Federal Aviation Administration (FAA) for developing software to be approved for use in commercial aircraft equipment (ref. 16). The DO-178B document describes objectives and design considerations to be used for the development of software as well as verification, configuration management, and quality assurance activities to be performed throughout the development process. The DO-178B guidelines were selected as the software development standard to be used for the GCS specimens.

The software application selected for the GCS project, as the title indicates, is a guidance and control function for controlling the terminal descent trajectory of a planetary lander vehicle. This terminal descent trajectory is the same fundamental trajectory referred to as the "seven minutes of terror" in the entry, descent, and landing phase of a planetary mission, such as the recent Phoenix Mars Lander (ref. 17). For the GCS project, the software requirements were reverse engineered from a simulation program used to study the probability of success of the original NASA Viking Lander mission to Mars in the 1970s (ref. 18). It is important to emphasize that the software requirements documented for the GCS project, while realistic, are not the actual software requirements used for NASA's Viking Lander or any other planetary landers.

For the GCS experiment, two[1] teams of software engineers were each tasked to independently design, code, and verify a GCS program, following the software development guidance in DO-178B, as closely as possible. In addition to those teams, another GCS version was produced, without the constraint of compliance with DO-178B, to aid development and verification of the requirements and simulation environment. Once all versions were complete, data on residual

[1] The original plan for the GCS project called for three independent teams. Due to funding constraints, only two teams were able to complete the project.

errors was supposed to be collected by running all the versions simultaneously in a simulation environment, and using any discrepancies among the results of the versions as possible indications of errors.

Results of the operational simulations and data collection are described in (ref. 15). The purpose of this report is not to repeat those results, but to disseminate some of the project documentation that has an unanticipated utility beyond its original project context. The project documentation of interest is the documentation developed by the teams required to comply with the DO-178B standard. That standard requires extensive records of all of the software development life cycle activities. For the GCS project, those records included 18 documents consisting of life cycle plans, development products including requirements and source code, verification cases and results, and configuration management and quality control data. Comparable data from a commercial avionics system would not be available for public review because of proprietary and other legal considerations. The GCS project documentation is not subject to those considerations because it is not data from an actual operational, or even prototype, system. But, the data has sufficient realism to provide a window into the types of activities and data involved in the production of DO-178 compliant software, which makes the GCS documentation desirable from a training perspective.

The remainder of this report provides a brief overview of aspects of the GCS project relevant to using the documentation for training. This information includes a description of the GCS application, a synopsis of the software development processes used to follow the DO-178B guidance, and the data that was generated as a result. Because the complete set of compliance documents is large, the documents have been divided into four sets (planning, development, verification, and other integral process documents) contained in separate volumes of this report. Volume 2 includes in Appendices A-C the requirements, design, and source code generated as part of the development processes.

2 Guidance and Control Software Application

The requirements for the GCS application focus on two primary functions: (1) to provide guidance and engine control of the lander vehicle during its terminal phase of descent onto the planet's surface, and (2) to communicate sensory information to an orbiting platform about the vehicle and its descent. Figure 1 shows a sketch of the lander vehicle, taken from (ref. 18), noting the location of the terminal descent propulsion systems.

The guidance package for the lander vehicle contains sensors that obtain information about the vehicle state and environment, a guidance and control computer, and actuators providing the thrust necessary for maintaining a safe descent. The vehicle has three accelerometers (one for each body axis), one Doppler radar with four beams, one altimeter radar, two temperature sensors, three strapped-down gyroscopes, three opposed pairs of roll engines, three axial thrust engines, one parachute release actuator, and a touch down sensor. The vehicle has a hexagonal, box-like shape; three legs and a surface sensing rod protrude from its undersurface.

In general, the requirements for the planetary lander only concern the final descent to the surface. Figure 2 shows a sketch of the phases of the terminal descent trajectory.

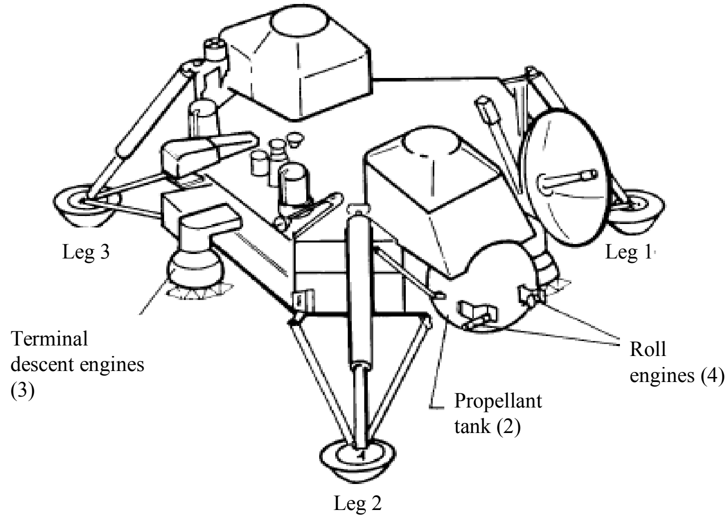


Figure 1. Lander with Terminal Descent Propulsion Systems

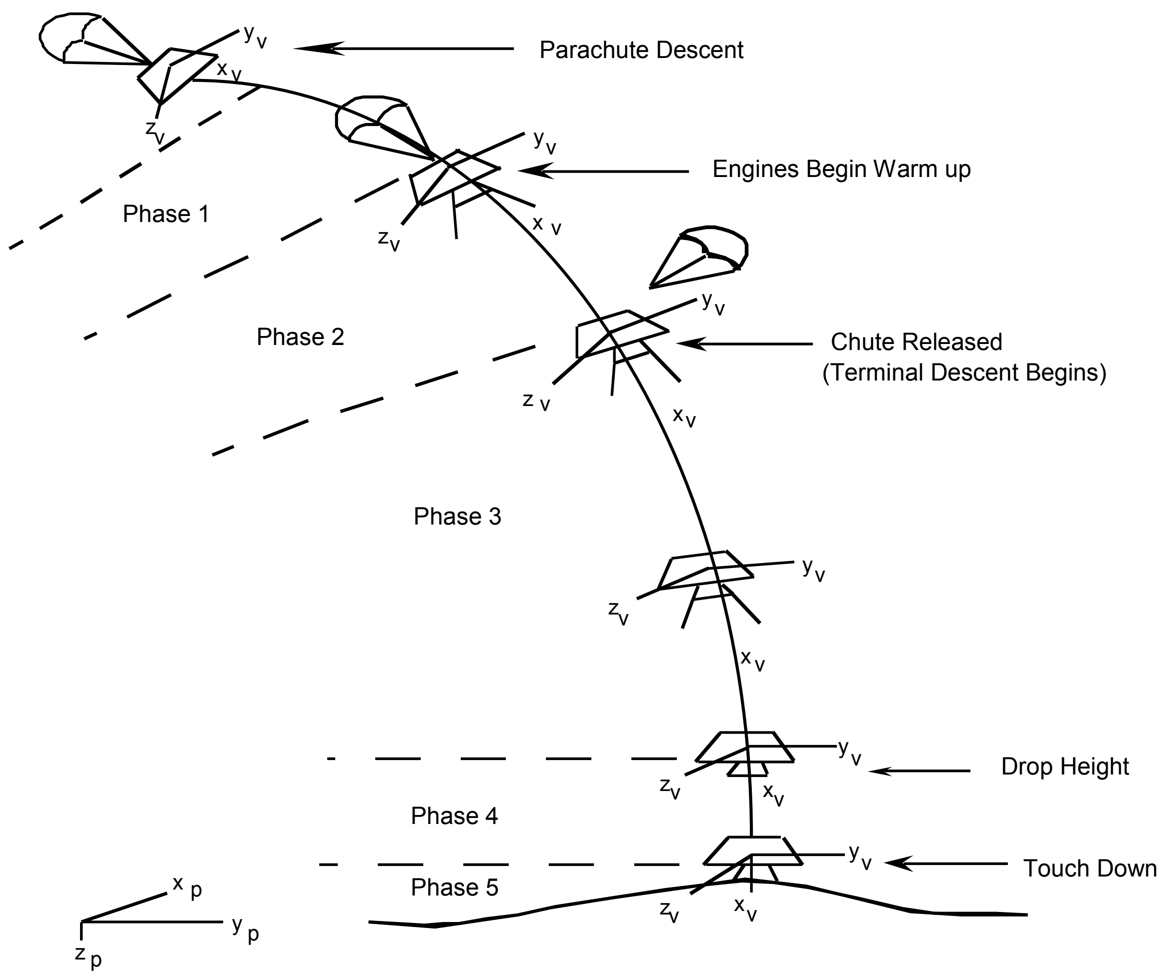


Figure 2. A Typical Terminal Descent Trajectory

After the lander has dropped from orbit, the software controls the engines of the vehicle to the surface of a planet. The initialization of the GCS starts the sensing of vehicle altitude. When a predefined engine ignition altitude is sensed by the altimeter radar, the GCS begins guidance and control of the lander. The axial and roll engines are ignited; while the axial engines are warming up, the parachute remains connected to the vehicle. During this engine warm-up phase, the aerodynamics of the parachute dictate the vehicle's trajectory. Vehicle attitude is maintained by firing the engines in a throttled-down condition. Once the main engines become hot, the parachute is released and the GCS performs an attitude correction maneuver and then follows a controlled acceleration descent until a predetermined velocity-altitude contour is crossed. The GCS then attempts to maintain the descent of the lander along this predetermined velocity-altitude contour. The lander descends along this contour until a predefined engine shut off altitude is reached or touchdown is sensed. After all engines are shut off, the lander free-falls to the surface.

The software requirements for this guidance and control application are contained in a document called the *Guidance and Control Development Specification* (in Volume 2). As mentioned earlier, the initial requirements for this application were reverse engineered from a simulation program used to study the probability of success of the original NASA Viking Lander mission to Mars. Prior to use in the experiment, the requirements were revised to make them suitable for use in an *n*-version software experiment. Each of the GCS programs for the experiment were developed from the same requirements document.

3 Software Life Cycle Processes and Documentation

Having some of the project teams adhere to the DO-178B guidelines as they created a software version for the experiment was a significant element of the GCS project, requiring the development and tracking of numerous software engineering artifacts not normally associated with a software engineering experiment. The purpose of DO-178B is to provide guidelines for the production of software such that the completed implementation performs its intended function with a level of confidence in safety satisfactory for airworthiness. Along with the production of software is the generation of an extensive set of documents recording the production activities.

DO-178B defines software development activities and objectives for the development life cycle of the software, and the evidence that is needed to show compliance. The life-cycle processes are divided into planning, development, and integral processes. The planning process defines and coordinates the software development processes and the integral processes. The software development processes involve identification of software requirements, software design and coding, and integration; that is, the development processes directly result in the software product. Finally, the integral processes function throughout the software development processes to ensure integrity of the software products. The integral processes include software verification, configuration management, and quality assurance processes. Section 11 of DO-178B describes data that should be produced as evidence of performing all of the life cycle process activities.

For the GCS project, some of this data was common for all of the teams, and other data was intended to be specific to each team. For example, each team worked with the same plans, standards, and requirements. Then, each individual team was responsible for independently developing their own design, code, and corresponding verification data. To distinguish the versions, each team was assigned a planetary name: Mercury, Venus, and Pluto[2].

[2] At the time the GCS experiment was conducted, Pluto had not yet been relegated to non-planet status.

Table 1. Life Cycle Data

Planning Process Documents	Development Process Documents	Integral Process Documents
• **Plan for Software Aspects of Certification**	• **Software Requirements Data**	• **Software Verification Cases and Procedures**
• **Software Development Plan**	• **Design Description**	• **Software Verification Results**
• **Software Verification Plan**	• **Source Code**	• **Software Life Cycle Environment Configuration Index**
• **Software Configuration Management Plan**	• **Executable Object Code**	• **Software Configuration Index**
• **Software Quality Assurance Plan**		• **Problem Reports**
• **Software Requirements Standards**		• **Software Configuration Management Records**
• **Software Design Standards**		• **Software Quality Assurance Records**
• **Software Code Standards**		• **Software Accomplishment Summary**

The DO-178B data associated with the development of the Pluto version of the GCS was selected for publication. For dissemination purposes, the Pluto data was divided into the following 4 subsets:

Volume 1: Planning Documents

- *Plan for Software Aspects of Certification of the Guidance and Control Software Project*
- *Software Configuration Management Plan for the Guidance and Control Software Project*
- *Software Quality Assurance Plan for the Guidance and Control Software Project*
- *Software Verification Plan for the Guidance and Control Software Project*
- *Software Development Standards for the Guidance and Control Software Project*

Volume 2: Development Documents

- *Guidance and Control Software Development Specification*
- *Design Description for the Pluto Implementation of the Guidance and Control Software*
- *Source Code for the Pluto Implementation of the Guidance and Control Software*

Volume 3: Verification Documents

- *Software Verification Cases and Procedures for the Guidance and Control Software Project*
- *Software Verification Results for the Pluto Implementation of GCS*
- *Review Records for the Pluto Implementation of the Guidance and Control Software*
- *Test Results Logs for the Pluto Implementation of the Guidance and Control Software*

Volume 4: Other Integral Processes Documents

- *Software Accomplishment Summary for the Guidance and Control Software Project*
- *Software Configuration Index for the Guidance and Control Software Project*
- *Problem Reports for the Pluto Implementation of the Guidance and Control Software*
- *Support Documentation Change Reports for the Guidance and Control Software Project*
- *Configuration Management Records for the Guidance and Control Software Project*
- *Software Quality Assurance Records for the Guidance and Control Software Project*

Appendices A-C contain the original development documents for the GCS project. The *Guidance and Control Software Development Specification*, in Appendix A, contains all of the high-level requirements for the guidance and control application, as well as instructions for interfacing with the experiment's simulator.. The low-level requirements and architecture are contained in the *Design Description for the Pluto Implementation of the Guidance and Control Software* in Appendix B. The design was developed using a structured analysis tool called Team*work* from Cadre Technologies. Finally, Appendix C contains the *Source Code for the Pluto Implementation of the Guidance and Control Software* with the Fortran source code for the Pluto implementation.

The content of the documents in the appendices has not been altered from the original versions produced during the project.

4 Role in Training

At the time of the GCS project, there was no publicly available information, such as templates, or examples, or training courses, to help a novice developer generate the type of evidence that a certificating authority would expect to see to demonstrate compliance with DO-178B. As mentioned earlier, compliance data from a real avionics system is not typically available for public review because of various legal and safety considerations. For example, an avionics manufacturer would likely consider the design and implementation of a system to be proprietary. Those considerations do not apply to the data from the GCS project, because neither the requirements nor the software versions represent an actual system with safety, liability, or other considerations.

In addition to the availability of data, the GCS requirements and DO-178B compliance data are sufficiently realistic to serve as an example of a DO-178B project: one that is small enough in scale to be studied in a training course. The GCS documentation provides a window into the activities and data produced throughout the development life cycle to comply with DO-178B. Because the Federal Aviation Administration (FAA) was aware of the GCS project, they recognized the potential value of the documentation for training. The FAA has designed software training to include a case study portion that addresses avionics software issues that arise from the application of the DO-178B guidelines. The case study gives students the opportunity to use auditing techniques to identify flaws in lifecycle data. Because the GCS data was produced by novices, there are plenty of flaws to find.

5 Summary

From 1977-1994, NASA Langley Research Center conducted a Software Error Studies program that generated data that provided insights into the software failure process and into conducting software engineering experiments as well. The GCS project was the final experiment in that program. A unique feature of the GCS project was the requirement for some of the software specimens used in the experiment to conform to the RTCA/DO-178B software standard, "Software Considerations in Airborne Systems and Equipment Certification," used in the civil aviation industry. The project documentation produced to meet that requirement has had the unanticipated benefit of serving as case study material in software certification training long after the conclusion of the original experiment. Volume 2 of this report contains all of the development artifacts from the GCS project.: requirements, design, and source code Other volumes of this report contain the rest of the GCS compliance data including planning, verification, and configuration management and quality assurance documents.

6 References

1. Littlewood, Bev, and Strigini, Lorenzo, Software Reliability and Dependability: a Roadmap, 22nd International Conference on Software Engineering, Future of Software Engineering Track, June 4-11, 2000, Limerick Ireland, pp. 175 – 188.
2. Software Considerations in Airborne Systems and Equipment Certification. Doc. No. RTCA/DO-178B, RTCA, Inc., Dec. 1, 1992.
3. Finelli, George B.: NASA Software Failure Characterization Experiments. Reliability Engineering & System Safety, vol. 32, pp. 155–169, 1991.
4. Hecht, H.; Sturm, W. A.; and Tratlner, S.: Reliability Measurement During Software Development. NASA CR-145205, 1977.
5. Hecht, H.: Measurement Estimation and Prediction of Software Reliability. NASA CR-145135, 1977.
6. Maxwell, F. D.: The Determination of Measures of Software Reliability. NASA CR-158960, 1978.
7. Nagel, Phyllis M.; and Skrivan, James A.: Software Reliability: Repetitive Run Experimentation and Modeling. NASA CR-165836, 1982.
8. Nagel, P. M.; Scholz, F. W.; and Skrivan, J. A.: Software Reliability: Additional Investigation Into Modeling With Replicated Experiments. NASA CR-172378, 1984.
9. Dunham, Janet R.: Experiments in Software Reliability: Life-Critical Applications. IEEE Transactions on Software Engineering, vol. SE-12, no. 1, Jan. 1986, pp. 110–123.
10. Dunham, J. R.; and Lauterbach, L. A.: An Experiment in Software Reliability Additional Analyses Using Data From Automated Replications. NASA CR-178395, 1987.
11. Dunham, Janet R.; and Pierce, John L.: An Empirical Study of Flight Control Software Reliability. NASA CR-178058, 1986.
12. Dunham, Janet R.; and Finelli, George B., Real-Time Software Failure Characterization, IEE Aerospace and Electronic Systems Magazine, pp. 38-44, November 1990.
13. Ammann, P. and Knight, J.: "Data Diversity: An Approach To Software Fault Tolerance", Digest of Papers FTCS-17: The 17th Annual International Symposium on Fault Tolerant Computing, Pittsburg, Pennsylvania, July 1987.
14. Finelli, George B, Results of Software Error-Data Experiments, AIAA/AHS/ASEE Aircraft Design, Systems and Operations Conference, September 7-9, 1988, Atlanta, Georgia, AIAA-88-4436.

15. Hayhurst, Kelly J., Framework for Small-Scale Experiments in Software Engineering, Guidance and Control Software Project: Software Engineering Case Study, NASA/TM-1998-207666, May 1998.
16. Federal Aviation Administration, Advisory Circular, 20-115B, January 11, 1993.
17. Tobin, Kate, NASA Preps for '7 Minutes of Terror' on Mars, May 23, 2008, http://www.cnn.com/2008/TECH/space/05/23/mars.lander/index.html.
18. Holmberg, Neil A.; Faust, Robert P.; and Holt, H. Milton: Viking '75 Spacecraft Design and Test Summary. Volume I—Lander Design. NASA RP-1027, 1980.

Appendix A: Guidance and Control Software Development Specification

Version 2.3 with Formal Modifications 2.3-1 through 2.3-7

Authors: B. Edward Withers, Research Triangle Institute
Bernice Becher, Lockheed Engineering & Sciences Corp.

This document was produced as part of Guidance and Control Software (GCS) Project conducted at NASA Langley Research Center. Although some of the requirements for the Guidance and Control Software application were derived from the NASA Viking Mission to Mars, this document does not contain data from an actual NASA mission.

A. ACKNOWLEDGEMENTS

We wish to acknowledge Peter Padilla for his guidance and assistance in reviewing the requirements for this document and in recommending revisions so that the requirements might more accurately reflect those of an actual lander. We wish to acknowledge Earle Migneault for his work on earlier experiments of this type and for his vision in suggesting the application to be used in this experiment. We also wish to acknowledge Don C. Rich, Douglas S. Lowman, R. C. Buckland, Anita M. Shagnea, and Janet R. Dunham, all of Research Triangle Institute, for their earlier work on this project.

A. FOREWORD

This specification defines a guidance and control system for a planetary landing vehicle during its terminal phase of descent. It is written for an experienced programmer with two or more years of full-time industrial programming experience using a scientific programming language. The programmer should have an adequate background, either through college courses or job training in mathematics, physics, differential equations, and numerical integration. The specification was written with the assumption that the implementation would be coded in FORTRAN; however, other languages can be used.

A. CONTENTS

A.1 INTRODUCTION A-7

A.2 LEVELS 0 AND 1 SPECIFICATION A-24

LEVEL 0 SPECIFICATION... A-24
LEVEL 1 SPECIFICATION... A-30

A.3 LEVEL 2 SPECIFICATION A-33

A.4 LEVEL 3 FLOW DIAGRAMS AND C-SPECS A-37

A.5 P-SPECS FOR LEVELS 3 AND 4 .... A-44

AECLP -- AXIAL ENGINE CONTROL LAW PROCESSING (P-SPEC 2.3.1)... A-44
ARSP -- ALTIMETER RADAR SENSOR PROCESSING (P-SPEC 2.1.2) A-50
ASP -- ACCELEROMETER SENSOR PROCESSING (P-SPEC 2.1.1 .... A-52
CP -- COMMUNICATIONS PROCESSING (P-SPEC 2.4)... A-56
CRCP -- CHUTE RELEASE CONTROL PROCESSING (P-SPEC 2.3.3)... A-61
GP -- GUIDANCE PROCESSING (P-SPEC 2.2) A-62
GSP -- GYROSCOPE SENSOR PROCESSING (P-SPEC 2.1.4) A-70
RECLP -- ROLL ENGINE CONTROL LAW PROCESSING (P-SPEC 2.3.2) A-72
TDLRSP -- TOUCH DOWN LANDING RADAR SENSOR PROCESSING (P-SPEC 2.1.3)... A-74
TDSP -- TOUCH DOWN SENSOR PROCESSING (P-SPEC 2.1.6)... A-78
TSP -- TEMPERATURE SENSOR PROCESSING (P-SPEC 2.1.5)... A-79

A.6 DATA REQUIREMENTS DICTIONARY A-82

PART I. DATA ELEMENT DESCRIPTIONS... A-82
PART II. CONTENTS OF DATA STORES... A-96
PART III. CONTROL SIGNALS, DATA CONDITIONS, AND GROUP FLOWS... A-100
A.7 NOTATION FOR LEVELS 0, 1, 2, AND 3 SPECIFICATION A-103
A.8 IMPLEMENTATION NOTES A-105
A.9 NUMERICAL INTEGRATION INSTRUCTIONS A-108
A.10 COMMUNICATIONS PACKET INSTRUCTIONS A-113

A.11 BIBLIOGRAPHY... A-117

List of Figures

A.1.1 THE LANDING VEHICLE DURING DESCENT A.8

A.1.2 A TYPICAL TERMINAL DESCENT TRAJECTORY A.9

A.1.3 ENGINEERING ILLUSTRATION OF VEHICLE A.14

A.2.1 STRUCTURE OF THE GCS SPECIFICATION A.26

A.2.2 DATA CONTEXT DIAGRAM: LANDER A.27

A.2.3 CONTROL CONTEXT DIAGRAM: LANDER A.28

A.2.4 DATA FLOW DIAGRAM (DFD) 0: GCS A.29

A.2.5 CONTROL FLOW DIAGRAM (CFD) 0: GCS A.30

A.3.1 DFD 2: RUN_GCS A.33

A.3.2 CFD 2: RUN_GCS A.34

A.4.1 DFD 2.1: SP -- SENSOR PROCESSING A.36

A.4.2 CFD 2.1: SP -- SENSOR PROCESSING A.37

A.4.3 DFD 2.3: CLP -- CONTROL LAW PROCESSING A.39

A.4.4 CFD 2.3: CLP -- CONTROL LAW PROCESSING A.40

A.5.1 VELOCITY-ALTITUDE CONTOUR A.64

A.5.2 GRAPH FOR DERIVING ROLL ENGINE COMMANDS A.71

A.5.3 DOPPLER RADAR BEAM LOCATIONS A.73

A.5.4 CALIBRATION OF THERMOCOUPLE PAIR A.78

A.7.1 GRAPHICAL SYMBOLS USED IN STRUCTURED ANALYSIS DIAGRAMS A.103

A.8.1 DIAGRAM OF STORAGE AS SEEN BY GCS IMPLEMENTATIONS A.105

List of Tables

A.1.1 ROTATION OF VARIABLES A-21

A.2.1 CONTROL SPECIFICATION (C-SPEC) 0: GCS A-31

A.3.1 C-Spec 2: RUN_GCS A-35

A.4.1 C-Spec 2.1: SP -- SENSOR PROCESSING A-38

A.4.2 C-Spec 2.3: CLP -- CONTROL LAW PROCESSING A-41

A.4.3 FUNCTIONAL UNIT SCHEDULING A-42

A.5.1 DETERMINATION OF AXIAL ENGINE TEMPERATURE A-45

A.5.2 DETERMINATION OF ERROR TERMS A-47

A.5.3 DETERMINATION OF AXIAL ENGINE COMMANDS A-47

A.5.4 DETERMINATION OF ALTITUDE STATUS A-50

A.5.5 PACKET VARIABLES A-57

A.5.6 SAMPLE MASK A-57

A.5.7 PACKET BYTE STRUCTURE A-58

A.5.8 DIFFERENTIAL EQUATIONS A-61

A.5.9 DETERMINATION OF AXIAL AND ROLL ENGINE ON/OFF SWITCHES A-62

A.5.10 DETERMINATION OF GUIDANCE PHASE A-65

A.5.11 DETERMINATION OF RADAR BEAM STATES A-74

A.5.12 PROCESSING OF DOPPLER RADAR BEAMS IN LOCK A-75

A.5.13 DETERMINATION OF TOUCH DOWN SENSOR AND STATUS A-76

A.6.1 DATA STORE: GUIDANCE_STATE A-94

A.6.2 DATA STORE: EXTERNAL A-95

A.6.3 DATA STORE: SENSOR_OUTPUT A-95

A.6.4 DATA STORE: RUN_PARAMETERS A-96

A.6.5 CONTROL SIGNALS A-97

A.6.6 DATA CONDITIONS A-98

A.6.7 INITIALIZATION DATA A-98

A.6.8 TEMP_DATA A-99

A.6.9 SENSOR_DATA A-101

A.6.10 OUTPUT_DATA A-101

A.6.11 OUTPUT_CONTROL A-101

A.6.12 FRAME_DATA A-101

A.9.1 INITIAL VALUES PROVIDED FOR USE IN INTEGRATION A-101

A.1 INTRODUCTION

PURPOSE OF THE GUIDANCE AND CONTROL SOFTWARE

The Guidance and Control Software (GCS) represents the Viking lander (ref. A.20) on-board navigational software. The purpose of this software is to:

1. provide guidance and engine control of the vehicle (shown in Figure A.1.1) during its terminal phase of descent onto a surface and
2. communicate sensory information about the vehicle and its descent to some other receiving device.

A typical descent trajectory is shown in Figure A.1.2.

The initialization of the GCS starts the sensing of vehicle altitude. When a predefined engine ignition altitude is sensed by the altimeter radar, the GCS begins guidance and control of the vehicle. The axial engines are ignited; while the axial engines are warming up, the parachute remains connected to the vehicle. During this engine warm-up phase, the aerodynamics of the parachute dictate the trajectory followed by the vehicle. Vehicle attitude is maintained by firing the engines in a throttled-down condition. Once the main engines become hot, the parachute is released and the GCS performs an attitude correction maneuver and then follows a controlled acceleration descent until a predetermined velocity-altitude contour is crossed (see Figure A.5.1). The GCS then attempts to maintain the descent of the vehicle along this predetermined velocity-altitude contour. The vehicle descends along this contour until a predefined engine shut off altitude is reached or touchdown is sensed. After all engines are shut off, the vehicle free-falls to the surface.

VEHICLE CONFIGURATION

The vehicle to be controlled is a guidance package containing sensors which obtain information about the vehicle state, a guidance and control computer, and actuators providing the thrust necessary for maintaining a safe descent. The vehicle has three accelerometers (one for each body axis), one doppler radar with four beams, one altimeter radar, two temperature sensors, three strapped-down gyroscopes, three opposed pairs of roll engines, three axial thrust engines, one parachute release actuator, and a touch down sensor. The vehicle has a hexagonal, box-like shape with three legs and a surface sensing rod protruding from its undersurface.

Figure A.1.1 THE LANDING VEHICLE DURING DESCENT

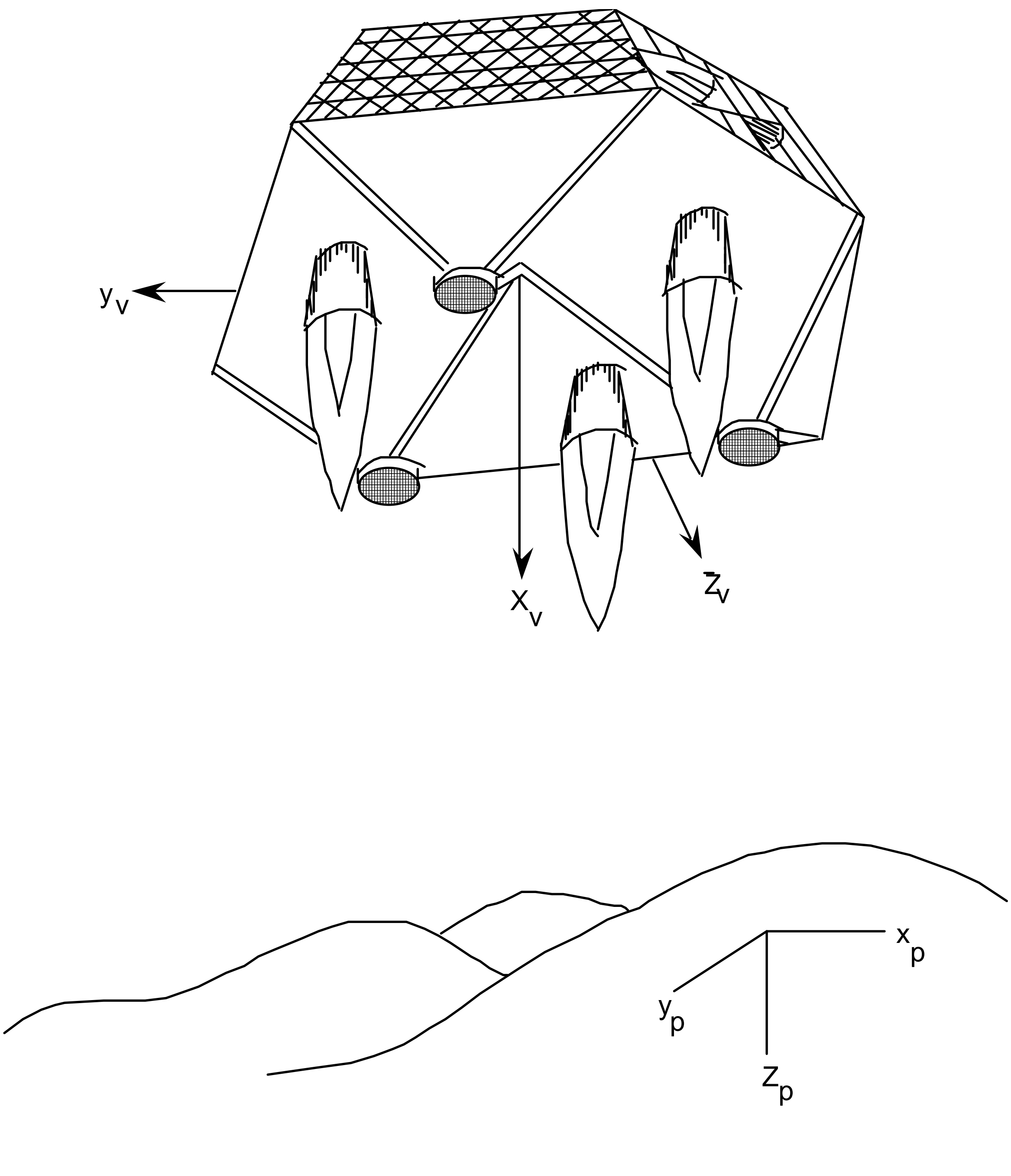
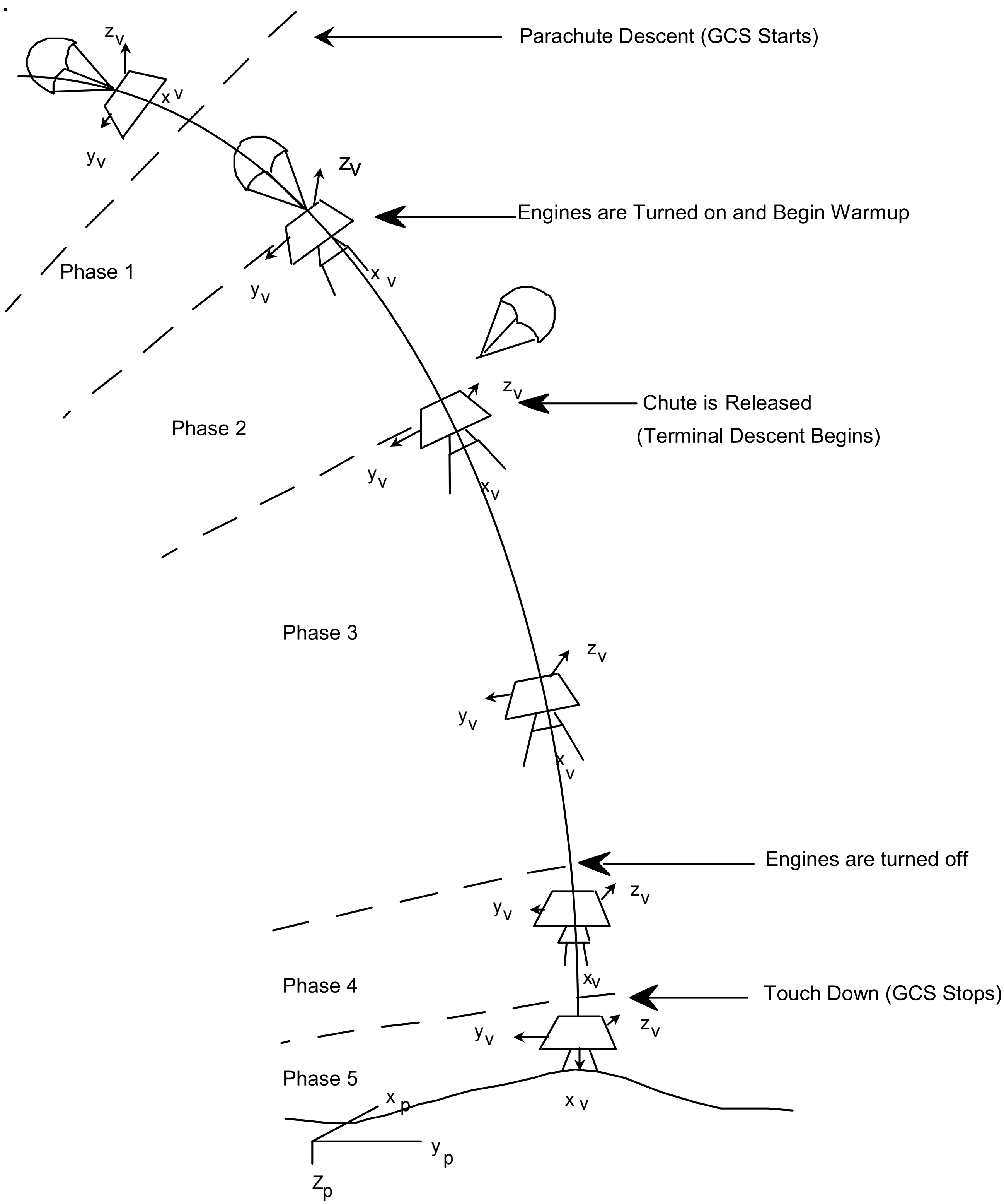


Figure A.1.2 A TYPICAL TERMINAL DESCENT TRAJECTORY



TERMINAL DESCENT

Prior to the terminal descent phase, the vehicle falls with a parachute attached. This parachute is released seconds after the engines ignite, and terminal descent begins. During terminal descent, the vehicle follows a modified gravity-turn guidance law until a predetermined altitude is reached. The atmosphere introduces drag forces, including the random effects of wind. Independently throttled engines slow the vehicle down. These engines can control the vehicle's orientation, and roll engines control the vehicle's roll rate. Roll control is necessary to keep the doppler radars in lock and insure that the desired touch down attitude (land on two legs prior to the third) is maintained.

The velocity during descent follows the predetermined velocity-altitude contour. At a specific altitude above the planet surface, the vehicle is maintained at a constant descent velocity. Once the surface is sensed, all engines are shut down and the vehicle free falls to the surface.

VEHICLE DYNAMICS

Frames of Reference

Terminal descent is described in terms of two coordinate systems:

1. the surface-oriented coordinate system, and
2. the vehicle-oriented coordinate system.

In the surface coordinate system, the $\vec{z}_p$ axis is viewed as normal to the surface and points down as shown in Figure A.1.2. The $\vec{x}_p$ axis points north, and the $\vec{y}_p$ points east.

By defining a *unit vector* as a vector of length equal to one unit along each axis in both the planetary and vehicular frames of reference, a relation between these two frames of reference may be established. Any vector can then be defined as a multiple of the unit vector along each of the axes defined in the frame of reference. Thus, the velocity of the vehicle $\vec{V}$ may be defined in the vehicle's frame of reference as: $V_{x_v}\hat{i}_v + V_{y_v}\hat{j}_v + V_{z_v}\hat{k}_v$, where $\hat{i}_v, \hat{j}_v,$ and $\hat{k}_v$ are the unit vectors in the *x*, y, and *z* directions of the vehicles coordinate system (unit vectors are usually represented by lower case i, j, or k with a hat to show that they are unit vectors). V_{x_v}, V_{y_v}, and V_{z_v} represent the components of the vehicle velocity in the given direction. At the same time, the velocity of the vehicle may be described in the planetary coordinate system as: $V_{x_p}\hat{i}_p + V_{y_p}\hat{j}_p + V_{z_p}\hat{k}_p$, where the subscript *p* represents planetary rather than vehicle coordinates. Note, since the two coordinate systems are not oriented in the same direction, the values of V_{x_v} will not be equal to V_{x_p}, but the magnitude of the total vector $\vec{V}$ will be the same in both systems.

Also the difference in the magnitudes of individual components represents the difference in relative orientation between the two coordinate systems.

The *dot product* $(\vec{a} \cdot \vec{b})$ is defined as the magnitude of $\vec{a}$ multiplied by the magnitude of $\vec{b}$ and then by the cosine of the angle between the vectors,

$$\vec{a} \cdot \vec{b} = |a||b| \cos \angle \vec{a}\vec{b}$$

The dot product is used to project $\vec{a}$ onto $\vec{b}$ and can be used to project a vector in one frame of reference onto another one. Rather than calculate the needed cosines each time a vector must be transformed from one frame of reference into another, the cosines of the angles between each unit vector of the vehicular and planetary coordinate systems are computed and placed into a *direction cosine matrix*. This matrix is then used along with the vector's magnitude in each dimension of the original frame of reference to compute a dot product. This product gives the vector's magnitude in each dimension of the new frame of reference.

The transformation between the vehicle and the surface coordinate systems at time *t* is specified by a matrix of direction cosines,

$$\begin{pmatrix} l_1 & l_2 & l_3 \\ m_1 & m_2 & m_3 \\ n_1 & n_2 & n_3 \end{pmatrix}_t = \begin{pmatrix} \cos\theta\left(\hat{i}_v, \hat{i}_p\right) & \cos\theta\left(\hat{i}_v, \hat{j}_p\right) & \cos\theta\left(\hat{i}_v, \hat{k}_p\right) \\ \cos\theta\left(\hat{j}_v, \hat{i}_p\right) & \cos\theta\left(\hat{j}_v, \hat{j}_p\right) & \cos\theta\left(\hat{j}_v, \hat{k}_p\right) \\ \cos\theta\left(\hat{k}_v, \hat{i}_p\right) & \cos\theta\left(\hat{k}_v, \hat{j}_p\right) & \cos\theta\left(\hat{k}_v, \hat{k}_p\right) \end{pmatrix}_t$$

where $\theta\left(\hat{i}, \hat{j}\right)$ denotes the angle between vectors $\hat{i}$ and $\hat{j}$, etc.

The change in orientation of the vehicle during descent makes the update of the direction cosine matrix necessary at each time step. This update is specified in the following equation:

$$d/dt \begin{pmatrix} l_1 & l_2 & l_3 \\ m_1 & m_2 & m_3 \\ n_1 & n_2 & n_3 \end{pmatrix}_t = \begin{pmatrix} 0 & r_v & -q_v \\ -r_v & 0 & p_v \\ q_v & -p_v & 0 \end{pmatrix}_t \begin{pmatrix} l_1 & l_2 & l_3 \\ m_1 & m_2 & m_3 \\ n_1 & n_2 & n_3 \end{pmatrix}_t$$

where the matrix containing the p_v, q_v, and r_v terms is the rate of rotation about the axes of the vehicle which may be obtained from sensor values.

Linear Velocity

The linear components of velocity for the vehicle during terminal descent are denoted by $\dot{x}_v, \dot{y}_v$, and $\dot{z}_v$ in the vehicle coordinate system and by $\dot{x}_p, \dot{y}_p$, and $\dot{z}_p$ in the surface coordinate system, where the dot $(\dot{})$ notation indicates derivatives with respect to time.

Vehicle Position

Vehicle position is expressed in terms of the surface coordinate system by transforming change in position (velocity) in the vehicle coordinate system into change in position in the surface frame and integrating as follows:

$$\begin{pmatrix} \dot{x}_p \\ \dot{y}_p \\ \dot{z}_p \end{pmatrix}_t = \begin{pmatrix} l_1 & m_1 & n_1 \\ l_2 & m_2 & n_2 \\ l_3 & m_3 & n_3 \end{pmatrix}_t \begin{pmatrix} \dot{x}_v \\ \dot{y}_v \\ \dot{z}_v \end{pmatrix}_t$$

and

$$\begin{pmatrix} x_p \\ y_p \\ z_p \end{pmatrix}_t = \int \begin{pmatrix} \dot{x}_p \\ \dot{y}_p \\ \dot{z}_p \end{pmatrix} d\tau \Bigg|_t$$

Angular Velocity

Roll, pitch, and yaw angular velocities are represented by the quantities p_v, q_v, and r_v in the vehicle frame of reference only. Roll is about the $\vec{x}_v$ axis, pitch is about the $\vec{y}_v$ axis, and yaw is about the $\vec{z}_v$ axis, as shown in Figure A.1.3. A more in-depth explanation of angular velocity naming conventions and other related material may be found in section II, part B of Reference (ref. A.3).

Vehicle Attitude

The vehicle attitude at time t is a function of the vehicle attitude (known by reference to celestial objects) at the start of descent at time t_0 and the cumulative changes in attitude from time t_0 to time t.

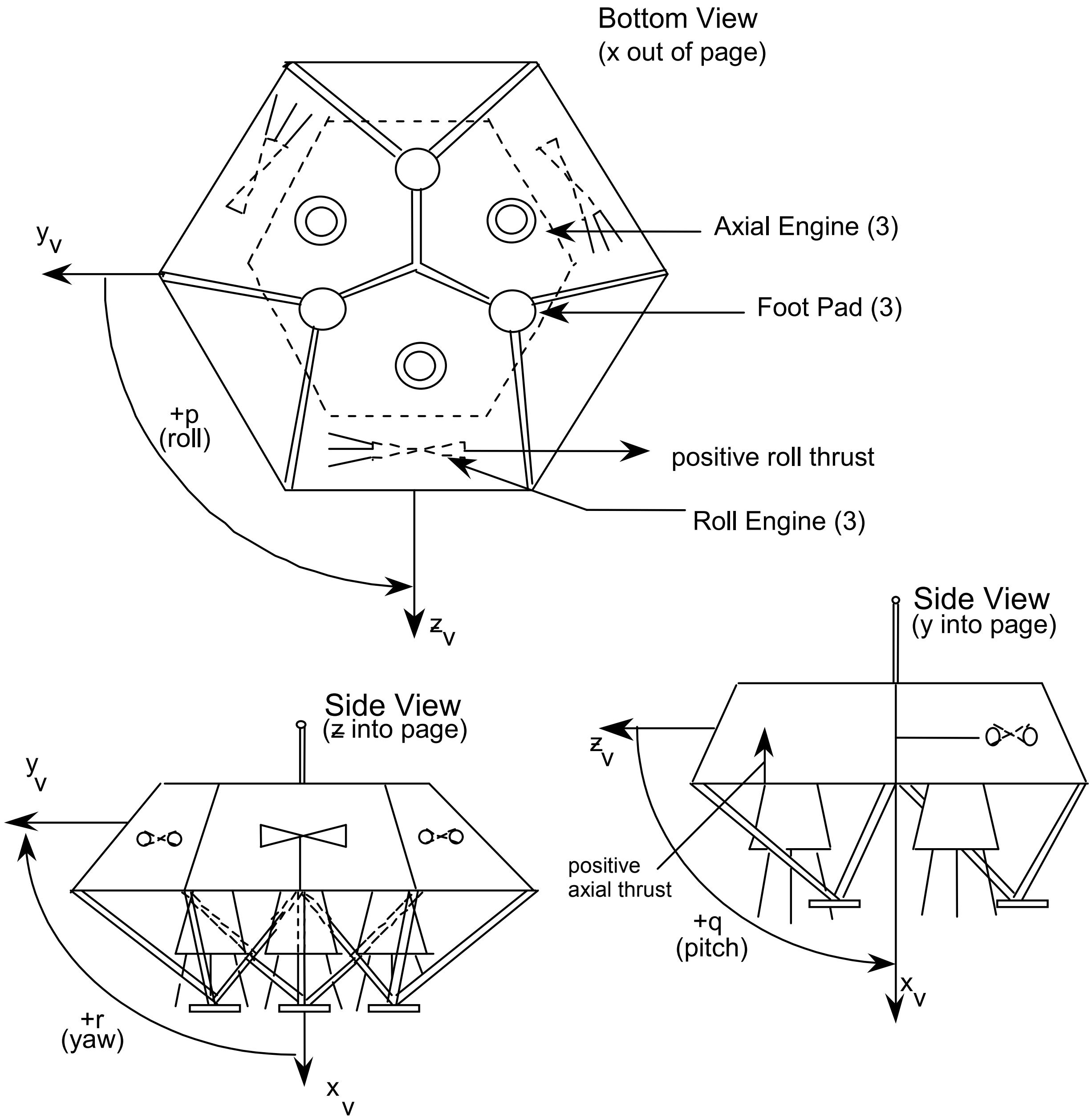
Acceleration

The linear components of acceleration for the vehicle in the vehicle frame of reference during terminal descent are denoted by $\ddot{x}_v$, $\ddot{y}_v$, and $\ddot{z}_v$, respectively.

Further Reading

The subjects of vector mathematics, transformations between frames of references, vector calculus, and rotating coordinate systems may not be sufficiently covered here for the user; however, such depth is not intended for this document. Chapter 4 of *Classical Mechanics* (ref. A.4) contains a detailed explanation of rigid body motion and transformation of vectors into multiple frames of reference or coordinate systems. Chapters 15 and 16 of *Engineering Mechanics* (ref. A.5) contains a more basic approach to the same ideas of multiple frames of reference and vector mechanics. Chapter 14 of (ref. A.6) and Chapter 5 of (ref. A.7) also discuss rotational motion and multiple frames of reference, as well as vector mechanics and calculus. Two other books of possible interest are (ref. A.8) and (ref. A.9). Both cover the mechanics of particles and dynamics, with strong references to particle trajectories and rocket dynamics. Also, these texts are basic in nature and require only a rudimentary knowledge of physics, math, or engineering.

Figure A.1.3 ENGINEERING ILLUSTRATION OF VEHICLE



VEHICLE GUIDANCE

Vehicle guidance is accomplished by varying the engine thrust so that the vehicle follows a single predetermined velocity-altitude contour. This contour is made available during GCS initialization. Applying too great a deceleration early in the descent brings the vehicle velocity to its terminal value too high above the surface, resulting in insufficient propellant for final descent. Applying too small a thrust lets the vehicle impact the surface with too great a velocity. Either condition could be disastrous. As soon as the touch down sensor touches the surface, the engines are shut off. Approximately ninety percent of propellant or thrust is used to minimize gravity losses; the remaining ten percent is used for steering.

A gravity-turn steering law is mechanized by rotating the vehicle in pitch and yaw until the body's lateral axis velocities are zero (causing the thrust axis to point along the total velocity vector). The action of gravity causes the thrust axis to rotate toward the vertical as the total velocity is reduced. An arbitrary roll orientation is maintained with an attitude hold mode during the descent.

ENGINES

The vehicle has three axial engines that supply the force necessary to slow the vehicle and allow it to safely land. Roll is controlled by three pairs of roll engines on the lander supplying rotational thrust. Figure A.1.3 shows the axial and roll engines and the resulting thrust forces they impart to the vehicle.

Axial Engine (Thrust) Control

Three thrust engines first orient the vehicle so that their combined thrust vector opposes the vehicle's velocity vector. Thrust (axial direction) engine control is a function of pitch error, yaw error, thrust error, and deviation from the velocity-altitude contour. A combination of proportional and integral control (PI) logic is applied to pitch and yaw control. The integral portion helps to reduce the steady-state pitch and yaw error.

If no thrust error or velocity-altitude contour deviation occurs, then axial engine response provides only pitch and yaw control via the PI control law. Use of this control law implies that the overshoot problem for pitch-yaw control is probably small.

Thrust control is implemented by a proportional-integral-derivative (PID) control law. The derivative control added here damps out overshoot.

Roll Engine Control

Roll control is attained by pulsing the three pairs of roll engines and is a function of roll angle deviation and roll rate (p_V) about the x axis. Roll engine specific impulse and thrust per unit time are constant with the integrated thrust controlled by pulse rate. Angle deviations are controlled within a very small range of 0.25 to 0.35 degrees.

GENERAL INFORMATION

NOTATION

Matrices and Arrays

It should be noted that throughout this specification, the words matrix and array are often interchanged. No significance should be placed upon the use of one word as opposed to use of the other.

All matrices are referenced with the row index first and the column index second. In the cases where there is a time history (see definition of history variable below), the last index is the time index.

When the name of an array which contains a time history is given without any index for the time history being specified, the most recent value is implied.

Operators

Throughout this specification, matrix operations (particularly multiplication) are required, and on some occasions, non-standard operations are used upon matrices. The following symbols are used to denote the types of multiplication to be applied.

Dots $(\cdot)$ Small dots are used to denote scalar multiplication. For example:

$$3 \cdot 4 = 12$$

Multiplication sign $(\times)$ This symbol is used to denote standard matrix multiplication. This does NOT imply a cross product, nor strictly a dot product. The definition of this type of operation is given below:

$$A \times B = C$$

where

$$\forall \text{i} \quad \text{and} \quad \forall \text{j}\,, \quad C_{ij} = \sum_{\forall k} A_{ik} \cdot B_{kj}$$

Asterisks (*) Asterisks are used in conjunction with index markers to show that the operations are to be conducted on individual elements of arrays or vectors as if they were scalars. This is often the case when calculating sensor values or other similar functions when multiple scalars are grouped together for convenience. For example, the following equation is listed in ASP:

$$A\_ACCELERATION\_M(i) = A\_BIAS(i) + A\_GAIN(i) * A\_COUNTER(i)$$

where i ranges from 1 to 3 and represents the three directions x, y, and z. In this case, the first element of A_ACCELERATION_M would be calculated as follows:

$$A\_ACCELERATION\_M(1) = A\_BIAS(1) + A\_GAIN(1) \cdot A\_COUNTER(1)$$

No Operator In those cases where variables, matrices, or scalars are located directly beside each other with no operator between, standard multiplication is implied. Thus two matrices collocated would be multiplied as if they had the $\times$ operator between them, while two scalars would be multiplied as if they had the $\cdot$ operator between them. Also, if a scalar and a matrix (of one or more dimensions) were collocated, then the scalar would be multiplied by each element of the matrix and a new matrix of equal dimensions would be generated.

DEFINITIONS

Implementation

Computer code which fulfills all of the requirements outlined in the GCS Development Specification.

Functional Unit

Section A.5 is divided into eleven subsections, each of which describes the requirements for a particular function to be performed by the GCS software. Throughout this specification, the term "functional unit" will be used to refer to one of these eleven functions. Note that there is not necessarily a one-to-one correspondence between a "functional unit" and a distinct unit or module of software code in an implementation.

Frame

A frame is the length of time necessary to execute all scheduled functional units. Each frame has two different time values associated with it. The first is the actual c.p.u. time that it takes to execute the GCS software on the simulation host computer, while the second is the allotted time for a frame on the actual lander. The global variable DELTA_T represents the time for one frame on the actual lander and is needed in the GCS code for the integration of the dynamic equations for the lander.

Subframe

A subframe is one of the three individual units of time which together make up a frame. The three subframes are named the Sensor Processing subframe (subframe 1), the Guidance Processing subframe (subframe 2), and the Control Law Processing subframe (subframe 3). In each frame, subframe 1 is executed first, subframe 2 is executed second, and subframe 3 is the last subframe executed.

Data Store

The definition for a data or control store given in Hatley (ref. A.13) is "A data or control store is simply a data or control flow frozen in time. The data or control information it contains may be used any time after that information is stored and in any order." In this specification, all stores contain data, while some also contain data conditions. For the purposes of this specification, the term "data store" will be used to refer to any store which contains some combination of data and data conditions. Thus, all four stores listed in the Data Requirements Dictionary part II will be referred to as "data stores".

Global Data Store Variable

A global data store variable is any variable listed in any of the four global data stores in Section A.6, namely GUIDANCE_STATE data store (Table A.6.1), EXTERNAL data store (Table A.6.2), SENSOR_OUTPUT data store (Table A.6.3), or RUN_PARAMETERS data store (Table A.6.4).

History Variable

Within this specification, a particular array, hereafter referred to as a "history variable" is one which contains a time history dimension; that is, it contains values for the current frame as well as for previous frames. The history variables are the following:

A_ACCELERATION (1:3,0:4)
A_STATUS (1:3,0:3)
AR_ALTITUDE (0:4)
AR_STATUS (0:4)
G_ROTATION (1:3,0:4)
GP_ALTITUDE (0:4)
GP_ATTITUDE (1:3,1:3,0:4)
GP_VELOCITY (1:3,0:4)
K_ALT (0:4)
K_MATRIX (1:3,1:3,0:4)
TDLR_VELOCITY (1:3,0:4)

In each case, the last dimension is the time dimension. The first subscript in a time history dimension is always declared to be zero. The time dimension contains a set of scalars, vectors, or arrays, depending on whether the total number of dimensions is one, two, or three, respectively. Let the term "object" denote a scalar, vector, or array, as appropriate for the particular variable. Each of these variables contains either four or five objects, depending on whether the last dimension is declared to be 0:3 or 0:4 respectively. The variable A_STATUS contains four objects, while each of the other time history variables contains five objects.

Each of the variables listed contains a most recent object and either three or four previous objects. The object with a time subscript of zero is the most recent object; the object with a time subscript of one is the object which is one frame older; the object with a time subscript of two is the object which is two frames older, etc.; the object with the largest time subscript (three or four) is the oldest object.

CONVENTIONS

FORTRAN Convention

This specification was written with the assumption that the implementation would be coded in FORTRAN. If the development language used is something other than FORTRAN, the programmer must investigate the possibility of differences between FORTRAN and the development language chosen.

REQUIREMENTS

Order of Processing

Within each functional unit in Chapter 5, the processing steps are given in a particular order. If the implementation uses the same order as that given in the specification, then correct results should be obtained; however, the programmer is free to use a different order as long as the change in order does not affect the outputs.

Calls to GCS_SIM_RENDEZVOUS

There must be a call to GCS_SIM_RENDEZVOUS prior to the execution of each subframe. See section A.2 and section A.8 for discussions regarding GCS_SIM_RENDEZVOUS.

Control Signals

The control signals listed in Table A.6.5 in Part III of the Data Requirements Dictionary may be implemented by the programmer in any form desired, or they may be completely ignored and the control of the program may be conducted through other means.

Number Representations

When variables are given in sign-magnitude or other unusual formats, conversion or manipulation may be necessary.

Conversion of Units

It is the responsibility of the programmer to be sure that any implied conversion of units is performed.

Global Data Store Organization

Part II of the Data Requirements Dictionary contains descriptions of four required data stores. Each of these data stores is to be located in a separate, globally accessible data region. The division of the global data stores into four separate regions illustrates the fact these regions have a direct mapping to a specific implementation of GCS on hardware components of an actual lander. (See Figure A.B.1).

If the implementation is being written in FORTRAN, four labeled common blocks should be declared with the labels GUIDANCE_STATE, EXTERNAL, SENSOR_OUTPUT, and RUN_PARAMETERS, respectively (See Tables 6.1, 6.2, 6.3, and 6.4). The variables declared in each labeled common block must be in the same order as those in the corresponding table.

Use of Variables That Are Not in the Global Data Stores

A programmer may use variables in addition to the global data store variables; however, if the value of such a variable is dependent upon the values of any global data store variable(s), then the programmer should only use the value of such a variable in the same subframe of the same frame in which it was calculated.

Use of Tables

Some tables have the heading "CURRENT STATE" and "ACTIONS". If the actual state of the variables appears under the "CURRENT STATE" section in the table, then the actions listed in the same line are to be performed. If the actions in one line of the table are performed, then none of the actions in any other line of the table should be performed in the same subframe. If the actual current state is not represented in any line under the "CURRENT STATE" section of the table, then no action is to be taken.

Rotation of History Variables

In Chapter 5, in certain functional units, an instruction is given to "rotate" specific variables. Table A.1.1 illustrates what is meant by rotation. The table is given for a variable with a time dimension of 0:4. For a variable with a time dimension of 0:3, the last line of the table should be ignored. Note that after the variable has been rotated, the new or current object is calculated and placed into the zeroth time history position.

Table A.1.1 ROTATION OF VARIABLES

TIME HISTORY SUBSCRIPT	VALUES BEFORE ROTATION	VALUES AFTER ROTATION	VALUE AFTER CALCULATIONS FOR CURRENT FRAME
0	O_{n-1}	X	O_n
1	O_{n-2}	O_{n-1}	O_{n-1}
2	O_{n-3}	O_{n-2}	O_{n-2}
3	On-4	O_{n-3}	O_{n-3}
4	On-5	On-4	On-4

Note: O_i denotes object that was calculated in frame i

n = current frame number

X = denotes that any value is acceptable

Precision

All calculations involving floating point variables should be done with precision equivalent to that of FORTRAN D-floating (REAL*8).

EXCEPTION HANDLING

During the execution of a computer program, exception conditions may sometimes occur. The implementation should anticipate or detect certain types of exception conditions and take specific actions. The relevant exception conditions and the actions to be taken are listed below.

Exception Conditions

DIVIDE BY ZERO

A division is performed, but the divisor is equal to zero.

NEGATIVE SQUARE ROOT

A square root is taken, but the argument for the square root is negative.

UPPER OR LOWER LIMIT EXCEEDED

The current value for a data element exceeds its upper or lower limit as specified in the range section in the Data Requirements Dictionary Part I.

Only certain data elements under certain conditions are to be checked for limits exceeded. The criteria for which elements are to be checked, in what context they are to be checked, and when they must be checked is as follows:

Which data elements:

A particular data element is to be checked for limits exceeded only if it is of data type REAL*8, and is in either of the two global data stores GUIDANCE_STATE or SENSOR_OUTPUT.

Context for check:

A data element is to be checked only when it is being used as an input. Rotation of a data element is not considered to be a use as an input for the purposes of limit checking. If the data element is a vector or array, then each element in the vector or array that is being used as input must be checked, including history values. It is not necessary for the functional unit CP to check any of its input data elements for limit exceeded.

When data element must be checked:

When an input data element is to be used or processed in a given subframe, then it must be checked sometime within that same subframe before it is used. If the data element is also being updated or changed in the same subframe before it is being used as an input, then it must be checked sometime between the time it is updated and the time it is used.

Action to be Taken for Each Specified Exception Condition

Write the appropriate output as specified below to the FORTRAN Logical Unit Number 6 and then continue. In the case of UPPER/LOWER LIMIT EXCEEDED, do not modify the data element. Note that to "continue" implies that the divide will be executed, or the square root will be taken, or the data element with exceeded limit will be used.

Output to be Generated for Each Exception Condition

The first line of the exception message should appear as follows:

" %EXCEPTIONAL-CONDITION-GCS-"<insert specific condition here>

where the specific condition is one of the following:

"DIVIDE_BY_ZERO"
"NEGATIVE_SQUARE_ROOT"
"LOWER_LIMIT_EXCEEDED"
"UPPER _LIMIT_EXCEEDED"

The second line of the exception message should contain the name of the functional unit where the exception condition occurred (i.e. AECLP, ASP, etc.), the name of the actual subroutine where the exception condition occurred, and the current value of the frame counter. Implementations that are coded in FORTRAN should use the following FORTRAN format statement:

FORMAT (x, a6, x, a32, x, i4)

A third line of the exception message containing information that is specific to the individual error type may be required as specified below.

Divide By Zero

No additional output necessary.

Negative Square Root

Display the value of the argument to the square root operation.
Use FORTRAN format statement FORMAT (x, e23.14).

Lower Limit Exceeded

Display the name of the data element in question and the value of the data element.
Use FORTRAN format statement FORMAT (x, a32, e23.14) for type real elements.

Upper Limit Exceeded

Display the name of the data element in question and the value of the data element.
Use FORTRAN format statement FORMAT (x, a32, e23.14) for type real elements.

A.2 LEVELS 0 AND 1 SPECIFICATION

LEVEL 0 SPECIFICATION

The GCS will provide an interface between the sensors (rate of descent, attitude, etc.) and the engines (roll and axial). The purpose of the GCS is to keep the vehicle descending along the predetermined velocity-altitude contour which has been chosen to conserve enough fuel to effect a safe attitude and touch down.

The GCS effects this control by:

- processing the following sensor information:
  - acceleration data from the three accelerometers -- one for each vehicle axis,
  - range rate data from four splayed doppler radar beams,
  - altitude data from one altimeter radar,
  - temperature data from a solid-state temperature sensor and a thermocouple pair temperature sensor,
  - rates of rotation from three strapped-down gyroscopes -- one for each vehicle axis, and
  - sensing of touch down by the touch down sensor.

- determining the appropriate commands for the axial and roll engines and the chute release mechanism and issuing them to keep the vehicle on a predetermined velocity-altitude contour.

The GCS also transmits telemetry data and synchronizes through a rendezvous routine (GCS_SIM_RENDEZVOUS) with GCS_SIM (ref. A.10), the simulator and controller.

Note that implementations of the GCS developed from this specification may be executed singly or in parallel. Consequently, only specific system services can be used in an implementation. In particular, a rendezvous routine will be provided and should be invoked, as specified in the implementation notes in section A.8. In addition, FORTRAN Intrinsic Functions may be used. Other system services and library routines are explicitly excluded from use by the programmer.

Figures 2.2 through 2.5, 3.1, 3.2, and 4.1 through 4.4, and Tables 2.1, 3.1, 4.1, and 4.2 follow Hatley's extension to Structured Analysis (see section A.7), with the following exceptions and assumptions.

Exceptions:

1. Any data store may appear at more than one level because the processes specified do not communicate directly but only through data stores.
2. Any unlabeled flow between a process and a data store may not necessarily carry all the information in the data store (the actual flow content is defined by the process specification and the Data Requirements Dictionary Part II).

Assumptions:

1. The initial value for control signals is assumed to be "FALSE".
2. In a process activation table (PAT), an empty process cell indicates the process is deactivated.
3. In a PAT, an empty output cell indicates the control signal value remains unchanged.
4. In a PAT, output control signals receive values before any processes are activated and therefore may delay the activation of processes by deactivating their parent process.

An example of assumption 4 is Table A.3.1 where setting RENDEZVOUS to "TRUE" delays the activation of the processes of which RUN_GCS is composed until GCS_SIM sets RENDEZVOUS to "FALSE".

Figure A.2.1 STRUCTURE OF THE GCS SPECIFICATION

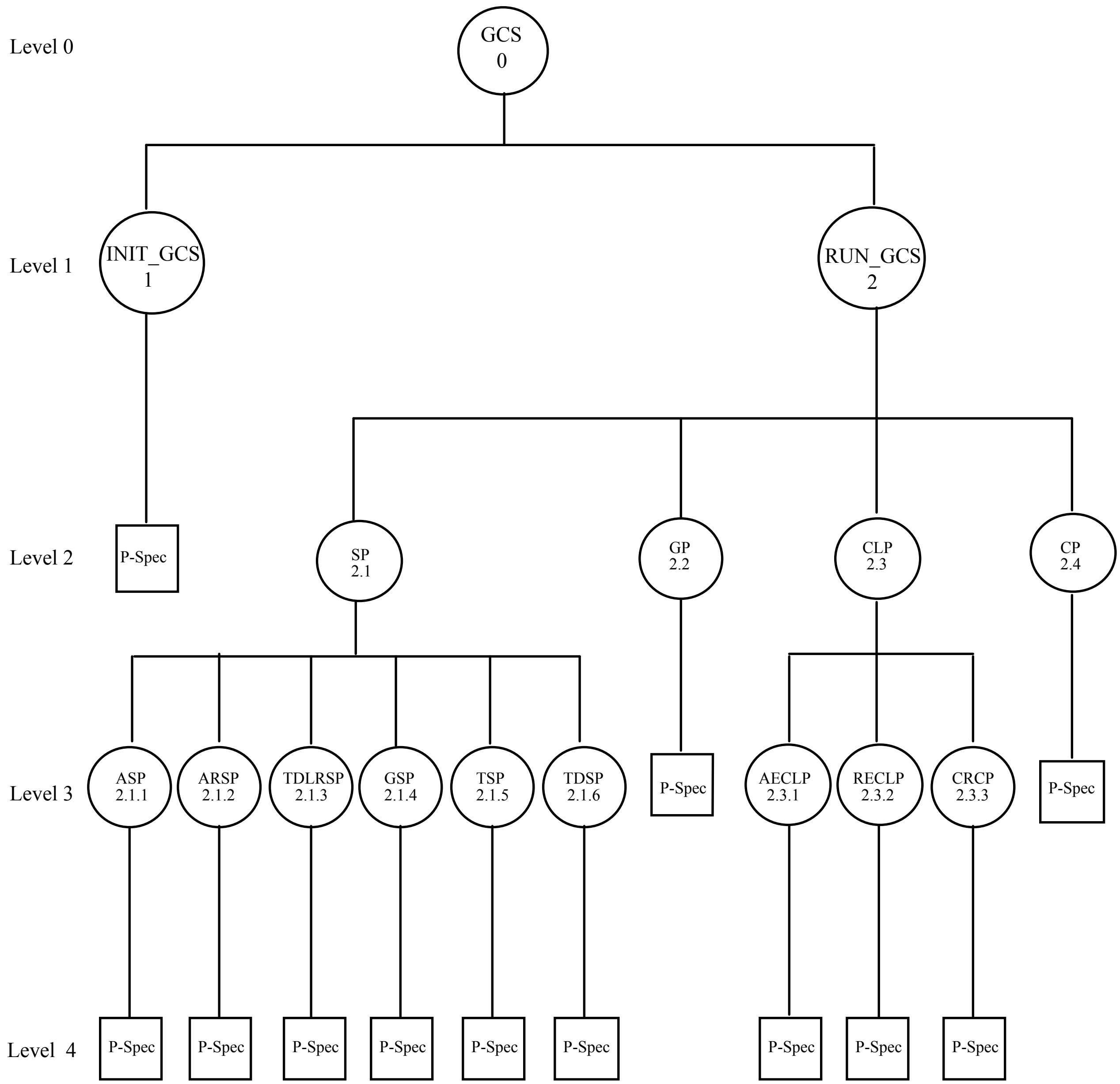


Figure A.2.2 DATA CONTEXT DIAGRAM: LANDER

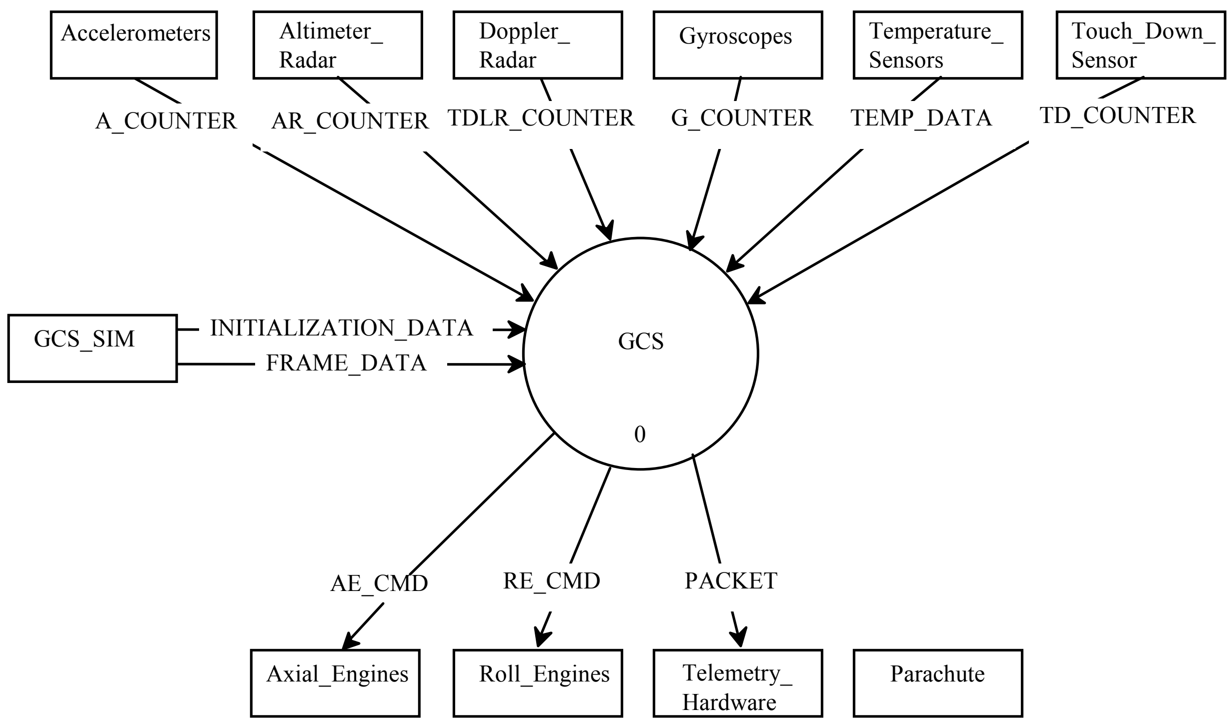
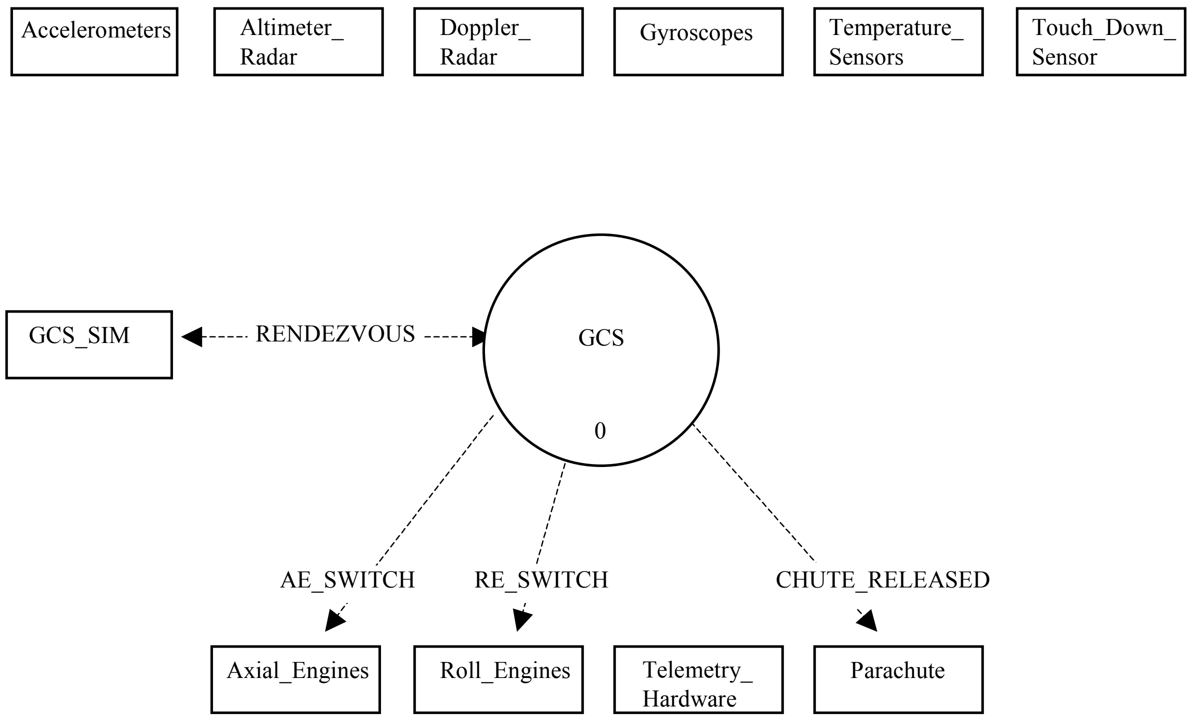


Figure A.2.3 CONTROL CONTEXT DIAGRAM: LANDER



LEVEL 1 SPECIFICATION

Figure A.2.4 DATA FLOW DIAGRAM (DFD) 0: GCS

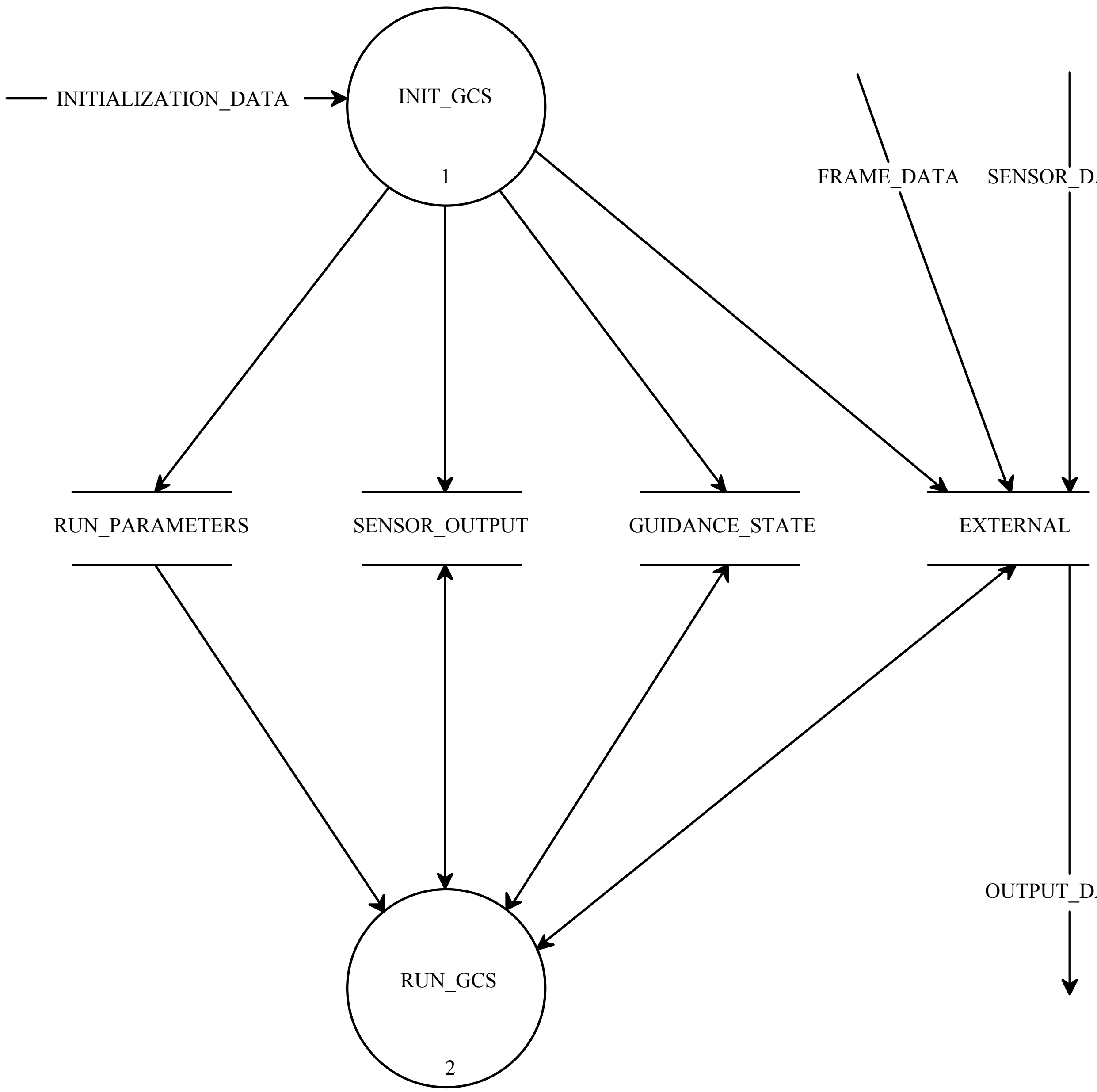
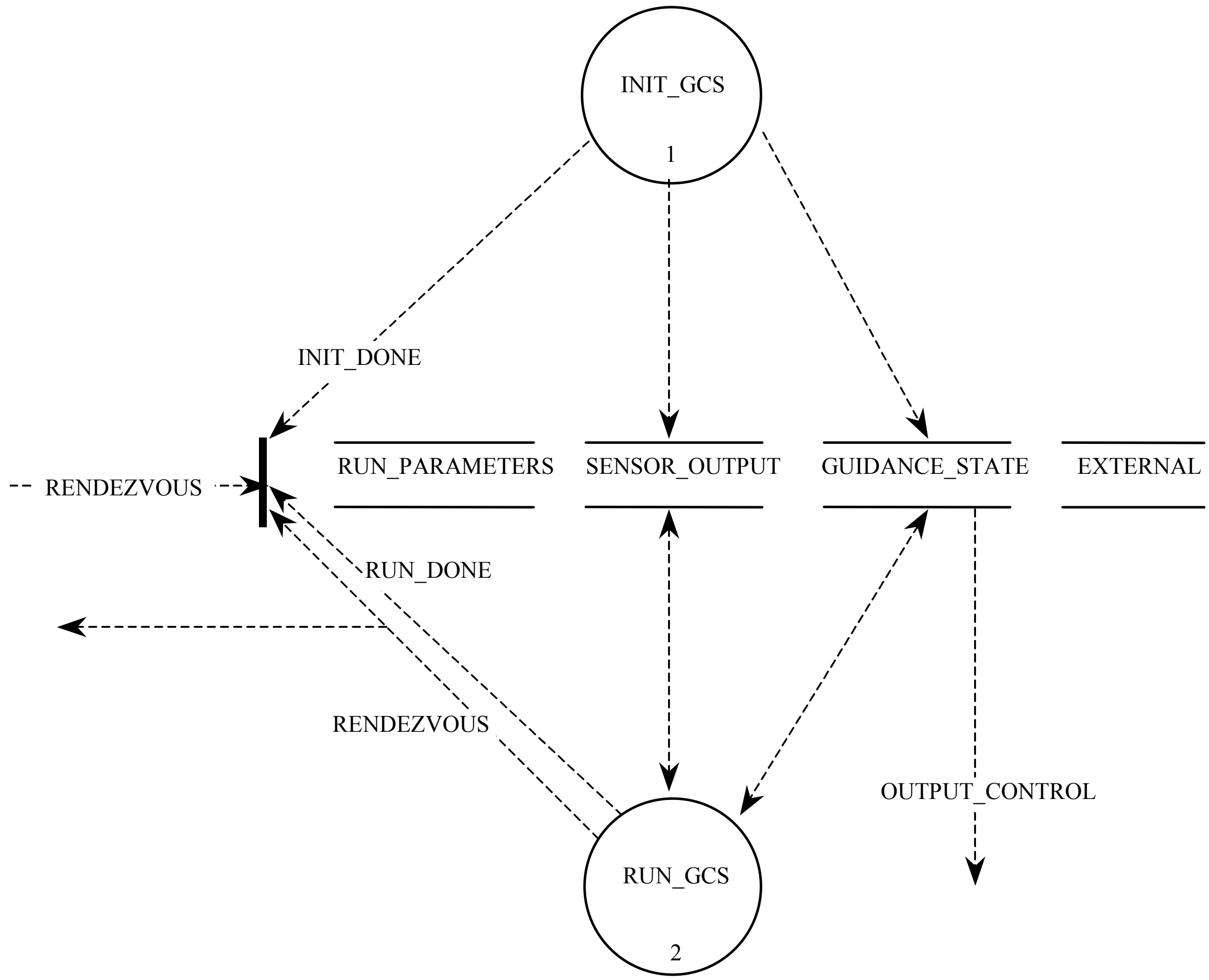


Figure A.2.5 CONTROL FLOW DIAGRAM (CFD) 0: GCS



RENDEZVOUS is only set to "TRUE" by RUN_GCS and it is only set to "FALSE" by GCS_SIM.

Table A.2.1 CONTROL SPECIFICATION (C-SPEC) 0: GCS

	"INIT_GCS"	"RUN_GCS"
~RENDEZVOUS & ~RUN_DONE		1
RENDEZVOUS & ~INIT_DONE & ~RUN_DONE	1	
(RENDEZVOUS & INIT_DONE) | RUN_DONE		

A.3 LEVEL 2 SPECIFICATION**PROCESS SPECIFICATION (P-Spec) 1: INIT_GCS**

PURPOSE INIT_GCS initializes the guidance and control software.

INPUT

INITIALIZATION_DATA

OUTPUT

INITIALIZATION_DATA

PROCESS INIT_GCS is actually a part of GCS_SIM_RENDEZVOUS, which will be supplied to the programmer; thus the functions performed by INIT_GCS are listed here for information only, but are not the responsibility of the programmer. There should be a call to GCS_SIM_RENDEZVOUS, prior to executing each subframe. The first call to GCS_SIM_RENDEZVOUS will cause INIT_GCS to automatically be executed. INIT_GCS will initialize all variables in the group flow INITIALIZATION_DATA, which is defined in Table A.6.7 in the Data Requirements Dictionary Part III. Since the variables FRAME_COUNTER and SUBFRAME_COUNTER are part of INITIALIZATION_DATA, they will be initialized at this time. FRAME_COUNTER will be initialized to a value representing the next frame to be executed, while SUBFRAME_COUNTER will always be initialized to the value one, which implies that the first subframe of the first frame to be executed will always be the sensor processing subframe. Although a terminal descent trajectory begins with FRAME_COUNTER initialized to the value one, the option exists for starting execution at some point other than at the beginning of the trajectory, i.e., FRAME_COUNTER may be initialized to a value greater than one.

Figure A.3.1 DFD 2: RUN_GCS

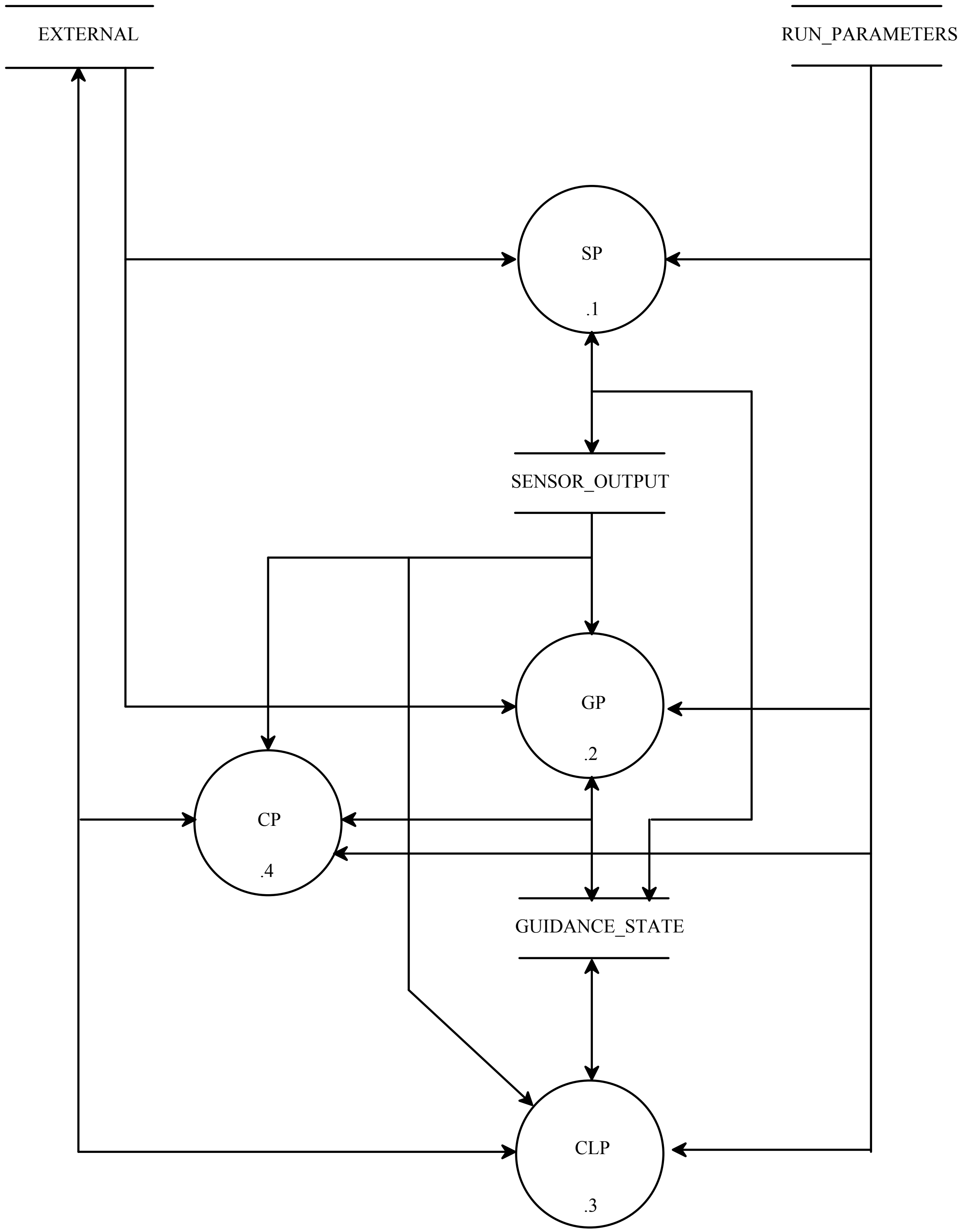


Figure A.3.2 CFD 2: RUN_GCS

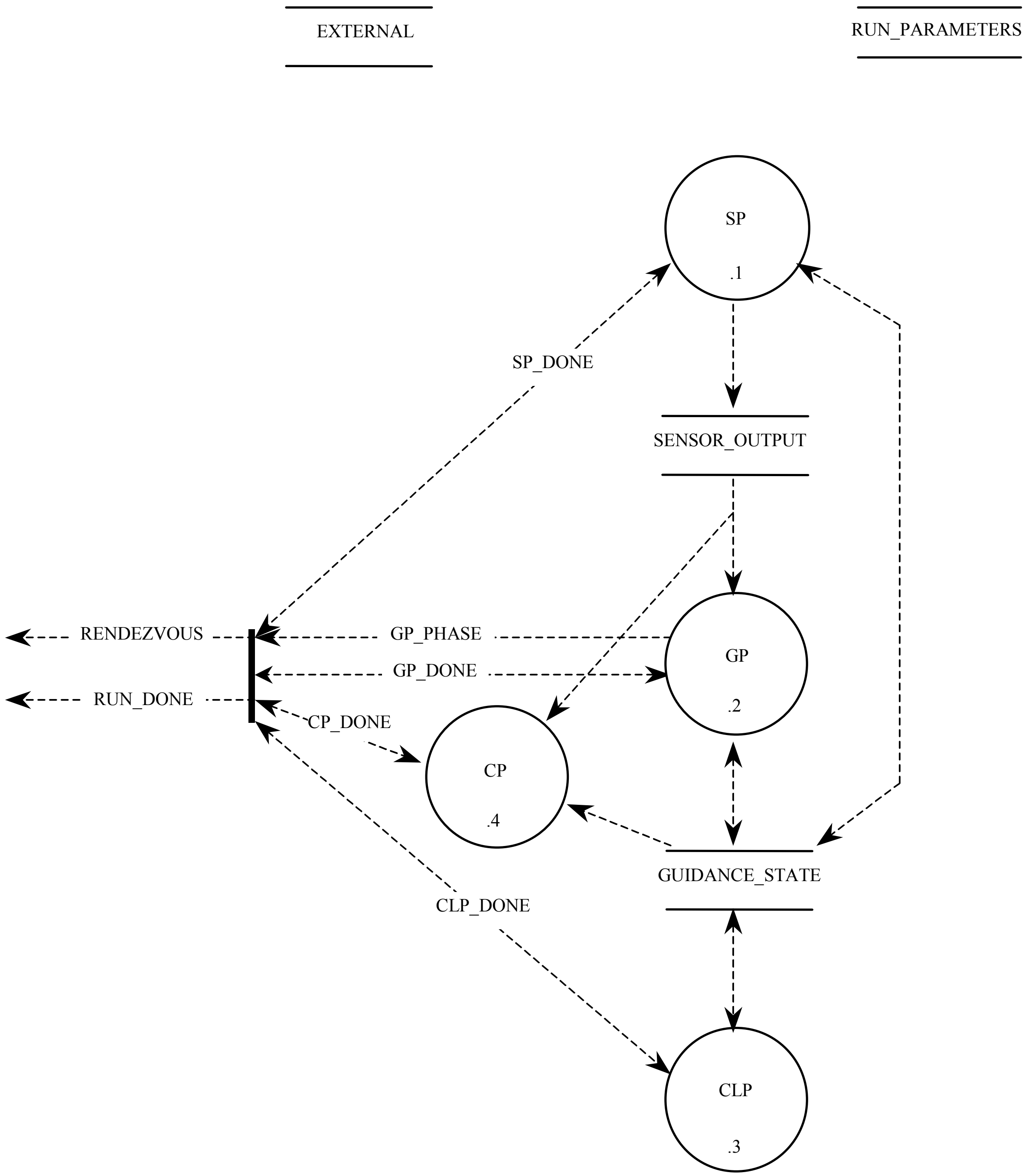


Table A.3.1 C-Spec 2: RUN_GCS

	"SP"	"GP"	"CLP"	"CP"	SP_DONE	GP_DONE	CLP_DONE	CP_DONE	RENDEZVOUS	RUN_DONE
~SP_DONE & ~GP_DONE & ~CLP_DONE & ~CP_DONE	1			2					"TRUE"	
SP_DONE & CP_DONE		1		2	"FALSE"			"FALSE"	"TRUE"	
GP_DONE & CP_DONE & GP_PHASE ~= 5			1	2		"FALSE"		"FALSE"	"TRUE"	
CLP_DONE & CP_DONE	1			2			"FALSE"	"FALSE"	"TRUE"	
GP_DONE & CP_DONE & GP_PHASE = 5										"TRUE"

A.4 LEVEL 3 FLOW DIAGRAMS AND C-SPECS

Figure A.4.1 DFD 2.1: SP -- SENSOR PROCESSING

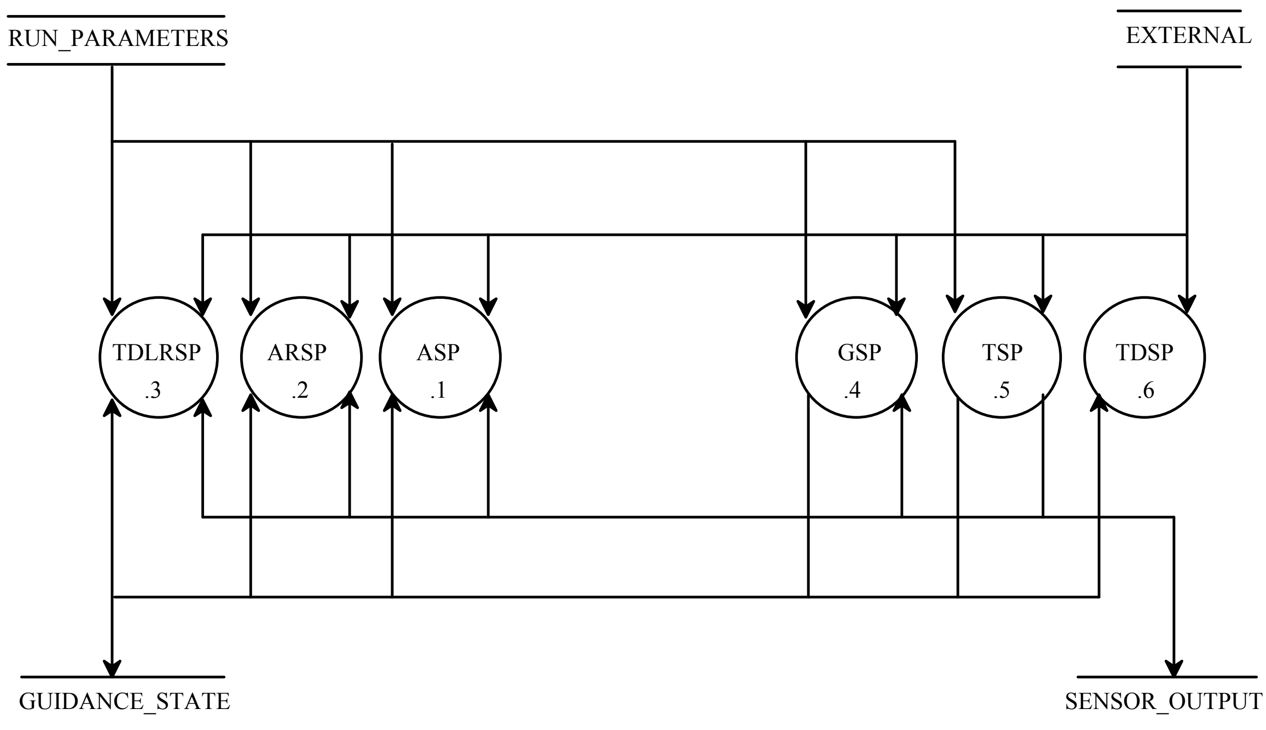


Figure A.4.2 CFD 2.1: SP -- SENSOR PROCESSING

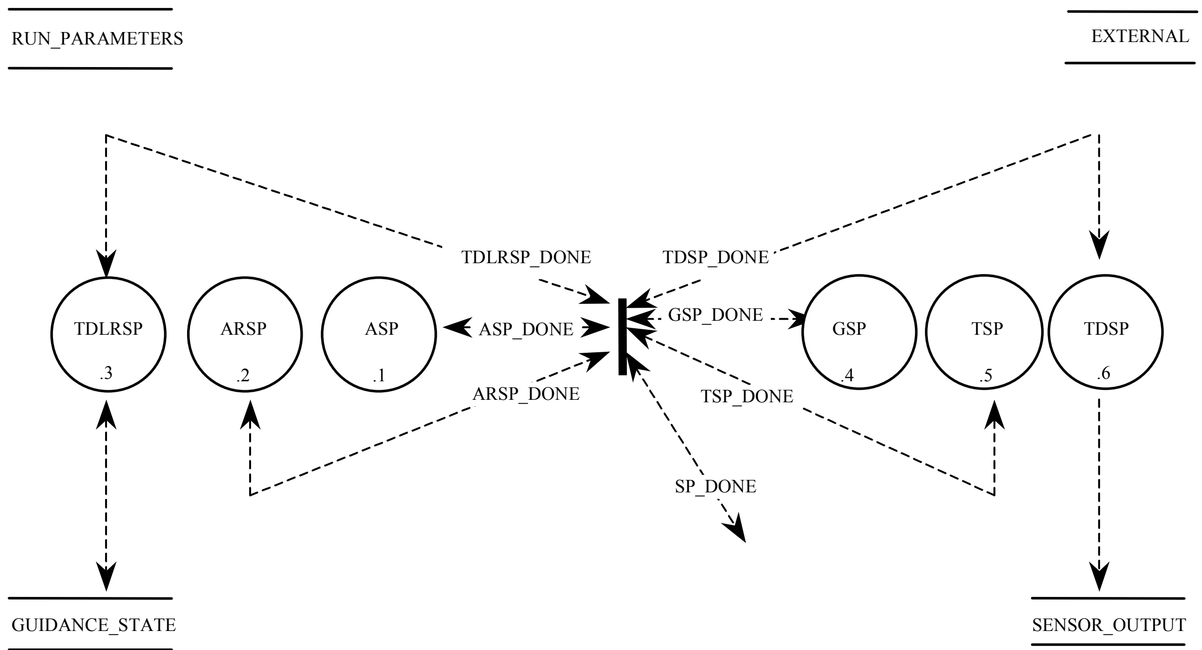


Table A.4.1 C-Spec 2.1: SP -- SENSOR PROCESSING

	"ASP "	ARSP"	"TDLRSP"	"GSP"	"TSP"	"TDSP"	ASP_ DONE	ARSP_ DONE	TDRLSP_ DONE	GSP_ DONE	TSP_ DONE	TDSP_ DONE	SP_ DONE
~ASP_DONE & ~ARSP_DONE & ~TDLRSP_DONE & ~GSP_DONE & ~TSP_DONE & ~TDSP_DONE & ~SP_DONE	2	2	2	2	1	2							
ASP_DONE & ARSP_DONE & TDLRSP_DONE & GSP_DONE & TSP_DONE & TDSP_DONE & ~SP_DONE							"FALSE"	"FALSE"	"FALSE"	"FALSE"	"FALSE"	"FALSE"	"TRUE"

Figure A.4.3 DFD 2.3: CLP -- CONTROL LAW PROCESSING

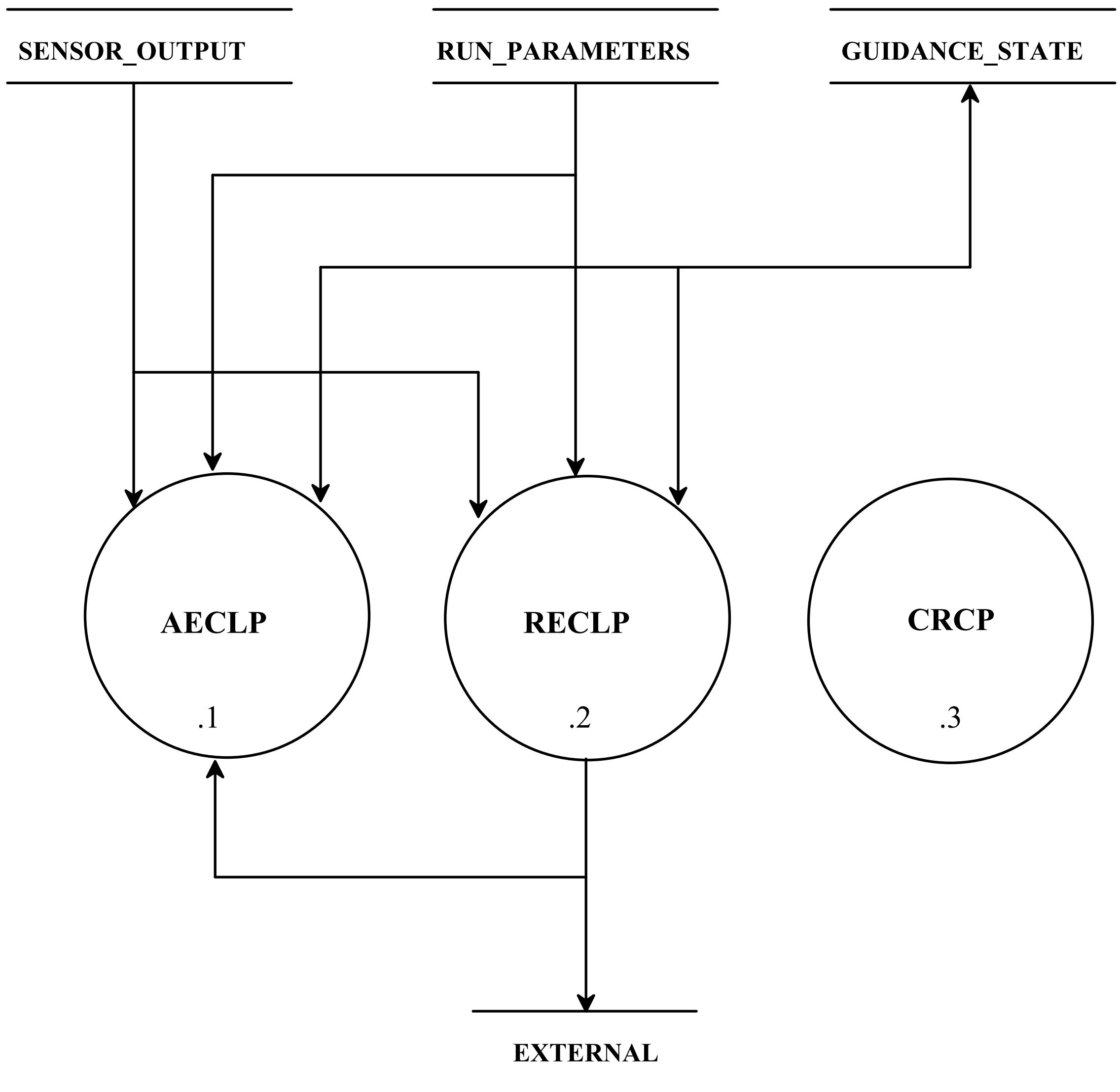


Figure A.4.4 CFD 2.3: CLP -- CONTROL LAW PROCESSING

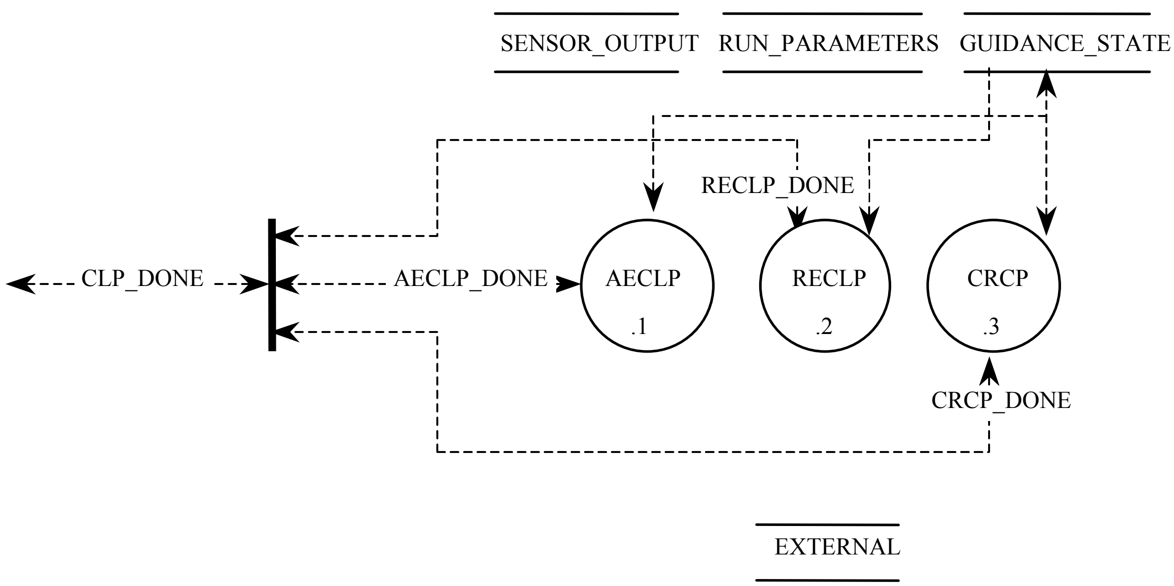


Table A.4.2 C-Spec 2.3: CLP -- CONTROL LAW PROCESSING

	"AECLP"	"RECLP"	"CRCP"	AECLP_DONE	RECLP_DONE	CRCP_DONE	CLP_DONE
~AECLP_DONE & ~RECLP_DONE & ~CRCP_DONE & ~CLP_DONE	1	1					
AECLP_DONE & ~CRCP_DONE & ~CLP_DONE		1	1				
AECLP_DONE & RECLP_DONE & CRCP_DONE & ~CLP_DONE				"FALSE"	"FALSE"	"FALSE"	"TRUE"

SCHEDULING

The execution of one frame consists of the execution of the Sensor Processing Subframe, the Guidance Processing Subframe, and the Control Law Processing Subframe, in that order. Within each subframe, the functional units which are to be executed are listed in Table A.4.3. Within each of the three subframes, GCS_SIM_RENDEZVOUS should be executed before executing any of the functional units, and the functional unit CP should be executed last. In the first and third subframes, there are also some sequencing constraints to be imposed upon certain functional units due to the fact that certain data is output from one unit and input to another unit: In the sensor processing subframe, TSP should be executed before ASP, and TSP should be executed before GSP. In the control law processing subframe, AECLP should be executed before CRCP. Within any given subframe, the order of execution of functional units not specifically mentioned here is immaterial.

Each functional unit will be executed every frame in the particular subframe in which it is included in Table A.4.3. Execution of the GCS may begin at any frame number and should operate as if it had been running from the beginning of the trajectory (frame number 1). On the first, and subsequent, calls to GCS_SIM_RENDEZVOUS, FRAME_COUNTER and SUBFRAME_COUNTER will be returned to the implementation containing the correct values for operation.

Table A.4.3 FUNCTIONAL UNIT SCHEDULING

Sensor Processing Subframe (Subframe 1)
ARSP ASP CP GSP TDLRSP TDSP TSP
Guidance Processing Subframe (Subframe 2)
CP GP
Control Law Processing Subframe (Subframe 3)
AECLP CP CRCP RECLP

The GCS software must meet all the requirements for a particular frame for any specific value of the variable FRAME_COUNTER. The software must be capable of executing continuously one frame after another until specified termination conditions are met, at which time it must terminate itself according to specified termination procedures.

The termination conditions and procedures are: GCS should check whether to terminate itself in each frame immediately after executing the Control Law Processing subframe. At that time if the value of the variable GP_PHASE is equal to 5, then GCS should terminate itself gracefully (without any exception conditions). In this case, the implementation should terminate at the end of the present subframe, i.e., it should execute the functional unit Communications Processing and then terminate without calling GCS_SIM_RENDEZVOUS.

5 P-SPECS FOR LEVELS 3 and 4

AECLP -- Axial Engine Control Law Processing (P-Spec 2.3.1)

PURPOSE The AECLP functional unit computes the valve settings for each of the three main (axial) engines. Measurements of the vehicle's velocity, acceleration, and roll rates are combined to produce error signals for the pitch, yaw, and thrust of the vehicle. These error signals are then mixed to produce the axial engine valve settings.

INPUT

AE_SWITCH	AE_TEMP
A_ACCELERATION	CHUTE_RELEASED
CL	CONTOUR_CROSSED
DELTA_T	ENGINES_ON_ALTITUDE
FRAME_COUNTER	FRAME_ENGINES_IGNITED
FULL_UP_TIME	GA
GAX	GP1
GP2	GPY
GP_ALTITUDE	GP_ATTITUDE
GP_ROTATION	GP_VELOCITY
GQ	GR
GRAVITY	GV
GVE	GVEI
GVI	GW
GWI	INTERNAL_CMD
OMEGA	PE_INTEGRAL
PE_MAX	PE_MIN
TE_DROP	TE_INIT
TE_INTEGRAL	TE_LIMIT
TE_MAX	TE_MIN
VELOCITY_ERROR	YE_INTEGRAL
YE_MAX	YE_MIN

OUTPUT

AE_CMD	AE_STATUS
AE_TEMP	INTERNAL_CMD
PE_INTEGRAL	TE_INTEGRAL

TE_LIMIT	YE_INTEGRAL

PROCESS The reader should refer to section A.9 for notes on integration. Note that once the correct value of AE_CMD has been determined, it will automatically be transmitted to the engines during the next call to the GCS_SIM_RENDEZVOUS routine provided in the GCS_SIM rendezvous package. (See section A.8. Implementation Notes). Computation of the axial engine valve settings requires the following steps:

✓ **PROCESSING WHEN AXIAL ENGINES ARE OFF**

- IF AE_SWITCH is set to OFF, then perform the following steps:
  - •• Set all elements of AE_CMD to 0
  - •• Proceed directly to the step "SET AXIAL ENGINE STATUS TO HEALTHY."

✓ **PROCESSING WHEN AXIAL ENGINES ARE ON**

The variable CL is used here as a subscript. Explanations for the variables CL and VELOCITY_ERROR are provided in functional unit 2.6 GP. The variables PE_INTEGRAL, YE_INTEGRAL, and TE_INTEGRAL will be initialized by INIT_GCS.

- **If AE_SWITCH is set to ON then perform the following steps:**

(Note: p_v, q_v, and r_v are the current elements of GP_ROTATION; $\dot{x}_v$, $\dot{y}_v$, and $\dot{z}_v$ are the current elements of GP_VELOCITY; $\ddot{x}_v$ is the current x component of A_ACCELERATION.)

DETERMINE ENGINE TEMPERATURE

- •• Set AE_TEMP according to Table A.5.1

Table A.5.1 DETERMINATION OF AXIAL ENGINE TEMPERATURE

CURRENT STATE			ACTION
AE_TEMP	GP_ALTITUDE	$(\boldsymbol{FRAME\_COUNTER} - \boldsymbol{FRAME\_ENGINES\_IGNITED}) \cdot \boldsymbol{DELTA\_T}$	AE_TEMP
cold	≤ ENGINES_ON_ALTITUDE	< FULL_UP_TIME	warming-up
warming-up	≤ ENGINES_ON_ALTITUDE	≥ FULL_UP_TIME	hot

COMPUTE LIMITING ERRORS FOR PITCH

- •• $PE\_INTEGRAL = PE\_INTEGRAL + \int_{t_0}^{t} \frac{\dot{z}_v}{|\dot{x}_v|} dt$, where t_0 is the time at the beginning of this frame and t is the time at the end of this frame.

- $P_e^L = GQ(CL) \cdot q_v + GW(CL) \cdot \left(\frac{\dot{z}_v}{|\dot{x}_v|} \right) + GWI(CL) \cdot PE\_INTEGRAL$

- If $P_e^L < PE\_MIN(CL)$ then set P_e^L to PE_MIN(CL).

- If $P_e^L > PE\_MAX(CL)$ then set P_e^L to PE_MAX(CL).

COMPUTE LIMITING ERROR FOR YAW

- • $YE\_INTEGRAL = YE\_INTEGRAL + \int_{t_0}^{t} \frac{\dot{y}_v}{|\dot{x}_v|} dt$,

  where t_0 is the time at the beginning of this frame and t is the time at the end of this frame.

- • $Y_e^L = -GR(CL) \cdot r_v + GV(CL) \cdot \left(\frac{\dot{y}_v}{|\dot{x}_v|} \right) + GVI(CL) \cdot YE\_INTEGRAL$

- • If $Y_e^L < YE\_MIN(CL)$ then set Y_e^L to YE_MIN(CL).

- • If $Y_e^L > YE\_MAX(CL)$ then set Y_e^L to YE_MAX(CL).

COMPUTE LIMITING ERROR FOR THRUST

- • If CONTOUR_CROSSED is set to "contour not crossed", then proceed directly to the step "COMPUTE PITCH, YAW, AND THRUST ERRORS."

- • If CONTOUR_CROSSED is set to "contour crossed", then perform the following steps:

  - • $TE\_INTEGRAL = TE\_INTEGRAL + \int_{t_0}^{t} (VELOCITY\_ERROR) dt$

    where t_0 is the time at the beginning of this frame and t is the time at the end of this frame.
  - • Solve the following equation analytically in order to calculate the value for TE_LIMIT:

    $$\frac{\frac{d}{dt}(TE\_LIMIT) + OMEGA \cdot TE\_LIMIT}{GA} =$$
    $$- GAX \cdot (\ddot{x}_v + GRAVITY \cdot GP\_ATTITUDE(1,3,0)) +$$
    $$GVE \cdot VELOCITY\_ERROR + GVEI(CL) \cdot TE\_INTEGRAL$$

  - • If $TE\_LIMIT < TE\_MIN(CL)$ then set TE_LIMIT to TE_MIN(CL).

  - • If $TE\_LIMIT > TE\_MAX(CL)$ then set TE_LIMIT to TE_MAX(CL).

COMPUTE PITCH, YAW, AND THRUST ERRORS

- Compute pitch error (P_e), Yaw Error (Y_e), and Thrust Error (T_e), according to Table A.5.2

Table A.5.2 DETERMINATION OF ERROR TERMS

CURRENT STATE			ACTIONS		
AE_SWITCH	CHUTE_ RELEASED	CONTOUR_ CROSSED	P_e	Y_e	T_e
1	1	1	P_e^L	Y_e^L	TE_LIMIT
1	1	0	P_e^L	Y_e^L	TE_DROP
1	0	0,1	$\mathrm{GQ(CL)} \cdot q_v$	$-\mathrm{GR(CL)} \cdot r_v$	TE_INIT

COMPUTE AXIAL ENGINE VALVE SETTINGS

Given pitch, yaw, and thrust errors, (P_e, Y_e, T_e), the valve settings (AE_CMD) for each of the three main engines are calculated as:

$$\mathrm{INTERNAL\_CMD} = \begin{pmatrix} GP1 & 0 & 1 \\ GP2 & -GPY & 1 \\ GP2 & GPY & 1 \end{pmatrix} \times \begin{pmatrix} P_e \\ Y_e \\ T_e \end{pmatrix}$$

which will result in each element of the INTERNAL_CMD vector being a real value. This value should be converted into an integer value between 0 and 127 and placed into the appropriate element of the AE_CMD vector. The mapping for the conversion from real to integer values for each of the three elements should be as follows:

Table A.5.3 DETERMINATION OF AXIAL ENGINE COMMANDS

CURRENT STATE	ACTIONS
INTERNAL_CMD	AE_CMD
$I < 0.0$	$A = 0$
$0.0 \le I \le 1.0$	$0 \le A \le 127$
$1.0 < I$	$A = 127$

Note: "I" represents the appropriate element of the vector INTERNAL_CMD
"A" represents the appropriate element of the vector AE_CMD

with INTERNAL_CMD between 0 and 1.0 being converted *linearly* to a value of AE_CMD between 0 and 127. Each value for AE_CMD is to be rounded to the nearest integer, where rounding is defined as follows:

Let x represent the real value that is to be rounded

Then, AE_CMD = the integer part of (x+0.5)

✓ **SET AXIAL ENGINE STATUS TO HEALTHY**

- Set AE_STATUS to healthy.

ARSP -- Altimeter Radar Sensor Processing (P-Spec 2.1.2)

PURPOSE The vehicle has one altimeter radar. The ARSP functional unit reads the altimeter counter provided by this radar and converts the data into a measure of distance to the surface.

INPUT

AR_ALTITUDE AR_FREQUENCY K_ALT	AR_COUNTER AR_STATUS

OUTPUT

AR_ALTITUDE K_ALT	AR_STATUS

PROCESS The processing of the altimeter counter data (AR_COUNTER) into the vehicle's altitude above the planet's terrain depends on whether or not an echo is received by the altimeter radar for the current time step. The distance covered by the radio pulses emitted from the altimeter radar is directly proportional to the time between transmission and reception of its echo. A digital counter (AR_COUNTER) is started as the radar pulse is transmitted. The counter increments AR_FREQUENCY times per second. If an echo is received, the lower order fifteen bits of AR_COUNTER contain the pulse count, and the sign bit will contain the value zero. If an echo is not received, AR_COUNTER will contain sixteen one bits.

✓ **ROTATE VARIABLES**

- Rotate AR_ALTITUDE, AR_STATUS, AND K_ALT.

✓ **DETERMINE ALTITUDE**

- If an echo is received, perform the following:
  - •• Convert the AR_COUNTER value to a distance to be returned in the variable AR_ALTITUDE according to the following equation:

$$\mathrm{AR\_ALTITUDE} = \frac{\mathrm{AR\_COUNTER} \cdot 3 \times 10^{8} \frac{m}{\sec}}{\mathrm{AR\_FREQUENCY} \cdot 2}$$

- If an echo is not received, compute AR_ALTITUDE as follows:
  - •• If all four previous values of AR_STATUS are healthy:
    - ••• In order to smooth the estimate of altitude, fit a third-order polynomial to the previous four values of AR_ALTITUDE.
    - ••• Use this polynomial to extrapolate a value for AR_ALTITUDE for the current time step.
  - •• If any of the previous four values of AR_STATUS is failed:
    - ••• Set the current value of AR_ALTITUDE equal to the previous value of AR_ALTITUDE.

✓ **SET ALTIMETER RADAR STATUS**

- Set the current values for AR_STATUS and K_ALT according to TABLE A.5.4.

Table A.5.4 DETERMINATION OF ALTITUDE STATUS

CURRENT STATE		ACTIONS	
ECHO RETURNED?	All 4 previous AR_STATUS values healthy?	AR_STATUS	K_ALT
yes	d	healthy	1
no	yes	failed	1
no	no	failed	0

Note: "d" = don't care condition

ASP -- Accelerometer Sensor Processing (P-Spec 2.1.1

PURPOSE Three accelerometers, located at the vehicle's center of gravity, are slightly misaligned along the vehicle's $\vec{x}_v$, $\vec{y}_v$, and $\vec{z}_v$ axes. Each accelerometer produces a 16-bit binary value (A_COUNTER), represented as the magnitude portion of a sign magnitude number which is a linear function of the acceleration along its axis. The sign of the counter will always be positive, but the offset given in A_BIAS will be negative or zero. The Acceleration Sensor Processing (ASP) functional unit provides measures of the vehicle accelerations through the conversion and digital filtering of this raw accelerometer data.

INPUT

ALPHA_MATRIX	ATMOSPHERIC_TEMP
A_ACCELERATION	A_BIAS
A_COUNTER	A_GAIN_0
A_SCALE	A_STATUS
G1	G2

OUTPUT

A_ACCELERATION	A_STATUS

PROCESS The processing of the accelerometer data (A_COUNTER) into vehicle accelerations (A_ACCELERATION) requires the following steps:

✓ **ROTATE VARIABLES**

- Rotate A_ACCELERATION and A_STATUS.

✓ **ADJUST GAIN FOR TEMPERATURE**

The standard gain (A_GAIN_0) must be adjusted for the effects of temperature prior to the conversion of the raw accelerometer values. The adjusted gain is a quadratic function of the ambient temperature (ATMOSPHERIC_TEMP) and the standard gain.

- Adjust the gain for temperature as follows:

$$\begin{aligned}\text{A\_GAIN}(i) = \text{A\_GAIN\_0}(i) &+ (\text{G1} \cdot \text{ATMOSPHERIC\_TEMP}) \\ &+ (\text{G2} \cdot \text{ATMOSPHERIC\_TEMP}^2)\end{aligned}$$

where i ranges from 1 to 3 and represents the three directions x, y, and z, and where A_GAIN_0 is the standard gain.

✓ **REMOVE CHARACTERISTIC BIAS**

Each accelerometer has a characteristic DC bias (A_BIAS) which must be removed from the signal prior to conversion. The acceleration is a linear function of its A_COUNTER value where the gain specifies the slope and the offset (A_BIAS) specifies the intercept.

- Remove the bias as follows:

  A_ACCELERATION_M(i) = A_BIAS(i) + A_GAIN(i) * A_COUNTER(i)

  where i ranges from 1 to 3 and represents the three directions x, y, and z.

✓ **CORRECT FOR MISALIGNMENT**

Each accelerometer is slightly misaligned from the true vehicle axes. The multiplier matrix (ALPHA_MATRIX) which is shown below, is based on small angle approximations and corrects for this misalignment. It is used for transforming the measured acceleration data into the true vehicle accelerations.

$$\text{ALPHA\_MATRIX} = \begin{pmatrix} 1 & -\alpha_{xz} & \alpha_{xy} \\ \alpha_{yz} & 1 & -\alpha_{yx} \\ -\alpha_{zy} & \alpha_{zx} & 1 \end{pmatrix}$$

α_{xy} defines the angle of rotation about the vehicle's $\vec{y}_v$ axis between the $\vec{x}_v$ axis and the misaligned $\vec{x}_v$ axis. The other misalignment angles are defined similarly, based upon a right-handed coordinate system.

- Compute preliminary current value of A_ACCELERATION as follows:

  $$A\_ACCELERATION = ALPHA\_MATRIX \times A\_ACCELERATION\_M$$

✓ **DETERMINE ACCELERATIONS AND ACCELEROMETER STATUS**

The variable A_STATUS is a four-element array in each of the three physical dimensions, and contains the present and previous three values of status for each accelerometer. The variable A_ACCELERATION is a five-element array in each of the three dimensions (x, y, and z). A_ACCELERATION contains the present and previous four values of acceleration.

- The following steps are described for the x axis but should be performed for each axis:
  - If one or more of the previous three values of A_STATUS is unhealthy, leave the current value of A_ACCELERATION unchanged, set the current value of A_STATUS to healthy, and do no further processing for this axis.
  - If all three of the previous values of A_STATUS are healthy and all three of the previous values of A_ACCELERATION are equal to each other, leave the current

value of A_ACCELERATION unchanged, set the current value of A_STATUS to healthy, and do no further processing for this axis.

- If all three of the previous values of A_STATUS are healthy, and it is not true that all three of the previous values of A_ACCELERATION are equal to each other, check for extreme values and set A_STATUS and A_ACCELERATION according to the method described below. The accelerometer processing includes filtering of the calculated accelerations along each axis (i.e. filtering of $(\ddot{x}_v, \ddot{y}_v, \ddot{z}_v)_t$), and ignoring or eliminating calculated accelerations which are out of range. To effect this filtering, the means and standard deviations for each component of acceleration are to be computed using the calculated accelerations from the previous three time steps. That is, for the current time step t and the measurement of acceleration along the x axis:
  - Calculate

    $$\hat{\mu} = \sum_{i=t-3}^{t-1} \frac{\ddot{x}_{v(i)}}{3}$$

    which is the current sample mean

  - Calculate

    $$\hat{\sigma} = \sqrt{\frac{\sum_{i=t-3}^{t-1} \left(\ddot{x}_{v(i)} - \hat{\mu}\right)^2}{3}}$$

    which is the current sample standard deviation.

  - If $\left|\hat{\mu} - \ddot{x}_v(t)\right| > \text{A\_SCALE} \cdot \hat{\sigma}$

    set $\ddot{x}_v(t)$ to $\hat{\mu}$

    set A_STATUS to unhealthy

    where $\ddot{x}_v(t)$ is the acceleration along the x axis for the current time step. Similar equations hold for eliminating outliers in the measures of acceleration along the y and z axes.

    otherwise

    set A_STATUS to healthy

  In summary, if the calculated acceleration for the current time step for any component differs from the mean by more than A_SCALE times the standard deviation, then that component of acceleration should be replaced by its current mean and A_STATUS should be set to unhealthy.

If the calculated acceleration for any component is within the specified range of the mean, then the preliminary value of A_ACCELERATION should remain unchanged and A_STATUS should be set to healthy.

CP -- Communications Processing (P-Spec 2.4)

PURPOSE Data from the vehicle sensors and guidance processor is relayed back to the orbiting platform for later analysis. The CP functional unit converts the sensed data into a data packet appropriate for radio transmission.

INPUT

AE_CMD	AE_STATUS
AE_TEMP	AR_ALTITUDE
AR_STATUS	ATMOSPHERIC_TEMP
A_ACCELERATION	A_STATUS
CHUTE_RELEASED	COMM_SYNC_PATTERN
CONTOUR_CROSSED	C_STATUS
FRAME_COUNTER	GP_ALTITUDE
GP_ATTITUDE	GP_PHASE
GP_ROTATION	GP_VELOCITY
G_ROTATION	G_STATUS
K_ALT	K_MATRIX
PE_INTEGRAL	RE_CMD
RE_STATUS	SUBFRAME_COUNTER
TDLR_STATE	TDLR_STATUS
TDLR_VELOCITY	TDS_STATUS
TD_SENSED	TE_INTEGRAL
TS_STATUS	VELOCITY_ERROR
YE_INTEGRAL	

OUTPUT

C_STATUS	PACKET

PROCESS The data packet (PACKET) prepared for transmission is organized to sequentially contain a message and a checksum. The message consists of the synchronization pattern, sequence number, sample mask, and data section. The data packet created will automatically be transmitted during the next call to GCS_SIM_RENDEZVOUS.

✓ **SET COMMUNICATOR STATUS TO HEALTHY**

- Set C_STATUS to healthy.

The construction of the packet requires the following steps:

✓ **CONSTRUCT PACKET**:

- GET SYNCHRONIZATION PATTERN

The synchronization pattern is provided in the variable COMM_SYNC_PATTERN. It is a 16-bit pattern dictated by the design of the receiving communications equipment.

- DETERMINE SEQUENCE NUMBER

  The sequence number identifies the packet of data that is being sent. It is a byte value in the range 0..255. The sequence number will be 0 during the first subframe of frame number 1. Sequence numbers increase by one every subframe, except that the values repeat after the 256th packet. The sequence number can be calculated based on the values of the variables FRAME_COUNTER and SUBFRAME_COUNTER.

- PREPARE SAMPLE MASK

  The sample mask is a Boolean vector where "ones" represent variables that have been sampled since the previous transmission. Any variables listed in Table A.5.5 that may have changed during the present subframe should be marked in the mask and transmitted, with one exception. The variable TE_INTEGRAL may be changed by GP in the second subframe and by AECLP in the third subframe; however, TE_INTEGRAL should be transmitted by CP only during the third subframe, and not during the second subframe. In the case of any "history variable", that is, one which contains a time dimension, only the object (scalar, vector, or array) with a time subscript of zero should be transmitted. Each bit position in the mask represents a particular variable listed in Table A.5.5. The leftmost bit of the mask corresponds to AE_CMD, and moving across the mask from left to right, the next mask bit corresponds to the next variable in Table A.5.5 (in row order).

- PREPARE DATA SECTION

  The data section of the packet contains the sixteen bit values for the elements of the variables in Table A.5.5 that may have new samples available. Once it has been determined which variables should be transmitted for this particular subframe, those variables should be packed into the data section. Although the length of the variable PACKET is fixed, the number of bytes of PACKET which contain actual variables to be transmitted will vary depending on the values of FRAME_COUNTER and SUBFRAME_COUNTER. The variables to be transmitted should be concatenated so that there are no unused bytes between the data to be transmitted. There may however be unused bytes following the checksum. The data are concatenated in the order given by the sample mask, starting with the most significant bit (i.e. left most bit). Variables should be packed to the nearest byte boundary; thus, a single element of PACKET could contain a logical*1 and the first byte of the variable that follows it. Arrays should be sent with the first index changing most rapidly. It should be noted that some arrays have terms that are constant (e.g. the off-diagonal terms of K_MATRIX and the diagonal terms of GP_ROTATION) and since these terms can never have "new" values, they should not be transmitted. The values in Table A.5.5 should be sent in row order, starting at the top of the table. The first value in alphabetical order goes next to the mask in the packet.

- CALCULATE CHECKSUM

  The checksum is calculated for the message using the standard CRC-16 polynomial as defined in (ref. A.11). Table A.5.7 illustrates the byte structure of the packet. The unused part of PACKET should be ignored in the calculation of the checksum. The checksum should be placed in the two bytes immediately following the message for this subframe. Refer to Appendix D for a detailed description of the packet and for specific instructions on the checksum calculation.

Table A.5.5 PACKET VARIABLES

AE_CMD	AE_STATUS	AE_TEMP
AR_ALTITUDE	AR_STATUS	ATMOSPHERIC_TEMP
A_ACCELERATION	A_STATUS	CHUTE_RELEASED
CONTOUR_CROSSED	C_STATUS	GP_ALTITUDE
GP_ATTITUDE	GP_PHASE	GP_ROTATION
GP_VELOCITY	G_ROTATION	G_STATUS
K_ALT	K_MATRIX	PE_INTEGRAL
RE_CMD	RE_STATUS	TDLR_STATE
TDLR_STATUS	TDLR_VELOCITY	TDS_STATUS
TD_SENSED	TE_INTEGRAL	TS_STATUS
VELOCITY_ERROR	YE_INTEGRAL	

Note**:** when read by rows, this table represents the alphabetical listing of variables that are to appear in the data section of the packet.

Table A.5.6 SAMPLE MASK

INFORMATION SENT	A	B	C	...	Z
EXAMPLE MASK	1	1	0	...	1

Note: this table gives information only on the order of the packet. The packet should be packed to a byte-boundary limit into integer*2 elements.

Table A.5.7 PACKET BYTE STRUCTURE

Subframe 1 Byte Positions	Subframe 2 Byte Positions	Subframe 3 Byte Positions	CONTENTS (Cells in bold italics with double-line border constitute the
1 2	1 2	1 2	***SYNCHRONIZATION PATTERN***
3	3	3	***SEQUENCE NUMBER***
4 5 6 7	4 5 6 7	4 5 6 7	***SAMPLE MASK***
8 . . . 129	8 . . . 173	8 . . . 45	***DATA SECTION***
130 131	174 175	46 47	CHECKSUM
132 . . 512	176 . . 512	48 . . 512	NOT USED

Note: The variables inserted into PACKET are inserted in the VAX standard byte order.

CRCP -- Chute Release Control Processing (P-Spec 2.3.3)

PURPOSE The CRCP functional unit implements the release of the parachute which is attached prior to the beginning of the terminal descent phase.

INPUT

AE_TEMP	CHUTE_RELEASED

OUTPUT

CHUTE_RELEASED

PROCESS If the chute has been released, leave CHUTE_RELEASED unchanged and this signal will be automatically transmitted to the chute release mechanism during the next call to GCS_SIM_RENDEZVOUS. If the chute has not been released, the engine temperature will determine whether or not to release the chute. If the chute has not been released and the engines are hot (i.e. AE_TEMP is HOT), then release the chute by setting CHUTE_RELEASED to "chute released."

GP -- Guidance Processing (P-Spec 2.2)

PURPOSE GP uses the information available from ASP, ARSP, CRCP, GSP, TDLRSP, and TDSP and the results of its previous computations to control the vehicle's state during terminal descent.

INPUT

AE_SWITCH	AE_TEMP
AR_ALTITUDE	A_ACCELERATION
CHUTE_RELEASED	CL
CONTOUR_ALTITUDE	CONTOUR_CROSSED
CONTOUR_VELOCITY	DELTA_T
DROP_HEIGHT	DROP_SPEED
ENGINES_ON_ALTITUDE	FRAME_COUNTER
GP_ALTITUDE	GP_ATTITUDE
GP_PHASE	GP_VELOCITY
GRAVITY	G_ROTATION
K_ALT	K_MATRIX
MAX_NORMAL_VELOCITY	RE_SWITCH
TDLR_VELOCITY	TDS_STATUS
TD_SENSED	

OUTPUT

AE_SWITCH	CL
CONTOUR_CROSSED	FRAME_ENGINES_IGNITED
GP_ALTITUDE	GP_ATTITUDE
GP_PHASE	GP_ROTATION
GP_VELOCITY	RE_SWITCH
TE_INTEGRAL	VELOCITY_ERROR

ARRAYS The variables GP_ATTITUDE, GP_ALTITUDE, and GP_VELOCITY are five element arrays in each of their history dimensions and contain enough previous values to provide the required history for integration in updating the vehicle and guidance states.

PROCESS The Guidance Processor computes the velocity, altitude, and attitude to be used in controlling the engines.

✓ **ROTATE VARIABLES**

- Rotate GP_ATTITUDE, GP_ALTITUDE, and GP_VELOCITY.

✓ **SET UP THE GP_ROTATION MATRIX**

G_ROTATION contains three values: p, q, and r, in that order. These values must be placed into a 3 x 3 matrix (GP_ROTATION) in the correct positions for later calculations. Note that GP_ROTATION does not include any time histories; thus it may be convenient to use a temporary variable during calculation to hold the time histories of GP_ROTATION or to use elements directly from G_ROTATION; however, GP_ROTATION does describe the correct matrix orientation for operations and upon exiting from GP should contain the correct values for the present time step.

- Place the values from G_ROTATION into GP_ROTATION as shown:

$$\mathrm{GP\_ROTATION} = \begin{pmatrix} 0 & r_v & -q_v \\ -r_v & 0 & p_v \\ q_v & -p_v & 0 \end{pmatrix}$$

✓ **CALCULATE NEW VALUES OF ATTITUDE, VELOCITY, AND ALTITUDE**

The attitude, velocity, and altitude are each calculated by:

1. finding a rate of change from known values, and then
2. integrating this rate of change through one time step by some method of integration providing the accuracy specified. That is:

$$X_t = X_{t-1} + \int_{t-1}^{t} \dot{X} dt$$

where $\dot{X}$ represents the rate of change of attitude, velocity, or altitude.

Table A.5.8 gives the equations for the rates of change for each of the variables GP_ATTITUDE, GP_VELOCITY, and GP_ALTITUDE.

- Solve for the current values of GP_ATTITUDE, GP_VELOCITY, and GP_ALTITUDE using the equation for X_t given above, Table A.5.8, and an appropriate integration method (see section A.9 *Numerical Integration Instructions*).

Table A.5.8 DIFFERENTIAL EQUATIONS

$\frac{d(GP\_ATTITUDE)}{dt} = GP\_ROTATION \times GP\_ATTITUDE$
$\frac{d(GP\_VELOCITY)}{dt} = GP\_ROTATION \times GP\_VELOCITY + GP\_ATTITUDE \times \begin{pmatrix} 0 \\ 0 \\ GRAVITY \end{pmatrix} + A\_ACCELERATION + K\_MATRIX \times (TDLR\_VELOCITY - GP\_VELOCITY)$
$\frac{d(GP\_ALTITUDE)}{dt} = -\left(GP\_ATTITUDE \times \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \right)^T \times GP\_VELOCITY + K\_ALT \cdot (AR\_ALTITUDE - GP\_ALTITUDE)$

✓ **DETERMINE IF ENGINES SHOULD BE ON OR OFF**

Note that RE_SWITCH is initialized to on, while AE_SWITCH is initialized to off, and FRAME_ENGINES_IGNITED is initialized by INIT_GCS. Use Table A.5.9 to determine whether to turn axial engines on (set AE_SWITCH to on and set FRAME_ENGINES_IGNITED) or whether to turn axial and roll engines off (set AE_SWITCH and RE_SWITCH to off).

TABLE A.5.9 DETERMINATION OF AXIAL AND ROLL ENGINE ON/OFF SWITCHES

CURRENT STATE					ACTIONS		
AE_ SWITCH	GP_ ALTITUDE	$(\sqrt{velocity\_expression}$[1] $+$ $x component of\ GP\_VELOCITY) \leq MAX\_NORMAL\_VELOCITY$?	Have engines been turned off in any prior frame?	TD_ SENSED	FRAME_ ENGINES_ IGNITED	AE_ SWITCH	RE_ SWITCH
off	≤ ENGINES_ON_ ALTITUDE	d	no	not sensed	current FRAME_ COUNTER	on	
on	≤ DROP_ HEIGHT	yes	d	not sensed		off	off
on	d	d	d	sensed		off	off

[1] $velocity\_expression = 2 \cdot GRAVITY \cdot \text{maximum}(GP\_ALTITUDE, 0)$

Note: A blank box under "ACTIONS" indicates no action is to be taken

"d" = don't care condition

✓ **DETERMINE VELOCITY ERROR**

The velocity-altitude contour consists of a set of points of which one coordinate is the altitude of the craft and the other coordinate is the optimal x component of velocity at the altitude given by the first coordinate. The altitude and optimal velocity coordinates are held in the CONTOUR_ALTITUDE and CONTOUR_VELOCITY arrays respectively. The altitude coordinates are in the CONTOUR_ALTITUDE array contiguous to each other, in ascending numerical order, beginning with the first element of the array. Any unused elements of the array have been filled with zeroes (the value of zero will not be used as an actual value for altitude). There are at least two valid non-zero altitude values in the table. The two arrays are related such that for a given value of altitude in CONTOUR_ALTITUDE, the corresponding value in CONTOUR_VELOCITY is the optimal velocity x component at that altitude. For any altitude that is not explicitly listed in CONTOUR_ALTITUDE, the value for optimal velocity can be found by linear interpolation (or extrapolation if the value is outside the range of the altitude array). The velocity error (VELOCITY_ERROR) is the difference between the actual x component of the velocity of the craft (GP_VELOCITY) and the optimal velocity x component at the vehicle altitude. Figure A.5.1 illustrates the velocity-altitude contour.

- The optimal velocity should be calculated by finding the present altitude in CONTOUR_ALTITUDE and then locating the corresponding velocity in CONTOUR_VELOCITY, using interpolation or extrapolation if necessary. Let *optimal_velocity* represent the value obtained from the contour arrays, whether extracted, interpolated, or extrapolated.

- Calculate VELOCITY_ERROR as follows:

  VELOCITY_ERROR = x component of GP_VELOCITY - *optimal_velocity*

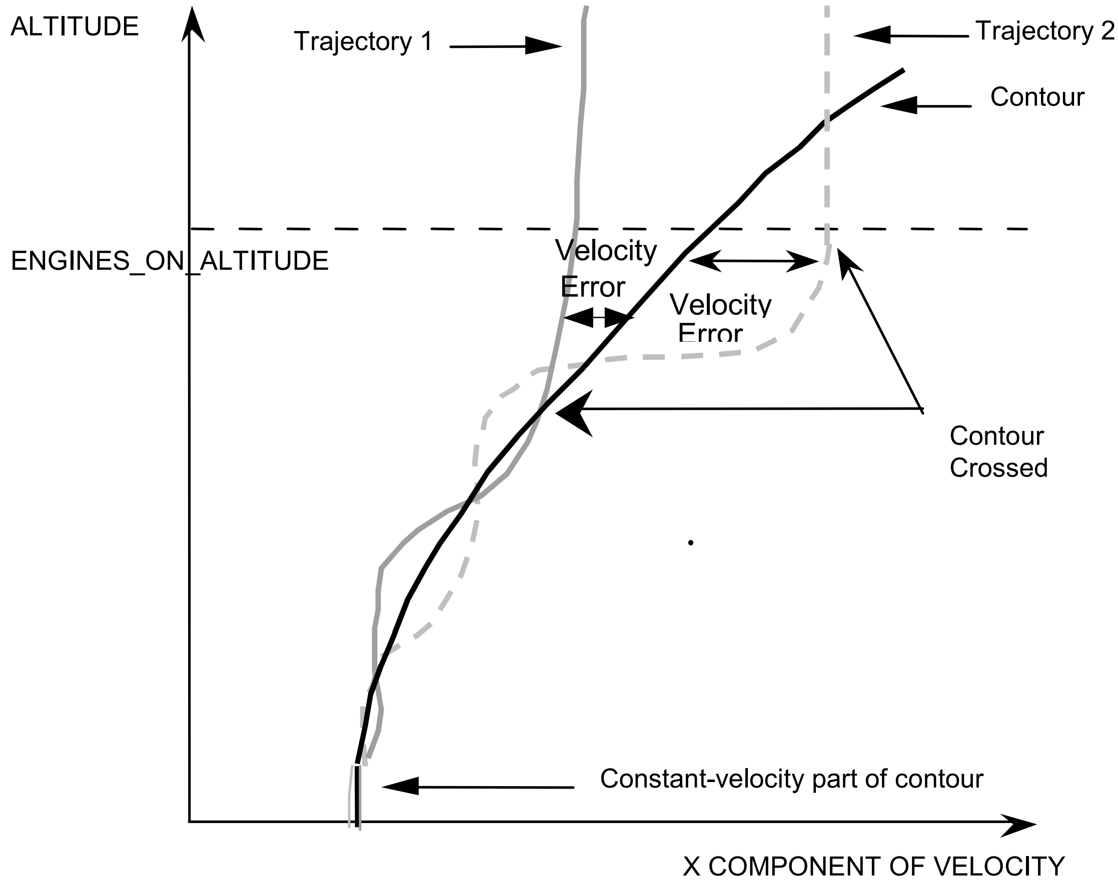
✓ **DETERMINE IF CONTOUR HAS BEEN CROSSED**

- If GP_ALTITUDE ≤ ENGINES_ON_ALTITUDE, then check whether the contour has been crossed as follows:

  •• If CONTOUR_CROSSED = "contour not crossed" and VELOCITY_ERROR ≥ 0, then set CONTOUR_CROSSED to "contour crossed". Otherwise CONTOUR_CROSSED should remain unchanged.

Figure A.5.1 shows two possible trajectories, with the point along each where the contour is first sensed and also an example of VELOCITY_ERROR. Note: the altitude where the engines are turned on should be the earliest point to check crossing the contour, even though the trajectory may have crossed the contour at some greater altitude.

Figure A.5.1 VELOCITY-ALTITUDE CONTOUR



✓ **DETERMINE GUIDANCE PHASE**

- The guidance phase (GP_PHASE) is determined according to the events in Table A.5.10. These phases are based upon information that may be provided by processes other than the guidance processor.

  The current phase (GP_PHASE) and the event are to be used where appropriate to reset GP_PHASE to the next phase. If there is no combination of current phase and event from the table that is true, then GP_PHASE should not be changed. Note that the two columns labeled "CURRENT STATE DESCRIPTION" and "NEXT STATE DESCRIPTION" are for informational purposes only, and are not used in the setting of GP_PHASE.

Table A.5.10 DETERMINATION OF GUIDANCE PHASE

CURRENT STATE			NEXT STATE	
			ACTION	
GP_ PHASE	CURRENT STATE DESCRIPTION	EVENT	GP_ PHASE	NEXT STATE DESCRIPTION
1	Chute attached Engines off Touch Down not sensed	Altitude for turning engines on is sensed	2	Chute attached Engines on Touch down not sensed
2	Chute attached Engines on Touch down not sensed	Axial Engines become hot and the chute is released	3	Chute released Axial Engines Hot Touch down not sensed
2	Chute attached Engines on Touch down not sensed	Touched down is sensed	**5**	Chute attached Engines off Touch down sensed
3	Chute released Axial Engines Hot Touch down not sensed	Altitude ≤ DROP_HEIGHT and TDS_STATUS = healthy and Touch down not sensed and $(\sqrt{\text{velocity\_expression}^1} + \text{xcomponentof GP\_VELOCITY}) \leq \text{MAX\_NORMAL\_VELOCITY}$	4	Chute released Engines off Touch down not sensed
3	Chute released Axial Engines Hot Touch down not sensed	Altitude ≤ DROP_HEIGHT and TDS_STATUS = failed	**5**	Chute released Engines off Touch down not sensed
3	Chute released Axial Engines Hot Touch down not sensed	Touch down is sensed	**5**	Chute released Engines off Touch down sensed
4	Chute released Engines off Touch down not sensed	Touch down is sensed	**5**	Chute released Engines off Touch down sensed
4	Chute released Engines off Touch down not sensed	TDS_STATUS = failed	**5**	Chute released Engines off Touch down not sensed

[1] $velocity\_expression = 2 \cdot GRAVITY \cdot \text{maximum}(GP\_ALTITUDE, 0)$

- PHASE 1: If the altitude provided by the guidance processor is less than or equal to the ENGINES_ON_ALTITUDE, set GP_PHASE = 2.
- PHASE 2: If the axial engines have become hot and the parachute has been released, set GP_PHASE = 3. If touch down is sensed, set GP_PHASE = 5.
- PHASE 3: If touch down has not been sensed and DROP_HEIGHT has not been reached, then control the axial and roll engines to cause the lander to follow a gravity-turn steering descent. If DROP_HEIGHT is reached and touch down is not sensed and

$$\sqrt{2 \cdot GRAVITY \cdot \text{maximum}(GP\_ALTITUDE, 0)} + x \ component \ of \ GP\_VELOCITY$$
$$\leq MAX\_NORMAL\_VELOCITY$$

and TDS_STATUS = healthy, then set GP_PHASE = 4. If DROP_HEIGHT is reached, and TDS_STATUS = failed, then set GP_PHASE = 5. If touch down is sensed, then set GP_PHASE = 5.

- PHASE 4: If touch down has not been sensed and TDS_STATUS is healthy, then take no action. If TDS_STATUS is failed, then set GP_PHASE to 5. If touch down has been sensed, set GP_PHASE to 5.

✓ **DETERMINE WHICH SET OF CONTROL LAW PARAMETERS TO USE**

The "Control Law Parameters" are a subset of the variables in the global data store named "RUN_PARAMETERS." This subset consists of the following variables: GVEI, GV, GVI, GR, GW, GWI, GQ, PE_MIN, PE_MAX, TE_MIN, TE_MAX, YE_MIN, and YE_MAX. Note that each one of these variables is an array of two elements. The elements with a subscript of one will be referred to as the "first" set of Control Law Parameters, while the elements with a subscript of two will be referred to as the "second" set of Control Law Parameters.

The variable CL is used to control which set of Control Law Parameters is used in the control laws at any given time by the functional unit AECLP. The functional unit GP must determine the value of CL for use by AECLP. The variable CL has two valid values, namely "first" which means that the first set of Control Law Parameters should be used by AECLP, and "second" which means that the second set of Control Law Parameters should be used by AECLP in the equations for P_e , Y_e , P_e^L , Y_e^L , and TE_LIMIT. See the Data Requirements Dictionary for the actual numeric values for CL which correspond to "first" and "second." The variable CL is initialized to the value "first" by INIT_GCS, and thus the first set of parameters will be used by AECLP until CL is changed. The second set of Control Law Parameters should be used by AECLP at the first point where the lander crosses the constant-velocity part of the Velocity-Altitude contour. The constant-velocity part of the contour consists of the four sets of coordinates with the smallest altitudes and for which the CONTOUR_VELOCITY elements are exactly equal to the value DROP_SPEED. The GUIDANCE PROCESSOR (GP) must determine when to begin using the second set of Control Law Parameters, as follows:

- If the following conditions are true:

  CL = first, and

  optimal_velocity = DROP_SPEED, and

  x component of GP_VELOCITY < DROP_SPEED

  Then

  Set CL = second

  Set TE_INTEGRAL = 0.0

GSP -- Gyroscope Sensor Processing (P-Spec 2.1.4)

PURPOSE Three fiber-optic ring gyroscopes are located on the lander, one for each of the *x, y,* and *z* axes. The Gyroscope Sensor Processing (GSP) functional unit provides a measure of the vehicle's rotation rates through the conversion and filtering of the raw gyroscope data.

INPUT

ATMOSPHERIC_TEMP	G3
G4	G_COUNTER
G_GAIN_0	G_OFFSET
G_ROTATION	

OUTPUT

G_ROTATION	G_STATUS

PROCESS The output from each of the gyroscopes is a 16-bit quantity (G_COUNTER) divided into 2 parts: the lower 14 bits represent the vehicle's rate of rotation about that axis and the high-order bit represents the direction of this rotation. This is a sign-magnitude representation of the counter value that only uses the lower 14 bits of the magnitude portion of the number. Following is a map of G_COUNTER:

15	14	13	12	11	...	0
D	X	MAGNITUDE				

where D = direction, and X = unused. The high bit set to 1 indicates a negative rotation consistent with a right-handed coordinate system.

✓ **ROTATE VARIABLES**

- Rotate G_ROTATION.

✓ **ADJUST GAIN**

The standard gain (G_GAIN_0) must be adjusted for the effects of temperature prior to the conversion of the raw gyroscope values. The adjusted gain is a quadratic function of the ambient temperature (ATMOSPHERIC_TEMP) and the standard gain.

That is,

$$\begin{aligned} \mathrm{G\_GAIN}(i) = \mathrm{G\_GAIN\_0}(i) &+ (\mathrm{G3} \cdot \mathrm{ATMOSPHERIC\_TEMP}) \\ &+ (\mathrm{G4} \cdot \mathrm{ATMOSPHERIC\_TEMP}^2) \end{aligned}$$

where i ranges from 1 to 3 and represents the three directions x, y, and z.

✓ **CONVERT G_COUNTER**

The rotation rate is linear with respect to the unprocessed gyroscope values, i.e. the lower 14 bits must be converted. G_GAIN is the multiplier for this conversion and G_OFFSET is the constant offset. The equation for converting counter to rotation then becomes:

G_ROTATION(*i*) = G_OFFSET(*i*) + G_GAIN(*i*) * (G_COUNTER(*i*))

where *i* ranges from 1 to 3 and represents the three directions x, y, and z.

✓ **SET GYROSCOPE STATUS TO HEALTHY.**

- Set G_STATUS to healthy.

RECLP -- Roll Engine Control Law Processing (P-Spec 2.3.2)

PURPOSE RECLP generates the roll engine command which controls the firing pulse and direction of the roll engines.

INPUT

DELTA_T	G_ROTATION
P1	P2
P3	P4
RE_SWITCH	THETA
THETA1	THETA2

OUTPUT

RE_CMD	RE_STATUS
THETA	

PROCESS Roll control of the lander is achieved by generating the roll commands as functions of the differences between the actual and desirable values for the roll angle and rate. These differences are limited, and the control commands are proportional to them. Note that once the roll command (RE_CMD) has been set with the correct value, it will automatically be sent to the engines during the next call to GCS_SIM_RENDEZVOUS. The steps to be performed are as follows:

✓ **DETERMINE IF ENGINES ARE ON**

- If RE_SWITCH is off, then set RE_CMD to 1, and proceed directly to the step "SET ROLL ENGINE STATUS TO HEALTHY."

✓ **DETERMINE PULSE INTENSITY AND DIRECTION**

- The pulse intensity and direction are derived from the graph shown in Figure A.5.2 using $(p_v)_t$. For each region of the graph, the intensity is given, followed by the direction inside parentheses. Note that the x axis represents the integral of the roll rate. This is really the present angle of roll. This integral should be calculated by Euler's method (see section A.9). As an example, THETA = THETA + (integral of roll rate for this frame). The variable THETA will be initialized by INIT_GCS. Note that when the vehicle status is located on a boundary between two or more roll command regions, the lowest intensity signal should be used to avoid over-commanding the engines. One should refer to the Data Requirements Dictionary under RE_CMD for the actual values for intensity and direction.

✓ **DETERMINE ROLL ENGINE COMMAND**

- The pulse intensity and direction are packed into the lowest three lower-order bits of the actual roll engine command (RE_CMD) as shown:

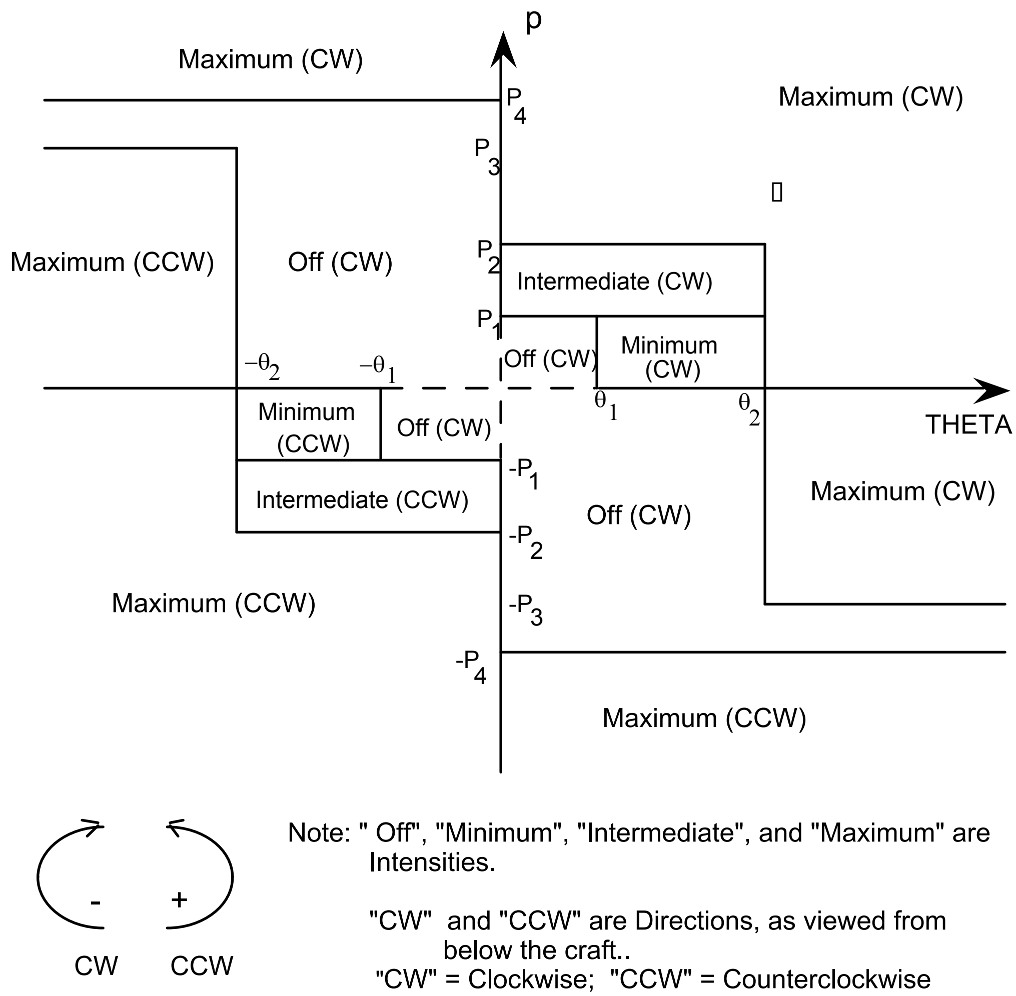
15	14	13	···	3	2	1	0
X	X	X	···	X	I	I	D

where X = unused, I = intensity, and D = direction. The bits marked "X = unused" in RE_CMD must be left at 0.

✓ **SET ROLL ENGINE STATUS TO HEALTHY**

- Set RE_STATUS to healthy.

Figure A.5.2 GRAPH FOR DERIVING ROLL ENGINE COMMANDS



Note: " Off", "Minimum", "Intermediate", and "Maximum" are Intensities.

"CW" and "CCW" are Directions, as viewed from below the craft..
"CW" = Clockwise; "CCW" = Counterclockwise

Note: $P1 < P2 < P3 < P4$ and $\theta 1 < \theta 2$

TDLRSP -- Touch Down Landing Radar Sensor Processing (P-Spec 2.1.3)

PURPOSE A single touch down landing radar (TDLR) gauges the velocity of the vehicle during terminal descent. This radar is a doppler radar with four radar beams, each of which emanates from the vehicle's center of gravity with a slight offset from the vehicle's $\vec{x}_v$ axis. The radar beams form the edges of the pyramid as shown in Figure A.5.3.

The Touch Down Landing Radar Sensor Processing (TDLRSP) functional unit converts measurements of the frequency shift of each beams reflection into vehicle velocities; however, the receivers associated with each beam may not find a usable reflection. If no usable reflection is found, the receiver returns a status of beam in search mode (unlocked).

INPUT

DELTA_T	FRAME_BEAM_UNLOCKED
FRAME_COUNTER	K_MATRIX
TDLR_ANGLES	TDLR_COUNTER
TDLR_GAIN	TDLR_LOCK_TIME
TDLR_OFFSET	TDLR_STATE
TDLR_VELOCITY	

OUTPUT

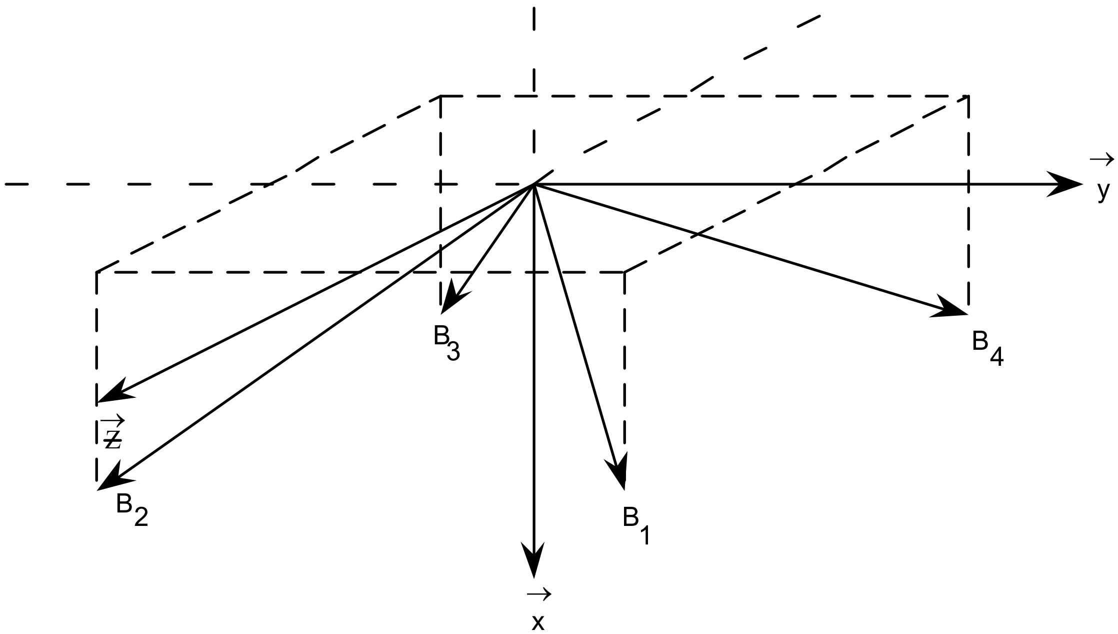
FRAME_BEAM_UNLOCKED	K_MATRIX
TDLR_STATE	TDLR_STATUS
TDLR_VELOCITY	

PROCESS The value returned by each beam (TDLR_COUNTER) is proportional to the beam frequency shift down that beam, which is, in turn, proportional to the velocity down that beam. The processing of the TDLR_COUNTER data into the component velocities along the vehicle's $\vec{x}$, $\vec{y}$, and $\vec{z}$ axes requires the following steps:

✓ **ROTATE VARIABLES**

- Rotate TDLR_VELOCITY and K_MATRIX.

Figure A.5.3 DOPPLER RADAR BEAM LOCATIONS



✓ **DETERMINE RADAR BEAM STATES**

The processing of the four radar beams depends on the current state of the radar, i.e. whether or not each of the four beams is searching or in lock, and also upon the previous states of the beams. Note that at the beginning of each trajectory, FRAME_BEAM_UNLOCKED will be set to zero, thus meaning that the beam has never been unlocked. If the receiver for a beam does not sense an echo (i.e. the beam is in search mode), the corresponding TDLR_COUNTER value will be zero. Note that a beam which becomes unlocked will be ignored for TDLR_LOCK_TIME seconds.

- Use Table A.5.11 to determine the state (TDLR_STATE and FRAME_BEAM_UNLOCKED) for each of the four beams.

Table A.5.11 DETERMINATION OF RADAR BEAM STATES

CURRENT STATE			ACTIONS	
TDLR_ STATE	TDLR_ COUNTER	$DELTA\_T \cdot (FRAME\_COUNTER - FRAME\_BEAM\_UNLOCKED) \geq TDLR\_LOCK\_TIME$?	TDLR_ STATE	FRAME_BEAM_ UNLOCKED
locked	0	d	unlocked	current FRAME_COUNTER
unlocked	$\neq$ 0	yes	locked	
unlocked	0	yes		current FRAME_COUNTER

Note: A blank box under "ACTIONS" indicates no action is to be taken
"d" = don't care condition

✓ **DETERMINE BEAM VELOCITIES**

A beam velocity is a linear function of its TDLR_COUNTER value where the gain (TDLR_GAIN) specifies the slope and the offset (TDLR_OFFSET) specifies the intercept.

- Calculate the beam velocities as follows:

  B(*i*) = TDLR_OFFSET + TDLR_GAIN * (TDLR_COUNTER(*i*))

  where *i* ranges from 1 to 4 and represents the four radar beams.

✓ **PROCESS THE BEAM VELOCITIES**

- Use Table A.5.12 to calculate values for $\hat{B}_x$, $\hat{B}_y$, and $\hat{B}_z$, which are the processed beam velocities. Note that in Table A.5.12, B_i is shorthand for B(*i*), where *i* ranges from 1 to 4. Note also that the knowledge of which beams are in lock is used to determine which line of the table to use in order to calculate $\hat{B}_x$, $\hat{B}_y$, and $\hat{B}_z$.

✓ **CONVERT TO BODY VELOCITIES**

- In order to convert the processed beam velocities to body velocities (TDLR_VELOCITY), use the following equations, which make use of the angles α, β and γ (TDLR_ANGLES) which are the offsets of the beams from the body axes:

$$TDLR\_VELOCITY(1) = \frac{\hat{B}_X}{\cos\alpha}$$

$$TDLR\_VELOCITY(2) = \frac{\hat{B}_y}{\cos\beta}$$

$$TDLR\_VELOCITY(3) = \frac{\hat{B}_z}{\cos\gamma}$$

✓ **SET VALUES IN K_MATRIX**

When calculating the vehicle velocity, the Guidance Processor must know which components of the body velocities are usable. A value of one in the diagonal element of the K_MATRIX indicates that the corresponding velocity should be used, while a value of zero indicates that it should not.

- Use Table A.5.12 to set the values for K_x, K_y, and K_z in K_MATRIX, (again on the basis of which beams are in lock), as follows:

$$K\_MATRIX = \begin{pmatrix} K_x & 0 & 0 \\ 0 & K_y & 0 \\ 0 & 0 & K_z \end{pmatrix}$$

The off-diagonal elements of K_MATRIX should not be updated.

✓ **SET TDLR_STATUS**

- Set all elements of TDLR_STATUS to healthy.

Table A.5.12 PROCESSING OF DOPPLER RADAR BEAMS IN LOCK

CURRENT STATE	A C T I O N S					
BEAMS IN LOCK	$\hat{B}_X$	K_x	$\hat{B}_y$	K_y	$\hat{B}_z$	K_z
none	0	0	0	0	0	0
B_1	0	0	0	0	0	0
B_2	0	0	0	0	0	0
B_3	0	0	0	0	0	0
B_4	0	0	0	0	0	0
B_1, B_2	0	0	$(B_1 - B_2)/2$	1	0	0
B_1, B_3	$(B_1 + B_3)/2$	1	0	0	0	0
B_1, B_4	0	0	0	0	$(B_1 - B_4)/2$	1
B_2, B_3	0	0	0	0	$(B_2 - B_3)/2$	1
B_2, B_4	$(B_2 + B_4)/2$	1	0	0	0	0
B_3, B_4	0	0	$(B_4 - B_3)/2$	1	0	0
B_1, B_2, B_3	$(B_1 + B_3)/2$	1	$(B_1 - B_2)/2$	1	$(B_2 - B_3)/2$	1
B_1, B_2, B_4	$(B_2 + B_4)/2$	1	$(B_1 - B_2)/2$	1	$(B_1 - B_4)/2$	1
B_1, B_3, B_4	$(B_1 + B_3)/2$	1	$(B_4 - B_3)/2$	1	$(B_1 - B_4)/2$	1
B_2, B_3, B_4	$(B_2 + B_4)/2$	1	$(B_4 - B_3)/2$	1	$(B_2 - B_3)/2$	1
B_1, B_2, B_3, B_4	$(B_1 + B_2 + B_3 + B_4)/4$	1	$(B_1 - B_2 - B_3 + B_4)/4$	1	$(B_1 + B_2 - B_3 - B_4)/4$	1

TDSP -- Touch Down Sensor Processing (P-Spec 2.1.6)

PURPOSE The touch down sensor is attached to the end of a rod which is attached to the bottom of the vehicle. Its purpose is to trigger engine shutdown when the vehicle is at the correct distance from the surface. This shutdown is necessary to:

- avoid the stirring up of dust and debris and
- avoid scorching immediate area of the experiment site.

INPUT

TDS_STATUS	TD_COUNTER

OUTPUT

TDS_STATUS	TD_SENSED

PROCESS The touch down sensor is a simple switch at the end of a pole on the underside of the lander. If the sensor is functioning properly, then TD_COUNTER will contain one of only two 16-bit values, namely sixteen "ones", which means that touch down has been sensed, or sixteen "zeroes", which means that touch down has not been sensed. If the sensor has failed due to electrical noise, TD_COUNTER will contain some combination of "ones" and "zeroes" other than all "ones" or all "zeroes".

✓ **DETERMINE STATUS OF TOUCH DOWN SENSOR AND WHETHER TOUCH DOWN HAS BEEN SENSED:**

- Use Table A.5.13 to determine whether the touch down sensor is functioning properly (set TDS_STATUS), and whether touch down has been sensed (set TD_SENSED). Note that if the sensor fails, the guidance processor will decide when the vehicle has touched down.

Table A.5.13 DETERMINATION OF TOUCH DOWN SENSOR AND STATUS

CURRENT STATE		ACTIONS	
TDS_STATUS	TD_COUNTER	TD_SENSED	TDS_STATUS
healthy	all zeroes	not sensed	
healthy	all ones	sensed	
healthy	mixture of ones & zeroes	not sensed	failed

Note: A blank block under "ACTIONS" indicates no action is to be taken

TSP -- Temperature Sensor Processing (P-Spec 2.1.5)

PURPOSE A temperature gauge on the vehicle is used to adjust the response of the accelerometers and gyroscopes. The gauge contains two temperature sensing devices, namely a solid-state sensor and a matched pair of thermocouples. The Temperature Sensor Processing (TSP) functional unit determines the ambient temperature, using either the solid-state sensor or the thermocouple pair in a manner maximizing the accuracy of the measurement.

INPUT

M1	M2
M3	M4
SS_TEMP	T1
T2	T3
T4	THERMO_TEMP

OUTPUT

ATMOSPHERIC_TEMP	TS_STATUS

PROCESS The temperature values from the solid-state sensor are highly quantized. The processing of raw temperature data from the solid-state sensor and thermocouple pair, SS_TEMP and THERMO_TEMP, is based on the solid-state sensor being less accurate than the thermocouple pair, but having a greater usable operating range.

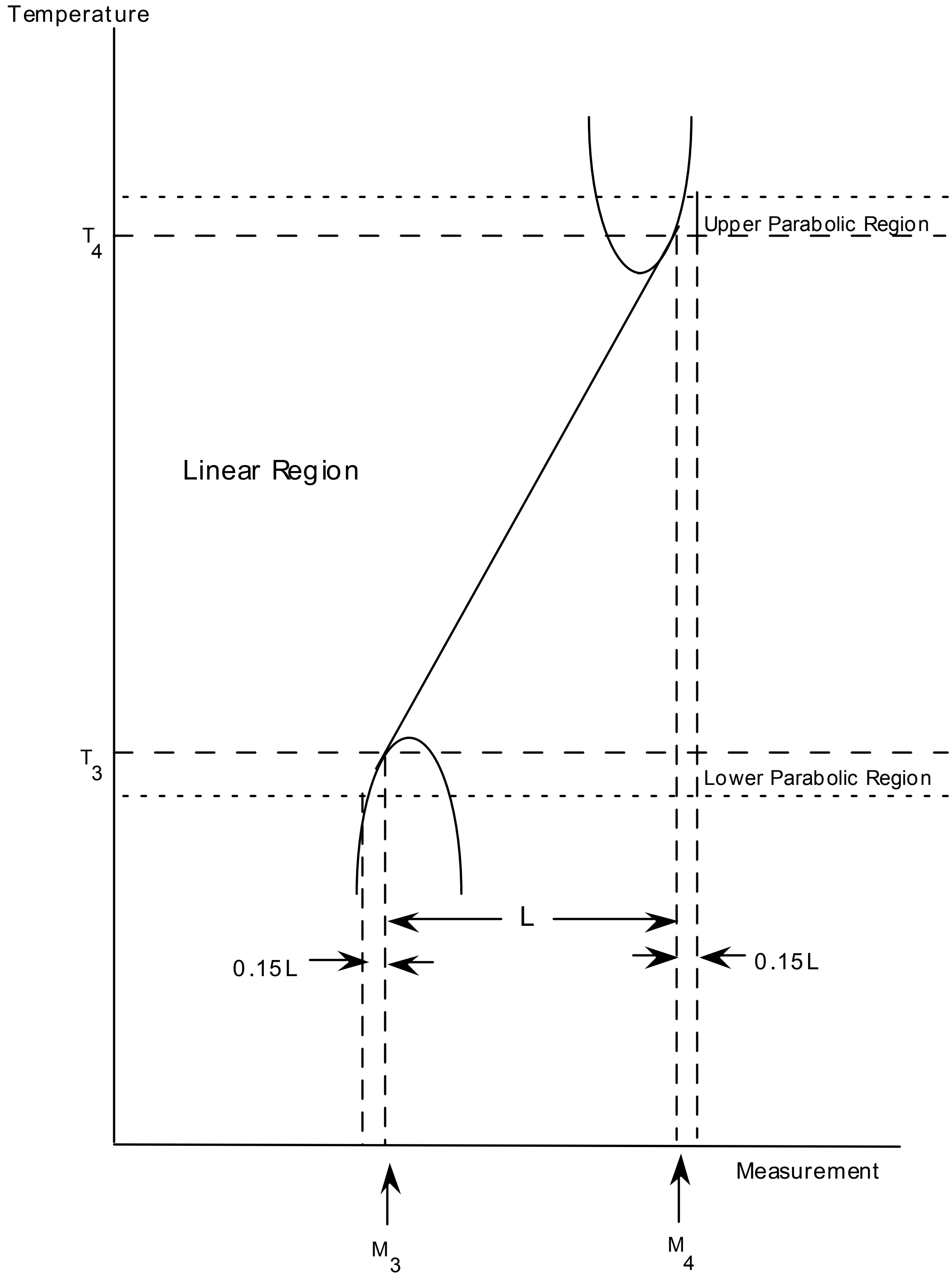
The ambient temperature (ATMOSPHERIC_TEMP) is to be calculated using either the solid state sensor value (SS_TEMP) or the thermocouple sensor value (THERMO_TEMP). Since the thermocouple sensor is more accurate, it should be used whenever possible; the solid state sensor should be used only if the temperature does not lie within the usable range of the thermocouple pair.

The response of the solid-state temperature sensor is linear with respect to the ambient temperature and is computed using the two calibration points (M1, T1) and (M2, T2) which characterize the line.

The response of the thermocouple pair is calibrated differently depending on the region (linear or parabolic) where the measurement lies (see Figure A.5.4):

> Thermocouple linear region - The linear region is bounded by the calibration points used by the thermocouple sensor (i.e., [M3, T3] and [M4, T4] inclusive). Temperatures measured within this region are calibrated accordingly.

Figure A.5.4 CALIBRATION OF THERMOCOUPLE PAIR



Note: $M_3 < M_4$ and $T_3 < T_4$

Thermocouple parabolic regions - The upper and lower parabolic regions extend plus or minus 15 percent of the difference between the measured calibration points, M4 and M3, respectively. These parabolic regions each intersect the line at the calibration points. The rate of change in temperature, with respect to the thermocouple measurements, is continuous at these intersections. The upper (and lower) parabolas are defined so that the temperature goes up (or down) as the square of the measurement value (THERMO_TEMP). The parabolas are offset along both the temperature and measurement axes. By using the values of T3, T4, M3, and M4, and the fact that the function is continuous at the endpoints, the offsets for the parabolas may be determined, and the equations for the parabolas may be generated. Note that the line in the linear region in Figure A.5.4 is tangent to both parabolas.

The processing of the values SS_TEMP and THERMO_TEMP into an accurate measure of ambient temperature (ATMOSPHERIC_TEMP) requires several steps, as follows:

✓ **CALCULATE THE SOLID STATE TEMPERATURE**

- Use the value of SS_TEMP and the equation appropriate to the solid-state linear region to compute the temperature.

✓ **DETERMINE WHETHER TO USE SOLID STATE OR THERMOCOUPLE TEMPERATURE**

- If the temperature derived from SS_TEMP in the previous step does not fall within the accurate temperature response zone of the thermocouple pair (the linear as well as parabolic regions), then set ATMOSPHERIC_TEMP to the temperature derived from SS_TEMP and proceed directly to the step labeled "SET STATUS TO HEALTHY"; otherwise, proceed to the step "CALCULATE THE THERMOCOUPLE TEMPERATURE".

✓ **CALCULATE THE THERMOCOUPLE TEMPERATURE**

- Use the value of THERMO_TEMP to determine whether the temperature lies in the thermocouple linear or the upper parabolic or the lower parabolic region.
- Use the value of THERMO_TEMP and the equation appropriate to the particular thermocouple region (as determined above) to calculate ATMOSPHERIC_TEMP.

✓ **SET STATUS TO HEALTHY**

- Set the values of both elements of TS_STATUS to healthy.

A.6 DATA REQUIREMENTS DICTIONARY

PART I. DATA ELEMENT DESCRIPTIONS

The following template has been constructed for defining the data elements in the four required global data stores and the optional variables shown in Table A.6.5:

NAME: DESCRIPTION: USED IN: UNITS: RANGE: DATA TYPE: ATTRIBUTE: DATA STORE LOCATION: ACCURACY:

NAME This field gives the name of the variable used in the specification. The variable name used during coding must be the same as specified.

DESCRIPTION This field gives a brief description of the variable.

USED IN This field provides a reference to the functional units using this variable.

UNITS This field indicates the unit of measure for the data contained in the variable being defined.

RANGE This field specifies the acceptable range of data values for the variable.

DATA TYPE The data type field specifies the data type to be used when declaring the variable during coding.

ATTRIBUTE This field indicates whether or not the variable contains data, control information, or a data condition.

DATA STORE LOCATION This field references the common region where the variable must be stored.

ACCURACY This field dictates the degree of accuracy required for output comparisons to be made between implementations. In the data dictionary, accuracy is listed as N/A where accuracy is not applicable, or TBD where accuracy is (T)o (B)e (D)etermined later. A formal modification will be released when the values of the accuracy requirements have been approved.

NAME: A_ACCELERATION
DESCRIPTION: vehicle accelerations
USED IN: AECLP, ASP, CP, GP
UNITS: $\frac{meters}{\sec^2}$
RANGE: [-20, 5]
DATA TYPE: array (1..3, 0..4) of real*8
ATTRIBUTE: data
DATA STORE LOCATION: SENSOR_OUTPUT
ACCURACY: TBD

NAME: A_BIAS
DESCRIPTION: characteristic bias in the accelerometer measurements
USED IN: ASP
UNITS: $\frac{meters}{\sec^2}$
RANGE: [-30, 0]
DATA TYPE: array (1..3) of real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: A_COUNTER
DESCRIPTION: accelerations along the $\vec{x}$, $\vec{y}$, and $\vec{z}$ axes
USED IN: ASP
UNITS: none
RANGE: [0, 2^{15} -1]
DATA TYPE: array (1..3) of Integer*2
ATTRIBUTE: data
DATA STORE LOCATION: EXTERNAL
ACCURACY: N/A

NAME: A_GAIN_0
DESCRIPTION: standard gain in the accelerations
USED IN: ASP
UNITS: $\frac{\text{meters}}{\text{sec}^2}$
RANGE: [0, 1]
DATA TYPE: array (1..3) of real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: A_SCALE
DESCRIPTION: multiplicative constant used to determine limit on deviation accelerometer values.
USED IN: ASP
UNITS: none
RANGE: [0, 3]
DATA TYPE: Integer*4
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: A_STATUS
DESCRIPTION: Flag indicating whether or not the accelerometers are working properly.
USED IN: ASP, CP
UNITS: none
RANGE: [0 : healthy, 1: unhealthy]
DATA TYPE: array (1..3, 0..3) of logical*1
ATTRIBUTE: data
DATA STORE LOCATION: GUIDANCE_STATE
ACCURACY: N/A

NAME: AECLP_DONE
DESCRIPTION: Flag indicating completion of AECLP task.
USED IN: 2. RUN_GCS
UNITS: none
RANGE: [FALSE: running of task AECLP incomplete, TRUE: running of task AECLP complete]
DATA TYPE: logical*1
ATTRIBUTE: control
DATA STORE LOCATION: none
ACCURACY: N/A

NAME: AE_CMD
DESCRIPTION: Valve settings for the axial engines.
USED IN: AECLP, CP
UNITS: none
RANGE: [0, 127]
DATA TYPE: array (1..3) of Integer*2
ATTRIBUTE: data
DATA STORE LOCATION: EXTERNAL
ACCURACY: TBD

NAME: AE_STATUS
DESCRIPTION: Status of axial engines.
USED IN: AECLP, CP
UNITS: none
RANGE: [0: Healthy, 1: Failed.]
DATA TYPE: logical*1
ATTRIBUTE: data
DATA STORE LOCATION: GUIDANCE_STATE
ACCURACY: N/A

NAME: AE_SWITCH
DESCRIPTION: Flag indicating whether or not axial engines are turned on.
USED IN: AECLP, GP
UNITS: none
RANGE: [0: axial engines are off, 1: axial engines are on.]
DATA TYPE: logical*1
ATTRIBUTE: data condition
DATA STORE LOCATION: GUIDANCE_STATE
ACCURACY: N/A

NAME: AE_TEMP
DESCRIPTION: Temperature of axial engines when they are turned on.
USED IN: AECLP, CP, CRCP, GP
UNITS: none
RANGE: [0: Cold, 1: Warming-Up, 2: Hot]
DATA TYPE: Integer*2
ATTRIBUTE: data condition
DATA STORE LOCATION: GUIDANCE_STATE
ACCURACY: N/A

NAME: ALPHA_MATRIX
DESCRIPTION: Matrix of misalignment angles
USED IN: ASP
UNITS: none
RANGE: [$-\pi$, π]
DATA TYPE: array (1..3, 1..3) of real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: AR_ALTITUDE
DESCRIPTION: altimeter radar height above terrain
USED IN: ARSP, CP, GP
UNITS: meters
RANGE: [0, 2000]
DATA TYPE: array (0..4) of real*8
ATTRIBUTE: data
DATA STORE LOCATION: SENSOR_OUTPUT
ACCURACY: TBD

NAME: AR_COUNTER
DESCRIPTION: counter containing elapsed time since transmission of radar pulse
USED IN: ARSP
UNITS: Cycles
RANGE: [-1, 2^{15}-1]
DATA TYPE: Integer*2
ATTRIBUTE: data
DATA STORE LOCATION: EXTERNAL
ACCURACY: N/A

NAME: AR_FREQUENCY
DESCRIPTION: increment frequency of AR_COUNTER
USED IN: ARSP
UNITS: $\frac{cycles}{\text{sec}}$
RANGE: [1, 2.45×10^9]
DATA TYPE: real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: AR_STATUS
DESCRIPTION: status of the altimeter radars
USED IN: ARSP, CP
UNITS: none
RANGE: [0 : healthy, 1: failed]
DATA TYPE: array (0..4) of logical*1
ATTRIBUTE: data
DATA STORE LOCATION: GUIDANCE_STATE
ACCURACY: N/A

NAME: ARSP_DONE
DESCRIPTION: Flag indicating completion of ARSP task.
USED IN: 2. RUN_GCS
UNITS: none
RANGE: [FALSE: running of task ARSP incomplete, TRUE: running of task ARSP complete]
DATA TYPE: logical*1
ATTRIBUTE: control
DATA STORE LOCATION: none
ACCURACY: N/A

NAME: ASP_DONE
DESCRIPTION: Flag indicating completion of ASP task.
USED IN: 2. RUN_GCS
UNITS: none
RANGE: [FALSE: running of task ASP incomplete, TRUE: running of task ASP complete]
DATA TYPE: logical*1
ATTRIBUTE: control
DATA STORE LOCATION: none
ACCURACY: N/A

NAME: ATMOSPHERIC_TEMP
DESCRIPTION: atmospheric temperature
USED IN: ASP, CP, GSP, TSP
UNITS: degrees C
RANGE: [-200, 25]
DATA TYPE: real*8
ATTRIBUTE: data
DATA STORE LOCATION: SENSOR_OUTPUT
ACCURACY: TBD

NAME: C_STATUS
DESCRIPTION: Flag indicating whether or not the communications processor is working properly.
USED IN: CP
UNITS: none
RANGE: [0 : healthy, 1: failed]
DATA TYPE: logical*1
ATTRIBUTE: data
DATA STORE LOCATION: GUIDANCE_STATE
ACCURACY: N/A

NAME: CHUTE_RELEASED
DESCRIPTION: signal indicating parachute has been released
USED IN: AECLP, CP, CRCP, GP
UNITS: none
RANGE: [0: Chute Attached, 1: Chute Released]
DATA TYPE: logical*1
ATTRIBUTE: data condition
DATA STORE LOCATION: EXTERNAL
ACCURACY: N/A

NAME: CL
DESCRIPTION: Index which specifies which set of Control Law Parameters to use
USED IN: AECLP, GP
UNITS: none
RANGE: [1: first, 2: second]
DATA TYPE: Integer*2
ATTRIBUTE: data
DATA STORE LOCATION: GUIDANCE_STATE
ACCURACY: N/A

NAME: CLP_DONE
DESCRIPTION: Control signal which indicates whether or not Control Law Processing function has completed.
USED IN: 2. RUN_GCS
UNITS: none
RANGE: [FALSE: running of Control Law Processing function incomplete, TRUE: running of Control Law Processing function complete]
DATA TYPE: logical*1
ATTRIBUTE: control
DATA STORE LOCATION: none
ACCURACY: N/A

NAME: COMM_SYNC_PATTERN
DESCRIPTION: sixteen bit synchronization pattern
USED IN: CP
UNITS: none
RANGE: [1101100110110010] (binary)
DATA TYPE: Integer*2
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: CONTOUR_ALTITUDE
DESCRIPTION: Altitude in velocity-altitude contour.
USED IN: GP
UNITS: kilometers
RANGE: [-.01, 2]
DATA TYPE: array (1..100) of real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: CONTOUR_CROSSED
DESCRIPTION: Indicates if the velocity-altitude contour has been sensed.
USED IN: AECLP, CP, GP
UNITS: none
RANGE: [0: contour not crossed, 1: contour crossed]
DATA TYPE: logical*1
ATTRIBUTE: data condition
DATA STORE LOCATION: GUIDANCE_STATE
ACCURACY: N/A

NAME: CONTOUR_VELOCITY
DESCRIPTION: Velocity in velocity-altitude contour.
USED IN: GP
UNITS: $\frac{kilometers}{\text{sec}}$
RANGE: [0, 0.5]
DATA TYPE: array (1..100) of real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: CP_DONE
DESCRIPTION: Flag indicating completion of CP task.
USED IN: 2. RUN_GCS
UNITS: none
RANGE: [FALSE: running of task CP incomplete, TRUE: running of task CP complete]
DATA TYPE: logical*1
ATTRIBUTE: control
DATA STORE LOCATION: none
ACCURACY: N/A

NAME: CRCP_DONE
DESCRIPTION: Flag indicating completion of CRCP task.
USED IN: 2. RUN_GCS
UNITS: none
RANGE: [FALSE: running of task CRCP incomplete, TRUE: running of task CRCP complete]
DATA TYPE: logical*1
ATTRIBUTE: control
DATA STORE LOCATION: none
ACCURACY: N/A

NAME: DELTA_T
DESCRIPTION: Time step duration.
USED IN: AECLP, GP, RECLP, TDLRSP
UNITS: seconds
RANGE: [0.005, 0.20]
DATA TYPE: real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: DROP_HEIGHT
DESCRIPTION: Height from which vehicle should free-fall to surface
USED IN: GP
UNITS: meters
RANGE: [0, 100]
DATA TYPE: real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: DROP_SPEED
DESCRIPTION: Optimal speed during constant velocity descent.
USED IN: GP
UNITS: $\frac{\text{meters}}{\text{sec}}$
RANGE: [0, 4.0]
DATA TYPE: real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: ENGINES_ON_ALTITUDE
DESCRIPTION: Altitude at which the axial engines are turned on.
USED IN: AECLP, GP
UNITS: meters
RANGE: [0, 2000]
DATA TYPE: real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: FRAME_BEAM_UNLOCKED
DESCRIPTION: Variable containing the number of the frame during which the radar beam unlocked
USED I N: TDLRSP
UNITS: none
RANGE: [0, 2^{31}-1]
DATA TYPE: array (1..4) of Integer*4
ATTRIBUTE: data
DATA STORE LOCATION: GUIDANCE_STATE
ACCURACY: TBD

NAME: FRAME_COUNTER
DESCRIPTION: Counter containing the number of the present frame
USED IN: AECLP, ARSP, CP, GP, TDLRSP
UNITS: none
RANGE: [1, 2^{31}-1]
DATA TYPE: Integer*4
ATTRIBUTE: data
DATA STORE LOCATION: EXTERNAL
ACCURACY: N/A

NAME: FRAME_ENGINES_IGNITED
DESCRIPTION: Variable containing the number of the frame during which the engines were ignited
USED IN: AECLP, GP
UNITS: none
RANGE: [0, 2^{31}-1]
DATA TYPE: Integer*4
ATTRIBUTE: data
DATA STORE LOCATION: GUIDANCE_STATE
ACCURACY: TBD

NAME: FULL_UP_TIME
DESCRIPTION: Time for axial engines to reach optimum operational condition
USED IN: AECLP
UNITS: seconds
RANGE: [0, 60]
DATA TYPE: real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: G1
DESCRIPTION: coefficient used to adjust A_GAIN
USED IN: ASP
UNITS: $\frac{\frac{meters}{\text{sec}^2}}{\deg reeC}$
RANGE: [-5, 5]
DATA TYPE: real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: G2
DESCRIPTION: coefficient used to adjust A_GAIN
USED IN: ASP
UNITS: $\frac{\frac{meters}{\text{sec}^2}}{\deg reeC^2}$
RANGE: [-5, 5]
DATA TYPE: real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: G3
DESCRIPTION: coefficient used to adjust G_GAIN
USED IN: GSP
UNITS: $\frac{\frac{radians}{\text{sec}}}{\deg reeC}$
RANGE: [-5, 5]
DATA TYPE: real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: G4
DESCRIPTION: coefficient used to adjust G_GAIN
USED IN: GSP
UNITS: $\frac{\frac{radians}{\sec}}{\deg ree\ C^2}$
RANGE: [-5, 5]
DATA TYPE: real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: G_COUNTER
DESCRIPTION: gyroscope measurement of vehicle rotation rates
USED IN : GSP
UNITS: none
RANGE: $[-(2^{14}-1), 2^{14}-1]$
DATA TYPE: array (1..3) of Integer*2
ATTRIBUTE: data
DATA STORE LOCATION: EXTERNAL
ACCURACY: N/A

NAME: G_GAIN_0
DESCRIPTION: standard gain in vehicle rotation rates as measured by the gyroscopes
USED IN: GSP
UNITS: $\frac{\text{radians}}{\sec}$
RANGE: [-1, 1]
DATA TYPE: array (1..3) of real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: G_OFFSET
DESCRIPTION: standard offset of the rotation raw values
USED IN: GSP
UNITS: $\frac{radians}{\sec}$
RANGE: [-0.5, 0.5]
DATA TYPE: array (1..3) of real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: G_ROTATION
DESCRIPTION: vehicle rotation rates
USED IN: CP, GSP, GP, RECLP
UNITS: $\frac{radians}{\sec}$
RANGE: [-1.0, 1.0]
DATA TYPE: array (1..3, 0..4) of real*8
ATTRIBUTE: data
DATA STORE LOCATION: SENSOR_OUTPUT
ACCURACY: TBD

NAME: G_STATUS
DESCRIPTION: status of the gyroscopes
USED IN: CP, GSP
UNITS: none
RANGE: [0 : healthy, 1: failed]
DATA TYPE: logical*1
ATTRIBUTE: data
DATA STORE LOCATION: GUIDANCE_STATE
ACCURACY: N/A

NAME: GA
DESCRIPTION: gain
USED IN: AECLP
UNITS: $\frac{\sec}{meter}$
RANGE: [0, 50]
DATA TYPE: real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: GAX
DESCRIPTION: gain
USED IN: AECLP
UNITS: none
RANGE: [0, 5]
DATA TYPE: real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: GP1
DESCRIPTION: gain
USED IN: AECLP
UNITS: none
RANGE: [-5, 5]
DATA TYPE: real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: GP2
DESCRIPTION: gain
USED IN: AECLP
UNITS: none
RANGE: [-5, 5]
DATA TYPE: real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: GP_ALTITUDE
DESCRIPTION: altitude as seen by guidance processor
USED IN: AECLP, CP, GP
UNITS: meters
RANGE: [0, 2000]
DATA TYPE: array (0..4) of real*8
ATTRIBUTE: data
DATA STORE LOCATION: GUIDANCE_STATE
ACCURACY: TBD

NAME: GP_ATTITUDE
DESCRIPTION: direction cosine matrix
USED IN: AECLP, CP, GP
UNITS: none
RANGE: [-1, 1]
DATA TYPE: array (1..3, 1..3, 0..4) real*8
ATTRIBUTE: data
DATA STORE LOCATION: GUIDANCE_STATE
ACCURACY: TBD

NAME: GP_DONE
DESCRIPTION: Flag indicating completion of GP task.
USED IN: 2. RUN_GCS
UNITS: none
RANGE: [FALSE: running of task GP incomplete, TRUE: running of task GP complete]
DATA TYPE: logical*1
ATTRIBUTE: control
DATA STORE LOCATION: none
ACCURACY: N/A

NAME: GP_PHASE
DESCRIPTION: phase of operation as seen by guidance processor
USED IN: CP, GP
UNITS: none
RANGE: [1, 5]
DATA TYPE: integer*4
ATTRIBUTE: data condition
DATA STORE LOCATION: GUIDANCE_STATE
ACCURACY: TBD

NAME: GP_ROTATION
DESCRIPTION: rotation rates as determined by the guidance processing functional unit
USED IN: AECLP, CP, GP
UNITS: $\frac{radians}{\text{sec}}$
RANGE: [-1.0, 1.0]
DATA TYPE: array (1..3, 1..3) real*8
ATTRIBUTE: data
DATA STORE LOCATION: GUIDANCE_STATE
ACCURACY: TBD

NAME: GP_VELOCITY
DESCRIPTION: Velocity as corrected by the guidance algorithm.
USED IN: AECLP, CP, GP
UNITS: $\frac{meters}{\text{sec}}$
RANGE: [-100, 100]
DATA TYPE: array (1..3, 0..4) of real*8
ATTRIBUTE: data
DATA STORE LOCATION: GUIDANCE_STATE
ACCURACY: TBD

NAME: GPY
DESCRIPTION: gain
USED IN: AECLP
UNITS: none
RANGE: [-5, 5]
DATA TYPE: real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: GQ
DESCRIPTION: gain
USED IN: AECLP
UNITS: seconds
RANGE: [-5, 8]
DATA TYPE: array (1..2) of real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: GR
DESCRIPTION: gain
USED IN: AECLP
UNITS: seconds
RANGE: [-5, 8]
DATA TYPE: array (1..2) of real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: GRAVITY
DESCRIPTION: gravity of planet
USED IN: AECLP, GP
UNITS: $\frac{meters}{\text{sec}^2}$
RANGE: [0, 100]
DATA TYPE: real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: GSP_DONE
DESCRIPTION: Flag indicating completion of GSP task.
USED IN: 2. RUN_GCS
UNITS: none
RANGE: [FALSE: running of task GSP incomplete, TRUE: running of task GSP complete]
DATA TYPE: logical*1
ATTRIBUTE: control
DATA STORE LOCATION: none
ACCURACY: N/A

NAME: GV
DESCRIPTION: gain
USED IN: AECLP
UNITS: $\frac{\text{sec}}{\textit{meter}}$
RANGE: [-5, 8]
DATA TYPE: array (1..2) of real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: GVE
DESCRIPTION: gain
USED IN: AECLP
UNITS: /second
RANGE: [0, 500]
DATA TYPE: real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: GVEI
DESCRIPTION: gain
USED IN: AECLP
UNITS: /second^2
RANGE: [-5, 40]
DATA TYPE: array (1..2) of real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: GVI
DESCRIPTION: gain
USED IN: AECLP
UNITS: /meter
RANGE: [-5, 5]
DATA TYPE: array (1..2) of real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: GW
DESCRIPTION: gain
USED IN: AECLP
UNITS: $\frac{\text{sec}}{\textit{meter}}$
RANGE: [-5, 8]
DATA TYPE: array (1..2) of real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: GWI
DESCRIPTION: gain
USED IN: AECLP
UNITS: /meter
RANGE: [-5, 5]
DATA TYPE: array (1..2) of real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: INIT_DONE
DESCRIPTION: Flag indicating completion of GCS initialization.
USED IN: 0. GCS
UNITS: none
RANGE: [FALSE: initialization incomplete, TRUE: initialization complete]
DATA TYPE: logical*1
ATTRIBUTE: control
DATA STORE LOCATION: none
ACCURACY: N/A

NAME: INTERNAL_CMD
DESCRIPTION: Real vector containing the command to be sent to the axial engines
USED IN: AECLP
UNITS: none
RANGE: [-0.7, 1.7]
DATA TYPE: array (1..3) of real*8
ATTRIBUTE: data
DATA STORE LOCATION: GUIDANCE_STATE
ACCURACY: TBD

NAME: K_ALT
DESCRIPTION: Determines use of altimeter radar by guidance processor
USED IN: ARSP, CP, GP
UNITS: none
RANGE: [0, 1]
DATA TYPE: array (0..4) of Integer*4
ATTRIBUTE: data
DATA STORE LOCATION: GUIDANCE_STATE
ACCURACY: N/A

NAME: K_MATRIX
DESCRIPTION: Determines use of doppler radar by guidance processor.
USED IN: CP, GP, TDLRSP
UNITS: none
RANGE: [0, 1]
DATA TYPE: array (1..3, 1..3, 0..4) Integer*4
ATTRIBUTE: data
DATA STORE LOCATION: GUIDANCE_STATE
ACCURACY: N/A

NAME: M1
DESCRIPTION: lower measured temperature calibration point for solid state temperature sensor
USED IN: TSP
UNITS: none
RANGE: $[0, 2^{15}-1]$
DATA TYPE: Integer*2
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: M2
DESCRIPTION: upper measured temperature calibration point for solid state temperature sensor
USED IN: TSP
UNITS: none
RANGE: $[0, 2^{15}-1]$
DATA TYPE: Integer*2
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: M3
DESCRIPTION: lower measured temperature calibration point for thermocouple pair temperature sensor
USED IN: TSP
UNITS: none
RANGE: $[0, 2^{15}-1]$
DATA TYPE: Integer*2
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: M4
DESCRIPTION: upper measured temperature calibration point for thermocouple pair temperature sensor
USED IN: TSP
UNITS: none
RANGE: $[0, 2^{15}-1]$
DATA TYPE: Integer*2
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: MAX_NORMAL_VELOCITY
DESCRIPTION: Maximum vertical velocity for safe landing
USED IN: GP
UNITS: $\frac{\text{meters}}{\text{sec}}$
RANGE: [0, 3.35]
DATA TYPE: real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: OMEGA
DESCRIPTION: gain of angular velocity
USED IN: AECLP
UNITS: /second
RANGE: [-50, 50]
DATA TYPE: real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: P1
DESCRIPTION: pulse rate boundary
USED IN: RECLP
UNITS: $\frac{radians}{\text{sec}}$
RANGE: [0, 0.05]
DATA TYPE: real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: P2
DESCRIPTION: pulse rate boundary
USED IN: RECLP
UNITS: $\frac{radians}{\text{sec}}$
RANGE: [0, 0.05]
DATA TYPE: real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: P3
DESCRIPTION: pulse rate boundary
USED IN: RECLP
UNITS: $\frac{radians}{\text{sec}}$
RANGE: [0, 0.05]
DATA TYPE: real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: P4
DESCRIPTION: pulse rate boundary
USED IN: RECLP
UNITS: $\frac{radians}{\text{sec}}$
RANGE: [0, 0.05]
DATA TYPE: real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: PACKET
DESCRIPTION: Packet of telemetry data
USED IN: CP
UNITS: N/A
RANGE: N/A
DATA TYPE: array (1..256) of Integer*2
ATTRIBUTE: data
DATA STORE LOCATION: EXTERNAL
ACCURACY: N/A

NAME: PE_INTEGRAL
DESCRIPTION: Integral portion of Pitch error equation
USED IN: AECLP, CP
UNITS: meters
RANGE: [-100, 100]
DATA TYPE: real*8
ATTRIBUTE: data
DATA STORE LOCATION: GUIDANCE_STATE
ACCURACY: TBD

NAME: PE_MAX
DESCRIPTION: Maximum pitch error tolerable
USED IN: AECLP
UNITS: none
RANGE: [0, 1]
DATA TYPE: array(1..2) of real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: PE_MIN
DESCRIPTION: Minimum pitch error tolerable.
USED IN: AECLP
UNITS: none
RANGE: [-1, 0]
DATA TYPE: array(1..2) of real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: RE_CMD
DESCRIPTION: roll engine command
USED IN: CP, RECLP
UNITS: none
RANGE:
[1: off,
2: minimum, counterclockwise,
3: minimum, clockwise,
4; intermediate, counterclockwise,
5: intermediate, clockwise,
6: maximum, counterclockwise,
7: maximum, clockwise]
note: the values above for Range have been derived from range of intensity and direction as follows:
Intensity [00:off, 01:minimum, 10:intermediate, 11:maximum](binary)
Direction [0:counterclockwise (positive), 1:clockwise (negative)] (binary)
DATA TYPE: Integer*2
ATTRIBUTE: data
DATA STORE LOCATION: EXTERNAL
ACCURACY: TBD

NAME: RE_STATUS
DESCRIPTION: status of the roll engines
USED IN: CP, RECLP
UNITS: none
RANGE: [0 : healthy, 1: failed]
DATA TYPE: logical*1
ATTRIBUTE: data
DATA STORE LOCATION: GUIDANCE_STATE
ACCURACY: N/A

NAME: RE_SWITCH
DESCRIPTION: Flag indicating whether or not the roll engines are turned on.
USED IN: GP, RECLP
UNITS: none
RANGE: [0: roll engines are off, 1: roll engines are on.]
DATA TYPE: logical*1
ATTRIBUTE: data condition
DATA STORE LOCATION: GUIDANCE_STATE
ACCURACY: N/A

NAME: RECLP_DONE
DESCRIPTION: Flag indicating completion of RECLP task.
USED IN: 2. RUN_GCS
UNITS: none
RANGE: [FALSE: running of task RECLP incomplete, TRUE: running of task RECLP complete]
DATA TYPE: logical*1
ATTRIBUTE: control
DATA STORE LOCATION: none
ACCURACY: N/A

NAME: RENDEZVOUS
DESCRIPTION: Control signal which indicates whether or not GCS_SIM_RENDEZVOUS is to be activated.
USED IN: 2. RUN_GCS
UNITS: none
RANGE: [FALSE: GCS_SIM_RENDEZVOUS is not to be activated, TRUE: GCS_SIM_RENDEZVOUS is to be activated]
DATA TYPE: logical*1
ATTRIBUTE: control
DATA STORE LOCATION: none
ACCURACY: N/A

NAME: RUN_DONE
DESCRIPTION: Flag indicating completion of GCS.
USED IN: 0. GCS
UNITS: none
RANGE: [FALSE: running of GCS incomplete, TRUE: running of GCS complete]
DATA TYPE: logical*1
ATTRIBUTE: control
DATA STORE LOCATION: none
ACCURACY: N/A

NAME: SP_DONE
DESCRIPTION: Control signal which indicates whether or not Sensor Processing function has been completed.
USED IN: 2. RUN_GCS
UNITS: none
RANGE: [FALSE: running of Sensor Processing function incomplete, TRUE: running of Sensor Processing function complete]
DATA TYPE: logical*1
ATTRIBUTE: control
DATA STORE LOCATION: none
ACCURACY: N/A

NAME: SS_TEMP
DESCRIPTION: Solid state temperature data
USED IN: TSP
UNITS: none
RANGE: $[0, 2^{15}-1]$
DATA TYPE: Integer*2
ATTRIBUTE: data
DATA STORE LOCATION: EXTERNAL
ACCURACY: N/A

NAME: SUBFRAME_COUNTER
DESCRIPTION: Counter containing the number of the present subframe.
USED IN: CP
UNITS: none
RANGE: [1, 3]
DATA TYPE: Integer*2
ATTRIBUTE: data
DATA STORE LOCATION: EXTERNAL
ACCURACY: N/A

NAME: T1
DESCRIPTION: lower ambient temperature calibration point for solid state temperature sensor
USED IN: TSP
UNITS: degrees C
RANGE: [-250, 250]
DATA TYPE: real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: T2
DESCRIPTION: upper ambient temperature calibration point for solid state temperature sensor
USED IN: TSP
UNITS: degrees C
RANGE: [-250, 250]
DATA TYPE: real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: T3
DESCRIPTION: lower ambient temperature calibration point for thermocouple pair temperature sensor
USED IN: TSP
UNITS: degrees C
RANGE: [-50, 50]
DATA TYPE: real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: T4
DESCRIPTION: upper ambient temperature calibration point for thermocouple pair temperature sensor
USED IN: TSP
UNITS: degrees C
RANGE: [-50, 50]
DATA TYPE: real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: TD_COUNTER
DESCRIPTION: value returned by Touch Down Sensor
USED I N: TDSP
UNITS: none
RANGE: $[-2^{15}, 2^{15}-1]$
DATA TYPE: Integer*2
ATTRIBUTE: data
DATA STORE LOCATION: EXTERNAL
ACCURACY: N/A

NAME: TD_SENSED
DESCRIPTION: Flag indicating whether or not touch down has been sensed.
USED IN: CP, GP, TDSP
UNITS: none
RANGE: [0: touch down not sensed, 1: touch down sensed]
DATA TYPE: logical*1
ATTRIBUTE: data condition
DATA STORE LOCATION: SENSOR_OUTPUT
ACCURACY: N/A

NAME: TDLR_ANGLES
DESCRIPTION: vector of doppler radar beam offset angles (i.e., α , β, γ)
USED IN: TDLRSP
UNITS: radians
RANGE: $[0, \frac{\pi}{2})$
DATA TYPE: array (1..3) real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: TDLR_COUNTER
DESCRIPTION: value returned by Doppler radar
USED IN: TDLRSP
UNITS: none
RANGE: $[0, 2^{15}-1]$
DATA TYPE: array (1..4) Integer*2
ATTRIBUTE: data
DATA STORE LOCATION: EXTERNAL
ACCURACY: N/A

NAME: TDLR_GAIN
DESCRIPTION: gain in doppler radar beam
USED IN: TDLRSP
UNITS: $\frac{meters}{\text{sec}}$
RANGE: [-1, 1]
DATA TYPE: real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: TDLR_LOCK_TIME
DESCRIPTION: locking time of doppler radar beam
USED IN: TDLRSP
UNITS: seconds
RANGE: [0, 60]
DATA TYPE: real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: TDLR_OFFSET
DESCRIPTION: offset in doppler radar beam
USED IN: TDLRSP
UNITS: $\frac{meters}{\text{sec}}$
RANGE: [-100, 0]
DATA TYPE: real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: TDLR_STATE
DESCRIPTION: state of the touch down landing radar beams.
USED IN: CP, TDLRSP
UNITS: none
RANGE: [0: Beam unlocked, 1: Beam locked]
DATA TYPE: array (1..4) logical*1
ATTRIBUTE: data condition
DATA STORE LOCATION: GUIDANCE_STATE
ACCURACY: N/A

NAME: TDLR_STATUS
DESCRIPTION: status of the doppler radar
USED IN: CP, TDLRSP
UNITS: none
RANGE: [0 : healthy, 1: failed]
DATA TYPE: array (1..4) of logical*1
ATTRIBUTE: data
DATA STORE LOCATION: GUIDANCE_STATE
ACCURACY: N/A

NAME: TDLR_VELOCITY
DESCRIPTION: Velocity as computed by the touch down landing radar.
USED IN: CP, GP, TDLRSP
UNITS: $\frac{meters}{\text{sec}}$
RANGE: [-100, 100]
DATA TYPE: array (1..3, 0..4) of real*8
ATTRIBUTE: data
DATA STORE LOCATION: SENSOR_OUTPUT
ACCURACY: TBD

NAME: TDLRSP_DONE
DESCRIPTION: Flag indicating completion of TDLRSP task.
USED IN: 2. RUN_GCS
UNITS: none
RANGE: [FALSE: running of task TDLRSP incomplete, TRUE: running of task TDLRSP complete]
DATA TYPE: logical*1
ATTRIBUTE: control
DATA STORE LOCATION: none
ACCURACY: N/A

NAME: TDSP_DONE
DESCRIPTION: Flag indicating completion of TDSP task.
USED IN: 2. RUN_GCS
UNITS: none
RANGE: [FALSE: running of task TDSP incomplete, TRUE: running of task TDSP complete]
DATA TYPE: logical*1
ATTRIBUTE: control
DATA STORE LOCATION: none
ACCURACY: N/A

NAME: TDS_STATUS
DESCRIPTION: status of the touch down sensor
USED IN: CP, GP, TDSP
UNITS: none
RANGE: [0 : healthy, 1: failed]
DATA TYPE: logical*1
ATTRIBUTE: data
DATA STORE LOCATION: GUIDANCE_STATE
ACCURACY: N/A

NAME: TE_DROP
DESCRIPTION: The axial thrust error when axial engines are warm and the velocity altitude contour has not been intersected.
USED IN: AECLP
UNITS: none
RANGE: [-2, 2]
DATA TYPE: real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: TE_INIT
DESCRIPTION: The axial thrust error when the axial engines are cold.
USED IN: AECLP
UNITS: none
RANGE: [-2, 2]
DATA TYPE: real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: TE_INTEGRAL
DESCRIPTION: Integral portion of Thrust error equation
USED IN: AECLP, CP, GP
UNITS: meters
RANGE: [-100, 100]
DATA TYPE: real*8
ATTRIBUTE: data
DATA STORE LOCATION: GUIDANCE_STATE
ACCURACY: TBD

NAME: TE_LIMIT
DESCRIPTION: Limiting thrust error
USED IN: AECLP
UNITS: none
RANGE: [-100, 100]
DATA TYPE: real*8
ATTRIBUTE: data
DATA STORE LOCATION: GUIDANCE_STATE
ACCURACY: TBD

NAME: TE_MAX
DESCRIPTION: Maximum thrust error tolerable
USED IN: AECLP
UNITS: none
RANGE: [-2, 2]
DATA TYPE: array(1..2) of real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: TE_MIN
DESCRIPTION: Minimum thrust error tolerable.
USED IN: AECLP
UNITS: none
RANGE: [-2, 2]
DATA TYPE: array(1..2) of real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: THERMO_TEMP
DESCRIPTION: thermocouple pair temperature
USED IN: TSP
UNITS: none
RANGE: [0, 2^{15}-1]
DATA TYPE: Integer*2
ATTRIBUTE: data
DATA STORE LOCATION: EXTERNAL
ACCURACY: N/A

NAME: THETA
DESCRIPTION: roll angle
USED IN: RECLP
UNITS: radians
RANGE: [$-\pi$, π]
DATA TYPE: real*8
ATTRIBUTE: data
DATA STORE LOCATION: GUIDANCE_STATE
ACCURACY: TBD

NAME: THETA1
DESCRIPTION: pulse angle boundary
USED IN: RECLP
UNITS: radians
RANGE: [0, 0.05]
DATA TYPE: real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: THETA2
DESCRIPTION: pulse angle boundary
USED IN: RECLP
UNITS: radians
RANGE: [0, 0.05]
DATA TYPE: real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: TS_STATUS
DESCRIPTION: status of the temperature sensors in solid state, then thermocouple pair order
USED IN: CP, TSP
UNITS: none
RANGE: [0 : healthy, 1: failed]
DATA TYPE: array (1..2) of logical*1
ATTRIBUTE: data
DATA STORE LOCATION: GUIDANCE_STATE
ACCURACY: N/A

NAME: TSP_DONE
DESCRIPTION: Flag indicating completion of TSP task.
USED IN: 2. RUN_GCS
UNITS: none
RANGE: [FALSE: running of task TSP incomplete, TRUE: running of task TSP complete]
DATA TYPE: logical*1
ATTRIBUTE: control
DATA STORE LOCATION: none
ACCURACY: N/A

NAME: VELOCITY_ERROR
DESCRIPTION: Distance from velocity-altitude contour. (Difference in velocities from actual to desired on contour.)
USED IN: AECLP, CP, GP
UNITS: $\frac{meters}{\text{sec}}$
RANGE: [-300, 20]
DATA TYPE: real*8
ATTRIBUTE: data
DATA STORE LOCATION: GUIDANCE_STATE
ACCURACY: TBD

NAME: YE_INTEGRAL
DESCRIPTION: Integral portion of Yaw error equation
USED IN: AECLP, CP
UNITS: meters
RANGE: [-100, 100]
DATA TYPE: real*8
ATTRIBUTE: data
DATA STORE LOCATION: GUIDANCE_STATE
ACCURACY: TBD

NAME: YE_MAX
DESCRIPTION: Maximum yaw error tolerable
USED IN: AECLP
UNITS: none
RANGE: [-1, 1]
DATA TYPE: array(1..2) of real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

NAME: YE_MIN
DESCRIPTION: Minimum yaw error tolerable.
USED IN: AECLP
UNITS: none
RANGE: [-1, 1]
DATA TYPE: array(1..2) of real*8
ATTRIBUTE: data
DATA STORE LOCATION: RUN_PARAMETERS
ACCURACY: N/A

PART II. CONTENTS OF DATA STORES

Table A.6.1 DATA STORE: GUIDANCE_STATE

VARIABLE NAME	USED BY:
A_STATUS	ASP, CP
AE_STATUS	AECLP, CP
AE_SWITCH	AECLP, GP
AE_TEMP	AECLP, CP, CRCP, GP
AR_STATUS	ARSP, CP
C_STATUS	CP
CL	AECLP, GP
CONTOUR_CROSSED	AECLP, CP, GP
FRAME_BEAM_UNLOCKED	TDLRSP
FRAME_ENGINES_IGNITED	AECLP, GP
G_STATUS	CP, GSP
GP_ALTITUDE	CP, GP, AECLP
GP_ATTITUDE	AECLP, CP, GP
GP_PHASE	CP, GP
GP_ROTATION	AECLP, CP, GP
GP_VELOCITY	AECLP, CP, GP
INTERNAL_CMD	AECLP
K_ALT	ARSP, CP, GP
K_MATRIX	CP, GP, TDLRSP
PE_INTEGRAL	AECLP, CP
RE_STATUS	CP, RECLP
RE_SWITCH	GP, RECLP
TDLR_STATE	CP, TDLRSP
TDLR_STATUS	CP, TDLRSP
TDS_STATUS	CP, GP, TDSP
TE_INTEGRAL	AECLP, CP, GP
TE_LIMIT	AECLP
THETA	RECLP
TS_STATUS	CP, TSP
VELOCITY_ERROR	AECLP, CP, GP
YE_INTEGRAL	AECLP, CP

Table A.6.2 DATA STORE: EXTERNAL

VARIABLE NAME	USED BY
A_COUNTER	ASP
AE_CMD	AECLP, CP
AR_COUNTER	ARSP
CHUTE_RELEASED	AECLP, CP, CRCP, GP
FRAME_COUNTER	AECLP, ARSP, CP, GP, TDLRSP
G_COUNTER	GSP
PACKET	CP
RE_CMD	RECLP, CP
SS_TEMP	TSP
SUBFRAME_COUNTER	CP
TD_COUNTER	TDSP
TDLR_COUNTER	TDLRSP
THERMO_TEMP	TSP

Table A.6.3 DATA STORE: SENSOR_OUTPUT

VARIABLE NAME	USED BY:
A_ACCELERATION	AECLP, ASP, CP, GP
AR_ALTITUDE	ARSP, CP, GP
ATMOSPHERIC_TEMP	ASP, CP, GSP, TSP
G_ROTATION	CP, GSP, GP, RECLP
TD_SENSED	CP, GP, TDSP
TDLR_VELOCITY	CP, GP, TDLRSP

Table A.6.4 DATA STORE: RUN_PARAMETERS

VARIABLE NAME	USED BY
A_BIAS	ASP
A_GAIN_0	ASP
A_SCALE	ASP
ALPHA_MATRIX	ASP
AR_FREQUENCY	ARSP
COMM_SYNC_PATTERN	CP
CONTOUR_ALTITUDE	GP
CONTOUR_VELOCITY	GP
DELTA_T	AECLP, GP, RECLP, TDLRSP
DROP_HEIGHT	GP
DROP_SPEED	GP
ENGINES_ON_ALTITUDE	AECLP, GP
FULL_UP_TIME	AECLP
G1	ASP
G2	ASP
G3	GSP
G4	GSP
G_GAIN_0	GSP
G_OFFSET	GSP
GA	AECLP
GAX	AECLP
GP1	AECLP
GP2	AECLP
GPY	AECLP
GQ	AECLP
GR	AECLP
GRAVITY	AECLP, GP
GV	AECLP
GVE	AECLP
GVEI	AECLP
GVI	AECLP
GW	AECLP
GWI	AECLP
M1	TSP
M2	TSP
M3	TSP
M4	TSP
MAX_NORMAL_VELOCITY	GP
OMEGA	AECLP
P1	RECLP
P2	RECLP
P3	RECLP
P4	RECLP
PE_MAX	AECLP
PE_MIN	AECLP
T1	TSP
T2	TSP
T3	TSP
T4	TSP

Table A.6.4 (continued) DATA STORE: RUN_PARAMETERS

VARIABLE NAME	USED BY
TDLR_ANGLES	TDLRSP
TDLR_GAIN	TDLRSP
TDLR_LOCK_TIME	TDLRSP
TDLR_OFFSET	TDLRSP
TE_DROP	AECLP
TE_INIT	AECLP
TE_MAX	AECLP
TE_MIN	AECLP
THETA1	RECLP
THETA2	RECLP
YE_MAX	AECLP
YE_MIN	AECLP

PART III. CONTROL SIGNALS, DATA CONDITIONS, AND GROUP FLOWS

Table A.6.5 CONTROL SIGNALS (OPTIONAL USAGE)

CONTROL SIGNAL NAME
AECLP_DONE
ARSP_DONE
ASP_DONE
CLP_DONE
CP_DONE
CRCP_DONE
GP_DONE
GSP_DONE
INIT_DONE
RECLP_DONE
RENDEZVOUS
RUN_DONE
SP_DONE
TDLRSP_DONE
TDSP_DONE
TSP_DONE

Note: These variables are not in the required global data stores.

Table A.6.6 DATA CONDITIONS (REQUIRED USAGE)

DATA CONDITION VARIABLE NAME
AE_SWITCH
AE_TEMP
CHUTE_RELEASED
CONTOUR_CROSSED
GP_PHASE
RE_SWITCH
TD_SENSED
TDLR_STATE

Table A.6.7 INITIALIZATION DATA

VARIABLE NAME	USED BY
A_ACCELERATION	AECLP, ASP, CP, GP
A_BIAS	ASP
A_COUNTER	ASP
A_GAIN_0	ASP
A_SCALE	ASP
A_STATUS	ASP, CP
AE_STATUS	AECLP, CP
AE_SWITCH	AECLP, GP
AE_TEMP	AECLP, CP, CRCP, GP
ALPHA_MATRIX	ASP
AR_ALTITUDE	ARSP, CP, GP
AR_COUNTER	ARSP
AR_FREQUENCY	ARSP
AR_STATUS	ARSP, CP
ATMOSPHERIC_TEMP	ASP, CP, GSP, TSP
C_STATUS	CP
CHUTE_RELEASED	AECLP, CP, CRCP, GP
CL	AECLP, GP
COMM_SYNC_PATTERN	CP
CONTOUR_ALTITUDE	GP
CONTOUR_CROSSED	AECLP, CP, GP
CONTOUR_VELOCITY	GP
DELTA_T	AECLP, GP, RECLP, TDLRSP
DROP_HEIGHT	GP
DROP_SPEED	GP
ENGINES_ON_ALTITUDE	AECLP, GP
FRAME_BEAM_UNLOCKED	TDLRSP
FRAME_COUNTER	AECLP, ARSP, CP, GP, TDLRSP
FRAME_ENGINES_IGNITED	AECLP, GP
FULL_UP_TIME	AECLP
G1	ASP
G2	ASP
G3	GSP
G4	GSP
G_COUNTER	GSP
G_GAIN_0	GSP
G_OFFSET	GSP
G_ROTATION	CP, GSP, GP, RECLP
G_STATUS	CP, GSP
GA	AECLP
GAX	AECLP
GP1	AECLP
GP2	AECLP
GP_ALTITUDE	AECLP, CP, GP
GP_ATTITUDE	AECLP, CP, GP
GP_PHASE	CP, GP
GP_ROTATION	AECLP, CP, GP
GP_VELOCITY	AECLP, CP, GP
GPY	AECLP
GQ	AECLP
GR	AECLP
GRAVITY	AECLP, GP
GV	AECLP

Table A.6.7 (continued) INITIALIZATION DATA

VARIABLE NAME	USED BY
GVE	AECLP
GVEI	AECLP
GVI	AECLP
GW	AECLP
GWI	AECLP
K_ALT	ARSP, CP, GP
K_MATRIX	CP, GP, TDLRSP
M1	TSP
M2	TSP
M3	TSP
M4	TSP
MAX_NORMAL_VELOCITY	GP
OMEGA	AECLP
P1	RECLP
P2	RECLP
P3	RECLP
P4	RECLP
PE_INTEGRAL	AECLP, CP
PE_MAX	AECLP
PE_MIN	AECLP
RE_STATUS	CP, RECLP
RE_SWITCH	GP, RECLP
SS_TEMP	TSP
SUBFRAME_COUNTER	CP
T1	TSP
T2	TSP
T3	TSP
T4	TSP
TD_COUNTER	TDSP
TD_SENSED	CP, GP, TDSP
TDLR_ANGLES	TDLRSP
TDLR_COUNTER	TDLRSP
TDLR_GAIN	TDLRSP
TDLR_LOCK_TIME	TDLRSP
TDLR_OFFSET	TDLRSP
TDLR_STATE	CP, TDLRSP
TDLR_STATUS	CP, TDLRSP
TDLR_VELOCITY	CP, GP, TDLRSP
TDS_STATUS	CP, GP, TDSP
TE_DROP	AECLP
TE_INIT	AECLP
TE_INTEGRAL	AECLP, CP, GP
TE_LIMIT	AECLP
TE_MAX	AECLP
TE_MIN	AECLP
THERMO_TEMP	TSP
THETA	RECLP
THETA1	RECLP
THETA2	RECLP
TS_STATUS	CP, TSP
VELOCITY_ERROR	AECLP, CP, GP
YE_INTEGRAL	AECLP, CP

YE_MAX	AECLP
YE_MIN	AECLP

Table A.6.8 TEMP_DATA

VARIABLE NAME
SS_TEMP
THERMO_TEMP

Table A.6.9 SENSOR_DATA

VARIABLE NAME
A_COUNTER
AR_COUNTER
TDLR_COUNTER
G_COUNTER
TEMP_DATA
TD_COUNTER

Table A.6.10 OUTPUT_DATA

VARIABLE NAME
AE_CMD
RE_CMD
PACKET

Table A.6.11 OUTPUT_CONTROL

VARIABLE NAME
AE_SWITCH
RE_SWITCH
CHUTE_RELEASED

Table A.6.12 FRAME_DATA

VARIABLE NAME
FRAME_COUNTER
SUBFRAME_COUNTER

A.7 NOTATION FOR LEVELS 0, 1, 2, AND 3 SPECIFICATION

This specification was developed using the extended structured analysis method advocated by Hatley (ref. A.12, A.13) and Cadre's team*work* (ref. A.19). This method is based on a hierarchical approach to defining processes and the associated data and control flows.

The documents constructed as a part of this specification include data context and flow diagrams, control context and flow diagrams, process and control specifications, and a Data Requirements Dictionary. Figure A.7.1 defines the graphical symbols used in the data flow and control flow diagrams**,** respectively.

The data flow diagrams describe the processes, data flows, and data stores. The data context diagram is the highest-level data flow diagram and represents the data flow between the system and the external entities.

The control flow diagrams describe processes, control signal and data condition flows, control specifications, and data stores. The control signal and data condition flows are depicted using directed arcs with broken lines. The control signals listed in the data dictionary may be implemented by the programmer in any form desired, or they may be completely ignored and the control of the program conducted through other means. The control flow diagrams show what the process structure must do under all conditions. Signal flows between the control flow diagram and the control specification have a short bar at the end of the directed arc. The control flow diagrams contain duplicate descriptions of the processes represented on the data flow diagrams. The control context diagram representing the most abstract control flow is similar to the data context diagram.

The control specifications describe the control requirements of a system. These specifications contain the conditions under which the processes detailed in the data and control flow diagrams are activated and de-activated, and in some cases also contain output values for control signals.

The Data Requirements Dictionary contains definitions for data, data conditions, control signals, and group flows.

Following is a list of definitions and explanations for the structured analysis diagrams:

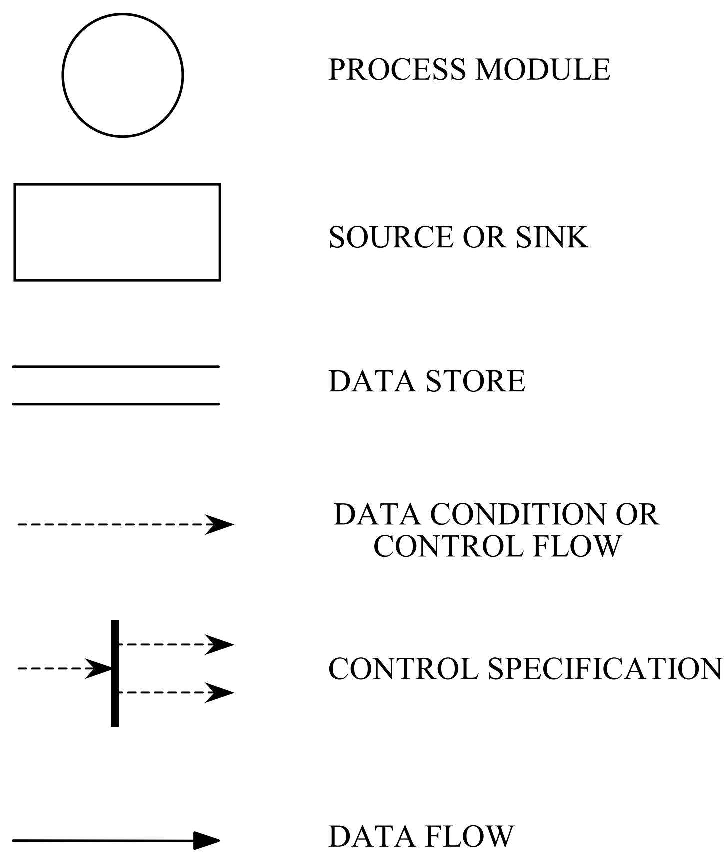
1. The data and control flow names on the directed arcs in the structured analysis figures can be found in the Data Requirements Dictionary Part I, while the group flow names on the arcs can be found in the Data Requirements Dictionary Part III.
2. In the Process Activation Tables, the first column contains the inputs. The second set of columns (separated by two vertical lines) contains the cells which indicate whether a process is to be activated or deactivated. A blank cell indicates that the process is deactivated. An integer indicates that the process is activated. A process whose cell

contains the integer "n" must complete before the process with integer "n+1" is activated. All processes whose cells contain the same integer can be activated in any order. The third set of columns, if present, represents the output values for control signals.

3. The meanings for the symbols used in the expressions for inputs are:

=	equal
~=	not equal
~	logical NOT
&	logical AND
|	logical OR
()	grouping (expression inside parentheses is evaluated first)

Figure A.7.1 GRAPHICAL SYMBOLS USED IN STRUCTURED ANALYSIS DIAGRAMS



A.8 IMPLEMENTATION NOTES

INTERFACE

Background

For the purposes of this research experiment, each GCS implementation must function as if it were actually controlling a planetary lander. In reality, each GCS implementation will be interacting with a software simulator (GCS_SIM) that *models* the behavior of a physical lander when exposed to the environmental forces of a planet.

Due to the fact that each GCS implementation must interact with GCS_SIM as if it were connected to the lander hardware, there are some additional requirements that are placed on a GCS implementation that help define a *software* interface. The software interface to the simulator replaces the physical connection to planetary lander hardware through the use of a simulator support utility and an additional requirement involving the organization of the global data stores.

Simulator Support Utility

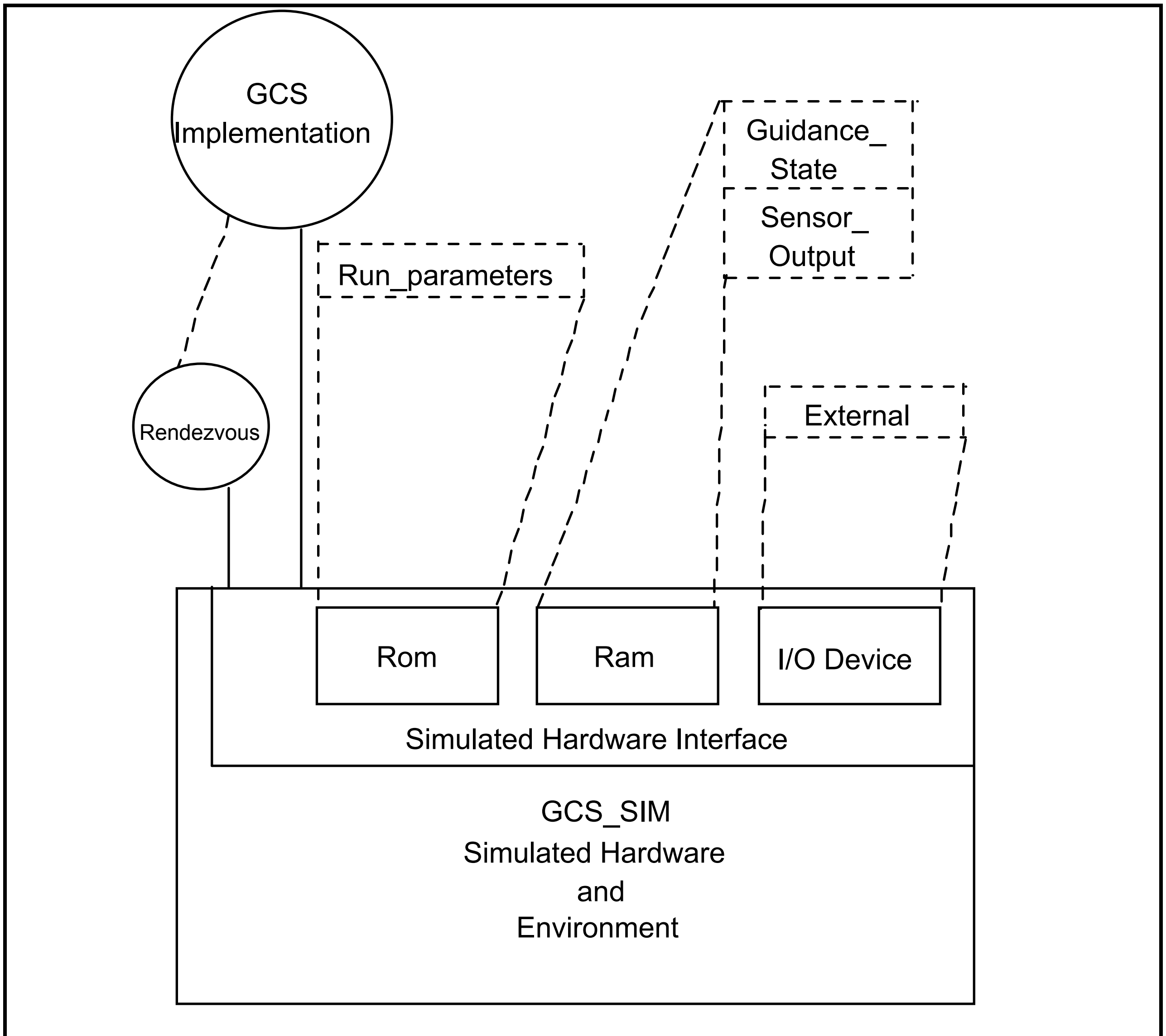
A single simulator support utility (GCS_SIM_RENDEZVOUS) is provided to form a uniform interface between the GCS implementation and the simulation environment (GCS_SIM). This utility is a routine which simplifies the interface between the GCS implementations and the simulation of the vehicle sensing and control mechanisms. This utility also includes a synchronization mechanism for the configurations using more than one version of the GCS. This routine provides the following support functions:

- Initialization for the Beginning of Terminal Descent
- Simulator Rendezvous Synchronization
- GCS Interface for Simulated Reads and Writes

Input/Output

The GCS_SIM_RENDEZVOUS routine simulates all of the input/output operations for each GCS implementation. When using the rendezvous routine with a GCS implementation, all data needed by rendezvous is passed via the four global data stores and there are no additional parameters required. All information *read from* or *written to* each GCS implementation will be transferred through the four global data stores defined in the data dictionary.

Figure A.8.1 DIAGRAM OF STORAGE AS SEEN BY GCS IMPLEMENTATIONS



Hardware Component	Global Data Store (Software Interface)
Input/Output Device	EXTERNAL
Read-Only Memory	RUN_PARAMETERS
On-board Random Access Memory	GUIDANCE_STATE
On-board Random Access Memory	SENSOR_OUTPUT

Process

The GCS uses the sensor input values in order to calculate control commands which are used by GCS_SIM to manipulate the actuators. Since GCS_SIM handles the *orbit to terminal descent* portion of each trajectory, a rendezvous must be issued at the start of each trajectory to load initial sensor values into each GCS implementation. Following the first call to rendezvous, all GCS implementations will synchronize themselves by calling rendezvous prior to the execution of each subframe. This rendezvous, in effect, suspends the GCS implementations until the other GCS implementations have processed this subframe.

The calling convention for this GCS_SIM provided support utility is as follows:

- GCS_SIM_RENDEZVOUS (*requires no parameters*)

GCS Initialization

During the initialization phase of each GCS trajectory (the first call to GCS_SIM_RENDEZVOUS) the frame counter (FRAME_COUNTER) will be updated with the starting frame number for the particular trajectory, and the subframe counter (SUBFRAME_COUNTER) will be initialized to the value one. Under *normal* circumstances, the value of the frame counter will be "1," but the programmer should not rely on that.

By using the interface described above, the simulator can be transparent to the implementation.

A.9 NUMERICAL INTEGRATION INSTRUCTIONS

Within the Guidance Processing functional unit, the calculations of GP_VELOCITY, GP_ALTITUDE, and GP_ATTITUDE require the use of a highly accurate integration method. To maintain the necessary degree of accuracy, three methods of numerical integration have been designated as acceptable for coding, namely Adams-Moulton method, Hamming's method, and the Runge-Kutta fourth-order method for simultaneous equations. If the Runge-Kutta method is used, it is required that the three equations be solved as a set of simultaneous equations.

Each method is briefly described in the following paragraphs, and references to numerical analysis texts describing the method are provided. Algorithms specified in either a text listed or another suitable numerical analysis text should be used during coding.

Adams-Moulton Method

The Adams-Moulton Method requires values from the previous four time steps to calculate the value at the next time step. The Adams-Moulton method is a predictor/corrector method. Both (ref. A.14) (pp. 346-7) and (ref. A.16) (pp. 478-81) explain the Adams-Moulton method.

Hamming's Method

The Hamming method uses a predictor/corrector method similar to that of Adams-Moulton. Hamming's method uses the same predictor as Milne's, but uses a much simpler corrector formula. Milne's method of integration was deemed too unstable for use, but Hamming's method with the simpler corrector is sufficiently stable. A description of both Hamming's method and Milne's method can be found in (ref. A.14) (pp. 347-8).

Runge-Kutta Fourth-Order Method for Simultaneous Equations

The well-known Runge-Kutta fourth-order method for simultaneous equations requires only the previous two values to calculate the next value. References can be found in many texts including: (ref. A.15)(pp. 356-60), (ref. A.17) (pp. 240-6; pp. 282-5), (ref. A.18) (pg. 447; pp. 471-3)

During the first time step, using a numerical integration method necessitates some specification of previous values. These values will be provided during initialization for the data elements provided in Table A.9.1.

TABLE A.9.1 INITIAL VALUES PROVIDED FOR USE IN INTEGRATION

A_ACCELERATION (1..3, 0..4)
AR_ALTITUDE (0..4)
GP_ALTITUDE (0..4)
GP_ATTITUDE (1..3, 1..3, 0..4)
GP_VELOCITY (1..3, 0..4)
G_ROTATION (1..3, 0..4)
K_ALT (0..4)
K_MATRIX (1..3, 1..3, 0..4)
TDLR_VELOCITY (1..3, 0..4)

Note that not all integration required by the GCS specification requires the use of one of the methods listed in this appendix. More specifically, in computing THETA, TE_INTEGRAL, PE_INTEGRAL, and YE_INTEGRAL, Euler's method provides sufficient accuracy and simplicity and should be used. Information on Euler's method may be found in: (ref. A.14)(pp. 318-22), (ref. A.15)(pg. 223), and (ref. A.16)(pp. 462-3).

ADAPTATION OF RUNGE-KUTTE FOURTH-ORDER METHOD FOR SIMULTANEOUS EQUATIONS TO THE GCS SOFTWARE

In the case where the Runge-Kutte method has been selected for integration in the Guidance Processing functional unit, the following gives information on how it is to be applied to GCS. The notation and formulas presented here are merely one representation of the Runge-Kutte method and its adaptation to GCS. The software designer/implementer may vary the notation and/or the form of the equations as long as the algorithm used is equivalent to the one presented here.

The Runge-Kutte fourth-order method (for one dependent variable only) can be summarized as follows:

Given:

Let $dy/dx = f(x,y)$

Let h represent the interval between equidistant values of x

Let the initial values for x and y be x_0 and y_0 respectively

Let $x_1 = x_0 + h$

The problem is to estimate y_1

The solution is:

$y_1 = y_0 + k$

$k = 1/6 \times (k_1 + 2 \times (k_2 + k_3) + k_4)$

where:

$$k_1 = h \times f(x_0, y_0)$$
$$k_2 = h \times f(x_0 + h/2,\ y_0 + k_1/2)$$
$$k_3 = h \times f(x_0 + h/2,\ y_0 + k_2/2)$$
$$k_4 = h \times f(x_0 + h,\ y_0 + k_3)$$

The GCS problem to be solved is as follows:

Simultaneously calculate current values for the variables GP_ATTITUDE, GP_VELOCITY, and GP_ALTITUDE, using the equations for the corresponding derivatives given in GUIDANCE PROCESSING (P-Spec 2.2), Table A.5.8.

Adaptation to GCS of the Runge-Kutte fourth-order method for simultaneous equations

In the discussion that follows, let the "dependent" variables refer to GP_ATTITUDE, GP_VELOCITY, and GP_ALTITUDE, and let the "sensor" variables refer to G_ROTATION, A_ACCELERATION, K_MATRIX, TDLR_VELOCITY, K_ALT, and AR_ALTITUDE. In the Runge-Kutte method, it is assumed that the derivative for y can be obtained as a function of the dependent and independent variables. In GCS, the derivative for each of the dependent variables is a function of some subset of the dependent variables and some subset of the sensor variables. The values for the sensor variables are only available to GCS at discrete values of time, namely at any time which is an integer multiple of the value of DELTA_T. It is therefore not possible to calculate derivatives at the midpoint between two frames. The mapping of the Runge-Kutte independent variable to the GCS time interval is shown below. This mapping should be used, as it will ensure that derivatives can be calculated as required.

Runge-Kutte

```
< ----------------------------- h ---------------------------- >
|________________________________|_____________________________|
x0                          x0 + h/2                          x1
```

GCS

```
< -------DELTA_T--------->< --------DELTA_T------- >
|________________________|_________________________|
t2                       t1                        t0   Time
2                                                  1    0    History
Subscript
n-2                      n-1                       n    Frame
Number
```

where:

$h = 2 \times$ DELTA_T

t_0 = present time (time for the current frame)

$t_1 = t_0$ - DELTA_T (time one frame ago)

$t_2 = t_0$ - (2 x DELTA_T) (time two frames ago)

The Algorithm

The following is intended to be a conceptual representation of the Runge-Kutte algorithm as applied to GCS. It is not intended to be pseudo code or actual code. In this discussion, the subscripts for arrays have been omitted except for the history subscript which appears as "(j)" where j is 0, 1, or 2. This has been done here in order to present the concepts involved concisely, but without low-level details. The previously calculated values of the dependent variables at t_1, although available, are not to be used. Also note that the history values of the dependent and sensor variables with subscripts of 3 and 4 are not used in this adaptation of Runge-Kutte to GCS.

Notation

Let k_1, k_2, k_3 k_4 each represent a 3 x 3 array to hold estimate for change in attitude.
Let l_1, $l_{2,}$ l_3, l_4 each represent a vector of size 3 to hold estimate for change in velocity.
Let m_1, m_2, m_3 m_4 each represent a scalar to hold estimate for change in altitude.

Let SENS_ATT(j) represent the G_ROTATION array with time history subscript j, where j is 0, 1, or 2.
Let SENS_VEL(j) represent the G_ROTATION, A_ACCELERATION, K_MATRIX, and TDLR_VELOCITY arrays with time history subscript (j), where j = 0, 1, or 2.
Let SENS_ALT(j) represent the K_ALT and AR_ALTITUDE arrays with time history subscript j, where
j = 0, 1, or 2.

Let f_att represent the function for derivative of attitude with respect to time.
Let f_vel represent the function for derivative of velocity with respect to time.
Let f_alt represent the function for derivative of altitude with respect to time.

Algorithm

Do first estimates of changes using derivatives calculated at t_2:

$k_1 = h \times$ f_att (GP_ATTITUDE(2), SENS_ATT(2))
$l_1 = h \times$ f_vel (GP_ATTITUDE(2), GP_VELOCITY(2), SENS_VEL(2))
$m_1 = h \times$ f_alt (GP_ATTITUDE(2), GP_VELOCITY(2), GP_ALTITUDE(2), SENS_ALT(2))

Do second estimates of changes using derivatives calculated at t_1:
$k_2 = h \times$ f_att (GP_ATTITUDE(2) + $k_1/2$, SENS_ATT(1))
$l_2 = h \times$ f_vel (GP_ATTITUDE(2) + $k_1/2$, GP_VELOCITY(2) + $l_1/2$, SENS_VEL(1))
$m_2 = h \times$ f_alt (GP_ATTITUDE(2) + $k_1/2$, GP_VELOCITY(2) + $l_1/2$, GP_ALTITUDE(2) + $m_1/2$, SENS_ALT(1))

Do third estimates of changes using derivatives calculated at t_1:
$k_3 = h \times$ f_att (GP_ATTITUDE(2) + $k_2/2$, SENS_ATT(1))
$l_3 = h \times$ f_vel (GP_ATTITUDE(2) + $k_2/2$, GP_VELOCITY(2) + $l_2/2$, SENS_VEL(1))
$m_3 = h \times$ f_alt (GP_ATTITUDE(2) + $k_2/2$, GP_VELOCITY(2) + $l_2/2$, GP_ALTITUDE(2) + $m_2/2$, SENS_ALT(1))

Do fourth estimates of changes using derivatives calculated at t_0 :
$k_4 = h \times$ f_att (GP_ATTITUDE(2) + k_3, SENS_ATT(0))
$l_4 = h \times$ f_vel (GP_ATTITUDE(2) + k_3, GP_VELOCITY(2) + l_3, SENS_VEL(0))
$m_4 = h \times$ f_alt (GP_ATTITUDE(2) + k_3, GP_VELOCITY(2) + l_3, GP_ALTITUDE(2) + m_3, SENS_ALT(0))

Add weighted average of four change estimates to previous value of dependent variable to get current dependent variable:
GP_ATTITUDE(0) = GP_ATTITUDE(2) + 1/6 $\times$ $(k_1 + 2 \times (k_2 + k_3) + k_4)$
GP_VELOCITY(0) = GP_VELOCITY(2) + 1/6 $\times$ $(l_1 + 2 \times (l_2 + l_3) + l_4)$
GP_ALTITUDE(0) = GP_ALTITUDE(2) + 1/6 $\times$ $(m_1 + 2 \times (m_2 + m_3) + m_4)$

A.10 COMMUNICATIONS PACKET INSTRUCTIONS

STRUCTURE OF PACKET

The global variable PACKET is defined in the data dictionary as an array of 256 elements of type Integer*2. The actual memory which holds this array can also be thought of as an array of 512 elements of type Byte. The message to be transmitted can therefore be thought of as a series of contiguous bytes, as illustrated in Table A.5.7. The message on which the checksum is to be calculated consists of the synchronization pattern, the sequence number, the sample mask, and the data section. The data section always begins in the eighth byte, but the position of the last byte of the data section depends upon the particular subframe in which the packet is being transmitted. The checksum is always in the two bytes immediately following the last used byte of the data section for the subframe, or in other words, immediately following the message. The bytes of PACKET following the checksum are unused.

Subframe	Byte Position of Message	Position of Least Significant Byte of Checksum	Position of Most Significant Byte of Checksum
1	1 - 129	130	131
2	1 - 173	174	175
3	1 - 45	46	47

PROCEDURE FOR CALCULATING THE CHECKSUM

The message polynomial is to be formed as described below, and then is to be multiplied by 2^{16}. This product polynomial is to be divided by the generator polynomial, using modulo 2 arithmetic. The 16-bit remainder obtained from this division (with its bits in reverse order) is the checksum, and is to be placed into the packet immediately following the message.

Conventions

Byte 1 of the synchronization pattern will be referred to as the first byte of the message, while the last used byte of the data section will be referred to as the last byte of the message. Each number appearing below is given with the most significant digit on the left, and the least significant digit on the right. When bit numbers are referenced, they are the VAX FORTRAN bit numbers (bit 0 is the least significant bit, while bit 7 is the most significant bit of the byte).

Form the Message Polynomial

Let n represent the number of bytes in the message

Let $pbyte_i$ represent byte i of the packet

Let $bit_{i,j}$ represent bit j of byte i of the packet. Then,

$<pbyte_i> = <bit_{i,7}><bit_{i,6}><bit_{i,5}><bit_{i,4}><bit_{i,3}><bit_{i,2}><bit_{i,1}><bit_{i,0}>$

$<mbyte_i> = <bit_{i,0}><bit_{i,1}><bit_{i,2}><bit_{i,3}><bit_{i,4}><bit_{i,5}><bit_{i,6}><bit_{i,7}>$

$<\text{Message Polynomial}> = <mbyte_1><mbyte_2>.....<mbyte_n>$

In other words, the message polynomial is formed by taking the bytes of the message in order from the first to the last, but within each byte, taking the bits in order from the least to the most significant.

Form the Dividend

<Dividend> = <Message Polynomial><0000000000000000>

The dividend is formed by multiplying the message polynomial by 2^{16}, or in other words, by appending 16 zeroes to the end of the polynomial.

Form the Divisor

<Divisor> = <11000000000000101>

The divisor is the CRC-16 generator polynomial, which is $x^{16} + x^{15} + x^2 + x^0$

Perform the Long Division

Divide the dividend by the divisor, using modulo two arithmetic.

Form the Checksum and Place it into the Packet

Let R represent the final 16-bit remainder from the long division.

Let $<R_i>$ represent bit i of R. Then,

$<R> = <R_{15}><R_{14}><R_{13}><R_{12}><R_{11}><R_{10}><R_9><R_8><R_7><R_6><R_5><R_4><R_3><R_2><R_1><R_0>$

$<\text{Checksum}> = <R_0><R_1><R_2><R_3><R_4><R_5><R_6><R_7><R_8><R_9><R_{10}><R_{11}><R_{12}><R_{13}><R_{14}><R_{15}>$

Thus, the checksum is the final 16-bit remainder, with the bits reversed.
The checksum is to be placed into the packet in standard VAX byte order, immediately following the last used byte of the message.

CHECKSUM ALGORITHMS
While different algorithms exist for calculating the checksum, any algorithm used in an implementation must be equivalent to, or accomplish the same results as the procedure described above.

EXAMPLE OF THE CALCULATION OF A CHECKSUM
Assume that the message to be sent consists of four bytes (this message is obviously shorter than any message to be sent in any GCS subframe, but it is infeasible to present an example with a message of 45 bytes or more).
Assume the message to be sent is:

Byte Position	Contents in Hexadecimal	Contents in Binary
1	44	01000100
2	4F	01001111
3	56	01010110
4	45	01000101

In this example, the long division(in binary) is as follows:

```
                      111100101000110011100011110110
11000000000000101  )00100010111100100110101010100010000000000000000
                      11000000000000101
                         10010111100101100
                         11000000000000101
                          10101111001010011
                          11000000000000101
                           11011110010101100
                           11000000000000101
                              11110010101001101
                              11000000000000101
                                11001010100100001
                                11000000000000101
                                    10101001001000001
                                    11000000000000101
                                     11010010010001000
                                     11000000000000101
                                        10010010001101000
                                        11000000000000101
                                         10100100011011010
                                         11000000000000101
                                          11001000110111110
                                          11000000000000101
                                              10001101110110000
                                              11000000000000101
                                               10011011101101010
                                               11000000000000101
                                                10110111011011110
                                                11000000000000101
                                                 11101110110110110
                                                 11000000000000101
                                                   10111011011001100
                                                   11000000000000101
                                                    11110110110010010
                                                    11000000000000101
                                                      1101101100101110
```

The remainder is 1101101100101110

The checksum is then the remainder with the bits reversed, or: 0111010011011011

The two bytes of the checksum will then be placed into the bytes immediately following the data portion, in standard VAX byte order (low order byte first followed by high order byte) as follows:

Byte Position	Contents in Hexadecimal	Contents in Binary
1	44	01000100
2	4F	01001111
3	56	01010110
4	45	01000101
5	DB	11011011
6	74	01110100

A.11 BIBLIOGRAPHY

A.1 George B. Finelli. Results of software error-data experiments. In AIAA/AHS/ASEE Aircraft Design, Systems and Operations Conference, Atlanta, GA, September 1988.

A.2 Special Committee 167 of Requirements and Technical Concepts for Aviation Inc. (RTCA, Inc.). Software Considerations in Airborne Systems and Equipment Certification, DOCUMENT NO. RTCA/DO-178B. RTCA Inc., Washington, D. C., 1992

A.3 Harm Buning and D. T. Greenwood. Flight mechanics for space and re-entry vehicles. Technical report, The University of Michigan Engineering Summer Conferences, Summer 1964.

A.4 Herbert Goldstein. Classical Mechanics. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, USA, 1959.

A.5 Irving H. Shames. Engineering Mechanics -- Statics and Dynamics. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1980.

A.6 Dan Edwin Christie. Vector Mechanics. McGraw-Hill Inc., New York, 1964

A.7 David Hestenes. New Foundations for Classical Mechanics. D. Reidel Publishing Company, Boston, 1986

A.8 D. N. Burghes and A. M. Downs. Classical Mechanics and Control. Ellis Horwood Limited, Coll House, Westergate, England, 1975.

A.9 G. S. Light and J. B. Higham. Theoretical Mechanics. Longman Inc., New York, 1975.

A.10 Don C. Rich and J. R. Dunham. Guidance and Control Software Simulator (GCS_SIM) Software Specification. Technical Report NASA Contract NAS1-17964; Task Assignment No. 8, Research Triangle Institute, Research Triangle Park, NC, 1987.

A.11 Andrew S. Tanenbaum. Computer Networks. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

A.12 Derek J. Hatley. The use of structured methods in the development of large, software-based avionics systems. In Proceedings of the AIAA/IEEE 6th Digital Avionics Systems Conference, New York, December 1984.

A.13 Derek J. Hatley and Imtiaz A. Pirbhai, Strategies for Real-Time System Specification. Dorset House Publishing Company, New York, New York, 1987.

A.14 W. Allen Smith. Elementary Numerical Analysis. Harper and Row, New York, 1979.

A.15 J. B. Scarborough. Numerical Mathematical Analysis. The Johns Hopkins Press, Baltimore, 1962

A.16 Stephen M. Pizer. Numerical Computing and Mathematical Analysis. Science Research Associates, Inc., Chicago, 1975

A.17 Richard L. Burden and J. Douglas Faires. Numerical Analysis. PWS-KENT Publishing Company, Boston, 1989.

A.18 Melvin J. Maron and Robert J. Lopez. Numerical Analysis: A Practical Approach. Wadsworth Publishing Company, Belmont, California, 1990.

A.19 team*work*/SA team*work*/RT User's Guide, Cadre Technologies, Inc., Providence, Rhode Island, Release 4.0, 1990.

A.20 Neil A. Holmberg, Robert P. Faust, and H. Milton Holt, Viking '75 Spacecraft Design and Test Summary, Volume I - Lander Design, NASA Reference Publication 1027, Langley Research Center, Hampton, Virginia, 1980

Appendix B: Design Description for the Pluto Implementation of the Guidance and Control Software

Authors: Philip Morris and Rob Angellatta, Lockheed Martin Engineering and Sciences Corp.

This document was produced as part of Guidance and Control Software (GCS) Project conducted at NASA Langley Research Center. Although some of the requirements for the Guidance and Control Software application were derived from the NASA Viking Mission to Mars, this document does not contain data from an actual NASA mission.

B. CONTENTS

B.1 INTRODUCTION TO PLUTO GCS DESIGN....B-3

B.1.1 TOP-LEVEL DESCRIPTION....B-3
B.1.2 DESIGN METHODOLOGY....B-3
B.1.3 DESIGN SYNTAX SPECIFICATIONS....B-4

B.2 DESIGN STRUCTURE....B-4

B.2.1 HIGH-LEVEL SOFTWARE DESIGN....B-4
B.2.2 DATA AND CONTROL FLOW....B-6
B.2.3 MODULE DESCRIPTION....B-6
B.2.4 PROCESS SCHEDULING....B-16
B.2.5 DATA DICTIONARY....B-16
B.2.6 DERIVED REQUIREMENTS....B-16

B.3 REFERENCES....B-16

B.4 TEAMWORK DESIGN....B-17

B.1 Introduction to Pluto GCS Design

This document contains a detailed description of the Pluto software design. The Pluto software design fully encompasses all software requirements as presented in the GCS Guidance and Control Software Development Specification (ref. B.3) defining a GCS implementation. The Pluto design provides full and complete software design specifications suitable for coding a GCS implementation.

B.1.1 Top-Level Description

A GCS implementation represents the guidance and control subsystem of a planetary landing vehicle. The guidance and control subsystem provides navigation, guidance, and attitude control for the lander during the terminal phase of a planetary landing. The Terminal phase of a planetary landing refers to the vehicle events beginning with the separation from the aeroshell to the actual contact with the planet surface.

The overall objective of the guidance and control subsystem is to effect a safe landing and to communicate the lander's telemetry data to a remote receiving station. Pluto implements a velocity-altitude contour (VAC) strategy for fulfilling the guidance and control responsibilities. The VAC strategy consists of attempting to match the vehicle's actual velocity-altitude contour with a predetermined descent contour stored in the flight software.

The communication task consists of preparing the appropriate telemetry data for transmission by the on-board communication gear. Preparing the telemetry data involves building a communications packet containing various vehicle guidance and control information. Periodically, a communications packet is prepared and provided to the communications gear for transmission.

B.1.2 Design Methodology

The Pluto design specification has been developed using the structured analysis with real time extensions (SA/RT) methodology as embodied in Cadre's Teamwork/SA (ref. B.4) and Teamwork/RT software development tools. Cadre's Teamwork/SA implements the Structured Analysis (SA) approach to systems analysis as described by DeMarco (ref. B.5). Cadre's Teamwork/RT is the companion product of Teamwork/SA implementing the real-time extensions to SA as described by Hatley (ref. B.6).

The SA/RT software specification methodology emphasizes data flow between processes. Individual processes are activated when their input data is available. In addition, explicit control specifications are available for describing process sequencing which is often necessary in real-time systems.

Note that both the SA and SA/RT methodologies are intended to describe software development specifications. The Structured Design (SD) methodology, as described by Page-Jones (ref. B.7), is more appropriate for describing the Pluto design specifications. However, the Pluto design was originally developed using SA/RT and during the transition to in-house software development, a decision was made to stay with SA/RT. The potential loss in design description capability directly attributable to the SA/RT methodology as compared with the SD methodology is minimal as compared to the loss of development man-hours it would cost to convert the design from SA/RT to SD during the transition phase.

B.1.3 Design Syntax Specifications

The main criterion for choosing an algorithm language for describing the Pluto design was that the algorithms should be clear, concise, and easy to read. No specific language was chosen to describe the algorithms of the Pluto design; rather, a "structured English" approach is followed. The language, with a few exceptions, is very similar to the Pascal programming language.

The algorithm description language's major deviations from Pascal are as follows: First, blocks are not delimited by "BEGIN/END" pairs. Blocks are readily apparent and an "END" appears in a few instances to clarify the end of a block. Semicolons are not used to signify the end of a statement. Again, the end of the statements is quite obvious.

A few conventions were borrowed from the C programming language. Hexadecimal value notation appears as 0xdddd where "0x" identifies "dddd" as a hexadecimal value. A few bitwise binary operators are introduced; "&" signifies bitwise *and* operation, "XOR" signifies bitwise *exclusive or* operation, and ">>" represents the bitwise *shift right* operation.

Two syntax features are peculiar to P-Spec 1.8 CP. The at sign "@" was selected to serve as an indirection operator. It has the same semantics as Pascal's "^". The reason for not maintaining the caret "^" for specifying indirection is that it was previously chosen to signify exponentiation. The Modula-2 record syntax was selected for specifying records primarily for it's ability to support deviate record structure. Also, when specifying the records, it is necessary to define the size of particular data types. Several terms were introduced for specifying the size of particular data elements: byte - an 8-bit quantity, word - a 16-bit quantity, longword a 32-bit quantity, and quadword a 64-bit quantity.

B.2 Design Structure

In the Teamwork representation of the Pluto design given in B.4, the SA/RT software specification methodology organizes the design as a top-down functional decomposition. As such, the Pluto design described in this section follows the top-down functional decomposition.

B.2.1 High-level Software Design

The ultimate goal of the guidance and control subsystem is to safely land the vehicle onto the planet's surface. Pluto attempts to safely land the vehicle by sensing the vehicle's position relative to the designated landing surface and commanding the vehicle's locomotive resources in an effort to maintain a predetermined descent contour. A secondary goal of the guidance and control subsystem is to provide periodic communications of the vehicle's telemetry data.

The design context diagram depicts Pluto as a process transforming raw sensor data into various output data. The raw sensor data originates from the on-board sensors. As a process, Pluto transforms the incoming raw sensor data into engine data which is passed on to the engine controller and packet data which is passed on to the communications gear, and when appropriate issues the chute_released signal.

Pluto organizes the vehicle terminal descent as a sequence of time slices. That is, Pluto divides the vehicle's journey into regular intervals of time. Each "time slice", or frame, has a well defined time duration as specified by the constant DELTA_T. During each frame, Pluto first determines the vehicle's position relative to the planet's surface and computes the vehicle's actual descent contour, then Pluto decides how closely the actual descent contour matches the predetermined VAC, and finally computes and issues the necessary corrective action.

As depicted in DFD 0, Pluto processing is partitioned into three processes. Process 1, the Sensor Processing Subframe, is responsible for gathering and transforming, if necessary, the current information available from the vehicle's sensors. Process 2, the Guidance Processing Subframe, is responsible for determining if the vehicle's actual VAC matches the preprogrammed VAC. Process 3, the Control Law Subframe is responsible for maneuvering necessary to put the vehicle closer to the preprogrammed VAC.

Each frame consists of performing the Sensor Processing Subframe processing, followed by the Guidance Processing Subframe processing, followed by the Control Law Subframe processing. This control structure is represented by PAT 0-s1. When Pluto processing is started, the data element GP_PHASE is initialized to 1. Pluto processing always begins at the beginning of a frame and will always terminate at the completion of a frame. Termination occurs at the completion of Control Law Processing Subframe processing during the frame in which Guidance Processing Subframe processing asserts the signal GP_PHASE to 5.

The Sensor Processing Subframe is responsible for collecting the information provided by the on-board sensors. The vehicle's sensors include accelerometers, gyroscopes, temperature sensors, an altimeter radar, a four-beam Doppler radar, and a touch-down switch. Sensor Processing Subframe processing is decomposed into eight distinct tasks as represented by the eight processes of DFD 1. The specific responsibilities assigned to each process are detailed in section 2.3 below.

PAT 1-s1 contains the control specification for the processes of DFD 1, Sensor Processing Subframe. It is not immediately obvious why the data element SUBFRAME_COUNTER was selected as the input to PAT 1-s1. Within each of the three "subframe" processes, a specific order of process activation is required. This particular ordering is necessary when the activation of some process depends upon the completion of another process.

The PAT is a control specification designed specifically for representing the ordering of process activation. The Pat specifies dependencies in the ordering of process activation via the conditions of the input signals. Although in PAT 1-s1, signal conditions are not necessary for determining the sequencing of process activation's, the Teamwork SA/RT implementation of the PAT requires an input signal. So, an input signal, the data element SUBFRAME_COUNTER, and it's value have been selected which always evaluates to "true". This is also the case with PAT 2-s1 and PAT 3-s1.

The major responsibilities of the Guidance Processing Subframe are to determine the current state of the vehicle and to determine how closely the actual vehicle VAC matches the preprogrammed VAC. These tasks are partitioned into three processes as depicted on DFD 2. The process named GCS_SIM_RENDEZVOUS appears on DFD 1, DFD 2 and DFD 3. All three "bubbles" represent the same process. At the beginning of each "subframe", Pluto is required to contact the other vehicle subsystems. GCS_SIM_RENDEZVOUS processing provides the interface to the other vehicle subsystems. The requirements for GCS_SIM_RENDEZVOUS are described in section 2.3 below.

Once Pluto has determined the present vehicle state, it is necessary to command the vehicle's locomotive resources, if available, in an effort to maintain the desired VAC. This responsibility is charged to the Control Law Processing Subframe. This processing is responsible for releasing the parachute and computing the appropriate engine commands. The process named CP appears on DFD 1, DFD 2 and DFD 3. All three "bubbles" represent the same process. At the end of each "subframe", Pluto is required to transmit particular telemetry data to a remote receiving station. CP processing is delegated the task of periodic telemetry communications.

B.2.2 Data and Control Flow

Consistent with the Software Development Specification, Pluto organizes its global storage into four data stores labeled EXTERNAL, GUIDANCE_STATE, SENSOR_OUTPUT, RUN_PARAMETERS. The data dictionary describes the organization of each data store and describes each of the data elements comprising the data stores.

The data stores are represented on DFD 1, DFD 2, and DFD 3. Each DFD clearly depicts the data flows between the represented processes and the data stores. It is important to note that a non-labeled data flow indicates that all data elements contained in the data store are available in the flow. The data flows originating and terminating in the process GCS_SIM_RENDEZVOUS are not labeled. All data elements stored in each of the data stores is available as input and output to the process GCS_SIM_RENDEZVOUS. However, GCS_SIM_RENDEZVOUS does not necessarily process as input or update as output all of the elements of each data store. The Pluto control flow is described above in section 2.1 High-level software design.

B.2.3 Module Description

Process specifications, better known as P-Specs, reside at the lowest level of decomposition in the SA/RT development methodology. P-Specs provide a functional description of the necessary processing within a process. A map to the P-Specs found in Pluto is presented below.

The Sensor Processing Subframe provides the guidance and control subsystem with an interface to the vehicle's on-board sensors. The vehicle's sensors provide Pluto with information pertaining to the lander's current state within the terminal descent operation. Sensor Processing Subframe processing is decomposed into eight distinct tasks as described below.

The GCS_SIM_RENDEZVOUS process is responsible for the Pluto communications with other vehicle subsystems. GCS_SIM_RENDEZVOUS has both read and write access to all four of the global stores. The actual implementation of the GCS_SIM_RENDEZVOUS functionality will be provided to the implementer. GCS_SIM_RENDEZVOUS is represented in the Sensor Processing subframe by DFD 1.1, in the Guidance Processing subframe by DFD 2.1, and in the Control Law Processing subframe by DFD 3.1. The functional processing of GCS_SIM_RENDEZVOUS is represented in P-Spec 1.1.

The ARSP process is responsible for determining the distance from the vehicle to the landing surface. ARSP processes data originating from the on-board altimeter radar sensor and reports the vehicle's altitude above the planet's surface. DFD 1.2 represents the role of ARSP in the Sensor processing subframe and P-Spec 1.2 specifies the ARSP functional processing.

ARSP processing requires an extrapolation algorithm for computing the value of AR_ALTITUDE. The development specifications calls for extrapolating a value for AR_ALTITUDE from a third-order polynomial fitted to the previous four values of AR_ALTITUDE. Given four equally spaced values, we can approximate the third order function representing the polynomial containing the given values. The fifth value in the series may then be extrapolated from this function. The value of DELTA_T represents the spacing of the values stored in AR_ALTITUDE.

ARSP employs the divided difference technique for performing the necessary extrapolation. Begin by constructing a difference table for the given values. The first column represents the given values of AR_ALTITUDE reported in the most recent previous four frames. The second column entries are computed as the difference between adjacent column one entries. Similarly, the third column entries are computed as the difference between adjacent column two entries.

The fourth column is computed as the difference between the column three entries. Letting "A" represent the data element "AR_ALTITUDE" and "t" represent the current frame, the table appears as:

```
Frame
number                         Column
------   --------------------------------------------------
         1      2                3                  4

  t-4    A[4]
            A[3]-A[4]
  t-3    A[3]          (A[2]-A[3])-(A[3]-A[4])
            A[2]-A[3]    ((A[1]-A[2])-(A[2]-A[3]))-((A[2]-A[3])-(A[3]-A[4]))
  t-2    A[2]          (A[1]-A[2])-(A[2]-A[3])
             A[1]-A[2]
  t-1    A[1]

  t-0
```

An extrapolation of the altitude for the current frame is constructed by summing the last element of each column:

```
A[0] = A[1] + A[1]-A[2] + (A[1]-A[2])-(A[2]-A[3]) +
       ((A[1]-A[2])-(A[2]-A[3]))-((A[2]-A[3])-(A[3]-A[4]))
```

Simplifying the equation yields:

```
A[0] =  4*A[1] - 6*A[2] + 4*A[3] - A[4]
```

The ASP process is responsible for determining the vehicle accelerations along each of it's three axes. ASP processes data originating from the on-board accelerometers and reports the vehicle accelerations. DFD 1.3 represents the role of ASP in the Sensor processing subframe and P-Spec 1.3 specifies the ASP functional processing.

The CP process is responsible for preparing a data packet suitable for transmission by the on-board communications gear. CP collects the appropriate data from the four global stores and arranges them into a data packet. CP is represented in the Sensor Processing subframe by DFD 1.8, in the Guidance Processing subframe by DFD 2.3, and in the Control Law Processing subframe by DFD 3.5. The functional processing of CP is described in P-Spec 1.8.

The GSP process is responsible for determining the vehicle's rotation rates. GSP processes data originating from the on-board gyroscope sensors and reports the vehicle rotation rates. DFD 1.4 represents the role of GSP in the Sensor processing subframe and P-Spec 1.4 specifies the GSP functional processing.

The TDLRSP process is responsible for computing vehicle's descent velocities. TDLRSP processes data originating from the on-board touch down landing radar sensor and reports the vehicle descent velocities. DFD 1.5 represents the role of TDLRSP in the Sensor processing subframe and P-Spec 1.5 specifies the TDLRSP functional processing.

The TDSP process is responsible for determining the vehicle's touch down status. TDSP processes data originating from the on-board touch down sensor and reports the vehicle's touch down status. DFD 1.6 represents the role of TDSP in the Sensor processing subframe and P-Spec 1.6 specifies the TDSP functional processing.

The TSP process is responsible for determining the ambient atmospheric temperature. TSP processes data originating from the two on-board temperature sensors and reports the ambient atmospheric temperature. DFD 1.7 represents the role of TSP in the Sensor processing subframe and P-Spec 1.7 specifies the TSP functional processing.

TSP contains four algorithms for computing the atmospheric temperature. The algorithm for computing the temperature based on the solid state (SS) sensor has been derived as follows. The task is to determine the linear function specifying the linear equation containing the points (M1, T1) and (M2, T2).

$$
\begin{aligned}
&y = mx + b \\
&\text{where: } m \text{ is the slope of the line} \\
&\qquad\quad\; b \text{ is the y intercept}
\end{aligned}
$$

substituting the given point (M1, T1) for (x, y):

$$
\begin{aligned}
T1 &= m \cdot M1 + b \\
b &= T1 - m \cdot M1
\end{aligned}
$$

the slope of the line is expressed by the delta y divided by delta x:

$$m = \frac{T2 - T1}{M2 - M1}$$

substituting into the point-slope equation gives:

$$solid\_state\_temp = \frac{T2 - T1}{M2 - M1} \cdot SS\_TEMP + T1 - \frac{T2 - T1}{M2 - M1} \cdot M1$$

The algorithm for converting a sensor measure residing in the lower parabolic region of the thermo-couple (TC) sensor was developed as follows. The first task is to determine the function which describes the lower parabolic region of the TC sensor:

$$
\begin{aligned}
&y = \frac{1}{4p} \cdot (x - h)^2 + k \\
&\text{where: } (h,k) \text{ is the vertex} \\
&\qquad\quad y = (k - p) \text{ is the directrix}
\end{aligned}
$$

Given that "the temperature goes down as the square of the measurement":

$$
\begin{aligned}
&y = \frac{1}{4p} \cdot (x - h)^2 + k && \text{standard equation of parabola} \\
&y = -(x - h)^2 + k && \text{from spec. "goes down as the square"} \\
&\frac{1}{4p} = -1 \\
&p = -\frac{1}{4}
\end{aligned}
$$

The derivative of a function at a given point is equivalent to the slope of the line tangent to the function at the given point.

$$
\begin{aligned}
f(x) &= y = -(x - h)^2 + k \\
f'(x) &= -2(x - h)
\end{aligned}
$$

The slope of the line tangent to point (M3, T3):

$$
\begin{aligned}
m &= -2(M3 - h) \\
h &= M3 + \frac{m}{2}
\end{aligned}
$$

Note, the slope of the line tangent to the point is equivalent to the slope of the line containing the points (M3, T3) and (M4, T4), so:

$$m = \frac{T4 - T3}{M4 - M3}$$

substituting the given point (M3, T3) for (x, y):

$$T3 = -(M3 - (M3 + \frac{m}{2}))^2 + k$$
$$k = T3 + \left(\frac{m}{2}\right)^2$$

The function representing the TC sensor lower parabolic region:

$$y = \frac{1}{4p} \cdot (x - h)^2 + k$$
$$where:\ p = -\frac{1}{4},\ \ h = M3 + \frac{m}{2},\ \ k = T3 + \left(\frac{m}{2}\right)^2,\ and\ \ m = \frac{T4 - T3}{M4 - M3}$$
$$lower\_parabolic\_function = -\left(x - \left(M3 + \left(\frac{\frac{T4 - T3}{M4 - M3}}{2}\right)\right)\right)^2 + T3 + \left(\frac{\frac{T4 - T3}{M4 - M3}}{2}\right)^2$$

Similarly, the algorithm for converting a sensor measure residing in the upper parabolic region of the thermo-couple sensor was developed as follows. The function which describes the upper parabolic region of the TC sensor:

$$y = \frac{1}{4p} \cdot (x - h)^2 + k$$
where: (h,k) is the vertex
y = (k - p) is the directrix

Given "the temp. goes up as the square of the measurement":

$$y = \frac{1}{4p} \cdot (x - h)^2 + k \qquad \textit{standard equation of parabola}$$
$$y = (x - h)^2 + k \qquad \textit{from spec. "goes up as the square"}$$
$$\frac{1}{4p} = 1$$
$$p = \frac{1}{4}$$

The derivative of a function at a given point is equivalent to the slope of the line tangent to the function at the given point.

$$f(x) = y = (x - h)^2 + k$$
$$f'(x) = 2(x - h)$$

The slope of the line tangent to point (M4, T4):

$$m = 2(M4 - h)$$
$$h = M4 - \frac{m}{2}$$

Note, the slope of the line tangent to the point is equivalent to the slope of the line containing the points (M3, T3) and (M4, T4), so:

$$m = \frac{T4 - T3}{M4 - M3}$$

substituting the given point (M4, T4) for (x, y):

$$T4 = (M4 - (M4 + \frac{m}{2}))^2 + k$$
$$k = T4 - \left(\frac{m}{2}\right)^2$$

The function representing the TC sensor upper parabolic region:

$$y = \frac{1}{4p} \cdot (x - h)^2 + k$$
$$where:\ p = \frac{1}{4},\quad h = M4 - \frac{m}{2},\quad k = T4 - \left(\frac{m}{2}\right)^2,\ and\ m = \frac{T4 - T3}{M4 - M3}$$
$$upper\_parabolic\_function = \left(x - \left(M4 - \left(\frac{\frac{T4 - T3}{M4 - M3}}{2} \right) \right) \right)^2 + T4 - \left(\frac{\frac{T4 - T3}{M4 - M3}}{2} \right)^2$$

And finally, the algorithm for converting a sensor measure residing in the linear region of the thermo-couple sensor was developed as follows. The task is to determine the linear function specifying the linear equation containing the points (M3, T3) and (M4, T4).

$$y = mx + b$$
$$where:\quad m\ is\ the\ slope\ of\ the\ line$$
$$b\ is\ the\ y\ intercept$$

substituting the given point (M3, T3) for (x, y):

$$T3 = m \cdot M3 + b$$
$$b = T3 - m \cdot M3$$

the slope of the line is expressed by the delta y divided by delta x:

$$m = \frac{T4 - T3}{M4 - M3}$$

substituting into the point-slope equation gives:

$$tc\_linear\_temp = \frac{T4 - T3}{M4 - M3} \cdot THERMO\_TEMP + T3 - \frac{T4 - T3}{M4 - M3} \cdot M3$$

The GP process is responsible for the guidance tasks of the vehicle. Guidance tasks include determining the current vehicle VAC, determining how closely the actual vehicle VAC matches the preprogrammed VAC, determining which set of engine control law should be in effect, and determining the appropriate state for the engines. GP processes data originating in the Sensor Processing Subframe, the preprogrammed run parameters, and the engine state data in performing the various guidance tasks. DFD 2.2 represents the role of GP in the Sensor processing subframe and P-Spec 2.2 specifies the GP functional processing.

When computing the optimal velocity during the GP processing, there is one case where interpolation is necessary and two cases where extrapolation is required. Note, in order to implement the following routines, it is assumed that the velocity altitude array data contains at least two valid entries.

Given the point (x_0, $f(x_0)$), the point (x_1, $f(x_1)$), and the x value of the desired point (x, f(x)) where $x_0 < x < x_1$, interpolate to find f(x)

$$\frac{f(x)-f(x_0)}{x-x_0}=\frac{f(x_1)-f(x_0)}{x_1-x_0}$$
$$f(x)=f(x_0)+\frac{f(x_1)-f(x_0)}{x_1-x_0}\cdot\left(x-x_0\right)$$

There are two cases for extrapolation. First, if the desired value is greater than the largest value stored in the velocity-altitude contour table, then a value must be extrapolated above the largest value stored in the table. Letting (x_0, $f(x_0)$) refer to the second largest value stored in the velocity-altitude contour table and (x_1, $f(x_1)$) refer to the largest value stored in the velocity-altitude contour table, the formula for extrapolation is:

$$\frac{f(x)-f(x_0)}{x-x_0}=\frac{f(x_1)-f(x_0)}{x_1-x_0}$$
$$f(x)=f(x_0)+\frac{f(x_1)-f(x_0)}{x_1-x_0}\cdot\left(x-x_0\right)$$

This is not misprinted, it is indeed the same formula displayed above describing the interpolation.

In the case where the desired value is less than the smallest value stored in the velocity-altitude contour table, then a value must be extrapolated below the smallest value stored in the table. Letting (x_0, $f(x_0)$) refer to the smallest value stored in the velocity-altitude contour table and (x_1, $f(x_1)$) refer to the second smallest value stored in the velocity-altitude contour table, the formula for extrapolation is:

$$\frac{f(x_0)-f(x)}{x_0-x}=\frac{f(x_1)-f(x_0)}{x_1-x_0}$$
$$f(x)=f(x_0)-\frac{f(x_1)-f(x_0)}{x_1-x_0}\cdot\left(x_0-x\right)$$

GP is responsible for computing the current values of the data elements GP_ATTITUDE, GP_VELOCITY, and GP_ALTITUDE. The current value of GP_ATTITUDE is expressed by the following formula:

$$GP\_ATTITUDE_t = GP\_ATTITUDE_{t-2} + \int_{t-2}^{t} \left(GP\_A\dot{T}TITUDE \right) dt$$

where

$$GP\_A\dot{T}TITUDE = GP\_ROTATION \times GP\_ATTITUDE$$

The current value of GP_VELOCITY is expressed by the following formula:

$$GP\_VELOCITY_t = GP\_VELOCITY_{t-2} + \int_{t-2}^{t} \left(GP\_VE\dot{L}OCITY \right) dt$$

where

$$\begin{aligned} GP\_VE\dot{L}OCITY = {} & GP\_ROTATION \times GP\_VELOCITY + ACCEL + \\ & \left(K\_MATRIX \times \left(TDLR\_VELOCITY - GP\_VELOCITY \right) \right) \end{aligned}$$

where ACCEL is a 3x1 matrix specified as:

$$\begin{aligned} & do\ i = 1\ to\ 3 \\ & \quad ACCEL[i] = GRAVITY \cdot GP\_ATTITUDE[i,3] + A\_ACCELERATION[i] \\ & end\ do \end{aligned}$$

The current value of GP_ALTITUDE is expressed by the following formula:

$$GP\_ALTITUDE_t = GP\_ALTITUDE_{t-2} + \int_{t-2}^{t} \left(GP\_AL\dot{T}ITUDE \right) dt$$

where

$$\begin{aligned} GP\_AL\dot{T}ITUDE = {} & (-GP\_ATTITUDE[1,3] \cdot GP\_VELOCITY[1] + \\ & -GP\_ATTITUDE[2,3] \cdot GP\_VELOCITY[2] + \\ & -GP\_ATTITUDE[3,3] \cdot GP\_VELOCITY[3]) + \\ & K\_ALT \cdot (AR\_ALTITUDE - GP\_ALTITUDE) \end{aligned}$$

Notice that the formula for computing the rate of change of GP_ALTITUDE contains references to both GP_ATTITUDE and GP_VELOCITY. Because of these references the solution for computing the current values of these data elements must solve these three equations simultaneously -- a system of equations.

The solution to computing this system of equations proposed in Pluto is based on the fourth-order Runge-Kutta (RK) method. Operating on a single equation, the RK method computes four estimates for the incremental value and then uses a weighted average of the four estimates to compute the result. The solution employs the RK method to the three equations simultaneously by computing the first estimate for each equation first, then computing the second estimate for each equation second, and so on, finally performing the weighted averages. In this manner, the intermediately computed estimates are available to the "downstream" computations of other further estimates as necessary.

The typical application of the RK method involves computing the new value of a function given the current value of the function and a step size. The first estimate, k_1, of the incremental value is determined by multiplying the rate-of-change of the function at the current value by the step size. The second estimate, k_2, of the incremental value is determined by multiplying the rate-of-change of the function at the midpoint of the line connecting the known value and the estimated new value determined by k_1. The straight forward application of the RK method to GCS is to treat the value for the current frame as the "new value," the value at the previous frame as the "old value," and the step size as DELTA_T.

But, there is a problem implementing this straight forward approach. In order to determine the rate-of-change, that is the first derivative of the function, for a specific instance in time, it is necessary to know specific sensor measurements at that point in time. The equations for rate-of-change presented above depict the necessary sensor measurements. So, if DELTA_T is chosen as the step-size, it is not possible to compute the rate-of-change at the "midpoint" of the line connecting $GP\_VELOCITY_{t-1}$ and the first estimate of $GP\_VELOCITY_{t}$

Likewise in the case of the computation of GP_ALTITUDE. In order for the necessary sensor information to be available for computations involving the "midpoint," the "midpoint" must coincide with the execution of the sensor processing subframe. Thus, if a step-size of 2 * DELTA_T is chosen, the "midpoint" falls on a frame boundary, and the necessary sensor information is available. The Pluto design implements the RK method with 2 * DELTA_T as the step-size, the data element value computed two frames previously as the "old value," and the data element current value as the "new value."

P-Spec 2.2 GP, presented in B.4, contains a detailed description of the application of the modified RK method for computing the current values of GP_ATTITUDE, GP_VELOCITY, and GP_ALTITUDE.

The Control Law Processing Subframe provides the guidance and control subsystem with an interface to the vehicle's locomotive resources, namely the axial engines, the roll engines and the parachute. The vehicle's locomotive resources provide Pluto with the means of maneuvering the lander. Control Law Processing Subframe processing is decomposed into several distinct tasks.

The AECLP process is responsible for generating the appropriate axial engine commands. AECLP processes data originating from the Sensor Processing Subframe and Guidance Processing Subframe processing and computes the axial engine commands. DFD 3.2 represents the role of AECLP in the Control Law Processing Subframe and P-Spec 3.2 specifies the AECLP functional processing.

The development specifications present the following formula as a solution for determining a value for the data element TE_LIMIT, note that the following data elements are abbreviated GRAVITY as GRAV, GP_ATTITUDE(1,3,0) as ATT, VELOCITY_ERROR as VEL_ERROR , and OMEGA as Ω:

$$\frac{\frac{d}{dt}(TE\_LIMIT) + \Omega \cdot TE\_LIMIT}{GA} =$$

$$-GAX \cdot \left(\ddot{\chi}_v + GRAV \cdot ATT\right) + GVE \cdot VEL\_ERROR + GVEI \cdot TE\_INTEGRAL$$

rewriting the equation yields:

$$\frac{d}{dt}(TE\_LIMIT) + \Omega \cdot TE\_LIMIT =$$

$$GA \cdot (-GAX \cdot \left(\ddot{\chi}_v + GRAV \cdot ATT\right) + GVE \cdot VEL\_ERROR + GVEI \cdot TE\_INTEGAL)$$

Letting $Q =$

$$GA \cdot (-GAX \cdot \left(\ddot{\chi}_v + GRAV \cdot ATT\right) + GVE \cdot VEL\_ERROR + GVEI \cdot TE\_INTEGAL)$$

gives:

$$\frac{d}{dt}(TE\_LIMIT) + \Omega \cdot TE\_LIMIT = Q$$

which happens to be a first order linear equation. Multiplying each term by the integrating factor:

$$e^{\int \Omega dt} \cdot \frac{d}{dt}(TE\_LIMIT) + e^{\int \Omega dt} \cdot \Omega \cdot TE\_LIMIT = e^{\int \Omega dt} \cdot Q$$

simplify:

$$\frac{d}{dt}\left(TE\_LIMIT \cdot e^{\int \Omega dt}\right) = e^{\int \Omega dt} \cdot Q$$

integrating both terms:

$$TE\_LIMIT \cdot e^{\int \Omega dt} = \int \left(Q \cdot e^{\int \Omega dt} \right) dt$$

$$TE\_LIMIT = e^{-\int \Omega dt} \cdot Q \cdot \int e^{\int \Omega dt} dt$$

$$e^{\int \Omega dt} = e^{\Omega t}$$

$$let\ u = \Omega t\ and\ du = \Omega dt$$

$$TE\_LIMIT = e^{-\Omega t} \cdot Q \cdot \int e^{\Omega t} dt$$

$$\begin{aligned} \int e^{\Omega t} dt &= \frac{1}{\Omega} \cdot \int \left(e^{\Omega t} \cdot Q \right) dt \\ &= \frac{1}{\Omega} \cdot \int e^{u} du \\ &= \frac{1}{\Omega} \cdot e^{u} + C \\ &= \frac{1}{\Omega} \cdot e^{\Omega t} + C \end{aligned}$$

$$TE\_LIMIT = \frac{Q}{\Omega} + C \cdot Q \cdot e^{-\Omega t}$$

Solving for C based on the following initial conditions:

$$t = t_0 = 0$$

$TE\_LIMIT = TE\_LIMIT_0$ where TE_LIMIT_0 is the previously computed TE_LIMIT

$$C = \frac{\left(TE\_LIMIT_0 - \frac{Q}{\Omega} \right)}{Q}$$

The final result is:

$$TE\_LIMIT = \frac{Q}{\Omega} + \left(TE\_LIMIT_0 - \frac{Q}{\Omega} \right) \cdot e^{-\Omega t}$$

where $Q =$

$$GA \cdot \left(-GAX \cdot \left(\ddot{X}_v + GRAV \cdot ATT \right) + GVE \cdot VEL\_ERROR + GVEI \cdot TE\_INTEGRAL \right)$$

The CRCP process is responsible for determining whether or not to release the parachute. CRCP determines to release the parachute based on the current state of the parachute and the axial engine temperature. DFD 3.3 represents the role of CRCP in the Control Law Processing Subframe and P-Spec 3.2 specifies the CRCP functional processing.

The RECLP process is responsible for generating the appropriate roll engine commands. RECLP processes data originating from the Sensor Processing Subframe and Guidance Processing Subframe processing and computes the roll engine commands. DFD 3.4 represents the role of RECLP in the Control Law Processing Subframe and P-Spec 3.4 specifies the RECLP functional processing.

B.2.4 Process scheduling

The Pluto software design specification contains no explicit process scheduling needs.

B.2.5 Data Dictionary

The data dictionary contains formal definitions of all the data items presented in the data-flow and control-flow diagrams. Teamwork provides an integrated data dictionary for use with the SA/RT software development tools. A copy of Pluto's data dictionary as stored in Teamwork may be found in B.4.

B.2.6 Derived Requirements

According to DO-178B (ref. B.1) derived requirements are those requirements which are not directly traceable to higher level requirements. The GCS Software Development Specification goes to great length in presenting the software specifications for a GCS implementation. As such, it has not been necessary for the Pluto software design specification to create any derived requirements.

B.3 References

B.1 RTCA Special Committee 152. Software Considerations in Airborne Systems and Equipment Certification. Technical Report RTCA/DO-178B, Requirements and Technical Concepts for Aviation, December 1992.

B.2 George B. Finelli. Results of software error-data experiments. In AIAA/AHS/ASEE Aircraft Design, Systems and Operations Conference, Atlanta, GA, September 1988.

B.3 Withers, B. Edward; and Becher, Bernice: *Guidance and Control Software Development Specification*. version 2.3 with formal modifications through 2.3-3, 1994

B.4 *Teamwork/SA Teamwork/RT User's Guide*. Release 4.0, Cadre Technologies Inc., 1990

B.5 DeMarco, Tom: *Structured Analysis and System Specification*. Prentice-Hall Inc., 1979.

B.6 Hatley, Derek J.; and Imtiaz A. Pirbhai: *Strategies for Real-Time System Specification*. Dorset House Publishing Co., 1988

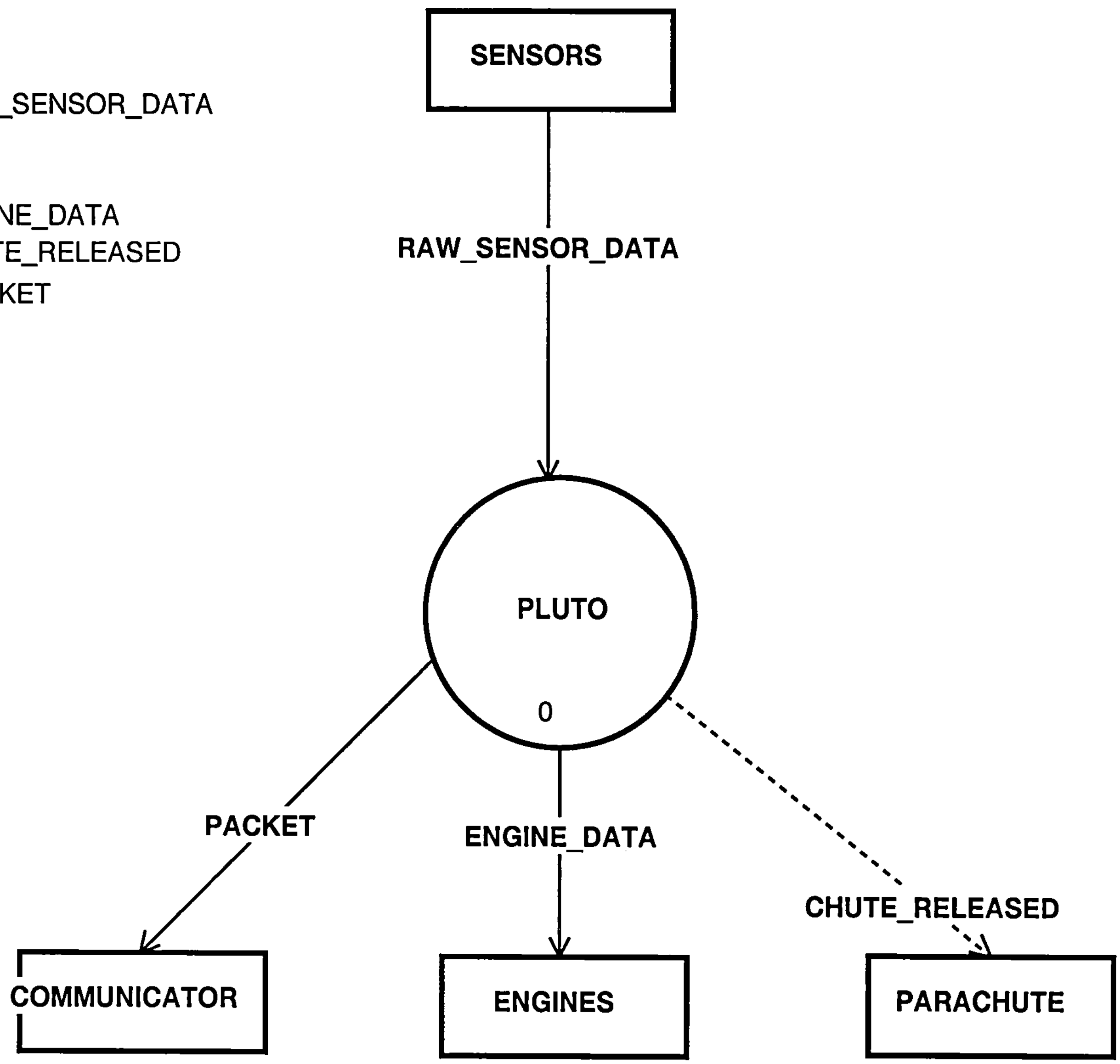
B.7 Page-Jones, Meilir: *The Practical Guide to Structured Systems Design*. Yourdon Press, 1980

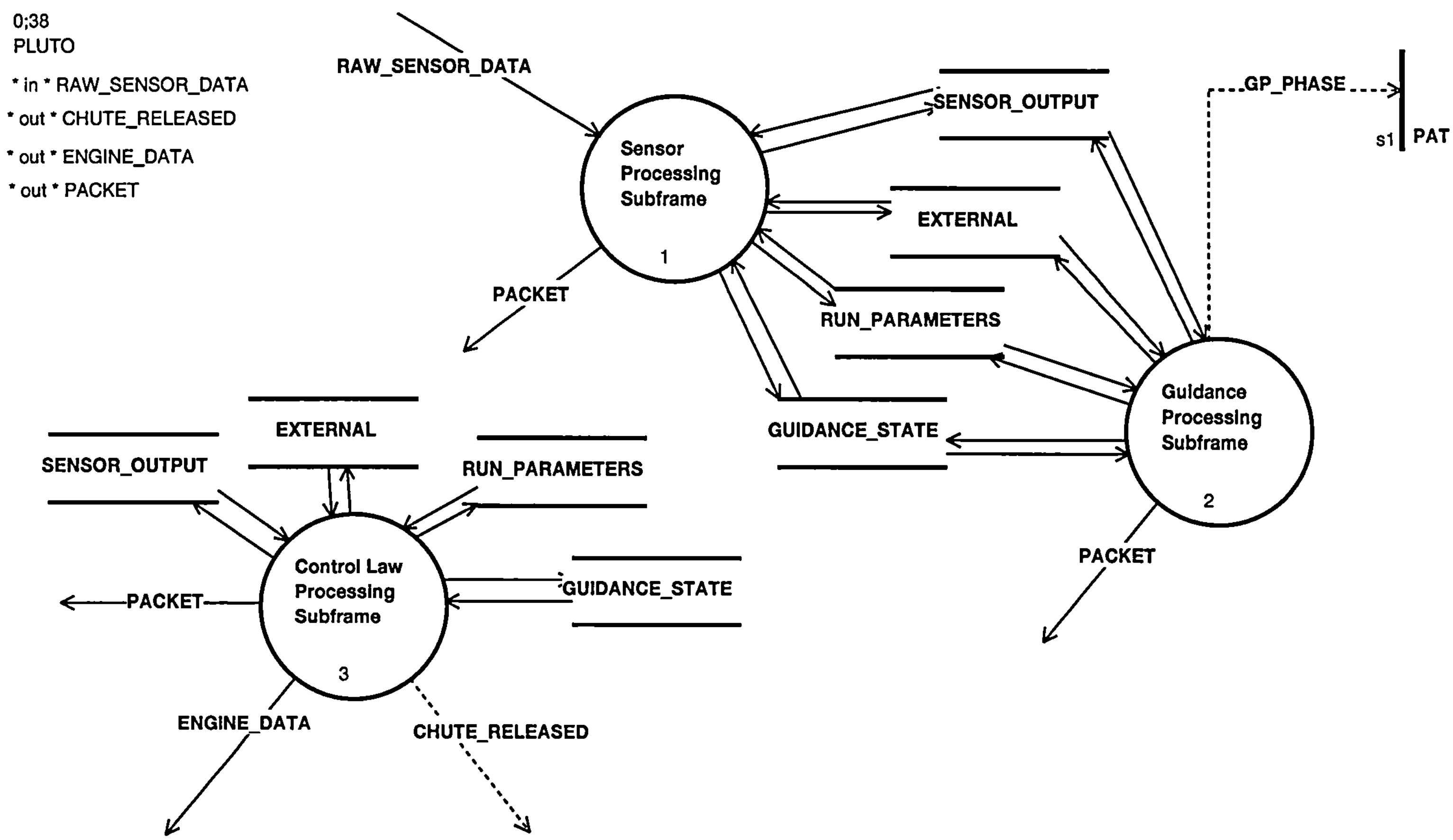
B.4 Teamwork Design

Context-Diagram;22
PLUTO

* in *
RAW_SENSOR_DATA

* out *
ENGINE_DATA
CHUTE_RELEASED
PACKET



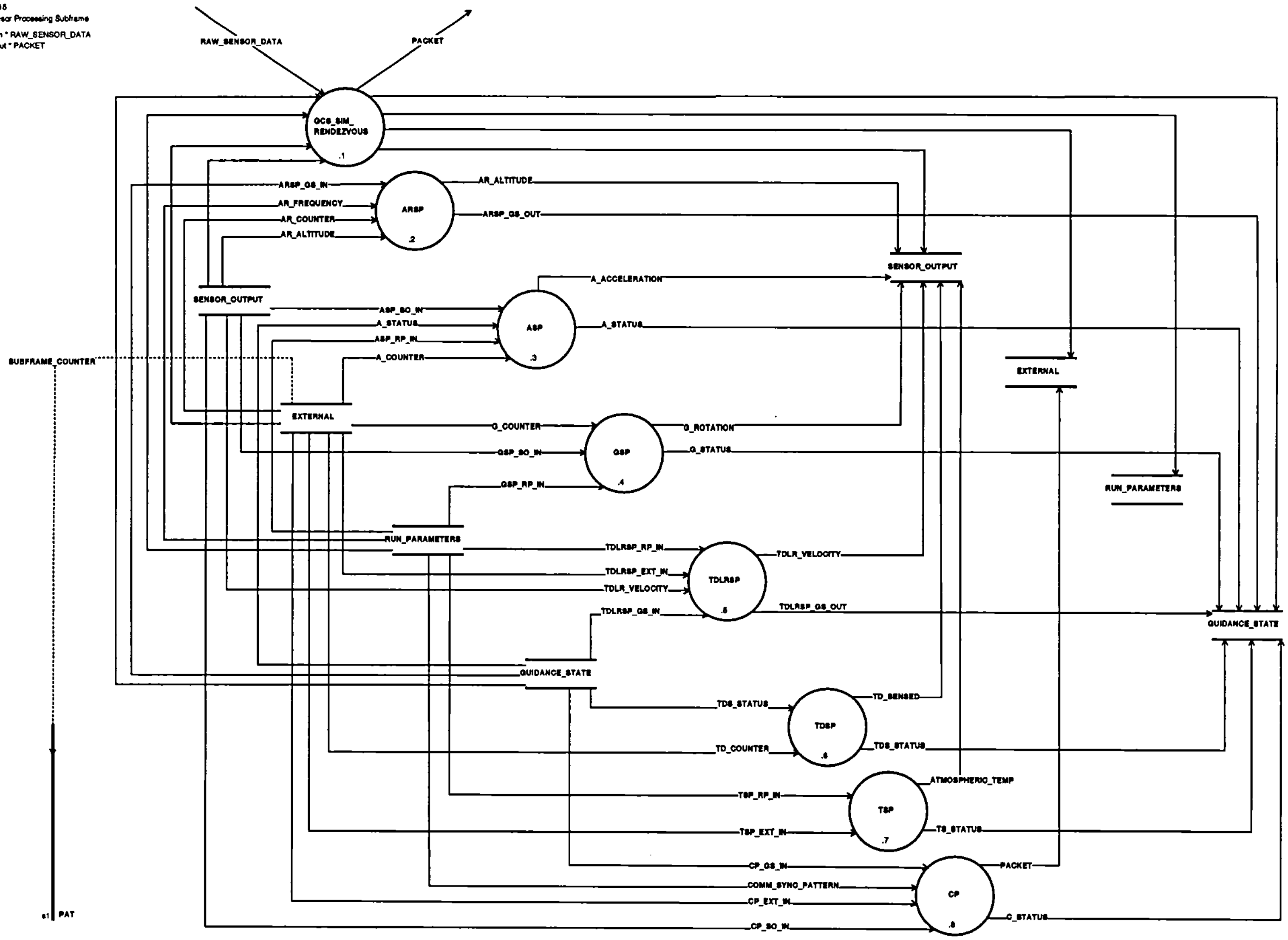

0;38
PLUTO
* in * RAW_SENSOR_DATA
* out * CHUTE_RELEASED
* out * ENGINE_DATA
* out * PACKET
RAW_SENSOR_DATA
Sensor Processing Subframe
1
PACKET
SENSOR_OUTPUT
EXTERNAL
RUN_PARAMETERS
GUIDANCE_STATE
GP_PHASE
s1
PAT
Guidance Processing Subframe
2
PACKET
EXTERNAL
SENSOR_OUTPUT
RUN_PARAMETERS
Control Law Processing Subframe
3
PACKET
GUIDANCE_STATE
ENGINE_DATA
CHUTE_RELEASED

0-s1;23
PLUTO PAT

GP_PHASE	"Sensor Processing Subframe"	"Guidance Processing Subframe"	"Control Law Processing Subframe"
"5"	0	0	0
Others	1	2	3

1;115
Sensor Processing Subframe

* in * RAW_SENSOR_DATA
* out * PACKET



1-s1;22
PAT - Sensor Processing Subframe

SUBFRAME_COUNTER	"GCS_SIM_RENDEZVOUS"	"ARSP"	"ASP"	"GSP"	"TDLRSP"	"TDSP"	"TSP"	"CP"
"1"	1	3	3	3	3	3	2	4

NAME:
1.1;5

TITLE:
GCS_SIM_RENDEZVOUS

INPUT/OUTPUT:
RAW_SENSOR_DATA: data_in
GUIDANCE_STATE : data_in
RUN_PARAMETERS : data_in
EXTERNAL : data_in
SENSOR_OUTPUT : data_in
SENSOR_OUTPUT : data_out
GUIDANCE_STATE : data_out
RUN_PARAMETERS : data_out
PACKET: data_out
EXTERNAL : data_out

BODY:

```
BEGIN P-Spec

GCS_SIM_RENDEZVOUS provides the interface to the vehicle.  This module is provided by
the systems group.

Bubbles 1.1, 2.1, 3.1 and the associated P-Specs represent a single process.

END P-Spec
```

ARSP

NAME:
1.2;31

TITLE:
ARSP

INPUT/OUTPUT:
AR_ALTITUDE : data_in

AR_COUNTER : data_in

AR_FREQUENCY : data_in

AR_STATUS : data_in

K_ALT : data_in

AR_ALTITUDE : data_out

AR_STATUS : data_out

K_ALT : data_out

BODY:
```
BEGIN P_SPEC
(***************************************************************************
 *  ARSP -- Altimeter Radar Sensor Processing
 *
 *  ARSP processing is responsible for:
 *  1) maintaining the history of the altitude and altimeter sensor data
 *     elements,
 *  2) determining the operational status of the altimeter radar sensor, and
 *  3) Reporting the current altitude.
 ***************************************************************************)

(***************************************************************************
 *  1) Maintain the history of the altitude and the sensor status by
 *  "rotating variables." Each of the three data elements AR_ALTITUDE,
 *  AR_STATUS, and K_ALT are defined as five element arrays.  The first
 *  element of each array, element zero, holds the most recently computed
 *  value.  The last element of each array, element four, holds the
 *  oldest maintained value.  In shifting the values stored in these
 *  data elements, a multi-frame history is maintained.
 ***************************************************************************)

   AR_ALTITUDE[4] := AR_ALTITUDE[3]
   AR_ALTITUDE[3] := AR_ALTITUDE[2]
   AR_ALTITUDE[2] := AR_ALTITUDE[1]
   AR_ALTITUDE[1] := AR_ALTITUDE[0]

   AR_STATUS[4] := AR_STATUS[3]
   AR_STATUS[3] := AR_STATUS[2]
   AR_STATUS[2] := AR_STATUS[1]
   AR_STATUS[1] := AR_STATUS[0]
```

```
  K_ALT[4] := K_ALT[3]
  K_ALT[3] := K_ALT[2]
  K_ALT[2] := K_ALT[1]
  K_ALT[1] := K_ALT[0]

(**************************************************************************
 *  2) The data element AR_STATUS represents the operational status
 *     of the altimeter radar sensor.  If an echo has been received,
 *     then the  sensor status is deemed "healthy" (value 0).  If an
 *     echo has not been received, the sensor status is reported as
 *     "failed" (value 1).  The GP process references the data element
 *     K_ALT to determine which method  was used to determine the
 *     current altitude. If either method A or B, as described below, is
 *     employed to compute the  current altitude, the value for K_ALT is
 *     reported as "1." If method C is used to compute the altitude,
 *     the value for K_ALT is  reported as "0."
 *
 *  3) There are three methods for determining the altitude.
 *
 *  A) If an echo has been received, then the altitude will be computed
 *     from the sensor measurement.
 *
 *  B) If an echo has not been received, and all four of the maintained
 *     altitude sensor history statuses are "heathly," then the value
 *     for the current altitude is estimated by fitting a third-order
 *     polynomial to the altitude history data values (see description
 *     below).
 *
 *  C) If an echo has not been received, and at least one of the
 *     maintained altitude sensor history statuses is "failed," then
 *     the value for the the current altitude is estimated by reporting
 *     the mostly recently reported value for the altitude.
 *************************************************************************)

(**************************************************************************
 * If an echo has been recieved, then the lower order fifteen bits of
 * AR_COUNTER contain the raw sensor measurement, and the upper bit of
 * AR_COUNTER will be clear (ie: 0).  When an echo has not been received,
 * the AR_COUNTER will contain 16 set bits (ie: 0xFFFF).
 *
 * It is known that this design will be implemented on VAX/VMS in FORTRAN.
 * The data type of AR_COUNTER is integer*2 and the valid value range
 * is specified as [-1, 32767].  VAX/VMS uses twos complement method for
 * representing negative values.  Thus, when an echo has not been recieved,
 * the AR_COUNTER will contain the value of -1.  Similarly, when an echo
 * has been received, AR_COUNTER will contain a non-negative value.
 *************************************************************************)

  if  (an echo has been recieved) then                                 {1}

(*** report the sensor status as healthy ***)

     AR_STATUS[0]      := 0    (* healthy      *)
     K_ALT[0]          := 1    (* method A     *)

(*** A) compute the altitude from the sensor measurement ***)
```

```
    AR_ALTITUDE := (AR_COUNTER * 3x10^8) / (2 * AR_FREQUENCY)

  else        (* no echo received *)                            {1}

(*** report the sensor status as failed ***)

    AR_STATUS[0]      := 1    (* failed *)

*** if at least one of the history sensor status is "failed" ***)

    if (AR_STATUS[1] = 1 OR AR_STATUS[2] = 1 OR                  {2}
        AR_STATUS[3] = 1 OR AR_STATUS[4] = 1) then

      K_ALT[0]        := 0    (* method C *)

(*** C) return previously computed value ***)

(*** range check the altitude ****)

      if (AR_ALTITUDE[1] < 0)                                   {3}
        display-error("%EXCEPTIONAL-CONDITION-GCS-LOWER_LIMIT_EXCEEDED",
                      "ARSP ARSP", FRAME_COUNTER,
                      "AR_ALTITUDE", AR_ALTITUDE[1])

      else if (AR_ALTITUDE[1] > 2000)                           {3}
        display-error("%EXCEPTIONAL-CONDITION-GCS-UPPER_LIMIT_EXCEEDED",
                      "ARSP ARSP", FRAME_COUNTER,
                      "AR_ALTITUDE", AR_ALTITUDE[1])
      end if                                                    {3}

      AR_ALTITUDE[0] := AR_ALTITUDE[1]       (* this should already exist *)

    else      (* all sensor status histories are "healthy" *)      {2}

(*** B) extrapolate the altitude ***)

      K_ALT[0] := 1  (* method B *)

(*** range check the altitude ****)

      if (AR_ALTITUDE[1] < 0)                                   {3}
        display-error("%EXCEPTIONAL-CONDITION-GCS-LOWER_LIMIT_EXCEEDED",
                      "ARSP ARSP", FRAME_COUNTER,
                      "AR_ALTITUDE", AR_ALTITUDE)

      else if (AR_ALTITUDE[1] > 2000)                           {3}
        display-error("%EXCEPTIONAL-CONDITION-GCS-UPPER_LIMIT_EXCEEDED",
                      "ARSP ARSP", FRAME_COUNTER,
                      "AR_ALTITUDE", AR_ALTITUDE)
      end if                                                    {3}

      if (AR_ALTITUDE[2] < 0)                                   {3}
        display-error("%EXCEPTIONAL-CONDITION-GCS-LOWER_LIMIT_EXCEEDED",
                      "ARSP ARSP", FRAME_COUNTER,
                      "AR_ALTITUDE", AR_ALTITUDE)
```

```
        else if (AR_ALTITUDE[2] > 2000)                              {3}
           display-error("%EXCEPTIONAL-CONDITION-GCS-UPPER_LIMIT_EXCEEDED",
                         "ARSP ARSP", FRAME_COUNTER,
                         "AR_ALTITUDE", AR_ALTITUDE)
        end if                                                       {3}

        if (AR_ALTITUDE[3] < 0)                                      {3}
           display-error("%EXCEPTIONAL-CONDITION-GCS-LOWER_LIMIT_EXCEEDED",
                         "ARSP ARSP", FRAME_COUNTER,
                         "AR_ALTITUDE", AR_ALTITUDE)

        else if (AR_ALTITUDE[3] > 2000)                              {3}
           display-error("%EXCEPTIONAL-CONDITION-GCS-UPPER_LIMIT_EXCEEDED",
                         "ARSP ARSP", FRAME_COUNTER,
                         "AR_ALTITUDE", AR_ALTITUDE)
        end if                                                       {3}

        if (AR_ALTITUDE[4] < 0)                                      {3}
           display-error("%EXCEPTIONAL-CONDITION-GCS-LOWER_LIMIT_EXCEEDED",
                         "ARSP ARSP", FRAME_COUNTER,
                         "AR_ALTITUDE", AR_ALTITUDE)

        else if (AR_ALTITUDE[4] > 2000)                              {3}
           display-error("%EXCEPTIONAL-CONDITION-GCS-UPPER_LIMIT_EXCEEDED",
                         "ARSP ARSP", FRAME_COUNTER,
                         "AR_ALTITUDE", AR_ALTITUDE)
        end if                                                       {3}

        AR_ALTITUDE[0] := 4*AR_ALTITUDE[1] - 6*AR_ALTITUDE[2] +
                         4*AR_ALTITUDE[3] - AR_ALTITUDE[4]

     end if                                                          {2}

  end if                                                             {1}

END P_SPEC
```

ASP

NAME:
1.3;36

TITLE:
ASP

INPUT/OUTPUT:
A_ACCELERATION : data_in

A_BIAS : data_in

A_COUNTER : data_in

A_GAIN_0 : data_in

A_SCALE : data_in

A_STATUS : data_in

ALPHA_MATRIX : data_in

ATMOSPHERIC_TEMP : data_in

G1 : data_in

G2 : data_in

A_ACCELERATION : data_out

A_STATUS : data_out

BODY:

```
BEGIN P_SPEC
(*****************************************************************************
 *  ASP -- Accelerometer Sensor Processing
 *
 *  ASP processing is responsible for:
 *  1) maintaining the history of the accelerations and accelerometer
 *     sensor statuses,
 *  2) determining the operational status of the accelerometer sensors, and
 *  3) Reporting the current vehicle accelerations along each of the
 *     vehicle's three axes.
 ****************************************************************************)

(*****************************************************************************
 *  1) Maintain the history of the vehicle accelerations and
 *  accelerometer sensor status by "rotating variables."
 *  Both of the data elements A_ACCELERATION and A_STATUS are defined as
 *  two diminsional arrays.  The first dimension of each array represents
 *  a vehicle axis: x-axis (1), y-axis (2), z-axis (3).  The second
 *  diminsion of each data element represents a history.  For
 *  A_ACCELERATION, the history is five deep and for A_STATUS the history
 *  is four deep.  The first element of each history, element zero, holds
 *  the most recently computed value.  The last element of each history,
 *  element four or three respectively, holds the oldest maintained value.
```

```
 *  In shifting the values stored in these data elements, a multi-frame
 *  history is maintained.
 ***************************************************************************)

   A_ACCELERATION[1, 4] := A_ACCELERATION[1, 3]
   A_ACCELERATION[1, 3] := A_ACCELERATION[1, 2]
   A_ACCELERATION[1, 2] := A_ACCELERATION[1, 1]
   A_ACCELERATION[1, 1] := A_ACCELERATION[1, 0]

   A_ACCELERATION[2, 4] := A_ACCELERATION[2, 3]
   A_ACCELERATION[2, 3] := A_ACCELERATION[2, 2]
   A_ACCELERATION[2, 2] := A_ACCELERATION[2, 1]
   A_ACCELERATION[2, 1] := A_ACCELERATION[2, 0]

   A_ACCELERATION[3, 4] := A_ACCELERATION[3, 3]
   A_ACCELERATION[3, 3] := A_ACCELERATION[3, 2]
   A_ACCELERATION[3, 2] := A_ACCELERATION[3, 1]
   A_ACCELERATION[3, 1] := A_ACCELERATION[3, 0]

   A_STATUS[1, 3] := A_STATUS[1, 2]
   A_STATUS[1, 2] := A_STATUS[1, 1]
   A_STATUS[1, 1] := A_STATUS[1, 0]

   A_STATUS[2, 3] := A_STATUS[2, 2]
   A_STATUS[2, 2] := A_STATUS[2, 1]
   A_STATUS[2, 1] := A_STATUS[2, 0]

   A_STATUS[3, 3] := A_STATUS[3, 2]
   A_STATUS[3, 2] := A_STATUS[3, 1]
   A_STATUS[3, 1] := A_STATUS[3, 0]

(***************************************************************************
 *  2) and 3), determine the operational status and the vehicle
 *  accelerations for each axis.
 ***************************************************************************)

(*** range check the atmospheric temperature ****)

   if (ATMOSPHERIC_TEMP < -200) then                                 {1}
      display-error("%EXCEPTIONAL-CONDITION-GCS-LOWER_LIMIT_EXCEEDED",
                    "ASP ASP", FRAME_COUNTER,
                    "ATMOSPHERIC_TEMP", ATMOSPHERIC_TEMP)

   else if  (ATMOSPHERIC_TEMP > 25) then                             {1}
      display-error("%EXCEPTIONAL-CONDITION-GCS-UPPER_LIMIT_EXCEEDED",
                    "ASP ASP", FRAME_COUNTER,
                    "ATMOSPHERIC_TEMP", ATMOSPHERIC_TEMP)
   end if                                                            {1}

(*** compute the preliminary value for the accelerations ***)

   Let accel_m be a 3x1 matrix

   compute each element (i := 1 to 3) of accel_m as follows:

      accel_m[i]  := A_BIAS[i] + a_gain * A_COUNTER[i]
```

```
    where
          a_gain := A_GAIN_0[i] + (G1 * ATMOSPHERIC_TEMP) +
                    (G2 * ATMOSPHERIC_TEMP^2)

  viewing the "current" elements of A_ACCELERATION as a 3x1 matrix,

       -                       -
       | A_ACCELERATION[1, 0] |
       | A_ACCELERATION[2, 0] |
       | A_ACCELERATION[3, 0] |
       -                       -

  the prelimenary values for A_ACCELERATION are computed from the
  matrix multication:

       A_ACCELERATION := ALPHA_MATRIX X accel_m

(*************************************************************************
 *  Determine whether or not the preliminary values for the
 *  accelerations are reasonable. The preliminary value for an
 *  acceleration is deemed reasonable: 1) if it differs from the mean
 *  of the previous three measurements by not more than A_SCALE
 *  standard deviations; 2) when any of the three accelerometer
 *  history statuses is "unhealthy" (value 1). If a preliminary
 *  acceleration value is found to be reasonable,
 *  then it is reported as the acceleration for it's axis.  If a
 *  preliminary value is not found to be reasonable, then the
 *  mean of the previous three measurements is reported as the
 *  acceleration for that axis.
 *
 *  The current value for the sensor status is determined directly
 *  from the reasonableness of the value of the preliminary
 *  accleration.  If the preliminary acceleration is reasonable, the
 *  sensor status is deemed "healthy " (value 0).  If the preliminary
 *  acceleration is not reasonable, the sensor status is deemed
 *  "unhealthy."
 *************************************************************************)

  do for each axis (i := 1 to 3)

     A_STATUS[i, 0] := 0;      (* set sensor status to "healthy" *)

     if (A_STATUS[i, 1] = 0) AND A_STATUS[i, 2] = 0 AND              {1}
         A_STATUS[i, 3] = 0) then

        if ((A_ACCELERATION[i,1] <> A_ACCELERATION[i,2]) OR
           (A_ACCELERATION[i,1] <> A_ACCLERATION[i,3])) then          {2}


(*** compute the mean of the previous three values ***)

(*** range check the acceleration values ****)

        if (A_ACCELERATION[i, 1] < -20) then                          {3}
           display-error("%EXCEPTIONAL-CONDITION-GCS-LOWER_LIMIT_EXCEEDED",
```

```
                       "ASP ASP", FRAME_COUNTER,
                       "A_ACCELERATION", A_ACCELERATION[i, 1])

      else if (A_ACCELERATION[i, 1] > 5) then                    {3}
         display-error("%EXCEPTIONAL-CONDITION-GCS-UPPER_LIMIT_EXCEEDED",
                       "ASP ASP", FRAME_COUNTER,
                       "A_ACCELERATION", A_ACCELERATION[i, 1])
      end if                                                     {3}

      if (A_ACCELERATION[i, 2] < -20) then                       {3}
         display-error("%EXCEPTIONAL-CONDITION-GCS-LOWER_LIMIT_EXCEEDED",
                       "ASP ASP", FRAME_COUNTER,
                       "A_ACCELERATION", A_ACCELERATION[i, 2])

      else if (A_ACCELERATION[i, 2] > 5) then                    {3}
         display-error("%EXCEPTIONAL-CONDITION-GCS-UPPER_LIMIT_EXCEEDED",
                       "ASP ASP", FRAME_COUNTER,
                       "A_ACCELERATION", A_ACCELERATION[i, 2])
      end if                                                     {3}

      if (A_ACCELERATION[i, 3] < -20) then                       {3}
         display-error("%EXCEPTIONAL-CONDITION-GCS-LOWER_LIMIT_EXCEEDED",
                       "ASP ASP", FRAME_COUNTER,
                       "A_ACCELERATION", A_ACCELERATION[i, 3])

      else if (A_ACCELERATION[i, 3] > 5) then                    {3}
         display-error("%EXCEPTIONAL-CONDITION-GCS-UPPER_LIMIT_EXCEEDED",
                       "ASP ASP", FRAME_COUNTER,
                       "A_ACCELERATION", A_ACCELERATION[i, 3])
      end if                                                     {3}

      mean := ((A_ACCELERATION[i, 1] + A_ACCELERATION[i, 2] +
               A_ACCELERATION[i, 3]) / 3

(*** compute the standard deviation ***)

          temp := ((A_ACCELERATION[i, 1] - mean)^2 +
                   (A_ACCELERATION[i, 2] - mean)^2 +
                   (A_ACCELERATION[i, 3] - mean)^2) / 3

          sd := SQRT(temp)

      if (ABS(mean - A_ACCELERATION[i, 0]) > A_SCALE * sd) then  {3}
         A_ACCELERATION[i, 0] := mean
         A_STATUS[i, 0] := 1 (* set sensor status to "unhealthy" *)

      end if                                                     {3}

    end if                                                       {2}

  end if                                                         {1}

 end do (* for each axis *)

END P_SPECA_STATUS[i, 0] := 0;  (* set sensor status to "healthy" *)
```

GSP

NAME:
1.4;16

TITLE:
GSP

INPUT/OUTPUT:
ATMOSPHERIC_TEMP : data_in

G3 : data_in

G4 : data_in

G_COUNTER : data_in

G_GAIN_0 : data_in

G_OFFSET : data_in

G_ROTATION : data_in

G_ROTATION : data_out

G_STATUS : data_out

BODY:

```
BEGIN P_SPEC

(***************************************************************************
 *  GSP -- Gyroscopy Sensor Processing
 *
 *  GSP processing is responsible for:
 *  1) maintaining the history of the vehicle rotation rates,
 *  2) determining the operational status of the gyroscope sensors, and
 *  3) Reporting the current vehicle rotation rates along each of the
 *     vehicle's three axes.
 ***************************************************************************)

(***************************************************************************
 *  1) Maintain the history of the vehicle rotation rates by "rotating
 *  variables." The data element G_ROTATION is defined as a  two
 *  diminsional array.  The first dimension represents a vehicle axis:
 *  x-axis (1), y-axis (2), and  z-axis (3).  The second diminsion
 *  represents a five deep history.  The first element of the history (0),
 *  holds the most recently computed value.  The last element of the
 *  history (4), holds the oldest maintained value.  In shifting the
 *  values stored in these data elements, a multi-frame history is
 *  maintained.
 ***************************************************************************)

   G_ROTATION[1, 4] := G_ROTATION[1, 3]
   G_ROTATION[1, 3] := G_ROTATION[1, 2]
   G_ROTATION[1, 2] := G_ROTATION[1, 1]
   G_ROTATION[1, 1] := G_ROTATION[1, 0]
```

```
   G_ROTATION[2, 4] := G_ROTATION[2, 3]
   G_ROTATION[2, 3] := G_ROTATION[2, 2]
   G_ROTATION[2, 2] := G_ROTATION[2, 1]
   G_ROTATION[2, 1] := G_ROTATION[2, 0]

   G_ROTATION[3, 4] := G_ROTATION[3, 3]
   G_ROTATION[3, 3] := G_ROTATION[3, 2]
   G_ROTATION[3, 2] := G_ROTATION[3, 1]
   G_ROTATION[3, 1] := G_ROTATION[3, 0]

(***************************************************************************
 *  2) determining the operational status of the gyroscope sensors.
 *  The operational status of the gyroscope sensors is always reported
 *  as "healthy" (value 0).
 ***************************************************************************)

   G_STATUS     := 0

(***************************************************************************
 *  3) Reporting the current vehicle rotation rates along each of the
 *  vehicle's three axes.
 ***************************************************************************)

(*** range check the atmospheric temperature ****)

    if  (ATMOSPHERIC_TEMP < -200) then
        display-error("%EXCEPTIONAL-CONDITION-GCS-LOWER_LIMIT_EXCEEDED",
                      "GSP GSP", FRAME_COUNTER,
                      "ATMOSPHERIC_TEMP", ATMOSPHERIC_TEMP)

    else if  (ATMOSPHERIC_TEMP > 25) then
        display-error("%EXCEPTIONAL-CONDITION-GCS-UPPER_LIMIT_EXCEEDED",
                      "GSP GSP", FRAME_COUNTER,
                      "ATMOSPHERIC_TEMP", ATMOSPHERIC_TEMP)
    end if

(***************************************************************************
 *  The raw sensor data stored in G_COUNTER represents the vehicle rate
 *  of rotation about a specific axis.  The sensor data is
 *  stored in a modified sign magnitude format.  The lower 14-bits
 *  represent the magnitude of the rotation and the most significant
 *  bit (bit 15) represents the sign.  Bit 14 is not used.  A
 *  positive value of G_COUNTER indicates a positive rotation about
 *  the vehicle axis consistent with a right handed coordinate system,
 *  while a negative value indicates a negative rotation consistent
 *  with a right handed coordinate system.
 ***************************************************************************)

   do for each axis (i := 1 to 3)

(*** convert the raw sensor ... ***)

(***************************************************************************
 *  Convert the raw sensor data from the modified sign magnitude
 *  format into an appropriate format for use by the target CPU, in
 *  this case two's complement.  Positive values are represented in
```

```
 *  the same fashion in sign magnitude and two's complement, however,
 *  negative sensor values must be massaged.
 *
 *  Transfer the magitude of the rotation from G_COUNTER to the local
 *  data element named counter by masking bits 14 and 15 from
 *  G_COUNTER. If G_COUNTER bit 15 is clear, the data element counter
 *  now contains the properly converted value. If G_COUNTER bit 15 is
 *  set, the value of data element counter must be negated.
 ***************************************************************************)

(***************************************************************************
 *  The symbol '&' represents a bitwise AND operation
 *  the notation '0xdddd' represents a hexidecimal value
 ***************************************************************************)

      counter   := G_COUNTER[i] & 0x3FFF

      if  ((G_COUNTER[i] & 0x8000) = 1)
         counter        := 0 - counter
      end if

(*** compute the vehicle rotation from the sensor data ***)

      G_ROTATION[i, 0] := G_OFFSET[i] + g_gain * counter

      where
          g_gain := G_GAIN_0[i] + (G3 * ATMOSPHERIC_TEMP) +
                         (G4 * ATMOSPHERIC_TEMP^2)

   end do      (* for each axis *)

END P_SPEC
```

NAME:
1.5;27

TITLE:
TDLRSP

INPUT/OUTPUT:
DELTA_T : data_in
FRAME_BEAM_UNLOCKED : data_in
FRAME_COUNTER : data_in
K_MATRIX : data_in
TDLR_ANGLES : data_in
TDLR_COUNTER : data_in
TDLR_GAIN : data_in
TDLR_LOCK_TIME : data_in
TDLR_OFFSET : data_in

TDLR_STATE : data_in

TDLR_VELOCITY : data_in
FRAME_BEAM_UNLOCKED : data_out

K_MATRIX : data_out
TDLR_STATE : data_out

TDLR_STATUS : data_out

TDLR_VELOCITY : data_out

BODY:

```
BEGIN P_SPEC

(**************************************************************************
 *  TDLRSP -- Touch Down Landing Radar Sensor Processing
 *
 *  TDLRSP processing is responsible for:
 *  1) Maintaining the history of the vehicle velocities and the
 *     velocity computation indicator,
 *  2) Determining the operational status of touch down landing radar
 *     sensor, and
 *  3) Reporting the current vehicle velocities along each of the
 *     vehicle's three axes, and
 *  4) Reporting the velocity computation indicators.
 *
 **************************************************************************)

(**************************************************************************
 *  1) Maintain the history of the vehicle velocities and the
 *  velocity computation indicator by "rotating variables."  The data
 *  element TDLR_VELOCITY is defined as a two dimensional array.  The
 *  first dimension represents a vehicle axis: x-axis (1), y-axis
 *  (2), and z-axis (3).  The second dimension represents a five deep
 *  history. The data element K_MATRIX is defined as a three
 *  dimensional array (1..3, 1..3, 0..4). The velocity computation
 *  indicators are arranged as a 3x3 matrix, represented by the first
```

```
* two dimensions of K_MATRIX.  The third dimension represents a
* five deep history.  The first element of the history, element
* zero, holds the most recently computed value.  The last element
* of the history, element four, holds the oldest maintained value.
* In shifting the values stored in these data elements, a
* multi-frame history is maintained.
************************************************************************)

   TDLR_VELOCITY[1, 4] := TDLR_VELOCITY[1, 3]
   TDLR_VELOCITY[1, 3] := TDLR_VELOCITY[1, 2]
   TDLR_VELOCITY[1, 2] := TDLR_VELOCITY[1, 1]
   TDLR_VELOCITY[1, 1] := TDLR_VELOCITY[1, 0]

   TDLR_VELOCITY[2, 4] := TDLR_VELOCITY[2, 3]
   TDLR_VELOCITY[2, 3] := TDLR_VELOCITY[2, 2]
   TDLR_VELOCITY[2, 2] := TDLR_VELOCITY[2, 1]
   TDLR_VELOCITY[2, 1] := TDLR_VELOCITY[2, 0]

   TDLR_VELOCITY[3, 4] := TDLR_VELOCITY[3, 3]
   TDLR_VELOCITY[3, 3] := TDLR_VELOCITY[3, 2]
   TDLR_VELOCITY[3, 2] := TDLR_VELOCITY[3, 1]
   TDLR_VELOCITY[3, 1] := TDLR_VELOCITY[3, 0]

   K_MATRIX[1, 1, 4]   := K_MATRIX[1, 1, 3]
   K_MATRIX[1, 2, 4]   := K_MATRIX[1, 2, 3]
   K_MATRIX[1, 3, 4]   := K_MATRIX[1, 3, 3]
   K_MATRIX[2, 1, 4]   := K_MATRIX[2, 1, 3]
   K_MATRIX[2, 2, 4]   := K_MATRIX[2, 2, 3]
   K_MATRIX[2, 3, 4]   := K_MATRIX[2, 3, 3]
   K_MATRIX[3, 1, 4]   := K_MATRIX[3, 1, 3]
   K_MATRIX[3, 2, 4]   := K_MATRIX[3, 2, 3]
   K_MATRIX[3, 3, 4]   := K_MATRIX[3, 3, 3]

   K_MATRIX[1, 1, 3]   := K_MATRIX[1, 1, 2]
   K_MATRIX[1, 2, 3]   := K_MATRIX[1, 2, 2]
   K_MATRIX[1, 3, 3]   := K_MATRIX[1, 3, 2]
   K_MATRIX[2, 1, 3]   := K_MATRIX[2, 1, 2]
   K_MATRIX[2, 2, 3]   := K_MATRIX[2, 2, 2]
   K_MATRIX[2, 3, 3]   := K_MATRIX[2, 3, 2]
   K_MATRIX[3, 1, 3]   := K_MATRIX[3, 1, 2]
   K_MATRIX[3, 2, 3]   := K_MATRIX[3, 2, 2]
   K_MATRIX[3, 3, 3]   := K_MATRIX[3, 3, 2]

   K_MATRIX[1, 1, 2]   := K_MATRIX[1, 1, 1]
   K_MATRIX[1, 2, 2]   := K_MATRIX[1, 2, 1]
   K_MATRIX[1, 3, 2]   := K_MATRIX[1, 3, 1]
   K_MATRIX[2, 1, 2]   := K_MATRIX[2, 1, 1]
   K_MATRIX[2, 2, 2]   := K_MATRIX[2, 2, 1]
   K_MATRIX[2, 3, 2]   := K_MATRIX[2, 3, 1]
   K_MATRIX[3, 1, 2]   := K_MATRIX[3, 1, 1]
   K_MATRIX[3, 2, 2]   := K_MATRIX[3, 2, 1]
   K_MATRIX[3, 3, 2]   := K_MATRIX[3, 3, 1]

   K_MATRIX[1, 1, 1]   := K_MATRIX[1, 1, 0]
   K_MATRIX[1, 2, 1]   := K_MATRIX[1, 2, 0]
   K_MATRIX[1, 3, 1]   := K_MATRIX[1, 3, 0]
```

```
  K_MATRIX[2, 1, 1]   := K_MATRIX[2, 1, 0]
  K_MATRIX[2, 2, 1]   := K_MATRIX[2, 2, 0]
  K_MATRIX[2, 3, 1]   := K_MATRIX[2, 3, 0]
  K_MATRIX[3, 1, 1]   := K_MATRIX[3, 1, 0]
  K_MATRIX[3, 2, 1]   := K_MATRIX[3, 2, 0]
  K_MATRIX[3, 3, 1]   := K_MATRIX[3, 3, 0]

(**************************************************************************
 *  2) Determine the operational status of touch down landing radar
 *  sensor.
 *
 *  The operational status of the TDLR sensor is always reported
 *  as "healthy" (value 0).
 **************************************************************************)

  TDLR_STATUS[1] := 0
  TDLR_STATUS[2] := 0
  TDLR_STATUS[3] := 0
  TDLR_STATUS[4] := 0

(**************************************************************************
 *  3) Reporting the current vehicle velocities along each of the
 *  vehicle's three axes and reporting the velocity computation
 *  indicators.
 **************************************************************************)

(**************************************************************************
 *  3A)  Determine the state of the four radar beams.
 *
 * The data element TDLR_STATE contains the state of the radar
 * beams.
 *
 * Valid radar beam states are "locked" (value 1) and "unlocked"
 * (value 0). The present state of a radar beam is determined from
 * the current value of the sensor data and the previous state of
 * the radar beam. A sensor measurement of zero indicates that the
 * radar beam echo was not received and the radar beam is considered
 * to be "unlocked."  A non-zero sensor measurement indicates that a
 * radar beam echo was received, but does not imply a radar beam
 * state of "locked."  Because, once a radar beam is declared
 * "unlocked," it is rendered unusable (remains "unlocked"
 * regardless of the sensor data value) for a specified period of
 * time.  This waiting period must be implemented in the software.
 *
 * A beam is deemed "locked" when 1) the current sensor value
 * contains a non-zero value and the beam's previous state was
 * "locked"; or 2) the current sensor value contains a non-zero
 * value and the beam's previous state was "unlocked" and the
 * elapsed time since the beam was determined "unlocked" is greater
 * than or equal to the sensor recovery period.
 *
 * The data element TDLR_LOCK_TIME specifies the unlocked sensor
 * recovery (waiting) period. The data element FRAME_BEAM_UNLOCKED
 * is updated with the value of the FRAME_COUNTER during the frame
 * in which  a radar beam state is first determined as "unlocked."
 * The data element DELTA_T specifies in seconds the duration of a
```

```
 * single frame.  Thus the elapsed time since a radar beam was
 * declared "unlocked" can be determined by subtracting the present
 * value of FRAME_COUNTER from the value of FRAME_BEAM_UNLOCKED and
 * multipling the result by the value of DELTA_T.
 ************************************************************************)

  do (for each radar beam i :=(1 to 4))                               {1}

    if (TDLR_COUNTER[i] = 0) then     (* beam is unlocked *)          {2}

      if (TDLR_STATE[i] = 1) then     (* beam was locked *)           {3}
        TDLR_STATE[i] := 0            (* set unlocked *)
        FRAME_BEAM_UNLOCKED[i] := FRAME_COUNTER

      else   (* the beam was unlocked *)                              {3}

        elapsed_time := DELTA_T * (FRAME_COUNTER - FRAME_BEAM_UNLOCKED[i])

        if (elapsed_time >= TDLR_LOCK_TIME) then                      {4}
          FRAME_BEAM_UNLOCKED[i] := FRAME_COUNTER
        end if                                                        {4}

      end if                                                          {3}

    else      (* the sensor measurement != 0 *)                       {2}

      if (TDLR_STATE[i] = 0) then     (* beam was unlocked *)         {3}

        elapsed_time := DELTA_T * (FRAME_COUNTER - FRAME_BEAM_UNLOCKED[i])

        if (elapsed_time >= TDLR_LOCK_TIME) then                      {4}
          TDLR_STATE[i] := 1          (* set locked *)

        end if                                                        {4}

      end if                                                          {3}

    end if                                                            {2}

  end do  (* for each beam i *)                                       {1}

(************************************************************************
 *  3B)  Determine the beam velocities.
 ************************************************************************)

  do (for each radar beam i := (1 to 4))
    b[i] := TDLR_OFFSET + TDLR_GAIN * TDLR_COUNTER[i]
  end do  (* for each beam *)

(************************************************************************
 *  3C)  Determine the "processed" beam velocities, and
 *   4)  Determine the velocity computation indicators.
 ************************************************************************)
(************************************************************************
 * Compute a "processed" beam velocity for each of the three axes as
 * specified by the following table:
```

```
*
*  Beams   |          PROCESSED  BEAM  VELOCITIES         |  K-MATRIX | Case
* in lock  |      pbvX       |      pbvY       |      pbvZ       | X   Y   Z | Number
* --------|---------------|---------------|---------------|----------------
*   none   |       0       |       0       |       0       | 0 | 0 | 0 |  0
*    1     |       0       |       0       |       0       | 0 | 0 | 0 |  1
*    2     |       0       |       0       |       0       | 0 | 0 | 0 |  2
*    3     |       0       |       0       |       0       | 0 | 0 | 0 |  4
*    4     |       0       |       0       |       0       | 0 | 0 | 0 |  8
* --------|---------------|---------------|---------------|----------------
*   1,2    |       0       | (b[1]-b[2])/2 |       0       | 0 | 1 | 0 |  3
*   1,3    | (b[1]+b[3])/2 |       0       |       0       | 1 | 0 | 0 |  5
*   1,4    |       0       |       0       | (b[1]-b[4])/2 | 0 | 0 | 1 |  9
*   2,3    |       0       |       0       | (b[2]-b[3])/2 | 0 | 0 | 1 |  6
*   2,4    | (b[2]+b[4])/2 |       0       |       0       | 1 | 0 | 0 | 10
*   3,4    |       0       | (b[4]-b[3])/2 |       0       | 0 | 1 | 0 | 12
* --------|---------------|---------------|---------------|----------------
*  1,2,3   | (b[1]+b[3])/2 | (b[1]-b[2])/2 | (b[2]-b[3])/2 | 1 | 1 | 1 |  7
*  1,2,4   | (b[2]+b[4])/2 | (b[1]-b[2])/2 | (b[1]-b[4])/2 | 1 | 1 | 1 | 11
*  1,3,4   | (b[1]+b[3])/2 | (b[4]-b[3])/2 | (b[1]-b[4])/2 | 1 | 1 | 1 | 13
*  2,3,4   | (b[2]+b[4])/2 | (b[4]-b[3])/2 | (b[2]-b[3])/2 | 1 | 1 | 1 | 14
* --------|---------------|---------------|---------------|----------------
* 1,2,3,4  |       a       |       b       |       c       | 1 | 1 | 1 | 15
*
* a) (b[1]+b[2]+b[3]+b[4])/4
* b) (b[1]-b[2]-b[3]+b[4])/4
* c) (b[1]+b[2]-b[3]-b[4])/4
*
* Each of the 16 possible cases has been assigned a case number to
* facilitate the description of the necessary processing.  The case
* number is found in the column labled "Case Number" in the table
* above.
*
* Determine the case number value for the current processing.
* Each of the four radar beams' state has been assigned a weight
* value:  beam 1: 1, beam 2: 2, beam 3: 4, beam 4: 8.  The "case
* number" is computed by  summing the radar beams multiplied by their
* their weight factors.
*************************************************************************)

  state_case := TDLR_STATE[1] + 2*TDLR_STATE[2] +
               4*TDLR_STATE[3] + 8*TDLR_STATE[4]

  case state_case of
     0, 1, 2, 4, 8:
          pbvX := 0
          pbvY := 0
          pbvZ := 0

          K_MATRIX[1, 1, 0] := 0
          K_MATRIX[2, 2, 0] := 0
          K_MATRIX[3, 3, 0] := 0
     end

     3:   pbvX := 0
          pbvY := (b1-b2)/2
```

```
          pbvZ := 0

          K_MATRIX[1, 1, 0] := 0
          K_MATRIX[2, 2, 0] := 1
          K_MATRIX[3, 3, 0] := 0
     end

     5:   pbvX := (b1+b3)/2
          pbvY := 0
          pbvZ := 0

          K_MATRIX[1, 1, 0] := 1
          K_MATRIX[2, 2, 0] := 0
          K_MATRIX[3, 3, 0] := 0
     end

     9:   pbvX := 0
          pbvY := 0
          pbvZ := (b1-b4)/2

          K_MATRIX[1, 1, 0] := 0
          K_MATRIX[2, 2, 0] := 0
          K_MATRIX[3, 3, 0] := 1
     end

     6:   pbvX := 0
          pbvY := 0
          pbvZ := (b2-b3)/2

          K_MATRIX[1, 1, 0] := 0
          K_MATRIX[2, 2, 0] := 0
          K_MATRIX[3, 3, 0] := 1
     end

     10:  pbvX := (b2+b4)/2
          pbvY := 0
          pbvZ := 0

          K_MATRIX[1, 1, 0] := 1
          K_MATRIX[2, 2, 0] := 0
          K_MATRIX[3, 3, 0] := 0
     end

     12:  pbvX := 0
          pbvY := (b4-b3)/2
          pbvZ := 0

          K_MATRIX[1, 1, 0] := 0
          K_MATRIX[2, 2, 0] := 1
          K_MATRIX[3, 3, 0] := 0
     end

     7:   pbvX := (b1+b3)/2
          pbvY := (b1-b2)/2
          pbvZ := (b2-b3)/2
```

```
          K_MATRIX[1, 1, 0] := 1
          K_MATRIX[2, 2, 0] := 1
          K_MATRIX[3, 3, 0] := 1
     end

     11:  pbvX := (b2+b4)/2
          pbvY := (b1-b2)/2
          pbvZ := (b1-b4)/2

          K_MATRIX[1, 1, 0] := 1
          K_MATRIX[2, 2, 0] := 1
          K_MATRIX[3, 3, 0] := 1
     end

     13:  pbvX := (b1+b3)/2
          pbvY := (b4-b3)/2
          pbvZ := (b1-b4)/2

          K_MATRIX[1, 1, 0] := 1
          K_MATRIX[2, 2, 0] := 1
          K_MATRIX[3, 3, 0] := 1
     end

     14:  pbvX := (b2+b4)/2
          pbvY := (b4-b3)/2
          pbvZ := (b2-b3)/2

          K_MATRIX[1, 1, 0] := 1
          K_MATRIX[2, 2, 0] := 1
          K_MATRIX[3, 3, 0] := 1
     end

     15:  pbvX := (b1+b2+b3+b4)/4
          pbvY := (b1-b2-b3+b4)/4
          pbvZ := (b1+b2-b3-b4)/4

          K_MATRIX[1, 1, 0] := 1
          K_MATRIX[2, 2, 0] := 1
          K_MATRIX[3, 3, 0] := 1
     end

(**************************************************************************
 *  3D)  Convert "processed" beam velocities into body velocites.
 **************************************************************************)

   TDLR_VELOCITY[1] := pbvX / cos(TDLR_ANGLES[1])
   TDLR_VELOCITY[2] := pbvY / cos(TDLR_ANGLES[2])
   TDLR_VELOCITY[3] := pbvZ / cos(TDLR_ANGLES[3])

 (*** where cos represents the cosine function. ***)

END P_SPEC
```

TDSP

NAME:
1.6;18

TITLE:
TDSP

INPUT/OUTPUT:
TD_COUNTER : data_in
TDS_STATUS : data_in
TD_SENSED :data_out

TDS_STATUS : data_out

BODY:

```
BEGIN P_SPEC

(**************************************************************************
 *  TDSP -- Touch Down Sensor Processing
 *
 *  TDSP processing is responsible for:
 *  1) Determining the operational status of the touch down sensor, and
 *  2) determining if touch down has been sensed.
 **************************************************************************)

(**************************************************************************
 *  1) Determining the operational status of the touch down sensor.
 *  and 2) determining if touch down has been sensed.
 *
 *  The data element TD_COUNTER represents the sensor's measurement.
 *  There are only two valid sensor measurements: A) all bits set
 *  which indicates touch down is sensed, and B) all bits clear which
 *  indicates touch down is not sensed.  If a valid sensor value
 *  exists, then the operation status of the touch down sensor is
 *  reported as "healthy" (value 0). Any other value of  TD_COUNTER
 *  indicates a faulty sensor in which case the touch down sensor
 *  status is reported as "failed" (value 1).
 *
 *  Note, once the touch down sensor has been determined to be
 *  faulty, it is considered to be failed for the duration of the
 *  mission -- no processing occurs once the sensor has failed.
 *
 *  The notation '0xdddd' represents a hexadecimal value
 **************************************************************************)


   if (TDS_STATUS = 0) then     (* healthy sensor *)

      if (TD_COUNTER = 0) then
         TD_SENSED := 0         (* TD not sensed *)

      else if (TD_COUNTER = 0xFFFF) then
         TD_SENSED := 1         (* TD sensed *)

      else                      (* faulty sensor *)
         TD_SENSED  := 0
         TDS_STATUS := 1        (* failed sensor *)
```

```
      end if

   end if

END P_SPEC
```

TSP

NAME:
1.7;21

TITLE:
TSP

INPUT/OUTPUT:
M1 : data_in

M2 : data_in

M3 : data_in

M4 : data_in

SS_TEMP : data_in

T1 : data_in

T2 : data_in

T3 : data_in

T4 : data_in

THERMO_TEMP : data_in

ATMOSPHERIC_TEMP : data_out

TS_STATUS : data_out

BODY:

```
BEGIN P_SPEC

(************************************************************************
 *  TSP  -- Temperature Sensor Processing
 *
 *  TSP is responsible for:
 *  1) Ascertaining the operational status of the temperature sensors, and
 *  2) Determining the current atmospheric temperature based on the
 *     measurements provided by two on-board temperature sensors.
 *
 * Notes:
 *  o  The constants associated with the solid state temperature sensor
 *     and the thermocouple pair would normally be calculated once and
 *     re-used in subsequent calls to this routine.  However, the GCS
 *     experiment methodology requires it to be calculated each call.
 ************************************************************************)

(************************************************************************
 * 1) Determine the operational status of the temperature sensors
 ************************************************************************)

(***  The status of both sensors is always reported as HEALTHY ***)
```

```
   TS_STATUS[1] := 0
   TS_STATUS[2] := 0

(************************************************************************
 * 2A) Compute the temperature based on the solid state sensor
 ************************************************************************)

   solid-state-temp :=
      ((T2 - T1)/(M2 - M1)) * SS_TEMP + T1 - ((T2 - T1)/(M2 - M1)) * M1

Implementation note, if M1 := M2 a divide by zero exception must be handled.

(************************************************************************
 * 2B) Determine if the temperature is within the valid range of the
 *    TC sensor;
 ************************************************************************)

   lower-parabolic-function :=
       -(x - (M3 + (((T4 - T3)/(M4 - M3))/2)))^2 + (T3 + (((T4 - T3)/(M4 - M3))/2)^2)

(************************************************************************
 * Once the function describing the parabola has been determined, the
 * temperature representing the lower limit of the parabolic region can
 * be determined.  The lower limit of the lower parabolic region is
 * specified as 15% of the difference of the two calibration
 * measurements less than the lower calibration point.
 ************************************************************************)

   lower-parabolic-temp-limit := lower-parabolic-function(M3 - 0.15*(M4 - M3))

(*** define the upper parabolic function ***)

   upper-parabolic-function :=
       (x - (M4 - (((T4 - T3)/(M4 - M3))/2)))^2 + (T4 - (((T4 - T3)/(M4 - M3))/2)^2)

(************************************************************************
 * Once the function describing the parabola has been determined, the
 * temperature representing the upper limit of the parabolic region can
 * be determined.  The upper limit of the upper parabolic region is
 * specified as 15% of the difference of the two calibration
 * measurements greater than the upper calibration point.
 ************************************************************************)

   upper-parabolic-temp-limit := upper-parabolic-function(M4 + 0.15*(M4 - M3))

(************************************************************************
 * Now determine sensor temperature measurement to report
 ************************************************************************)

   if  (solid-state-temp < lower-parabolic-temp-limit) OR              {1}
       (solid-state-temp > upper-parabolic-temp-limit) then

(*** the atmospheric temp is beyond the valid range of the TC sensor
     so return the solid-state-temp                                ***)

         ATMOSPHERIC_TEMP := solid-state-temp
```

```
    else                                                                  {1}

(************************************************************************
 * 2C) Compute the temperature based on the TC sensor
 ************************************************************************)

        if  (THERMO_TEMP < M3) then                                       {2}

(*** the atmospheric temp resides within the TC lower parabolic region ***)

             ATMOSPHERIC_TEMP :=  lower-parbolic-function(THERMO_TEMP)

        else if (THERMO_TEMP > M4) then                                   {2}

(*** the atmospheric temp resides within the TC upper parabolic region ***)

             ATMOSPHERIC_TEMP :=  upper-parabolic-function(THERMO_TEMP)

        else                                                              {2}

(*** The temperature resides within the TC sensor linear region ***)

(*** compute the temperature from the TC linear region ***)

            ATMOSPHERIC_TEMP :=
        ((T4 - T3)/(M4 - M3)) * THERMO_TEMP + T3 - ((T4 - T3)/(M4 - M3)) * M3

        end if                                                            {2}

    end if                                                                {1}

END P_SPEC
```

CP

NAME:
1.8;51

TITLE:
CP

INPUT/OUTPUT:
AE_CMD : data_in

AE_STATUS : data_in

AE_TEMP : data_in
AR_ALTITUDE : data_in

AR_STATUS: data_in

ATMOSPHERIC_TEMP : data_in

A_ACCELERATION : data_in

A_STATUS : data_in

CHUTE_RELEASED : data_in

COMM_SYNC_PATTERN : data_in

CONTOUR_CROSSED : data_in

C_STATUS :data_in

FRAME_COUNTER : data_in

GP_ALTITUDE : data_in

GP_ATTITUDE : data_in

GP_PHASE : data_in

GP_ROTATION : data_in

GP_VELOCITY : data_in

G_ROTATION : data_in

G_STATUS : data_in

K_ALT : data_in

K_MATRIX : data_in

PE_INTEGRAL : data_in

RE_CMD : data_in

RE_STATUS : data_in

SUBFRAME_COUNTER : data_in

TDLR_STATE : data_in

TDLR_STATUS : data_in

TDLR_VELOCITY : data_in

TDS_STATUS : data_in

TD_SENSED : data_in

TE_INTEGRAL : data_in

TS_STATUS : data_in

VELOCITY_ERROR : data_in

YE_INTEGRAL : data_in

C_STATUS : data_out

PACKET :data_out

BODY:

```
BEGIN P_SPEC

Note, bubbles 1.8, 2.3, 3.5 and the associated P-Specs represent a single
process, CP described below.

(*************************************************************************
 *  CP -- Communications Processing
 *
 *  CP processing is responsible for:
 *  1) determining the current operational status of the communicator, and
 *  2) constructing a telemetry data packet.
 *
 * CP processing is responsible for constructing a data packet
 * suitable for transmission via the on-board communications
 * equipment. The data element PACKET contains 512 bytes of storage
 * in which to construct the data packet.  Conceptually, the data
 * packet is merely the organization of particular data into the
 * storage defined by PACKET.
 *
 * A data packet is organized into five fields arranged in the
 * following sequence: a syncronization pattern, a sequence number,
 * a data mask, a data field, and a checksum. Constructing a data
 * packet consists of updating the five fields with the appropriate
 * data.
 *
 * CP has the capability to construct three specific types of
 * data packets, one each for reporting the completion of each
 * subframe. The distinguishing element of each packet type is
 * the contents of the data field and indirectly the value of the
 * data mask.  The data field is a composite
```

```
* field consisting of the values of specific data elements which
* were potentially altered during the processing of the specified
* subframe.
*
* The contents of the data field of a specific data packet is
* indicated by the data packet's data mask. The data mask is a
* 32-bit field.  Each of the 32 data elements which may be reported
* in the data packet has an associated bit in the data mask.  A set
* bit in the data mask indicates that the associated data element
* is stored in the data field portion of the data packet.
*
* At the completion of sensor processing subframe, the data field
* contains the data elements listed below in the order listed.
* Note the list of data elements below contains the derivation
* of the data mask value and derivation of the data field length,
* in bytes, which is used in contructing the packet. The
* synchronization pattern, sequence number, and data mask
* consume 7 bytes.
*
* data element       data       field length
*     name          bit mask     in bytes
*
* AR_ALTITUDE      0x10000000      8
* AR_STATUS        0x08000000      1
* ATMOSPHERIC_TEMP 0x04000000      8      122 bytes in this field
* A_ACCELERATION   0x02000000     24    +   7 bytes in preceeding fields
* A_STATUS         0x01000000      3      ---
* C_STATUS         0x00200000      1      129 bytes not including
* G_ROTATION       0x00008000     24          the checksum
* G_STATUS         0x00004000      1
* K_ALT            0x00002000      4
* K_MATRIX         0x00001000     12
* TDLR_STATE       0x00000100      4
* TDLR_STATUS      0x00000080      4
* TDLR_VELOCITY    0x00000040     24
* TDS_STATUS       0x00000020      1
* TD_SENSED        0x00000010      1
* TS_STATUS        0x00000004      2
*                               122 bytes
*
*                  0x1F20F1F4 data mask value for subframe #1
*
* At the completion of guidance processing subframe, the data field
* contains the following data elements:
*
* data element       data       field length
*     name          bit mask     in bytes
*
* CONTOUR_CROSSED  0x00400000      1
* C_STATUS         0x00200000      1
* GP_ALTITIUDE     0x00100000      8      166 bytes in this field
* GP_ATTITUDE      0x00080000     72    +   7 bytes in preceeding fields
* GP_PHASE         0x00040000      4      ---
* GP_ROTATION      0x00020000     48      173 bytes not including
* GP_VELOCITY      0x00010000     24          the checksum
* VELOCITY_ERROR   0x00000002      8
```

```
 *                              166 bytes
 *
 *                 0x007F0002 data mask value for subframe #2
 *
 * At the completion of control law processing subframe, the data field
 * contains the following data elements:
 *
 * data element       data       field length
 *     name          bit mask     in bytes
 *
 * AE_CMD            0x80000000     6
 * AE_STATUS         0x40000000     1
 * AE_TEMP           0x20000000     2       38 bytes in this field
 * CHUTE_RELEASED    0x00800000     1     +  7 bytes in preceeding fields
 * C_STATUS          0x00200000     1      ---
 * PE_INTEGRAL       0x00000800     8       45 bytes not including
 * RE_CMD            0x00000400     2          the checksum
 * RE_STATUS         0x00000200     1
 * TE_INTEGRAL       0x00000008     8
 * YE_INTEGRAL       0x00000001     8
 *                               38 bytes
 *
 *                 0xE0A00E09 data mask value for subframe #3
 *
 * It is not obvious why the checksum field is defined in three
 * places in the data structures presented below.  Notice that the
 * data field is variable in length.  The organization of the data
 * packet demands that the checksum field immediately follow the
 * last byte of the data field. In order to satisfy this
 * requirement, storage for the checksum field is defined as the
 * last element in each of the data field specifications.
 *
 * A function for computing the CRC-16 for a data packet is defined
 * below.  The function takes two arguments, the address of the byte
 * stream to process and integer specifing the length of the byte
 * stream.  The function returns an integer value which is the
 * CRC-16 of the specified byte stream.
 ***********************************************************************)

(*** the Sensor Processing subframe data field and checksum ***)

type sp_data_t Record
   AR_ALTITUDE      : quadword
   AR_STATUS        : byte
   ATMOSPHERIC_TEMP : quadword
   A_ACCELERATION   : array[1..3] of quadword
   A_STATUS         : array[1..3] of byte
   C_STATUS         : byte
   G_ROTATION       : array[1..3] of quadword
   G_STATUS         : byte
   K_ALT            : longword
   K_MATRIX         : array[1..3] of longword
   TDLR_STATE       : array[1..4] of byte
   TDLR_STATUS      : array[1..4] of byte
   TDLR_VELOCITY    : array[1..3] of quadword
   TDS_STATUS       : byte
```

```
   TD_SENSED         : byte
   TS_STATUS         : array[1..2] of byte
   CHECKSUM          : word
end {record}

(*** the Guidance Processing subframe data field and checksum ***)

type gp_data_t record
   CONTOUR_CROSSED   : byte
   C_STATUS          : byte
   GP_ALTITIUDE      : quadword
   GP_ATTITUDE       : array[1..9] of quadword
   GP_PHASE          : longword
   GP_ROTATION       : array[1..6] of quadword
   GP_VELOCITY       : array[1..3] of quadword
   VELOCITY_ERROR    : quadword
   CHECKSUM          : word
end {record}

(*** the Control Law Processing subframe data field and checksum ***)

type clp_data_t record
   AE_CMD            : array [1..3] of word
   AE_STATUS         : byte
   AE_TEMP           : word
   CHUTE_RELEASED    : byte
   C_STATUS          : byte
   PE_INTEGRAL       : quadword
   RE_CMD            : word
   RE_STATUS         : byte
   TE_INTEGRAL       : quadword
   YE_INTEGRAL       : quadword
   CHECKSUM          : word
end {record}

(*** the data packet structure ***)

type subframe_t = (sp_data_t, gp_data_t, clp_data_t)

type data_packet_t = record
   SYNC_PATTERN     : word
   SEQ_NUMBER       : byte
   DATA_MASK        : longword
   case subframe_t of
      sp            : sp_data_t
    | gp            : gp_data_t
    | clp           : clp_data_t
   end
end {record}

(************************************************************************
 * Overlay the data element PACKET with the data structure data_packet_t
 * and refer to it as "data_packet" for local processing.
 ************************************************************************)

(*** declare a variable of type pointer to data_packet_t ***)
```

```
  var data_packet:  @data_packet_t

  data_packet@ := PACKET       (* new variable "points to" PACKET *)

(*************************************************************************
 *  1) Determine the current operational status of the communicator.
 *
 *  The operational status of the communicator is always reported
 *  as "healthy" (value 0).
 *************************************************************************)

  C_STATUS := 0

(************************************************************************
 *  2) Construct a telemetry data packet.
 ************************************************************************)


(************************************************************************
 *  2A) Get synchronization pattern.
 *
 * The leading field is the synchronization pattern.  This 16-bit
 * pattern allows the recieving communications gear to recognize
 * the beginning of the data packet.  The bit pattern is stored in
 * the data element COMM_SYNC_PATTERN.
 ************************************************************************)

  data_packet.SYNC_PATTERN := COMM_SYNC_PATTERN

(************************************************************************
 *  2B) Determine the sequence number.
 *
 * The sequence number is an unsigned 8-bit value.  Conceptually,
 * the sequence number performs as an 8-bit counter.  The initial
 * packet is assigned a value of zero and the counter incremented
 * for each packet thereafter. However, an implementation does not
 * have access to it's own inter-subframe static storage, so the
 * "counter" is implemented as a function of the current frame
 * number and subframe number.
 ************************************************************************)

  data_packet.SEQ_NUMBER := (3*(FRAME_COUNTER-1)+(SUBFRAME_COUNTER-1)) MOD 256

  where MOD represents modulo division operation

(************************************************************************
 *  2C) Prepare the data mask,
 *  2D) Prepare the data, and
 *  2E) Compute the checksum.
 ************************************************************************)

  if  (SUBFRAME_COUNTER == 1) then            (* sp   *)
     data_packet.DATA_MASK := 0x1F20F1F4

     data_packet.DATA.SP.AR_ALTITUDE      := AR_ALTITUDE[0]
```

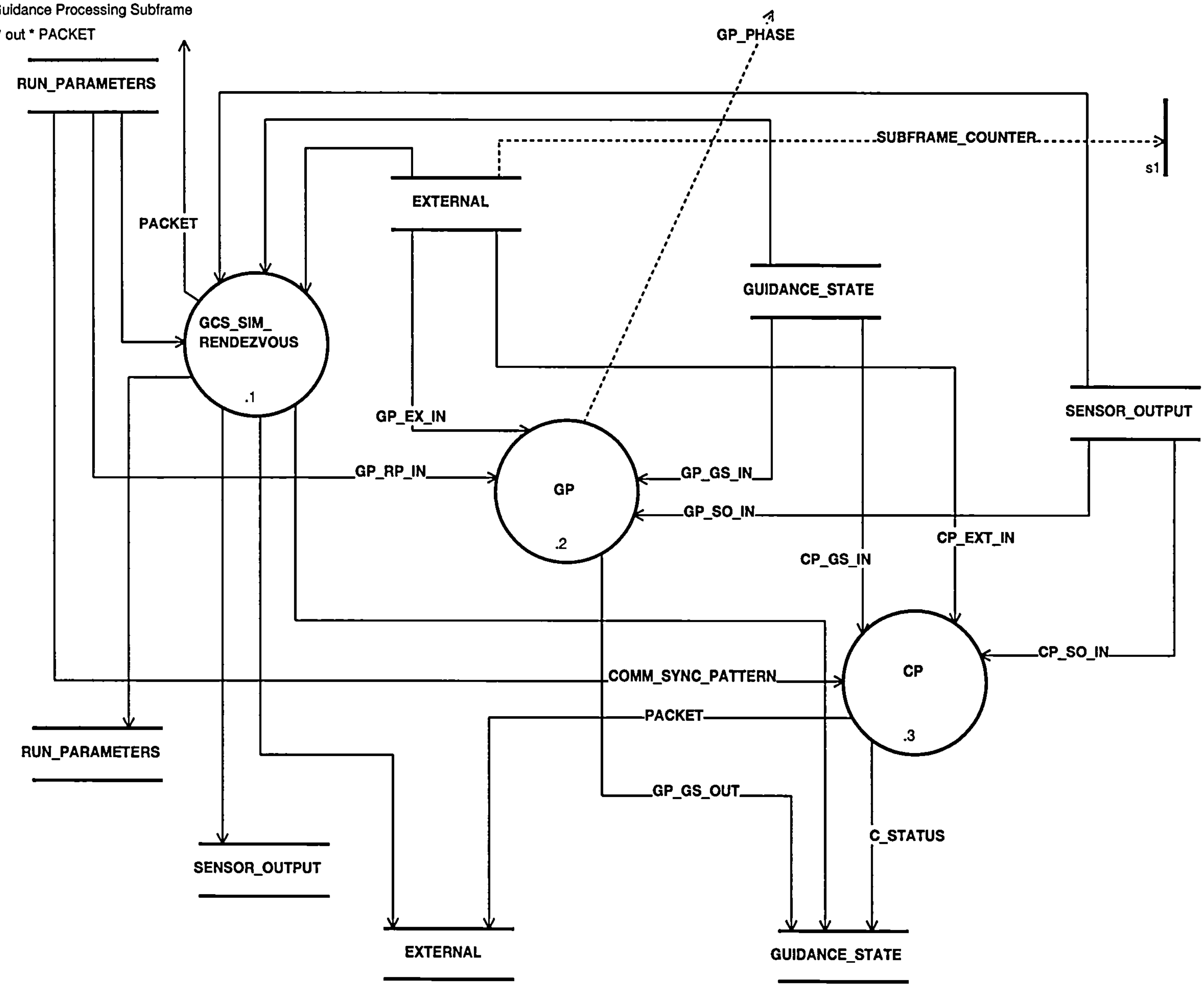
```
      data_packet.DATA.SP.AR_STATUS         := AR_STATUS[0]
      data_packet.DATA.SP.ATMOSPHERIC_TEMP  := ATMOSPHERIC_TEMP
      data_packet.DATA.SP.A_ACCELERATION[1]:= A_ACCELERATION[1,0]
      data_packet.DATA.SP.A_ACCELERATION[2]:= A_ACCELERATION[2,0]
      data_packet.DATA.SP.A_ACCELERATION[3]:= A_ACCELERATION[3,0]
      data_packet.DATA.SP.A_STATUS[1]       := A_STATUS[1,0]
      data_packet.DATA.SP.A_STATUS[2]       := A_STATUS[2,0]
      data_packet.DATA.SP.A_STATUS[3]       := A_STATUS[3,0]
      data_packet.DATA.SP.C_STATUS          := C_STATUS
      data_packet.DATA.SP.G_ROTATION[1]     := G_ROTATION[1,0]
      data_packet.DATA.SP.G_ROTATION[2]     := G_ROTATION[2,0]
      data_packet.DATA.SP.G_ROTATION[3]     := G_ROTATION[3,0]
      data_packet.DATA.SP.G_STATUS          := G_STATUS
      data_packet.DATA.SP.K_ALT             := K_ALT[0]
      data_packet.DATA.SP.K_MATRIX[1]       := K_MATRIX[1,1,0]
      data_packet.DATA.SP.K_MATRIX[2]       := K_MATRIX[2,2,0]
      data_packet.DATA.SP.K_MATRIX[3]       := K_MATRIX[3,3,0]
      data_packet.DATA.SP.TDLR_STATE[1]     := TDLR_STATE[1]
      data_packet.DATA.SP.TDLR_STATE[2]     := TDLR_STATE[2]
      data_packet.DATA.SP.TDLR_STATE[3]     := TDLR_STATE[3]
      data_packet.DATA.SP.TDLR_STATE[4]     := TDLR_STATE[4]
      data_packet.DATA.SP.TDLR_STATUS[1]    := TDLR_STATUS[1]
      data_packet.DATA.SP.TDLR_STATUS[2]    := TDLR_STATUS[2]
      data_packet.DATA.SP.TDLR_STATUS[3]    := TDLR_STATUS[3]
      data_packet.DATA.SP.TDLR_STATUS[4]    := TDLR_STATUS[4]
      data_packet.DATA.SP.TDLR_VELOCITY[1]  := TDLR_VELOCITY[1,0]
      data_packet.DATA.SP.TDLR_VELOCITY[2]  := TDLR_VELOCITY[2,0]
      data_packet.DATA.SP.TDLR_VELOCITY[3]  := TDLR_VELOCITY[3,0]
      data_packet.DATA.SP.TDS_STATUS        := TDS_STATUS
      data_packet.DATA.SP.TD_SENSED         := TD_SENSED
      data_packet.DATA.SP.TS_STATUS[1]      := TS_STATUS[1]
      data_packet.DATA.SP.TS_STATUS[2]      := TS_STATUS[2]
      data_packet.DATA.SP.CHECKSUM          := CRC16(data_packet.DATA.SP, 129)

   else if (SUBFRAME_COUNTER == 2) then        (* gp   *)
      data_packet.DATA_MASK := 0x007F0002

      data_packet.DATA.GP.CONTOUR_CROSSED   := CONTOUR_CROSSED
      data_packet.DATA.GP.C_STATUS          := C_STATUS
      data_packet.DATA.GP.GP_ALTITUDE       := GP_ALTITUDE[0]

(*** first element of array changes most rapidly ***)

      data_packet.DATA.GP.GP_ATTITUDE[1]    := GP_ATTITUDE[1, 1, 0]
      data_packet.DATA.GP.GP_ATTITUDE[2]    := GP_ATTITUDE[2, 1, 0]
      data_packet.DATA.GP.GP_ATTITUDE[3]    := GP_ATTITUDE[3, 1, 0]
      data_packet.DATA.GP.GP_ATTITUDE[4]    := GP_ATTITUDE[1, 2, 0]
      data_packet.DATA.GP.GP_ATTITUDE[5]    := GP_ATTITUDE[2, 2, 0]
      data_packet.DATA.GP.GP_ATTITUDE[6]    := GP_ATTITUDE[3, 2, 0]
      data_packet.DATA.GP.GP_ATTITUDE[7]    := GP_ATTITUDE[1, 3, 0]
      data_packet.DATA.GP.GP_ATTITUDE[8]    := GP_ATTITUDE[2, 3, 0]
      data_packet.DATA.GP.GP_ATTITUDE[9]    := GP_ATTITUDE[3, 3, 0]

      data_packet.DATA.GP.GP_PHASE          := GP_PHASE

      data_packet.DATA.GP.GP_ROTATION[1]    := GP_ROTATION[2, 1]
```

```
    data_packet.DATA.GP.GP_ROTATION[2]   := GP_ROTATION[3, 1]
    data_packet.DATA.GP.GP_ROTATION[3]   := GP_ROTATION[1, 2]
    data_packet.DATA.GP.GP_ROTATION[4]   := GP_ROTATION[3, 2]
    data_packet.DATA.GP.GP_ROTATION[5]   := GP_ROTATION[1, 3]
    data_packet.DATA.GP.GP_ROTATION[6]   := GP_ROTATION[2, 3]

    data_packet.DATA.GP.GP_VELOCITY[1]   :=  GP_VELOCITY[1, 0]
    data_packet.DATA.GP.GP_VELOCITY[2]   :=  GP_VELOCITY[2, 0]
    data_packet.DATA.GP.GP_VELOCITY[3]   :=  GP_VELOCITY[3, 0]

    data_packet.DATA.GP.VELOCITY_ERROR   := VELOCITY_ERROR
    data_packet.DATA.GP.checksum         := CRC16(data_packet.DATA.GP, 173)

  else                                        (* clp  *)
    data_packet.DATA_MASK := 0xE0A00E09

    data_packet.DATA.CLP.AE_CMD[1]       := AE_CMD[1]
    data_packet.DATA.CLP.AE_CMD[2]       := AE_CMD[2]
    data_packet.DATA.CLP.AE_CMD[3]       := AE_CMD[3]
    data_packet.DATA.CLP.AE_STATUS       := AE_STATUS
    data_packet.DATA.CLP.AE_TEMP         := AE_TEMP
    data_packet.DATA.CLP.CHUTE_RELEASED  := CHUTE_RELEASED
    data_packet.DATA.CLP.C_STATUS        := C_STATUS
    data_packet.DATA.CLP.PE_INTEGRAL     := PE_INTEGRAL
    data_packet.DATA.CLP.RE_CMD          := RE_CMD
    data_packet.DATA.CLP.RE_STATUS       := RE_STATUS
    data_packet.DATA.CLP.TE_INTEGRAL     := TE_INTEGRAL
    data_packet.DATA.CLP.YE_INTEGRAL     := YE_INTEGRAL
    data_packet.DATA.CLP.CHECKSUM        := CRC16(data_packet.DATA.CLP, 45)

  end if

  return

(************************************************************************
 *  Title:  CRC16
 *  Description:
 *      Compute the Cyclic Redundancy Code of the specified buffer using
 *      CRC-16 as the generator polynomial.
 *  Arguments:
 *      byte pointer buffer - address of first byte of message.
 *      longword     length - count of bytes in message.
 *
 *  Returns:
 *      word     crc    - the CRC-16 of the specified message.
 ************************************************************************)

(************************************************************************
 * The following CRC generation algorithm is described by Perez in
 * IEEE Micro, Volume 3, Number 3 (june 1983).
 *
 * The algorithm computes the CRC of the specified messages by
 * operating on a byte at a time.  Beginning with an initial value
 * for the CRC, each byte of the message in sequence is applied to
 * the computation of the new CRC value.
 *
```

```
 * In the bitwise approach to generating a CRC, Each bit within the
 * message is operated on individually.  Note, that after performing the
 * necessary bitwise operations on 8 successive bits, there are only
 * 256 possible distinct results.  This algorithm takes advantage of
 * this fact and operates on a single byte (8-bits) at time.
 *
 * For a given 17-bit generator polynomial a 256 (16-bits) entry
 * table is constructed for storing the polynomial "signature."
 * The contents of this table is used for processing the message.
 * An algorithm for constructing the table is presented in the reference
 * cited above.
 ***********************************************************************)

(*** The "signature" table for the CRC-16 generator polynomial 0x1A001 ***)

word crc16_table[0..255] :=
        0x0000, 0xc0c1, 0xc181, 0x0140, 0xc301, 0x03c0, 0x0280, 0xc241,
        0xc601, 0x06c0, 0x0780, 0xc741, 0x0500, 0xc5c1, 0xc481, 0x0440,
        0xcc01, 0x0cc0, 0x0d80, 0xcd41, 0x0f00, 0xcfc1, 0xce81, 0x0e40,
        0x0a00, 0xcac1, 0xcb81, 0x0b40, 0xc901, 0x09c0, 0x0880, 0xc841,
        0xd801, 0x18c0, 0x1980, 0xd941, 0x1b00, 0xdbc1, 0xda81, 0x1a40,
        0x1e00, 0xdec1, 0xdf81, 0x1f40, 0xdd01, 0x1dc0, 0x1c80, 0xdc41,
        0x1400, 0xd4c1, 0xd581, 0x1540, 0xd701, 0x17c0, 0x1680, 0xd641,
        0xd201, 0x12c0, 0x1380, 0xd341, 0x1100, 0xd1c1, 0xd081, 0x1040,
        0xf001, 0x30c0, 0x3180, 0xf141, 0x3300, 0xf3c1, 0xf281, 0x3240,
        0x3600, 0xf6c1, 0xf781, 0x3740, 0xf501, 0x35c0, 0x3480, 0xf441,
        0x3c00, 0xfcc1, 0xfd81, 0x3d40, 0xff01, 0x3fc0, 0x3e80, 0xfe41,
        0xfa01, 0x3ac0, 0x3b80, 0xfb41, 0x3900, 0xf9c1, 0xf881, 0x3840,
        0x2800, 0xe8c1, 0xe981, 0x2940, 0xeb01, 0x2bc0, 0x2a80, 0xea41,
        0xee01, 0x2ec0, 0x2f80, 0xef41, 0x2d00, 0xedc1, 0xec81, 0x2c40,
        0xe401, 0x24c0, 0x2580, 0xe541, 0x2700, 0xe7c1, 0xe681, 0x2640,
        0x2200, 0xe2c1, 0xe381, 0x2340, 0xe101, 0x21c0, 0x2080, 0xe041,
        0xa001, 0x60c0, 0x6180, 0xa141, 0x6300, 0xa3c1, 0xa281, 0x6240,
        0x6600, 0xa6c1, 0xa781, 0x6740, 0xa501, 0x65c0, 0x6480, 0xa441,
        0x6c00, 0xacc1, 0xad81, 0x6d40, 0xaf01, 0x6fc0, 0x6e80, 0xae41,
        0xaa01, 0x6ac0, 0x6b80, 0xab41, 0x6900, 0xa9c1, 0xa881, 0x6840,
        0x7800, 0xb8c1, 0xb981, 0x7940, 0xbb01, 0x7bc0, 0x7a80, 0xba41,
        0xbe01, 0x7ec0, 0x7f80, 0xbf41, 0x7d00, 0xbdc1, 0xbc81, 0x7c40,
        0xb401, 0x74c0, 0x7580, 0xb541, 0x7700, 0xb7c1, 0xb681, 0x7640,
        0x7200, 0xb2c1, 0xb381, 0x7340, 0xb101, 0x71c0, 0x7080, 0xb041,
        0x5000, 0x90c1, 0x9181, 0x5140, 0x9301, 0x53c0, 0x5280, 0x9241,
        0x9601, 0x56c0, 0x5780, 0x9741, 0x5500, 0x95c1, 0x9481, 0x5440,
        0x9c01, 0x5cc0, 0x5d80, 0x9d41, 0x5f00, 0x9fc1, 0x9e81, 0x5e40,
        0x5a00, 0x9ac1, 0x9b81, 0x5b40, 0x9901, 0x59c0, 0x5880, 0x9841,
        0x8801, 0x48c0, 0x4980, 0x8941, 0x4b00, 0x8bc1, 0x8a81, 0x4a40,
        0x4e00, 0x8ec1, 0x8f81, 0x4f40, 0x8d01, 0x4dc0, 0x4c80, 0x8c41,
        0x4400, 0x84c1, 0x8581, 0x4540, 0x8701, 0x47c0, 0x4680, 0x8641,
        0x8201, 0x42c0, 0x4380, 0x8341, 0x4100, 0x81c1, 0x8081, 0x4040

(*** crc is a 16-bit unsigned integer value ***)

   crc := 0                           (* initial crc value *)

(*** process every byte in the message ***)

   do next_byte := 1 to bytecount
```

```
    index := crc XOR next_byte (* bitwise exclusive OR the lower 8 bits of crc *)
    crc   := crc >> 8                 (* bitwise right shift crc 8 times *)
    crc   := crc XOR crc16_table[index](* bitwise exclusive OR all 16 bits of crc *)
  end do

  return crc

END P_SPEC
```

2;10
Guidance Processing Subframe
* out * PACKET



2-s1;3
PAT - Guidance Processing Subframe

SUBFRAME_COUNTER	"GCS_SIM_RENDEZVOUS"	"GP"	"CP"
"2"	1	2	3

NAME:
2.1;4

TITLE:
GCS_SIM_RENDEZVOUS

INPUT/OUTPUT:
SENSOR_OUTPUT : data_in
GUIDANCE_STATE : data_in
EXTERNAL : data_in
RUN_PARAMETERS : data_in
RUN_PARAMETERS : data_out
SENSOR_OUTPUT : data_out
EXTERNAL : data_out
PACKET: data_out
GUIDANCE_STATE : data_out

BODY:

```
BEGIN P-Spec

GCS_SIM_RENDEZVOUS provides the interface to the vehicle.  This module is provided by
the systems group.

Bubbles 1.1, 2.1, 3.1 and the associated P-Specs represent a single process.

END P-Spec
```

GP

NAME:
2.2;47

TITLE:
GP

INPUT/OUTPUT:
A_ACCELERATION : data_in

AE_SWITCH : data_in

AE_TEMP : data_in

AR_ALTITUDE : data_in

CHUTE_RELEASED : data_in

CL : data_in

CONTOUR_ALTITUDE : data_in

CONTOUR_CROSSED : data_in

CONTOUR_VELOCITY : data_in

DELTA_T : data_in

DROP_HEIGHT : data_in

DROP_SPEED : data_in

ENGINES_ON_ALTITUDE : data_in

FRAME_COUNTER : data_in

GP_ALTITUDE : data_in

GP_ATTITUDE : data_in

GP_PHASE : data_in

GP_VELOCITY : data_in

GRAVITY : data_in

G_ROTATION : data_in

K_ALT : data_in

K_MATRIX : data_in

MAX_NORMAL_VELOCITY : data_in

RE_SWITCH : data_in

TD_SENSED : data_in

TDLR_VELOCITY : data_in

TDS_STATUS : data_in

AE_SWITCH : data_out

CL : data_out

CONTOUR_CROSSED : data_out

FRAME_ENGINES_IGNITED : data_out

GP_ALTITUDE : data_out

GP_ATTITUDE : data_out

GP_PHASE : data_out

GP_ROTATION : data_out

GP_VELOCITY : data_out

RE_SWITCH : data_out

TE_INTEGRAL : data_out

VELOCITY_ERROR : data_out

BODY:

```
BEGIN P_SPEC

(************************************************************************
 *  GP -- Guidance Processing
 *
 *  GP is responsible for:
 *  1) Maintaining the history of the vehicle's altitude, velocities,
 *     and attitude,
 *  2) Computing the current vehicle altitude, velocities and attitude,
 *  3) Determining if the engines should be switched on or off,
 *  4) Computing the current velocity error,
 *  5) Determining if the predetermined velocity-altitude contour has
 *     been crossed,
 *  6) Determining the current guidance phase, and
 *  7) Determining the appropriate axial engine control law parameters.
 ************************************************************************)

(************************************************************************
 *  1) Maintain the history of the vehicle altitude, velocities,
 *     and attitude by "rotating variables."
 *
 *  The data element GP_ALTITUDE is defined as a single dimensional
 *  array. The data element GP_VELOCITY is defined as a two
 *  dimensional array. The first dimension of GP_VELOCITY represents
 *  a vehicle axis: x-axis (1), y-axis (2), z-axis (3). The data
```

```
* element GP_ATTITUDE is defined as a three dimensional array
* (1..3, 1..3, 0..4). The vehicle attitudes are arranged as a 3x3
* matrix, represented by the first two dimensions of GP_ATTITUDE.
* The first dimension of GP_ALTITUDE, the second dimension of
* GP_VELOCITY and the third diminsion of GP_ATTITUDE represent a
* history.  Each  history is five deep. The first element of each
* history, element zero, holds the most recently computed value.
* The last element of each history, element four holds the oldest
* maintained value.  In shifting the values stored in these data
* elements, a multi-frame history is maintained.
**********************************************************************)

  GP_ALTITUDE[4] := GP_ALTITUDE[3]
  GP_ALTITUDE[3] := GP_ALTITUDE[2]
  GP_ALTITUDE[2] := GP_ALTITUDE[1]
  GP_ALTITUDE[1] := GP_ALTITUDE[0]

  GP_VELOCITY[1, 4] := GP_VELOCITY[1, 3]
  GP_VELOCITY[1, 3] := GP_VELOCITY[1, 2]
  GP_VELOCITY[1, 2] := GP_VELOCITY[1, 1]
  GP_VELOCITY[1, 1] := GP_VELOCITY[1, 0]

  GP_VELOCITY[2, 4] := GP_VELOCITY[2, 3]
  GP_VELOCITY[2, 3] := GP_VELOCITY[2, 2]
  GP_VELOCITY[2, 2] := GP_VELOCITY[2, 1]
  GP_VELOCITY[2, 1] := GP_VELOCITY[2, 0]

  GP_VELOCITY[3, 4] := GP_VELOCITY[3, 3]
  GP_VELOCITY[3, 3] := GP_VELOCITY[3, 2]
  GP_VELOCITY[3, 2] := GP_VELOCITY[3, 1]
  GP_VELOCITY[3, 1] := GP_VELOCITY[3, 0]

  GP_ATTITUDE[1, 1, 4] := GP_ATTITUDE[1, 1, 3]
  GP_ATTITUDE[1, 2, 4] := GP_ATTITUDE[1, 2, 3]
  GP_ATTITUDE[1, 3, 4] := GP_ATTITUDE[1, 3, 3]
  GP_ATTITUDE[2, 1, 4] := GP_ATTITUDE[2, 1, 3]
  GP_ATTITUDE[2, 2, 4] := GP_ATTITUDE[2, 2, 3]
  GP_ATTITUDE[2, 3, 4] := GP_ATTITUDE[2, 3, 3]
  GP_ATTITUDE[3, 1, 4] := GP_ATTITUDE[3, 1, 3]
  GP_ATTITUDE[3, 2, 4] := GP_ATTITUDE[3, 2, 3]
  GP_ATTITUDE[3, 3, 4] := GP_ATTITUDE[3, 3, 3]

  GP_ATTITUDE[1, 1, 3] := GP_ATTITUDE[1, 1, 2]
  GP_ATTITUDE[1, 2, 3] := GP_ATTITUDE[1, 2, 2]
  GP_ATTITUDE[1, 3, 3] := GP_ATTITUDE[1, 3, 2]
  GP_ATTITUDE[2, 1, 3] := GP_ATTITUDE[2, 1, 2]
  GP_ATTITUDE[2, 2, 3] := GP_ATTITUDE[2, 2, 2]
  GP_ATTITUDE[2, 3, 3] := GP_ATTITUDE[2, 3, 2]
  GP_ATTITUDE[3, 1, 3] := GP_ATTITUDE[3, 1, 2]
  GP_ATTITUDE[3, 2, 3] := GP_ATTITUDE[3, 2, 2]
  GP_ATTITUDE[3, 3, 3] := GP_ATTITUDE[3, 3, 2]

  GP_ATTITUDE[1, 1, 2] := GP_ATTITUDE[1, 1, 1]
  GP_ATTITUDE[1, 2, 2] := GP_ATTITUDE[1, 2, 1]
  GP_ATTITUDE[1, 3, 2] := GP_ATTITUDE[1, 3, 1]
  GP_ATTITUDE[2, 1, 2] := GP_ATTITUDE[2, 1, 1]
```

```
  GP_ATTITUDE[2, 2, 2] := GP_ATTITUDE[2, 2, 1]
  GP_ATTITUDE[2, 3, 2] := GP_ATTITUDE[2, 3, 1]
  GP_ATTITUDE[3, 1, 2] := GP_ATTITUDE[3, 1, 1]
  GP_ATTITUDE[3, 2, 2] := GP_ATTITUDE[3, 2, 1]
  GP_ATTITUDE[3, 3, 2] := GP_ATTITUDE[3, 3, 1]

  GP_ATTITUDE[1, 1, 1] := GP_ATTITUDE[1, 1, 0]
  GP_ATTITUDE[1, 2, 1] := GP_ATTITUDE[1, 2, 0]
  GP_ATTITUDE[1, 3, 1] := GP_ATTITUDE[1, 3, 0]
  GP_ATTITUDE[2, 1, 1] := GP_ATTITUDE[2, 1, 0]
  GP_ATTITUDE[2, 2, 1] := GP_ATTITUDE[2, 2, 0]
  GP_ATTITUDE[2, 3, 1] := GP_ATTITUDE[2, 3, 0]
  GP_ATTITUDE[3, 1, 1] := GP_ATTITUDE[3, 1, 0]
  GP_ATTITUDE[3, 2, 1] := GP_ATTITUDE[3, 2, 0]
  GP_ATTITUDE[3, 3, 1] := GP_ATTITUDE[3, 3, 0]

(*************************************************************************
 *  2) Compute the current vehicle altitude, velocities and attitude.
 *
 *  For a more detailed description of this processing, see the section
 *  2.3 of the design introduction.
 *************************************************************************)

(*** simultanously compute the current vehicle attitude, velocities
     and altitude                                                    ***)

(*** range check the following data elements ***)

(*************************************************************************
 *  In all other instances of range checking, the actual range checking
 *  has been written out.  Here, however, writing out all of the range
 *  checking is prohibitively long.  So, it should be sufficient to say,
 *  range check the following data elements:
 *
 *    element                  lower    upper
 *************************************************************************)

  GP_ATTITUDE[1, 1, 2]          -1        1
  GP_ATTITUDE[1, 2, 2]          -1        1
  GP_ATTITUDE[1, 3, 2]          -1        1
  GP_ATTITUDE[2, 1, 2]          -1        1
  GP_ATTITUDE[2, 2, 2]          -1        1
  GP_ATTITUDE[2, 3, 2]          -1        1
  GP_ATTITUDE[3, 1, 2]          -1        1
  GP_ATTITUDE[3, 2, 2]          -1        1
  GP_ATTITUDE[3, 3, 2]          -1        1

  GP_VELOCITY[1, 2]           -100      100
  GP_VELOCITY[2, 2]           -100      100
  GP_VELOCITY[3, 2]           -100      100

  GP_ALTITUDE[2]                 0     2000

(*** sensor data ***)

  G_ROTATION[1, 0]              -1        1
```

```
  G_ROTATION[2, 0]              -1        1
  G_ROTATION[3, 0]              -1        1
  G_ROTATION[1, 1]              -1        1
  G_ROTATION[2, 1]              -1        1
  G_ROTATION[3, 1]              -1        1
  G_ROTATION[1, 2]              -1        1
  G_ROTATION[2, 2]              -1        1
  G_ROTATION[3, 2]              -1        1

  A_ACCELERATION[1, 0]         -20        5
  A_ACCELERATION[2, 0]         -20        5
  A_ACCELERATION[3, 0]         -20        5
  A_ACCELERATION[1, 1]         -20        5
  A_ACCELERATION[2, 1]         -20        5
  A_ACCELERATION[3, 1]         -20        5
  A_ACCELERATION[1, 2]         -20        5
  A_ACCELERATION[2, 2]         -20        5
  A_ACCELERATION[3, 2]         -20        5

  TDLR_VELOCITY[1, 0]         -100      100
  TDLR_VELOCITY[2, 0]         -100      100
  TDLR_VELOCITY[3, 0]         -100      100
  TDLR_VELOCITY[1, 1]         -100      100
  TDLR_VELOCITY[2, 1]         -100      100
  TDLR_VELOCITY[3, 1]         -100      100
  TDLR_VELOCITY[1, 2]         -100      100
  TDLR_VELOCITY[2, 2]         -100      100
  TDLR_VELOCITY[3, 2]         -100      100

  AR_ALTITUDE[0]                 0     2000
  AR_ALTITUDE[1]                 0     2000
  AR_ALTITUDE[2]                 0     2000

(*************************************************************************
 *   A five step implementation of the RK method.  The functions
 *   deriv_att(), deriv_vel(), and deriv_alt() are described below.
 *
 *  The interval begins at the current frame minus 2 frames.
 *
 *  1.  Compute the first estimate of the incremental value for
 *      GP_ATTITUDE, GP_VELOCITY, and GP_ALTITUDE based upon the
 *      rate of change at the beginning of the interval
 *      (2 frames ago):
 *
 *      estimate    := rate_of_change * step_size
 *************************************************************************)

  att_k1 := deriv_att(GP_ATTITUDE[1, 1, 2], 2) * 2*DELTA_T
  vel_k1 := deriv_vel(GP_VELOCITY[1, 2],
                   GP_ATTITUDE[1, 1, 2], 2) * 2*DELTA_T
  alt_k1 := deriv_alt(GP_ALTITUDE[2],
                   GP_VELOCITY[1, 2],
                   GP_ATTITUDE[1, 1, 2], 2) * 2*DELTA_T

(*************************************************************************
 *  2.  Compute the second estimate of the incremental value for
```

```
 *     GP_ATTITUDE, GP_VELOCITY, and GP_ALTITUDE based upon the
 *     rate of change at the midpoint of the first estimate k1:
 ***********************************************************************)

  att_k2 := deriv_att((GP_ATTITUDE[1, 1, 2] + att_k1/2), 1) * 2*DELTA_T
  vel_k2 := deriv_vel((GP_VELOCITY[1, 2] + vel_k1/2),
                     (GP_ATTITUDE[1, 1, 2] + att_k1/2), 1) * 2*DELTA_T
  alt_k2 := deriv_alt((GP_ALTITUDE[2] + alt_k1/2),
                     (GP_VELOCITY[1, 2] + vel_k1/2),
                     (GP_ATTITUDE[1, 1, 2] + att_k1/2), 1) * 2*DELTA_T

(***********************************************************************
 *  3.  Compute the third estimate of the incremental value for
 *      GP_ATTITUDE, GP_VELOCITY, and GP_ALTITUDE based upon the
 *      rate of change at the midpoint of the second estimate k2:
 ***********************************************************************)

  att_k3 := deriv_att((GP_ATTITUDE[1, 1, 2] + att_k2/2), 1) * 2*DELTA_T
  vel_k3 := deriv_vel((GP_VELOCITY[1, 2] + vel_k2/2),
                     (GP_ATTITUDE[1, 1, 2] + att_k2/2), 1) * 2*DELTA_T
  alt_k3 := deriv_alt((GP_ALTITUDE[2] + alt_k2/2),
                     (GP_VELOCITY[1, 2] + vel_k2/2),
                     (GP_ATTITUDE[1, 1, 2] + att_k2/2), 1) * 2*DELTA_T

(***********************************************************************
 *  4.  Compute the fourth estimate of the incremental value for
 *      GP_ATTITUDE, GP_VELOCITY, and GP_ALTITUDE based upon the
 *      the rate-of-change at the third estimate k3:
 ***********************************************************************)

  att_k4 := deriv_att((GP_ATTITUDE[1, 1, 2] + att_k3), 0) * 2*DELTA_T
  vel_k4 := deriv_vel((GP_VELOCITY[1, 2] + vel_k3),
                     (GP_ATTITUDE[1, 1, 2] + att_k3), 0) * 2*DELTA_T
  alt_k4 := deriv_alt((GP_ALTITUDE[2] + alt_k3),
                     (GP_VELOCITY[1, 2] + vel_k3),
                     (GP_ATTITUDE[1, 1, 2] + att_k3), 0) * 2*DELTA_T

(***********************************************************************
 *  5.  Perform a weighted average of the four previously computed
 *      estimates of the new value for  GP_ATTITUDE, GP_VELOCITY, and
 *      GP_ALTITUDE.
 *
 *  Note, the syntax [*, x] and [*, *, x] represent the xth history of
 *  the data element.
 ***********************************************************************)

  GP_ATTITUDE[*, *, 0] := GP_ATTITUDE[*, *, 2] +
                       (1/6)(att_k1 + 2*(att_k2 + att_k3) + att_k4)

  GP_VELOCITY[*, 0] := GP_VELOCITY[*, 2] +
                       (1/6)(vel_k1 + 2*(vel_k2 + vel_k3) + vel_k4)

  GP_ALTITUDE[0] := GP_ALTITUDE[2] +
                       (1/6)(alt_k1 + 2*(alt_k2 + alt_k3) + alt_k4)

(*** establish the "final" rotation matrix ***)
```

```
  GP_ROTATION[1, 1] :=  0
  GP_ROTATION[1, 2] :=  G_ROTATION[3, 0]
  GP_ROTATION[1, 3] := -G_ROTATION[2, 0]
  GP_ROTATION[2, 1] := -G_ROTATION[3, 0]
  GP_ROTATION[2, 2] :=  0
  GP_ROTATION[2, 3] :=  G_ROTATION[1, 0]
  GP_ROTATION[3, 1] :=  G_ROTATION[2, 0]
  GP_ROTATION[3, 2] := -G_ROTATION[1, 0]
  GP_ROTATION[3, 3] :=  0

(*************************************************************************
 *  3) Determine if the engines should be switched on or off.
 *************************************************************************)

(*************************************************************************
 * At the beginning of the trajectory, the axial engines are "off"
 * and the roll engines are "on."  During the course of the
 * trajectory the axial engines are "switched on."  Further along
 * the course of the trajectory, the axial engines and the roll
 * engines are "switched off."  These engine "switching" decisions
 * are made here.
 *************************************************************************)

(*** range check the current altitude ****)

    if  (GP_ALTITUDE[0] < 0)                                          {1}
        display-error("%EXCEPTIONAL-CONDITION-GCS-LOWER_LIMIT_EXCEEDED",
                      "GP GP", FRAME_COUNTER,
                      "GP_ALTITUDE", GP_ALTITUDE[0])

    else if  (GP_ALTITUDE[0] > 2000)                                  {1}
        display-error("%EXCEPTIONAL-CONDITION-GCS-UPPER_LIMIT_EXCEEDED",
                      "GP GP", FRAME_COUNTER,
                      "GP_ALTITUDE", GP_ALTITUDE[0])
    end if                                                            {1}

(*** range check the current x-axis vehicle velocity ****)

    if  (GP_VELOCITY[1, 0] < -100)                                    {1}
        display-error("%EXCEPTIONAL-CONDITION-GCS-LOWER_LIMIT_EXCEEDED",
                      "GP GP", FRAME_COUNTER,
                      "GP_VELOCITY", GP_VELOCITY[1, 0])

    else if  (GP_VELOCITY[1, 0] > 100)                                {1}
        display-error("%EXCEPTIONAL-CONDITION-GCS-UPPER_LIMIT_EXCEEDED",
                      "GP GP", FRAME_COUNTER,
                      "GP_VELOCITY", GP_VELOCITY[1, 0])
    end if                                                            {1}

(*** ***)

  if (AE_SWITCH = 0) then           (* axial engines are "off"    *)  {1}
     if (RE_SWITCH = 1) then        (* engines not prev. "off"    *)  {2}
        if (TD_SENSED = 0) then     (* touch down "not sensed"    *)  {3}
           if (GP_ALTITUDE[0] <= ENGINES_ON_ALTITUDE) then            {4}
```

```
            AE_SWITCH := 1          (* switch axial engines "on" *)
            FRAME_ENGINES_IGNITED := FRAME_COUNTER
          end if                                                  {4}
        end if                                                    {3}
      end if                                                      {2}

    else                              (* axial engines are "on"    *) {1}
      if (TD_SENSED = 1) then         (* touch down "sensed"       *) {2}
        AE_SWITCH := 0                (* switch axial engines "off"*)
        RE_SWITCH := 0                (* switch roll engines "off" *)

      else                            (* touch down "not sensed"   *) {2}
        if (GP_ALTITUDE[0] <= DROP_HEIGHT) then                   {3}
          temp := 2*GRAVITY*maximum(GP_ALTITUDE[0],0)

          if (temp < 0) then                                      {4}
            display-error("%EXCEPTIONAL-CONDITION-GCS-NEGATIVE_SQUARE_ROOT",
                 "GP GP", FRAME_COUNTER,
                  temp)
           end if                                                 {4}

           if (sqrt(temp)+GP_VELOCITY[1,0] <= MAX_NORMAL_VELOCITY) then {4}
            AE_SWITCH := 0             (* switch axial engines "off"*)
            RE_SWITCH := 0             (* switch roll engines "off" *)
          end if                                                  {4}
        end if                                                    {3}
      end if                                                      {2}
    end if                                                        {1}

(*************************************************************************
 *  4) Compute the current velocity error.
 *
 *  For a more detailed description of the interpolation and extrapolation
 *  algorithms, see the section labeled "" in the design overview.
 *************************************************************************)

(*** compute the optimal velocity ***)

(*** convert GP_ALTITUDE from meters to kilometers ***)

    cur_altitude := GP_ALTITUDE[0] / 1000

    do for i := 1 to 100                                          {1}
      if (CONTOUR_ALTITUDE[i] = cur_altitude) then                {2}

(*** found an exact match in the table ***)

        optimal_velocity := CONTOUR_VELOCITY(i)

        i := 101                       (* early do loop exit *)

      else if (CONTOUR_ALTITUDE[i] > cur_altitude) then           {2}
        if (i > 1) then                                           {3}

(*** interpolate between i-1 and i ***)
```

```
(*** check for potential divide by zero condition ***)

          if (CONTOUR_ALTITUDE[i] = CONTOUR_ALTITUDE[i-1]) then      {4}
             display-error("%EXCEPTIONAL-CONDITION-GCS-DIVIDE_BY_ZERO",
                           "GP GP", FRAME_COUNTER)
          end if                                                     {4}

(*** interpolation formula ***)

          optimal_velocity := CONTOUR_VELOCITY[i-1] +
                    ((CONTOUR_VELOCITY[i] - CONTOUR_VELOCITY[i-1]) /
                     (CONTOUR_ALTITUDE[i] - CONTOUR_ALTITUDE[i-1]) ) *
                     (cur_altitude - CONTOUR_ALTITUDE[i-1])

          i := 101                    (* early do loop exit *)

       else      (* i = 1 and altitude < table entry *)              {3}

(*** Extrapolate for altitude < smallest value in table entries ***)

(*** check for potential divide by zero condition ***)

          if (CONTOUR_ALTITUDE[2] = CONTOUR_ALTITUDE[1]) then        {4}
             display-error("%EXCEPTIONAL-CONDITION-GCS-DIVIDE_BY_ZERO",
                           "GP GP", FRAME_COUNTER)
          end if                                                     {4}

(*** Extrapolation formula ***)

          optimal_velocity := CONTOUR_VELOCITY[1] -
                    ((CONTOUR_VELOCITY[2] - CONTOUR_VELOCITY[1]) /
                     (CONTOUR_ALTITUDE[2] - CONTOUR_ALTITUDE[1]) ) *
                     (CONTOUR_ALTITUDE[1] - cur_altitude)

          i := 101                    (* early do loop exit *)

       end if                                                        {3}

    else              (* CONTOUR_ALTITUDE[i] < cur_altitude) *)      {2}

       if ((CONTOUR_ALTITUDE[i] = 0) OR (i = 100)) then              {3}

(*** Extrapolate for altitude > largest value in table entries ***)
(*** note, i points to first (lowest) "0" entry in the table ***)

(*** check for potential divide by zero condition ***)

          if (CONTOUR_ALTITUDE[i-1] = CONTOUR_ALTITUDE[i-2]) then    {4}
             display-error("%EXCEPTIONAL-CONDITION-GCS-DIVIDE_BY_ZERO",
                           "GP GP", FRAME_COUNTER)
          end if                                                     {4}

(*** Extrapolation formula ***)

          optimal_velocity := CONTOUR_VELOCITY[i-1] +
                    ((CONTOUR_VELOCITY[i-1] - CONTOUR_VELOCITY[i-2]) /
```

```
                    (CONTOUR_ALTITUDE[i-1] - CONTOUR_ALTITUDE[i-2]) ) *
                    (cur_altitude - CONTOUR_ALTITUDE[i-1])

          i := 101                  (* early do loop exit *)

        end if                                                      {3}

      end if                                                        {2}
    end do                                                          {1}

(*** convert optimal_velocity from km/sec to m/sec ***)

    optimal_velocity := optimal_velocity * 1000

(*** compute the velocity error ***)

    VELOCITY_ERROR := GP_VELOCITY[1, 0] - optimal_velocity

(************************************************************************
 *  5) Determine if the predetermined velocity-altitude contour has
 *     been crossed.
 ************************************************************************)

 (*** range check the VELOCITY_ERROR ****)

     if  (VELOCITY_ERROR < -300)                                    {1}
         display-error("%EXCEPTIONAL-CONDITION-GCS-LOWER_LIMIT_EXCEEDED",
                       "GP GP", FRAME_COUNTER,
                       "VELOCITY_ERROR", VELOCITY_ERROR)

     else if  (VELOCITY_ERROR > 20)                                 {1}
         display-error("%EXCEPTIONAL-CONDITION-GCS-UPPER_LIMIT_EXCEEDED",
                       "GP GP", FRAME_COUNTER,
                       "VELOCITY_ERROR", VELOCITY_ERROR)
     end if                                                         {1}

     if (GP_ALTITUDE[0] <= ENGINES_ON_ALTITUDE) then                {1}
       if (CONTOUR_CROSSED = 0) then    (* "contour not crossed" *) {2}
         if (VELOCITY_ERROR >= 0) then                              {3}
           CONTOUR_CROSSED := 1        (* set "contour crossed" *)
         end if                                                     {3}
       end if                                                       {2}
     end if                                                         {1}

(************************************************************************
 *  6) Determine the current guidance phase.
 ************************************************************************)

    case GP_PHASE of

(*** trans from 1 to 2 when the "engines on altitude" is reached ***)

      1:    if (GP_ALTITUDE[0] <= ENGINES_ON_ALTITUDE) then             {1}
               GP_PHASE := 2
            end if                                                  {1}
      end {of case 1}
```

```
(*** trans from 2 to 5 when touch down is sensed ***)

     2:    if (TD_SENSED = 1) then    (* "touch down sensed" *)         {1}
              GP_PHASE := 5
           else                                                          {1}

(*** trans from 2 to 3 when the engines are hot and the chute is released ***)

              if (AE_TEMP = 2) then           (* a-engines are "hot" *) {2}
                 if (CHUTE_RELEASED = 1) then (* "chute released"    *) {3}
                    GP_PHASE := 3
                 end if                                                  {3}
              end if                                                     {2}
           end if                                                        {1}

     end {of case 2}

(*** trans from 3 to 5 when touch down is sensed ***)

     3:    if (TD_SENSED = 1) then    (* touch down "sensed" *)         {1}
              GP_PHASE := 5

           else                        (* touch down "not sensed" *)    {1}

(*** trans from 3 to 5 when the TD sensor fails and altitude too low ***)
(*** trans from 3 to 4 when the TD sensor healthy and altitude too low ***)

              if (GP_ALTITUDE[0] =< DROP_HEIGHT) then (* too low *)     {2}

                 if (TDS_STATUS = 1) then     (* sensor "failed"*)      {3}
                    GP_PHASE := 5
                 else                          (* sensor "healthy" *)   {3}
                    temp := 2*GRAVITY*maximum(GP_ALTITUDE[0],0)

                    if (temp < 0) then                                   {4}
                       display-error("%EXCEPTIONAL-CONDITION-GCS-NEGATIVE_SQUARE_ROOT",
                       "GP GP", FRAME_COUNTER,
                        temp)
                    end if                                               {4}

                    if (sqrt(temp)+GP_VELOCITY[1,0] <=
                             MAX_NORMAL_VELOCITY) then                   {4}
                       GP_PHASE := 4
                    end if                                               {4}
                 end if                                                  {3}
              end if                                                     {2}
           end if                                                        {1}

     end {of case 3}

(*** trans from 4 to 5 when touch down is sensed ***)

     4:    if (TD_SENSED = 1) then     (* touch down "sensed" *)        {1}
              GP_PHASE := 5
```

```
          else                        (* touch down "not sensed" *)   {1}

(*** trans from 4 to 5 when the TD sensor fails ***)

             if (TDS_STATUS = 1) then     (* sensor "failed"*)        {2}
                GP_PHASE := 5
             end if                                                   {2}
          end if                                                      {1}

    end {of case 4}

  end of case statement

(************************************************************************
 *  7) Determine the appropriate axial engine control law parameter index.
 ************************************************************************)

(*** Note, the optimal_velocity is computed above during the computing of
     the current velocity error.                                     ***)

  if (CL = 1) then                                                    {1}
     if (optimal_velocity = DROP_SPEED) then                          {2}
        if (GP_VELOCITY[1, 0] < DROP_SPEED) then                      {3}
           CL := 2
           TE_INTEGRAL := 0
        end if                                                        {3}
     end if                                                           {2}
  end if                                                              {1}

 return

(************************************************************************
 *  Title:  deriv_att
 *  Description:
 *      Compute the derivative of the vehicle attitude.
 *
 *  Usage:
 *      rate-of-change := deriv_att(attitude, i]
 *
 *  Arguments:
 *      attitude input pointer to longword array[1..3, 1..3]
 *      i        input           longword integer    - time history index
 *
 *         where
 *              pv := G_ROTATION[1, i]
 *              qv := G_ROTATION[2, i]
 *              rv := G_ROTATION[3, i]
 *
 ************************************************************************)

                   -                -
                   |  0   rv  -qv  |
    deriv_att  :=  | -rv   0   pv  |  x attitude
                   |  qv  -pv   0  |
                   -                -

    return
```

```
(************************************************************************
 *  Title:  deriv_vel
 *  Description:
 *      Compute the derivative of the vehicle velocity.
 *
 *  Usage:
 *      rate-of-change := deriv_vel(velocity, attitude, i)
 *
 *  Arguments:
 *      velocity input pointer to longword array[1..3]
 *      attitude input pointer to longword array[1..3, 1..3]
 *      i        input           longword integer    - time history index
 ************************************************************************)

                 -                 -
                 |   0    rv   -qv |
   deriv_vel  := |  -rv    0    pv |  x velocity +
                 |   qv   -pv    0 |
                 -                 -

        -                                              -
        | GRAVITY * attitude[1,3] + A_ACCELERATION[1,i] |
        | GRAVITY * attitude[2,3] + A_ACCELERATION[2,i] |  +
        | GRAVITY * attitude[3,3] + A_ACCELERATION[3,i] |
        -                                              -
                           -                                  -
                           | TDLR_VELOCITY[1,i] - velocity[1] |
        K_MATRIX[i]    x   | TDLR_VELOCITY[2,i] - velocity[2] |
                           | TDLR_VELOCITY[3,i] - velocity[3] |
                           -                                  -

           where
               pv := G_ROTATION[1, i]
               qv := G_ROTATION[2, i]
               rv := G_ROTATION[3, i]

(* note that "K_MATRIX[i]" represents the "i" history of the 3x3 matrix  *)

   return

(************************************************************************
 *  Title:  deriv_alt
 *  Description:
 *      Compute the derivative of the vehicle altitude.
 *
 *  Usage:
 *      rate-of-change := deriv_att(attitude, i)
 *
 *  Arguments:
 *      altitude input           longword real
 *      velocity input pointer to longword array[1..3]
 *      attitude input pointer to longword array[1..3, 1..3]
 *      i        input           longword integer    - time history index
 ************************************************************************)
```

```
    deriv_alt =

        -                                                -
        | -attitude[1,3] -attitude[2,3] -attitude[3,3] |   x velocity +
        -                                                -

          K_ALT[i] * (AR_ALTITUDE[i] - altitude)

END P_SPEC
```

NAME:
2.3;2

TITLE:
CP

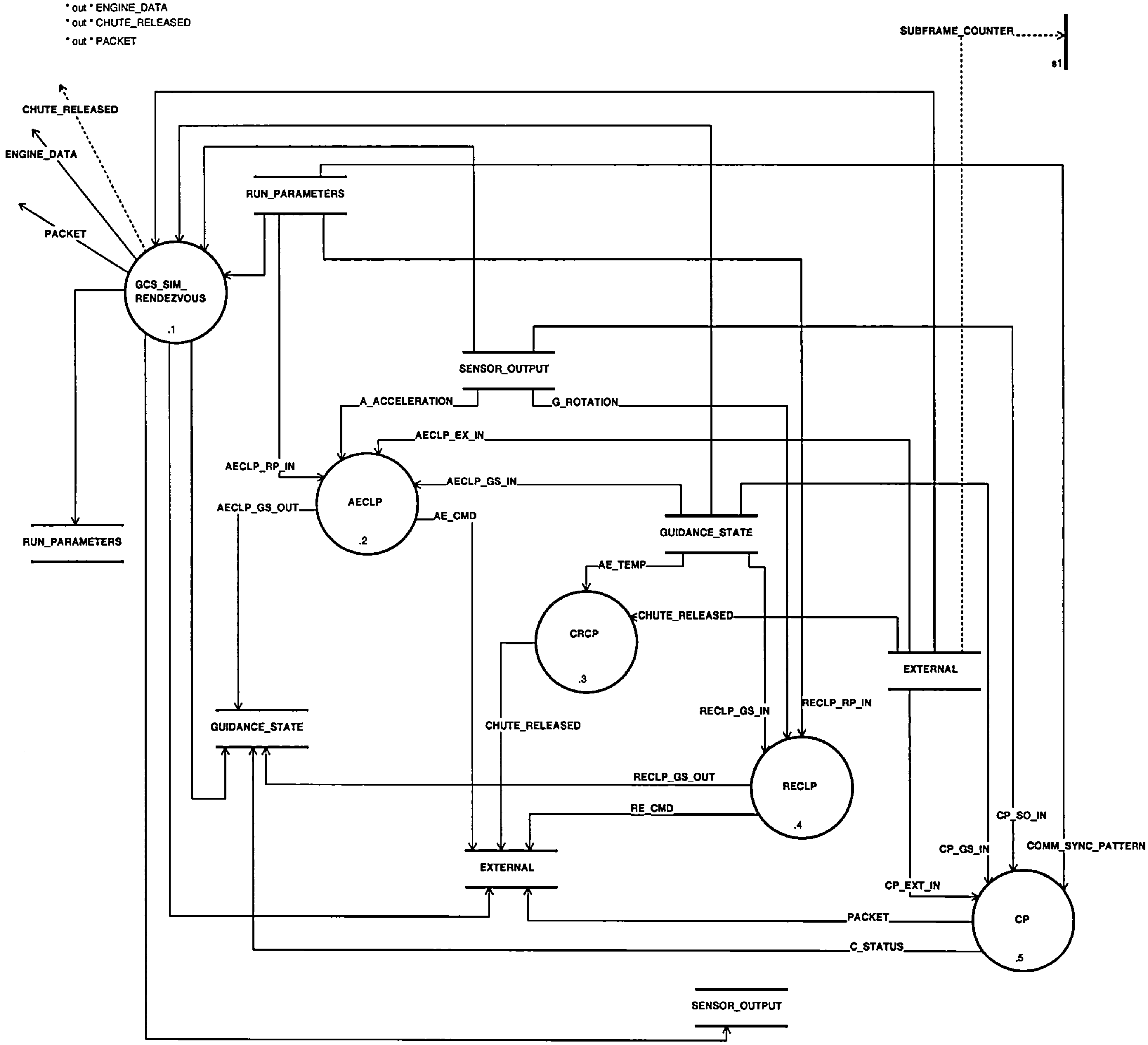
INPUT/OUTPUT:
CP_EXT_IN : data_in
CP_SO_IN : data_in
CP_GS_IN : data_in
COMM_SYNC_PATTERN : data_in
PACKET : data_out
C_STATUS : data_out

BODY:

```
Bubbles 1.8, 2.3, 3.5 and the associated P-Specs represent a single
process, CP as described in P-Spec 1.8.
```

3;9
Control Law Processing Subframe

* out * ENGINE_DATA
* out * CHUTE_RELEASED
* out * PACKET



3-s1;2
PAT - Control Law Processing Subframe

SUBFRAME_COUNTER	"GCS_SIM_RENDEZVOUS"	"AECLP"	"CRCP"	"RECLP"	"CP"
"3"	1	2	3	2	4

NAME:
3.1;4

TITLE:
GCS_SIM_RENDEZVOUS

INPUT/OUTPUT:
EXTERNAL : data_in
GUIDANCE_STATE : data_in
SENSOR_OUTPUT : data_in
RUN_PARAMETERS : data_in
GUIDANCE_STATE : data_out
EXTERNAL : data_out
SENSOR_OUTPUT : data_out
RUN_PARAMETERS : data_out
ENGINE_DATA: data_out
PACKET: data_out
CHUTE_RELEASED : data_out

BODY:

```
BEGIN P-Spec

GCS_SIM_RENDEZVOUS provides the interface to the vehicle.  This module is provided by
the systems group.

Bubbles 1.1, 2.1, 3.1 and the associated P-Specs represent a single process.

END P-Spec
```

NAME:
3.1;4

TITLE:
GCS_SIM_RENDEZVOUS

INPUT/OUTPUT:
EXTERNAL : data_in
GUIDANCE_STATE : data_in
SENSOR_OUTPUT : data_in
RUN_PARAMETERS : data_in
GUIDANCE_STATE : data_out
EXTERNAL : data_out
SENSOR_OUTPUT : data_out
RUN_PARAMETERS : data_out
ENGINE_DATA: data_out
PACKET: data_out
CHUTE_RELEASED : data_out

BODY:
```
BEGIN P-Spec

GCS_SIM_RENDEZVOUS provides the interface to the vehicle.  This module is provided by
the systems group.

Bubbles 1.1, 2.1, 3.1 and the associated P-Specs represent a single process.

END P-Spec
```

AECLP

NAME:
3.2;40

TITLE:
AECLP

INPUT/OUTPUT:
AE_SWITCH :data_in

AE_TEMP :data_in

A_ACCELERATION :data_in

CHUTE_RELEASED :data_in

CL :data_in

CONTOUR_CROSSED :data_in

DELTA_T :data_in

ENGINES_ON_ALTITUDE :data_in

FRAME_COUNTER :data_in

FRAME_ENGINES_IGNITED :data_in

FULL_UP_TIME :data_in

GA :data_in

GAX :data_in

GP1 :data_in

GP2 :data_in

GPY :data_in

GP_ALTITUDE :data_in

GP_ATTITUDE :data_in

GP_ROTATION :data_in

GP_VELOCITY :data_in

GQ :data_in

GR :data_in

GRAVITY :data_in

GV :data_in

GVE :data_in

GVEI :data_in

GVI :data_in

GW :data_in

GWI :data_in

INTERNAL_CMD: data_in

OMEGA :data_in

PE_INTEGRAL :data_in

PE_MAX :data_in

PE_MIN :data_in

TE_DROP :data_in

TE_INIT :data_in

TE_INTEGRAL :data_in

TE_LIMIT :data_in

TE_MAX :data_in

TE_MIN :data_in

VELOCITY_ERROR :data_in

YE_INTEGRAL :data_in

YE_MAX :data_in

YE_MIN :data_in

AE_CMD :data_out

AE_STATUS :data_out

AE_TEMP :data_out

INTERNAL_CMD :data_out

PE_INTEGRAL :data_out

TE_INTEGRAL :data_out

TE_LIMIT :data_out

YE_INTEGRAL :data_out

BODY:

```
BEGIN P_SPEC

(************************************************************************
 *  AECLP -- Axial Engine Control Law Processing
 *
 *  AECLP processing is responsible for:
 *  1) determining the current operational status of the axial engines, and
 *  2) generating the appropriate axial engine commands.
 ************************************************************************)

(*************************************************************************
 *  1) Determine the current operational status of the axial engines.
 *
 *  The operational status of the axial engines is always reported
 *  as "Healthy" (value 0).
 *************************************************************************)

   AE_STATUS := 0

(*************************************************************************
 *  2) Generate the appropriate axial engine commands.
 *
 *  Determine if the axial engines are on.  If the axial engines
 *  are "off" (value 0) then the axial engine commands are "0".
 *  Otherwise, further processing is required in order to determine
 *  the appropriate axial engine commands.
 *************************************************************************)

   if (AE_SWITCH = 0)   then           (* axial engines are off *)      {1}
      AE_CMD[1] := 0
      AE_CMD[2] := 0
      AE_CMD[3] := 0

   else                                                                  {1}

(*************************************************************************
 * The axial engines are "on" so further processing is required.
 *
 * 2A) determine the axial engine temperature.
 *************************************************************************)

(*** range check the current altitude ***)

      if  (GP_ALTITUDE[0] < 0) then                                     {2}
         display-error("%EXCEPTIONAL-CONDITION-GCS-LOWER_LIMIT_EXCEEDED",
                       "AECLP AECLP", FRAME_COUNTER,
                       "GP_ALTITUTE", GP_ALTITUDE[0])

      else if (GP_ALTITUDE[0] > 2000) then                              {2}
         display-error("%EXCEPTIONAL-CONDITION-GCS-UPPER_LIMIT_EXCEEDED",
                       "AECLP AECLP", FRAME_COUNTER,
                       "GP_ALTITUTE", GP_ALTITUDE[0])
      end if                                                            {2}

(*************************************************************************
```

```
 * The three possible engine temperature states are: "Cold" (value 0),
 * "Warming up" (value 1), and "Hot" (value 2).  The current temperature
 * of the axial engines is stored in the data element AE_TEMP.
 ***************************************************************************)

     if (GP_ALTITUDE[0] <= ENGINES_ON_ALTITUDE) then                      {2}

        if (AE_TEMP = 0) then          (* engines are "Cold" *)           {3}

           if ((FRAME_COUNTER - FRAME_ENGINES_IGNITED) * DELTA_T <        {4}
                     FULL_UP_TIME) then
              AE_TEMP := 1             (* "Warming up" *)
           end if                                                         {4}

        else if (AE_TEMP = 1) then     (* engines are "Warming up" *)     {3}

           if ((FRAME_COUNTER - FRAME_ENGINES_IGNITED) * DELTA_T >=       {4}
                     FULL_UP_TIME) then
              AE_TEMP := 2             (* "Hot" *)
           end if                                                         {4}

        end if                                                            {3}

     end if                                                               {2}

(***************************************************************************
 * 2B) Compute the pitch error limit.
 ***************************************************************************)

(*** range check the pitch error integral ***)

     if  (PE_INTEGRAL < -100) then                                        {2}
        display-error("%EXCEPTIONAL-CONDITION-GCS-LOWER_LIMIT_EXCEEDED",
                       "AECLP AECLP", FRAME_COUNTER,
                       "PE_INTEGRAL", PE_INTEGRAL)

     else if (PE_INTEGRAL > 100) then                                     {2}
        display-error("%EXCEPTIONAL-CONDITION-GCS-UPPER_LIMIT_EXCEEDED",
                       "AECLP AECLP", FRAME_COUNTER,
                       "PE_INTEGRAL", PE_INTEGRAL)
     end if                                                               {2}

(*** range check the x-axis roll rate ***)

     if  (GP_VELOCITY[1, 0] < -100) then                                  {2}
        display-error("%EXCEPTIONAL-CONDITION-GCS-LOWER_LIMIT_EXCEEDED",
                       "AECLP AECLP", FRAME_COUNTER,
                       "x-axis GP_VELOCITY", GP_VELOCITY[1, 0])

     else if (GP_VELOCITY[1, 0] > 100) then                               {2}
        display-error("%EXCEPTIONAL-CONDITION-GCS-UPPER_LIMIT_EXCEEDED",
                       "AECLP AECLP", FRAME_COUNTER,
                       "x-axis GP_VELOCITY", GP_VELOCITY[1, 0])
     end if                                                               {2}

(*** range check the z-axis roll rate ***)
```

```
     if  (GP_VELOCITY[3, 0] < -100) then                              {2}
        display-error("%EXCEPTIONAL-CONDITION-GCS-LOWER_LIMIT_EXCEEDED",
                      "AECLP AECLP", FRAME_COUNTER,
                      "z-axis GP_VELOCITY", GP_VELOCITY[3, 0])

     else if (GP_VELOCITY[3, 0] > 100) then                           {2}
        display-error("%EXCEPTIONAL-CONDITION-GCS-UPPER_LIMIT_EXCEEDED",
                      "AECLP AECLP", FRAME_COUNTER,
                      "z-axis GP_VELOCITY", GP_VELOCITY[3, 0])
     end if                                                           {2}

(*** check for potential divide by zero condition ***)

     if  (GP_VELOCITY[1, 0] = 0) then                                 {2}
        display-error("%EXCEPTIONAL-CONDITION-GCS-DIVIDE_BY_ZERO",
                      "AECLP AECLP", FRAME_COUNTER,
                      "x-axis GP_VELOCITY")
     end if                                                           {2}

(*** compute the current value for PE_INTEGRAL ***)

     PE_INTEGRAL := PE_INTEGRAL +
               (GP_VELOCITY[3, 0] / ABS(GP_VELOCITY[1, 0])) * DELTA_T

        where "ABS()" represents the absolute value operation

(*** range check the pitch error integral (again) ***)

     if  (PE_INTEGRAL < -100) then                                    {2}
        display-error("%EXCEPTIONAL-CONDITION-GCS-LOWER_LIMIT_EXCEEDED",
                      "AECLP AECLP", FRAME_COUNTER,
                      "PE_INTEGRAL", PE_INTEGRAL)

     else if (PE_INTEGRAL > 100) then                                 {2}
        display-error("%EXCEPTIONAL-CONDITION-GCS-UPPER_LIMIT_EXCEEDED",
                      "AECLP AECLP", FRAME_COUNTER,
                      "PE_INTEGRAL", PE_INTEGRAL)
     end if                                                           {2}

(*** range check the pitch rotational displacement ***)

     if  (GP_ROTATION[3, 1] < -1) then                                {2}
        display-error("%EXCEPTIONAL-CONDITION-GCS-LOWER_LIMIT_EXCEEDED",
                      "AECLP AECLP", FRAME_COUNTER,
                      "GP_ROTATION", GP_ROTATION[3, 1])

     else if (GP_ROTATION[3, 1] > 1) then                             {2}
        display-error("%EXCEPTIONAL-CONDITION-GCS-UPPER_LIMIT_EXCEEDED",
                      "AECLP AECLP", FRAME_COUNTER,
                      "GP_ROTATION", GP_ROTATION[3, 1])
     end if                                                           {2}

(*** compute the pitch error limit ***)

     pitch_error_limit := GQ[CL] * GP_ROTATION[3, 1] +
```

```
          GW[CL] * (GP_VELOCITY[3, 0] / ABS(GP_VELOCITY[1, 0])) +
          GWI[CL] * PE_INTEGRAL

      if (pitch_error_limit < PE_MIN[CL]) then                          {2}
         pitch_error_limit := PE_MIN[CL]

      else if (pitch_error_limit > PE_MAX[CL]) then                     {2}
         pitch_error_limit := PE_MAX[CL]

      end if                                                            {2}

(*************************************************************************
 * 2C) Compute the yaw error limit.
 *************************************************************************)

(*** range check the yaw error integral ***)

      if  (YE_INTEGRAL < -100) then                                     {2}
         display-error("%EXCEPTIONAL-CONDITION-GCS-LOWER_LIMIT_EXCEEDED",
                       "AECLP AECLP", FRAME_COUNTER,
                       "YE_INTEGRAL", YE_INTEGRAL)

      else if (YE_INTEGRAL > 100) then                                  {2}
         display-error("%EXCEPTIONAL-CONDITION-GCS-UPPER_LIMIT_EXCEEDED",
                       "AECLP AECLP", FRAME_COUNTER,
                       "YE_INTEGRAL", YE_INTEGRAL)
      end if                                                            {2}

(*** range check the y-axis roll rate ***)

      if  (GP_VELOCITY[2, 0] < -100) then                               {2}
         display-error("%EXCEPTIONAL-CONDITION-GCS-LOWER_LIMIT_EXCEEDED",
                       "AECLP AECLP", FRAME_COUNTER,
                       "y-axis GP_VELOCITY", GP_VELOCITY[2, 0])

      else if (GP_VELOCITY[2, 0] > 100) then                            {2}
         display-error("%EXCEPTIONAL-CONDITION-GCS-UPPER_LIMIT_EXCEEDED",
                       "AECLP AECLP", FRAME_COUNTER,
                       "y-axis GP_VELOCITY", GP_VELOCITY[2, 0])
      end if                                                            {2}

(*** check for potential divide by zero condition ***)

      if  (GP_VELOCITY[1, 0] = 0) then                                  {2}
         display-error("%EXCEPTIONAL-CONDITION-GCS-DIVIDE_BY_ZERO",
                       "AECLP AECLP", FRAME_COUNTER,
                       GP_VELOCITY[1, 0])
      end if                                                            {2}

(*** Compute the current value for YE_INTEGRAL ***)

      YE_INTEGRAL := YE_INTEGRAL +
          (GP_VELOCITY[2, 0] / ABS(GP_VELOCITY[1, 0])) * DELTA_T

(*** range check the yaw error integral (again) ***)
```

```
      if  (YE_INTEGRAL < -100) then                                   {2}
         display-error("%EXCEPTIONAL-CONDITION-GCS-LOWER_LIMIT_EXCEEDED",
                       "AECLP AECLP", FRAME_COUNTER,
                       "YE_INTEGRAL", YE_INTEGRAL)

      else if (YE_INTEGRAL > 100) then                                {2}
         display-error("%EXCEPTIONAL-CONDITION-GCS-UPPER_LIMIT_EXCEEDED",
                       "AECLP AECLP", FRAME_COUNTER,
                       "YE_INTEGRAL", YE_INTEGRAL)
      end if                                                          {2}

(*** range check the yaw rotational displacement ***)

      if  (GP_ROTATION[1, 2] < -1) then                               {2}
         display-error("%EXCEPTIONAL-CONDITION-GCS-LOWER_LIMIT_EXCEEDED",
                       "AECLP AECLP", FRAME_COUNTER,
                       "GP_ROTATION", GP_ROTATION[1, 2])

      else if (GP_ROTATION[1, 2] > 1) then                            {2}
         display-error("%EXCEPTIONAL-CONDITION-GCS-UPPER_LIMIT_EXCEEDED",
                       "AECLP AECLP", FRAME_COUNTER,
                       "GP_ROTATION", GP_ROTATION[1, 2])
      end if                                                          {2}

(*** compute the yaw error limit ***)

      yaw_error_limit := -GR[CL] * GP_ROTATION[1, 2] +
               GV[CL] * (GP_VELOCITY[2, 0] / ABS(GP_VELOCITY[1, 0])) +
               GVI[CL] * YE_INTEGRAL

      if (yaw_error_limit < YE_MIN[CL]) then                          {2}
         yaw_error_limit := YE_MIN[CL]

      else if (yaw_error_limit > YE_MAX[CL]) then                     {2}
         yaw_error_limit := YE_MAX[CL]

      end if                                                          {2}

(**************************************************************************
 * 2D) Compute the thrust limiting error.
 **************************************************************************)

      if (CONTOUR_CROSSED = 1)          (* "contour crossed" *)       {2}

(*** range check the thrust error integral ***)

         if (TE_INTEGRAL < -100) then                                 {3}
            display-error("%EXCEPTIONAL-CONDITION-GCS-LOWER_LIMIT_EXCEEDED",
                       "AECLP AECLP", FRAME_COUNTER,
                       "TE_INTEGRAL", TE_INTEGRAL)

         else if (TE_INTEGRAL > 100) then                             {3}
            display-error("%EXCEPTIONAL-CONDITION-GCS-UPPER_LIMIT_EXCEEDED",
                       "AECLP AECLP", FRAME_COUNTER,
                       "TE_INTEGRAL", TE_INTEGRAL)
         end if                                                       {3}
```

```
(*** range check the velocity error ***)

        if (VELOCITY_ERROR < -300) then                            {3}
           display-error("%EXCEPTIONAL-CONDITION-GCS-LOWER_LIMIT_EXCEEDED",
                       "AECLP AECLP", FRAME_COUNTER,
                       "VELOCITY_ERROR", VELOCITY_ERROR)

        else if (VELOCITY_ERROR > 20) then                         {3}
           display-error("%EXCEPTIONAL-CONDITION-GCS-UPPER_LIMIT_EXCEEDED",
                       "AECLP AECLP", FRAME_COUNTER,
                       "VELOCITY_ERROR", VELOCITY_ERROR)
     end if                                                        {3}

(*** Compute the current value for TE_INTEGRAL ***)

        TE_INTEGRAL := TE_INTEGRAL + VELOCITY_ERROR * DELTA_T

(*** range check the thrust error integral (again) ***)

        if (TE_INTEGRAL < -100) then                               {3}
           display-error("%EXCEPTIONAL-CONDITION-GCS-LOWER_LIMIT_EXCEEDED",
                       "AECLP AECLP", FRAME_COUNTER,
                       "TE_INTEGRAL", TE_INTEGRAL)

        else if (TE_INTEGRAL > 100) then                           {3}
           display-error("%EXCEPTIONAL-CONDITION-GCS-UPPER_LIMIT_EXCEEDED",
                       "AECLP AECLP", FRAME_COUNTER,
                       "TE_INTEGRAL", TE_INTEGRAL)
        end if                                                     {3}

(*** range check the attitude component ***)

        if (GP_ATTITUDE[1, 3, 0] < -1) then                        {3}
           display-error("%EXCEPTIONAL-CONDITION-GCS-LOWER_LIMIT_EXCEEDED",
                       "AECLP AECLP", FRAME_COUNTER,
                       "GP_ATTITUDE", GP_ATTITUDE[1, 3, 0])

        else if (GP_ATTITUDE[1, 3, 0] > 1) then                    {3}
           display-error("%EXCEPTIONAL-CONDITION-GCS-UPPER_LIMIT_EXCEEDED",
                       "AECLP AECLP", FRAME_COUNTER,
                       "GP_ATTITUDE", GP_ATTITUDE[1, 3, 0])
        end if                                                     {3}

(*** range check the x-axis acceleration ***)

        if (A_ACCELERATION[1, 0] < -20) then                       {3}
           display-error("%EXCEPTIONAL-CONDITION-GCS-LOWER_LIMIT_EXCEEDED",
                       "AECLP AECLP", FRAME_COUNTER,
                       "x-axis A_ACCELERATION", A_ACCELERATION[1, 0])

        else if (A_ACCELERATION[1, 0] > 5) then                    {3}
           display-error("%EXCEPTIONAL-CONDITION-GCS-UPPER_LIMIT_EXCEEDED",
                       "AECLP AECLP", FRAME_COUNTER,
                       "x-axis A_ACCELERATION", A_ACCELERATION[1, 0])
        end if                                                     {3}
```

```
(*** range check the thrust error limit ***)

        if (TE_LIMIT < -100) then                                      {3}
           display-error("%EXCEPTIONAL-CONDITION-GCS-LOWER_LIMIT_EXCEEDED",
                       "AECLP AECLP", FRAME_COUNTER,
                       "TE_LIMIT", TE_LIMIT)

        else if (TE_LIMIT > 100) then                                  {3}
           display-error("%EXCEPTIONAL-CONDITION-GCS-UPPER_LIMIT_EXCEEDED",
                       "AECLP AECLP", FRAME_COUNTER,
                       "TE_LIMIT", TE_LIMIT)
        end if                                                         {3}

(************************************************************************
 * Section "Algorithms not specificied in GCS Development
 * Specification" of the Design Overview contains a derivation of
 * the following equation used to compute TE_LIMIT.
 ************************************************************************)

       let e :=  2.718281828459045            { the number "e" }

        TE_LIMIT := (q / OMEGA) +
                   (TE_LIMIT - (q / OMEGA)) * e^(-OMEGA * DELTA_T)

           where q := GA * (-GAX * (A_ACCELERATION[1, 0] + GRAVITY *
                      GP_ATTITUDE[1, 3, 0]) + GVE * VELOCITY_ERROR +
                      GVEI[CL] * TE_INTEGRAL)

(*** range check the current value of for the thrust error limit ***)

        if (TE_LIMIT < -100) then                                      {3}
           display-error("%EXCEPTIONAL-CONDITION-GCS-LOWER_LIMIT_EXCEEDED",
                       "AECLP AECLP", FRAME_COUNTER,
                       "TE_LIMIT", TE_LIMIT)

        else if (TE_LIMIT > 100) then                                  {3}
           display-error("%EXCEPTIONAL-CONDITION-GCS-UPPER_LIMIT_EXCEEDED",
                       "AECLP AECLP", FRAME_COUNTER,
                       "TE_LIMIT", TE_LIMIT)
        end if                                                         {3}

        if (TE_LIMIT < TE_MIN[CL]) then                                {3}
           TE_LIMIT:= TE_MIN[CL]

        else if (TE_LIMIT > TE_MAX[CL]) then                           {3}
           TE_LIMIT := TE_MAX[CL]

        end if                                                         {3}

     end if                                                            {2}

(************************************************************************
 * 2E) Compute the pitch, yaw and thrust errors.
 ************************************************************************)
```

```
(*** Note, to get here (AE_SWITCH = 1) ***)

     if (CHUTE_RELEASED = 1) then      (* "Chute Released" *)          {2}

        if (CONTOUR_CROSSED = 0) then  (* "contour not crossed" *)     {3}
           pitch_error  := pitch_limiting_error
           yaw_error    := yaw_limiting_error
           thrust_error :=  TE_DROP

        else                           (* "contour crossed" *)         {3}
           pitch_error  := pitch_limiting_error
           yaw_error    := yaw_limiting_error
           thrust_error :=  TE_LIMIT
        end if                                                         {3}

     else                              (* "Chute Attached" *)
        pitch_error  :=  GQ[CL] * GP_ROTATION[3, 1]
        yaw_error    := -GR[CL] * GP_ROTATION[1, 2]
        thrust_error :=  TE_INIT

     end if                                                            {2}

(**************************************************************************
 * 2F) Compute the axial engine value settings.
 **************************************************************************)

     INTERNAL_CMD[1] := GP1 * pitch_error                  + thrust_error
     INTERNAL_CMD[2] := GP2 * pitch_error - GPY * yaw_error + thrust_error
     INTERNAL_CMD[3] := GP2 * pitch_error + GPY * yaw_error + thrust_error

(**************************************************************************
 * 2G) Convert the axial engine value settings to engine commands.
 **************************************************************************)

(*** range check the internal command ***)

     if  (INTERNAL_CMD[1] < -0.7) then                                 {2}
        display-error("%EXCEPTIONAL-CONDITION-GCS-LOWER_LIMIT_EXCEEDED",
                      "AECLP AECLP", FRAME_COUNTER,
                      "INTERNAL_CMD[1]", INTERNAL_CMD[1])

     else if (INTERNAL_CMD[1] > 1.7) then                              {2}
        display-error("%EXCEPTIONAL-CONDITION-GCS-UPPER_LIMIT_EXCEEDED",
                      "AECLP AECLP", FRAME_COUNTER,
                      "INTERNAL_CMD[1]", INTERNAL_CMD[1])
     end if                                                            {2}

     if  (INTERNAL_CMD[2] < -0.7) then                                 {2}
        display-error("%EXCEPTIONAL-CONDITION-GCS-LOWER_LIMIT_EXCEEDED",
                      "AECLP AECLP", FRAME_COUNTER,
                      "INTERNAL_CMD[2]", INTERNAL_CMD[2])

     else if (INTERNAL_CMD[2] > 1.7) then                              {2}
        display-error("%EXCEPTIONAL-CONDITION-GCS-UPPER_LIMIT_EXCEEDED",
                      "AECLP AECLP", FRAME_COUNTER,
                      "INTERNAL_CMD[2]", INTERNAL_CMD[2])
```

```
      end if                                                         {2}

      if  (INTERNAL_CMD[3] < -0.7) then                              {2}
         display-error("%EXCEPTIONAL-CONDITION-GCS-LOWER_LIMIT_EXCEEDED",
                       "AECLP AECLP", FRAME_COUNTER,
                       "INTERNAL_CMD[3]", INTERNAL_CMD[3])

      else if (INTERNAL_CMD[3] > 1.7) then                           {2}
         display-error("%EXCEPTIONAL-CONDITION-GCS-UPPER_LIMIT_EXCEEDED",
                       "AECLP AECLP", FRAME_COUNTER,
                       "INTERNAL_CMD[3]", INTERNAL_CMD[3])
      end if                                                         {2}

(*** first engine ***)

      if (INTERNAL_CMD[1] < 0) then                                  {2}
         AE_CMD[1] := 0

      else if (INTERNAL_CMD[1] <= 1) then                            {2}
         AE_CMD[1] := TRUNC(127 * INTERNAL_CMD[1] + 0.5)

             where TRUNC() represents the truncation operation
      else                                                           {2}
         AE_CMD[1] := 127

      end if                                                         {2}

(*** second engine ***)

      if (INTERNAL_CMD[2] < 0) then                                  {2}
         AE_CMD[2] := 0

      else if (INTERNAL_CMD[2] <= 1) then                            {2}
         AE_CMD[2] := TRUNC(127 * INTERNAL_CMD[2] + 0.5)

      else                                                           {2}
         AE_CMD[2] := 127

      end if                                                         {2}

(*** third engine ***)

      if (INTERNAL_CMD[3] < 0) then                                  {2}
         AE_CMD[3] := 0

      else if (INTERNAL_CMD[3] <= 1) then                            {2}
         AE_CMD[3] := TRUNC(127 * INTERNAL_CMD[3] + 0.5)

      else                                                           {2}
         AE_CMD[3] := 127

      end if                                                         {2}

   end if                                                            {1}

END P_SPEC
```

CRCP

NAME:
3.3;24

TITLE:
CRCP

INPUT/OUTPUT:
AE_TEMP : data_in
CHUTE_RELEASED :data_in
CHUTE_RELEASED:data_out

BODY:

```
BEGIN P_SPEC

(*************************************************************************
 *  CRCP -- Chute Release Control Processing
 *
 *  CRCP processing is responsible for:
 *  1) Determining whether or not to release the parachute.
 *************************************************************************)

(*************************************************************************
 *  The parachute is to be released during the same frame in which the
 *  axial engine temperature becomes "HOT" (2). Valid states for
 *  CHUTE_RELEASED are "Chute Attached" (0) and "Chute Released" (1).
 *************************************************************************)

(***  1) Determine whether or not to release the parachute. ***)

   if (CHUTE_RELEASED = 0) then        (* Chute Attached *)           {1}

      if (AE_TEMP = 2) then            (* engines are "HOT" *)        {2}
         CHUTE_RELEASED := 1           (* release the chute *)
      end if                                                          {2}

   end if                                                             {1}


END P_SPEC
```

NAME:
3.4;23

TITLE:
RECLP

INPUT/OUTPUT:
DELTA_T : data_in

G_ROTATION : data_in

P1 : data_in

P2 : data_in

P3 : data_in

P4 : data_in

RE_SWITCH : data_in

THETA : data_in

THETA1 : data_in

THETA2 : data_in

RE_CMD : data_out

RE_STATUS : data_out

THETA : data_out

BODY:
```
BEGIN P_SPEC
(***************************************************************************
 *  RECLP -- Roll Engine Control Law Processing
 *
 *  RECLP processing is responsible for:
 *  1) determining the current operational status of the roll engines, and
 *  2) generating the appropriate roll engine command.
 ***************************************************************************)

(***************************************************************************
 *  1) Determine the current operational status of the roll engines.
 *
 *  The operational status of the roll engines is always reported
 *  as "healthy" (value 0).
 ***************************************************************************)

   RE_STATUS := 0

(***************************************************************************
 *  2) Generate the appropriate roll engine command.
 *
```

```
 *  Determine if the roll engines are on.  If the roll engines
 *  are "switched off" (value 0) then the roll engine command is "1".
 *************************************************************************)

   if  (RE_SWITCH = 0)  then    (* roll engines are switched off *)      {1}
     RE_CMD := 1                                       (* off + cw *)

   else                                                                  {1}

(*** range check the x-axis vehicle rotation rate ***)

     if  (G_ROTATION[1 ,0] < -1.0) then                                  {2}
       display-error("%EXCEPTIONAL-CONDITION-GCS-LOWER_LIMIT_EXCEEDED",
                     "RECLP RECLP", FRAME_COUNTER,
                     "x-axis G_ROTATION", G_ROTATION[1 ,0])

     else if (G_ROTATION[1 ,0] > 1.0) then                               {2}
       display-error("%EXCEPTIONAL-CONDITION-GCS-UPPER_LIMIT_EXCEEDED",
                     "RECLP RECLP", FRAME_COUNTER,
                     "x-axis G_ROTATION", G_ROTATION[1 ,0])
     end if                                                              {2}

(*** range check the x-axis vehicle rotation displacement ***)

     let pi :=  3.14159265358979

     if  (THETA < (0 - pi)) then                                         {2}
       display-error("%EXCEPTIONAL-CONDITION-GCS-LOWER_LIMIT_EXCEEDED",
                     "RECLP RECLP", FRAME_COUNTER,
                     "THETA", THETA)

     else if (THETA > pi) then                                           {2}
       display-error("%EXCEPTIONAL-CONDITION-GCS-UPPER_LIMIT_EXCEEDED",
                     "RECLP RECLP", FRAME_COUNTER,
                     "THETA", THETA)
     end if                                                              {2}

(*************************************************************************
 * The roll engine command consists of two components: an
 * intensity, and a direction.  Taking into account the command data
 * encoding, the possible intensities are:  Off (0), Minimum (2),
 * Intermediate (4), and Maximum (6), and the possible directions
 * are CounterClockwise (0) and Clockwise (1).
 *
 * Both roll engine command components are determined from the
 * current value of the vehicle's roll rate and rotational
 * displacement about the x-axis.
 *
 * Employing Euler's method for differential equations, compute the
 * current x-axis angular displacement, theta.
 *************************************************************************)

     THETA     := THETA + G_ROTATION[1 ,0] * DELTA_T

(*** range check the theta again before use ***)
```

```
     if  (THETA < (0 - pi)) then                                    {2}
        display-error("%EXCEPTIONAL-CONDITION-GCS-LOWER_LIMIT_EXCEEDED",
                     "RECLP RECLP", FRAME_COUNTER,
                     "THETA", THETA)

     else if (THETA > pi) then                                      {2}
        display-error("%EXCEPTIONAL-CONDITION-GCS-UPPER_LIMIT_EXCEEDED",
                     "RECLP RECLP", FRAME_COUNTER,
                     "THETA", THETA)
     end if                                                         {2}

(************************************************************************
 * From figure 5.2 "Graph for Deriving Roll Engine Commands" of the
 * GCS development specifications, determine the appropriate roll
 * engine intensity and direction.
 ************************************************************************)

(*** check case when theta = 0 ***)

     if (THETA = 0) then                                            {2}

        if (G_ROTATION[1 ,0] > P4) then                             {3}
           RE_CMD := 6 + 1                          (* max + cw *)
        else if (G_ROTATION[1 ,0] < -P4) then                       {3}
           RE_CMD := 6 + 0                          (* max + ccw *)
        else                                                        {3}
           RE_CMD := 0 + 1                          (* off + cw *)
       end if                                                       {3}

(*** check first and fourth quadrants ***)

     else if (THETA > 0) then                                       {2}

        if (THETA <= THETA1) then                                   {3}

           if (G_ROTATION[1 ,0] > P2) then                          {4}
              RE_CMD := 6 + 1                       (* max + cw *)
           else if (G_ROTATION[1 ,0] > P1) then                     {4}
              RE_CMD := 4 + 1                       (* inter + cw *)
           else if (G_ROTATION[1 ,0] >= -P4) then                   {4}
              RE_CMD := 0 + 1                       (* off + cw *)
           else                                                     {4}
              RE_CMD := 6 + 0                       (* max + ccw *)
           end if                                                   {4}

        else if (THETA <= THETA2) then                              {3}

           if (G_ROTATION[1 ,0] > P2) then                          {4}
              RE_CMD := 6 + 1                       (* max + cw *)
           else if (G_ROTATION[1 ,0] > P1) then                     {4}
              RE_CMD := 4 + 1                       (* inter + cw *)
           else if (G_ROTATION[1 ,0] > 0.0) then                    {4}
              RE_CMD := 2 + 1                       (* min + cw *)
           else if (G_ROTATION[1 ,0] >= -P4) then                   {4}
              RE_CMD := 0 + 1                       (* off + cw *)
           else                                                     {4}
```

```
        RE_CMD := 6 + 0                              (* max + ccw *)
      end if                                                        {4}

    else  (* THETA > THETA2 *)                                      {3}

      if (G_ROTATION[1 ,0] > -P3) then                              {4}
        RE_CMD := 6 + 1                              (* max + cw *)
      else if (G_ROTATION[1 ,0] >= -P4) then                        {4}
        RE_CMD := 0 + 1                              (* off + cw *)
      else                                                          {4}
        RE_CMD := 6 + 0                              (* max + ccw *)
      end if                                                        {4}

     end if                                                         {3}

(*** check second and third quadrants ***)

  else (* THETA < 0 *)                                              {2}

    if (THETA >= -THETA1) then                                      {3}

      if (G_ROTATION[1 ,0] > p4) then                               {4}
        RE_CMD := 6 + 1                              (* max + cw *)
      else if (G_ROTATION[1 ,0] >= -P1) then                        {4}
        RE_CMD := 0 + 1                              (* off + cw *)
      else if (G_ROTATION[1 ,0] >= -P2) then                        {4}
        RE_CMD := 4 + 0                              (* inter + ccw *)
      else                                                          {4}
        RE_CMD = 6 + 0                               (* max + ccw *)
      end if                                                        {4}

    else if (THETA >= -THETA2) then                                 {3}

      if (G_ROTATION[1 ,0] > P4) then                               {4}
        RE_CMD := 6 + 1                              (* max + cw *)
      else if (G_ROTATION[1 ,0] >= 0.0) then                        {4}
        RE_CMD = 0 + 1                               (* off + cw *)
      else if (G_ROTATION[1 ,0] >= -P1) then                        {4}
        RE_CMD := 2 + 0                              (* min + ccw *)
      else if (G_ROTATION[1 ,0] >= -P2) then                        {4}
        RE_CMD := 4 + 0                              (* inter + ccw *)
      else                                                          {4}
        RE_CMD := 6 + 0                              (* max + ccw *)
      end if                                                        {4}

    else  (* THETA < -THETA2 *)                                     {3}

      if (G_ROTATION[1 ,0] > P4) then                               {4}
        RE_CMD := 6 + 1                              (* max + cw *)
      else if (G_ROTATION[1 ,0] >= P3) then                         {4}
        RE_CMD := 0 + 1                              (* off + cw *)
      else                                                          {4}
        RE_CMD = 6 + 0                               (* max + ccw *)
      end if                                                        {4}

    end if                                                          {3}
```

```
      end if                                                {2}

   end if                                                   {1}

END P_SPEC
```

NAME:
3.5;2

TITLE:
CP

INPUT/OUTPUT:
COMM_SYNC_PATTERN : data_in
CP_SO_IN : data_in
CP_GS_IN : data_in
CP_EXT_IN : data_in
PACKET : data_out
C_STATUS : data_out

BODY:

```
Bubbles 1.8, 2.3, 3.5 and the associated P-Specs represent a single
process, CP described in P-Spec 1.8.
```

A_ACCELERATION (data flow, cel) =

```
*A_ACCELERATION*


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


DESCRIPTION: vehicle accelerations
USED IN:     AECLP, ASP, CP, GP
UNITS:       meters/sec^2
RANGE:       [-20, 5]
DATA TYPE:   array(1..3, 0..4) of real-8
ATTRIBUTE:   data
DATA STORE:  SENSOR_OUTPUT
ACCURACY:    TBD
```

A_BIAS (data flow, cel) =

```
*A_BIAS*


--------


DESCRIPTION: characteristic bias in the accelerometer measurements.
USED IN:     ASP
UNITS:       meters/sec^2
RANGE:       [-30, 0]
DATA TYPE:   array (1..3) of real-8
ATTRIBUTE:   data
DATA STORE:  RUN_PARAMETERS
ACCURACY:    N/A
```

A_COUNTER (data flow, del) =

```
*A_COUNTER*


----------


DESCRIPTION: accelerating along the x ,y and z axes
USED IN:     ASP
UNITS:       none
RANGE:       [0, (2^15)-1]
DATA TYPE:   array(1..3) of Integer-2
ATTRIBUTE:   data
DATA STORE:  EXTERNAL
ACCURACY:    N/A
```

A_GAIN_0 (data flow, cel) =

```
*A_GAIN_0*


----------


DESCRIPTION: standard gain in the accelerations
USED IN:     ASP
UNITS:       meters/sec^2
```

RANGE: [0, 1]
DATA TYPE: array(1..3) of Real-8
ATTRIBUTE: data
DATA STORE: RUN_PARAMETERS
ACCURACY: N/A

A_SCALE (data flow, del) =

A_SCALE

DESCRIPTION: multiplicative constant used to determine limit on deviation accelerometer values
USED IN: ASP
UNITS: none
RANGE: [0, 3]
DATA TYPE: Integer-4
ATTRIBUTE: data
DATA STORE: RUN_PAREMETERS
ACCURACY: N/A

A_STATUS (data flow, del) =

```
[ "0"
  | "1"
]
*A_STATUS*
```

DESCRIPTION: flag indicating whether or not the accelerometers are working properly
USED IN: ASP, CP
UNITS: none
RANGE: [0:healthy, 1:unhealthy]
DATA TYPE: array(1..3, 0..3) of Logical-1
ATTRIBUTE: data
DATA STORE: GUIDANCE_STATE
ACCURACY: N/A

AE_CMD (data flow, del) =

AE_CMD

DESCRIPTION: valve settings for the axial engines
USED IN: AECLP, CP
UNITS: none
RANGE: [0, 127]
DATA TYPE: array(1..3) of Integer-2
ATTRIBUTE: data
DATA STORE: EXTERNAL
ACCURACY: TBD

.

AE_STATUS (data flow, del) =

```
[ "0"
  | "1"
]
*AE_STATUS*

-----------

DESCRIPTION: status of axial engines
USED IN:     AECLP, CP
UNITS:       none
RANGE:       [0:healthy, 1:failed]
DATA TYPE:   Logical-1
ATTRIBUTE:   data
DATA STORE:  GUIDANCE_STATE
ACCURACY:    N/A
```

AE_SWITCH (data flow, del) =

```
[ "0"
  | "1"
]
*AE_SWITCH*

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

DESCRIPTION: flag indicating whether or not axial engines are turned on
USED IN:     AECLP, GP
UNITS:       none
RANGE:       [0:AXIAL_ENGINES_ARE_OFF, 1:AXIAL_ENGINES_ARE_ON]
DATA TYPE:   Logical-1
ATTRIBUTE:   data
DATA STORE:  GUIDANCE_STATE
ACCURACY:    N/A
```

AE_TEMP (data flow, del) =

```
[ "0"
  | "1"
  | "2"
]
*AE_TEMP*

---------

DESCRIPTION: temperature of axial engines when they are turned on
USED IN:     AECLP, CP, CRCP, GP
UNITS:       none
RANGE:       [0:cold, 1:warming_up, 2:hot]
DATA TYPE:   Integer-2
ATTRIBUTE:   data
```

```
DATA STORE:  GUIDANCE_STATE
ACCURACY:    N/A
```

AECLP_EX_IN (data flow) =

```
CHUTE_RELEASED
+ FRAME_COUNTER
```

AECLP_GS_IN (data flow) =

```
AE_SWITCH
+ AE_TEMP
+ FRAME_ENGINES_IGNITED
+ CONTOUR_CROSSED
+ GP_ALTITUDE
+ GP_ATTITUDE
+ GP_ROTATION
+ GP_VELOCITY
+ INTERNAL_CMD
+ PE_INTEGRAL
+ TE_INTEGRAL
+ TE_LIMIT
+ VELOCITY_ERROR
+ YE_INTEGRAL
+ CL
```

AECLP_GS_OUT (data flow) =

```
AE_STATUS
+ AE_TEMP
+ INTERNAL_CMD
+ PE_INTEGRAL
+ TE_INTEGRAL
+ TE_LIMIT
+ YE_INTEGRAL
```

AECLP_RP_IN (data flow) =

```
ENGINES_ON_ALTITUDE
+ DELTA_T
+ FULL_UP_TIME
+ GAX
+ GP2
+ GQ
+ GRAVITY
+ GVE
+ GVI
+ GWI
+ PE_MIN
+ TE_INIT
+ TE_MIN
+ YE_MIN
+ GA
+ GP1
```

+ GPY
+ GR
+ GV
+ GVEI
+ GW
+ OMEGA
+ PE_MAX
+ TE_DROP
+ TE_MAX
+ YE_MAX

ALPHA_MATRIX (data flow, cel) =

ALPHA_MATRIX

DESCRIPTION: matrix of misalignment angles
USED IN: ASP
UNITS: none
RANGE: [-PI, PI] where PI = 3.141592653589793
DATA TYPE: array (1..3, 1..3) of Real-8
ATTRIBUTE: data
DATA STORE: RUN_PARAMENTERS
ACCURACY: N/A

AR_ALTITUDE (data flow, cel) =

AR_ALTITUDE

DESCRIPTION: altimeter radar height above terrain
USED IN: ARSP, CP, GP
UNITS: meters
RANGE: [0, 2000]
DATA TYPE: array(0..4) of Real-8
ATTRIBUTE: data
DATA STORE: SENSOR_OUTPUT
ACCURACY: TBD

AR_COUNTER (data flow, del) =

AR_COUNTER

DESCRIPTION: counter containing elapsed time since transmission of radar pulse
USED IN: ARSP
UNITS: cycles
RANGE: [-1, (2^15)-1]
DATA TYPE: Integer-2
ATTRIBUTE: data

```
DATA STORE:  EXTERNAL
ACCURACY:    N/A
```

AR_FREQUENCY (data flow) =

```
*AR_FREQUENCY*


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

DESCRIPTION: increment frequency of AR_COUNTER
USED IN:     ARSP
UNITS:       cycles/sec
RANGE:       [1, 2.45x10^9]
DATA TYPE:   Real-8
ATTRIBUTE:   data
DATA STORE:  RUN_PARAMETERS
ACCURACY:    N/A
```

AR_STATUS (data flow, del) =

```
[ "0"
  | "1"
]
*AR_STATUS*

-----------

DESCRIPTION: status of the altimeter radars
USED IN:     ARSP, CP
UNITS:       none
RANGE:       [0:healthy, 1:failed]
DATA TYPE:   array(0..4) of Logical-1
ATTRIBUTE:   data
DATA STORE:  GUIDANCE_STATE
ACCURACY:    N/A
```

ARSP_GS_IN (data flow) =

```
AR_STATUS
+ K_ALT
```

ARSP_GS_OUT (data flow) =

```
AR_STATUS
+ K_ALT
```

ASP_RP_IN (data flow) =

```
A_BIAS
+ A_GAIN_0
+ A_SCALE
+ ALPHA_MATRIX
+ G1
```

```
+ G2
```

ASP_SO_IN (data flow) =

```
A_ACCELERATION
+ ATMOSPHERIC_TEMP
```

ATMOSPHERIC_TEMP (data flow, cel) =

```
*ATMOSPHERIC_TEMP*


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

DESCRIPTION: atmospheric temperature
USED IN:     ASP, CP, GSP, TSP
UNITS:       degrees C
RANGE:       [-200, 25]
DATA TYPE:   Real-8
ATTRIBUTE:   data
DATA STORE:  SENSOR_OUTPUT
ACCURACY:    TBD
```

C_STATUS (data flow, del) =

```
[ "0"
  | "1"
]
*C_STATUS*

----------

DESCRIPTION: flag indicating whether or not the communications processor is working
             properly
USED IN:     CP
UNITS:       none
RANGE:       [0:healthy, 1:failed]
DATA TYPE:   Logical-1
ATTRIBUTE:   data
DATA STORE:  GUIDANCE_STATE
ACCURACY:    N/A
```

CHUTE_RELEASED (data/control flow, del) =

```
[ "0"
  | "1"
]
*CHUTE_RELEASED*

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

DESCRIPTION: signal indicating parachute has been released
USED IN:     AECLP, CP, CRCP, GP
UNITS:       none
RANGE:       [0:chute_attached, 1:chute_released]
DATA TYPE:   Logical-1
```

```
ATTRIBUTE:   data condition
DATA STORE:  EXTERNAL
ACCURACY:    N/A
```

CL (data flow, del) =

```
[ "1"
  | "2"
]
*CL*
- - DESCRIPTION : index which specifies which set of control law parameters to use USED IN : AECLP , GP UNITS
RANGE : [ 1 : first , 2 : second
        ] DATA TYPE : Integer-2 ATTRIBUTE : data DATA STORE : GUIDANCE_STATE ACCURACY : N / A
```

COMM_SYNC_PATTERN (data flow, pel) =

```
*COMM_SYNC_PATTERN*


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

DESCRIPTION: sixteen bit synchronization pattern
USED IN:     CP
UNITS:       none
RANGE:       [0xD9B2]    * hexadecimal notation *
DATA TYPE:   Integer-2
ATTRIBUTE:   data
DATA STORE:  RUN_PARAMETERS
ACCURACY:    N/A
```

CONTOUR_ALTITUDE (data flow, cel) =

```
*CONTOUR_ALTITUDE*


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

DESCRIPTION: altitude in velocity-altitude contour
USED IN:     GP
UNITS:       kilometers
RANGE:       [-.01, 2]
DATA TYPE:   array (1..100) of Real-8
ATTRIBUTE:   data
DATA STORE:  RUN_PARAMETERS
ACCURACY:    N/A
```

CONTOUR_CROSSED (data flow, del) =

```
[ "0"
  | "1"
]
*contour_crossed*

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

DESCRIPTION: indicates if the velocity-altitude contour has been sensed
USED IN: AECLP, CP, GP
UNITS: none
RANGE: [0:contour_not_crossed, 1:contour_crossed]
DATA TYPE: logical-1
ATTRIBUTE: data
DATA STORE: GUIDANCE_STATE
ACCURACY: N/A

CONTOUR_VELOCITY (data flow, cel) =

CONTOUR_VELOCITY

DESCRIPTION: velocity in velocity-altitude contour
USED IN: GP
UNITS: kilometers/sec
RANGE: [0, 0.5]
DATA TYPE: array(1..100) of Real-8
ATTRIBUTE: data
DATA STORE: RUN_PARAMETERS
ACCURACY: N/A

CP_EXT_IN (data flow) =

AE_CMD
+ CHUTE_RELEASED
+ FRAME_COUNTER
+ RE_CMD
+ SUBFRAME_COUNTER

CP_GS_IN (data flow) =

A_STATUS
+ AE_STATUS
+ AE_TEMP
+ AR_STATUS
+ C_STATUS
+ CONTOUR_CROSSED
+ G_STATUS
+ GP_ALTITUDE
+ GP_ATTITUDE
+ GP_PHASE

```
+ GP_ROTATION
+ GP_VELOCITY
+ K_ALT
+ K_MATRIX
+ PE_INTEGRAL
+ RE_STATUS
+ TDLR_STATE
+ TDLR_STATUS
+ TDS_STATUS
+ TE_INTEGRAL
+ TS_STATUS
+ VELOCITY_ERROR
+ YE_INTEGRAL
```

CP_SO_IN (data flow) =

```
AR_ALTITUDE
+ ATMOSPHERIC_TEMP
+ A_ACCELERATION
+ G_ROTATION
+ TDLR_VELOCITY
+ TD_SENSED
```

DELTA_T (data flow, del) =

```
*DELTA_T*


---------

DESCRIPTION: time step duration
USED IN:     AECLP, GP, RECLP, TDLRSP
UNITS:       seconds
RANGE:       [0.005, 0.20]
DATA TYPE:   Real-8
ATTRIBUTE:   data
DATA STORE:  RUN_PARAMETERS
ACCURACY:    N/A
```

DROP_HEIGHT (data flow, del) =

```
*DROP_HEIGHT*


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

DESCRIPTION: height from which vehicle should free-fall to surface
USED IN:     GP
UNITS:       meters
RANGE:       [0, 100]
DATA TYPE:   Real-8
ATTRIBUTE:   data
DATA STORE:  RUN_PARAMETERS
ACCURACY:    N/A.
```

DROP_SPEED (data flow, del) =

```
*DROP_SPEED*


----------

DESCRIPTION: optimal speed during constant velocity descent
USED IN:     GP
UNITS:       meters/sec
RANGE:       [0, 4.0]
DATA TYPE:   Real-8
ATTRIBUTE:   data
DATA STORE:  RUN_PARAMETERS
ACCURACY:    N/A
```

ENGINE_DATA (data flow) =

```
AE_CMD
+ RE_CMD
```

ENGINES_ON_ALTITUDE (data flow, del) =

```
*ENGINES_ON_ALTITUDE*


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

DESCRIPTION: altitude at which the axial engines are turned on;
USED IN:     AECLP, GP
UNITS:       meters
RANGE:       [0, 2000]
DATA TYPE:   Real-8
ATTRIBUTE:   data
DATA STORE:  RUN_PARAMETERS
ACCURACY:    N/A
```

EXTERNAL (store) =

```
*----------------------------------------------------------------------
 Implementaion of the data elements in the data store EXTERNAL must be
 arranged in the order specified below.
 ----------------------------------------------------------------------*


A_COUNTER
+ AE_CMD
+ AR_COUNTER
+ CHUTE_RELEASED
+ FRAME_COUNTER
+ G_COUNTER
+ PACKET
+ RE_CMD
+ SS_TEMP
```

+ SUBFRAME_COUNTER
+ TD_COUNTER
+ TDLR_COUNTER
+ THERMO_TEMP

FRAME_BEAM_UNLOCKED (data flow, del) =

FRAME_BEAM_UNLOCKED

DESCRIPTION: variable containing the number of the frame during which the radar beam unlocked
USED IN: TDLRSP
UNITS: none
RANGE: [0, (2^31)-1]
DATA TYPE: array(1..4) of Integer-4
ATTRIBUTE: data
DATA STORE: GUIDANCE_STATE
ACCURACY: TBD

FRAME_COUNTER (data flow) =

FRAME_COUNTER

DESCRIPTION: counter containing the number of the present frame;
USED IN: AECLP, CP, GP, TDLRSP, ARSP
UNITS: none
RANGE: [1, (2^31)-1]
DATA TYPE: Integer-4
ATTRIBUTE: data
DATA STORE: EXTERNAL
ACCURACY: N/A

FRAME_ENGINES_IGNITED (data flow, del) =

FRAME_ENGINES_IGNITED

DESCRIPTION: variable containing the number of the frame during which the engines were ignited
USED IN: AECLP, GP
UNITS: none
RANGE: [0, (2^31)-1]
DATA TYPE: Integer-4
ATTRIBUTE: data
DATA STORE: GUIDANCE_STATE
ACCURACY: TBD

FULL_UP_TIME (data flow, del) =

FULL_UP_TIME

DESCRIPTION: time for axial engines to reach optimum operational condition
USED IN: AECLP
UNITS: seconds
RANGE: [0, 60]
DATA TYPE: Real-8
ATTRIBUTE: data
DATA STORE: RUN_PARAMETERS
ACCURACY: N/A

G1 (data flow, del) =

G1

- - DESCRIPTION : coefficent used to adjust A_GAIN USED IN : ASP UNITS : (meters / sec ^ 2) / (degree_C) R
] DATA TYPE : Real-8 ATTRIBUTE : data DATA STORE : RUN_PARAMETERS ACCURACY : N / A

G2 (data flow, del) =

G2

- - DESCRIPTION : coefficent used to adjust A_GAIN USED IN : ASP UNITS : (meters / sec ^ 2) / degree_C ^ 2 R
] DATA TYPE : Real-8 ATTRIBUTE : data DATA STORE : RUN_PARAMETERS ACCURACY : N / A .

G3 (data flow, del) =

G3

- - DESCRIPTION : coefficent used to adjust G_GAIN USED IN : GSP UNITS : (radians / sec) / degree_C RANGE :
] DATA TYPE : Real-8 ATTRIBUTE : data DATA STORE : RUN_PARAMETERS ACCURACY : N / A

G4 (data flow, del) =

G4

- - DESCRIPTION : coefficient used to adjust G_GAIN USED IN : GSP UNITS : (radians / sec) / degree_C ^ 2 RAN
] DATA TYPE : Real-8 ATTRIBUTE : data DATA STORE : RUN_PARAMETERS ACCURACY : N / A

G_COUNTER (data flow, del) =

G_COUNTER

DESCRIPTION: gyroscope measurement of vehicle rotation rates;
USED IN: GSP
UNITS: none
RANGE: [-(2^14)-1, (2^14)-1]
DATA TYPE: array (1..3) of Integer-2
ATTRIBUTE: data;

DATA STORE: EXTERNAL
ACCURACY: N/A

G_GAIN_0 (data flow, del) =

G_GAIN_0

DESCRIPTION: standard gain in vehicle rotation rates as measured by the gyroscopes
USED IN: GSP
UNITS: radians/sec
RANGE: [-1, 1]
DATA TYPE: array (1..3) of Real-8
ATTRIBUTE: data
DATA STORE: RUN_PARAMETERS
ACCURACY: N/A

G_OFFSET (data flow, del) =

G_OFFSET

DESCRIPTION: standard offset of the rotation raw values
USED IN: GSP
UNITS: radians/sec
RANGE: [-0.5, 0.5]
DATA TYPE: array (1..3) of Real-8
ATTRIBUTE: data
DATA STORE: RUN_PARAMETERS
ACCURACY: N/A

G_ROTATION (data flow, cel) =

G_ROTATION

DESCRIPTION: vehicle rotation rates
USED IN: CP, GSP, GP, RECLP
UNITS: radians/sec
RANGE: [-1.0, 1.0]
DATA TYPE: array (1..3, 0..4) of Real-8
ATTRIBUTE: data
DATA STORE: SENSOR_OUTPUT
ACCURACY: TBD

G_STATUS (data flow, del) =

["0"

```
  | "1"
]
*G_STATUS*

----------

DESCRIPTION: status of the gyroscopes
USED IN:     CP, GSP
UNITS:       none
RANGE:       [0:healthy, 1:failed]
DATA TYPE:   Logical-1
ATTRIBUTE:   data
DATA STORE:  GUIDANCE_STATE
ACCURACY:    N/A
```

GA (data flow) =

GA

- - DESCRIPTION : gain USED IN : AECLP UNITS : sec / meter RANGE : [0 , 50
] DATA TYPE : Real-8 ATTRIBUTE : data DATA
RUN_PARAMETERS ACCURACY : N / A

GAX (data flow) =

GAX

- - - DESCRIPTION : gain USED IN : AECLP UNITS : none RANGE : [0 , 5
] DATA TYPE : Real-8 ATTRIBUTE : data DATA STORE
RUN_PARAMETERS ACCURACY : N / A

GP1 (data flow) =

GP1

- - - DESCRIPTION : gain USED IN : AECLP UNITS : none RANGE : [- , 5
] DATA TYPE : Real-8 ATTRIBUTE : data DATA STORE
RUN_PARAMETERS ACCURACY : N / A

GP2 (data flow) =

GP2

- - - DESCRIPTION : gain USED IN : AECLP UNITS : none RANGE : [- , 5
] DATA TYPE : Real-8 ATTRIBUTE : data DATA STORE
RUN_PARAMETERS ACCURACY : N / A

GP_ALTITUDE (data flow) =

```
*GP_ALTITUDE*


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

DESCRIPTION: altitude as seen by guidance processor
USED IN:     AECLP, CP, GP
```

UNITS: meters
RANGE: [0, 2000]
DATA TYPE: array (0..4) of Real-8
ATTRIBUTE: data
DATA STORE: GUIDANCE_STATE
ACCURACY: TBD

GP_ATTITUDE (data flow) =

GP_ATTITUDE

DESCRIPTION: direction cosine matrix
USED IN: AECLP, CP, GP
UNITS: none
RANGE: [-1, 1]
DATA TYPE: array (1..3, 1..3, 0..4) Real-8
ATTRIBUTE: data
DATA STORE: GUIDANCE_STATE
ACCURACY: TBD

GP_EX_IN (data flow) =

CHUTE_RELEASED
+ FRAME_COUNTER

GP_GS_IN (data flow) =

AE_SWITCH
+ AE_TEMP
+ CL
+ CONTOUR_CROSSED
+ GP_ALTITUDE
+ GP_ATTITUDE
+ GP_PHASE
+ GP_VELOCITY
+ K_ALT
+ K_MATRIX
+ RE_SWITCH
+ TDS_STATUS

GP_GS_OUT (data flow) =

AE_SWITCH
+ CONTOUR_CROSSED
+ CL
+ FRAME_ENGINES_IGNITED
+ GP_ALTITUDE
+ GP_ATTITUDE
+ GP_PHASE
+ GP_ROTATION
+ GP_VELOCITY

```
+ RE_SWITCH
+ TE_INTEGRAL
+ VELOCITY_ERROR
```

GP_PHASE (data/control flow, del) =

```
[ "1"
  | "2"
  | "3"
  | "4"
  | "5"
]
*GP_PHASE*

----------

DESCRIPTION: phase of operation as seen by guidance processor
USED IN:     CP, GP
UNITS:       none
RANGE:       [1, 5]
DATA TYPE:   Integer-4
ATTRIBUTE:   data condition
DATA STORE:  GUIDANCE_STATE
ACCURACY:    TBD
```

GP_ROTATION (data flow) =

```
*GP_ROTATION*


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

DESCRIPTION: rotation rates as determined by the guidance processing functional unit
USED IN:     AECLP, CP, GP
UNITS:       radians/sec
RANGE:       [-1.0, 1.0]
DATA TYPE:   array (1..3, 1..3) Real-8
ATTRIBUTE:   data
DATA STORE:  GUIDANCE_STATE
ACCURACY:    TBD
```

GP_RP_IN (data flow) =

```
CONTOUR_ALTITUDE
+ CONTOUR_VELOCITY
+ DELTA_T
+ DROP_HEIGHT
+ ENGINES_ON_ALTITUDE
+ GRAVITY
+ DROP_SPEED
+ MAX_NORMAL_VELOCITY
```

GP_SO_IN (data flow) =

```
A_ACCELERATION
+ AR_ALTITUDE
```

```
+ G_ROTATION
+ TD_SENSED
+ TDLR_VELOCITY
```

GP_VELOCITY (data flow) =

```
*GP_VELOCITY*


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

DESCRIPTION: velocity as corrected by the guidance algorithm
USED IN:     AECLP, CP, GP
UNITS:       meters/sec
RANGE:       [-100, 100]
DATA TYPE:   array (1..3, 0..4) of Real-8
ATTRIBUTE:   data
DATA STORE:  GUIDANCE_STATE
ACCURACY:    TBD
```

GPY (data flow) =

```
*GPY*

- - - DESCRIPTION : gain USED IN : AECLP UNITS : none RANGE : [ - , 5
                                                              ] DATA TYPE : Real-8 ATTRIBUTE : data DATA STORE
RUN_PARAMETERS ACCURACY : N / A
```

GQ (data flow) =

```
*GQ*

- - DESCRIPTION : gain USED IN : AECLP UNITS : seconds RANGE : [ - , 8
                                                               ] DATA TYPE : array ( 1 ..2 ) of Real-8 ATTRIBU
```

GR (data flow) =

```
*GR*

- - DESCRIPTION : gain USED IN : AECLP UNITS : seconds RANGE : [ - , 8
                                                               ] DATA TYPE : array ( 1 ..2 ) of Real-8 ATTRIBU
```

GRAVITY (data flow) =

```
*GRAVITY*


---------

DESCRIPTION: gravity of planet
USED IN:     AECLP, GP
UNITS:       meters/sec^2
RANGE:       [0, 100]
DATA TYPE:   Real-8
ATTRIBUTE:   data
DATA STORE:  RUN_PARAMETERS
```

ACCURACY: N/A

GSP_RP_IN (data flow) =

```
G3
+ G4
+ G_GAIN_0
+ G_OFFSET
```

GSP_SO_IN (data flow) =

```
ATMOSPHERIC_TEMP
+ G_ROTATION
```

GUIDANCE_STATE (store) =

```
*-----------------------------------------------------------------------
 Implementaion of the data elements in the data store GUIDANCE_STATE
 must be arranged in the order specified below.
 -----------------------------------------------------------------------*


A_STATUS
+ AE_STATUS
+ AE_SWITCH
+ AE_TEMP
+ AR_STATUS
+ C_STATUS
+ CL
+ CONTOUR_CROSSED
+ FRAME_BEAM_UNLOCKED
+ FRAME_ENGINES_IGNITED
+ G_STATUS
+ GP_ALTITUDE
+ GP_ATTITUDE
+ GP_PHASE
+ GP_ROTATION
+ GP_VELOCITY
+ INTERNAL_CMD
+ K_ALT
+ K_MATRIX
+ PE_INTEGRAL
+ RE_STATUS
+ RE_SWITCH
+ TDLR_STATE
+ TDLR_STATUS
+ TDS_STATUS
+ TE_INTEGRAL
+ TE_LIMIT
+ THETA
+ TS_STATUS
+ VELOCITY_ERROR
+ YE_INTEGRAL
```

GV (data flow) =

GV

- - DESCRIPTION : gain USED IN : AECLP UNITS : sec / meter RANGE : [- , 8
] DATA TYPE : array (1 ..2) of Real-8 ATT

GVE (data flow) =

GVE

- - - DESCRIPTION : gain USED IN : AECLP UNITS : / second RANGE : [0 , 500
] DATA TYPE : Real-8 ATTRIBUTE : data DATA S

RUN_PARAMETERS ACCURACY : N / A

GVEI (data flow) =

GVEI

- - - DESCRIPTION : gain USED IN : AECLP UNITS : / sec ^ 2 RANGE : [- , 40
] DATA TYPE : array (1 ..2) of Real-8 ATT

GVI (data flow) =

GVI

- - - DESCRIPTION : gain USED IN : AECLP UNITS : / meter RANGE : [- , 5
] DATA TYPE : array (1 ..2) of Real-8 ATTRI

GW (data flow) =

GW

- - DESCRIPTION : gain USED IN : AECLP UNITS : sec / meter RANGE : [- , 8
] DATA TYPE : array (1 ..2) of Real-8 ATT

GWI (data flow) =

GWI

- - - DESCRIPTION : gain USED IN : AECLP UNITS : / meter RANGE : [- , 5
] DATA TYPE : array (1 ..2) of Real-8 ATTRI

INTERNAL_CMD (data flow) =

INTERNAL_CMD

DESCRIPTION: real vector containing the command to be sent to the axial engines
USED IN: AECLP
UNITS: none
RANGE: [-0.7, 1.7]
DATA TYPE: array (1..3) of Real-8
ATTRIBUTE: data
DATA STORE: GUIDANCE_STATE
ACCURACY: TBD

K_ALT (data flow) =

K_ALT

- - - - DESCRIPTION : determines use of altimeter radar by guidance processor USED BY : ARSP , CP , GP UNITS :
RANGE : [0 , 1
] DATA TYPE : array (0 ..4) of integer-4 ATTRIBUTE : data DATA STORE : GUIDANCE_STATE ACCURACY : N /

K_MATRIX (data flow) =

K_MATRIX

```
DESCRIPTION: determines use of the doppler radar by guidance processor.
USED IN:     CP, GP, TDLRSP
UNITS:       none
RANGE:       [0, 1]
DATA TYPE:   array (1..3, 1..3, 0..4) integer-4
ATTRIBUTE:   data
DATA STORE:  GUIDANCE_STATE
ACCURACY:    N/A
```

M1 (data flow) =

M1

- - DESCRIPTION : lower measured temperature calibration point for solid state temperature sensor .USED IN : T
none RANGE : [0 , (2 ^ 15) -
] DATA TYPE : interger-2 ATTRIBUTE : data DATA STORE : RUN_PARAMETERS ACCURACY : N / A

M2 (data flow) =

M2

- - DESCRIPTION : upper measured temperature calibration point for solid state temperature sensor .USED IN : T
none RANGE : [0 , (2 ^ 15) -
] DATA TYPE : integer-2 ATTRIBUTE : data DATA STORE : RUN_PARAMETERS ACCURACY : N / A .

M3 (data flow) =

M3

- - DESCRIPTION : lower measured temperature calibration point for thermocouple pair temperature sensor USED I
UNITS : none RANGE : [0 , (2 ^ 15) -
] DATA TYPE : integer-2 ATTRIBUTE : data DATA STORE : RUN_PARAMETERS ACCURACY : N / A

M4 (data flow) =

M4

- - DESCRIPTION : upper measured temperature calibration point for thermocouple pair temperature sensor USED I

UNITS : none RANGE : [0 , (2 ^ 15) -
] DATA TYPE : integer-2 ATTRIBUTE : data DATA STORE : RUN_PARAMETERS ACCURACY : N / A

MAX_NORMAL_VELOCITY (data flow) =

MAX_NORMAL_VELOCITY

DESCRIPTION: maximum vertical velocity for safe landing
USED IN: GP
UNITS: meters/sec
RANGE: [0, 3.35]
DATA TYPE: real-8
ATTRIBUTE: data
DATA STORE: RUN_PARAMETERS
ACCURACY: N/A

OMEGA (data flow) =

OMEGA

- - - - DESCRIPTION : gain of angular velocity USED IN : AECLP UNITS : / second RANGE : [- 0 , 50
] DATA TYPE : real-8 ATTRIBUTE : data DATA STORE : RUN_PARAMETERS ACCURACY : N / A

P1 (data flow) =

P1

- - DESCRIPTION : pulse rate boundary USED IN : RECLP UNITS : radians / sec RANGE : [0 , 0 .05
] DATA TYPE : real-8 ATTRIBUTE : data DATA STORE : RUN_PARAMETERS ACCURACY : N / A

P2 (data flow) =

P2

- - DESCRIPTION : pulse rate boundary USED IN : RECLP UNITS : radians / sec RANGE : [0 , 0 .05
] DATA TYPE : real-8 ATTRIBUTE : data DATA STORE : RUN_PARAMETERS ACCURACY : N / A

P3 (data flow) =

P3

- - DESCRIPTION : pulse rate boundary USED IN : RECLP UNITS : radians / sec RANGE : [0 , 0 .05
] DATA TYPE : real-8 ATTRIBUTE : data DATA STORE : RUN_PARAMETERS ACCURACY : N / A

P4 (data flow) =

P4

- - DESCRIPTION : pulse rate boundary USED IN : RECLP UNITS : radians / sec RANGE : [0 , 0 .05
] DATA TYPE : real-8 ATTRIBUTE : data DATA STORE : RUN_PARAMETERS ACCURACY : N / A

PACKET (data flow) =

PACKET

```
DESCRIPTION: packet of telemetry data
USED IN:     CP
UNITS:       N/A
RANGE:       N/A
DATA TYPE:   array (1..256) of integer-2
ATTRIBUTE:   data
DATA STORE:  EXTERNAL
ACCURACY:    N/A
```

PE_INTEGRAL (data flow) =

PE_INTEGRAL

```
DESCRIPTION: integral portion of pitch error equation
USED IN:     AECLP, CP
UNITS:       meters
RANGE:       [-100, 100]
DATA TYPE:   real-8
ATTRIBUTE:   data
DATA STORE:  GUIDANCE_STATE
ACCURACY:    TBD
```

PE_MAX (data flow) =

PE_MAX

```
DESCRIPTION: maximum pitch error tolerable
USED IN:     AECLP
UNITS:       none
RANGE:       [0, 1]
DATA TYPE:   array (1..2) of real-8
ATTRIBUTE:   data
DATA STORE:  RUN_PARAMETERS
ACCURACY:    N/A
```

PE_MIN (data flow) =

PE_MIN

```
DESCRIPTION: minimum pitch error tolerable
USED IN:     AECLP
```

```
UNITS:        none
RANGE:        [-1, 0]
DATA TYPE:    array (1..2) of real-8
ATTRIBUTE:    data
DATA STORE:   RUN_PARAMETERS
ACCURACY:     N/A
```

RAW_SENSOR_DATA (data flow) =

```
A_COUNTER
+ AR_COUNTER
+ G_COUNTER
+ SS_TEMP
+ TD_COUNTER
+ TDLR_COUNTER
+ THERMO_TEMP
```

RE_CMD (data flow) =

```
[ "1"
  | "2"
  | "3"
  | "4"
  | "5"
  | "6"
  | "7"
]
*RE_CMD*

--------

DESCRIPTION: roll engine command
USED IN:      CP,RECLP
UNITS:        none
RANGE:        [1, 7]
                1 off
                2 minimum, counterclockwise
                3 minimum, clockwise
                4 intermediate, counterclockwise
                5 intermediate, clockwise
                6 maximum, counterclockwise
                7 maximum, clockwise
DATA TYPE:    integer-2
ATTRIBUTE:    data
DATA STORE:   EXTERNAL
ACCURACY:     TBD
```

RE_STATUS (data flow) =

```
[ "0"
  | "1"
]
```

```
*RE_STATUS*

-----------

DESCRIPTION: status of the roll engines
USED IN:     CP, RECLP
UNITS:       none
RANGE:       [0:healthy, 1:failed]
DATA TYPE:   logical-1
ATTRIBUTE:   data
DATA STORE:  GUIDANCE_STATE
ACCURACY:    N/A
```

RE_SWITCH (data flow) =

```
[ "0"
  | "1"
]
*RE_SWITCH*

-----------

DESCRIPTION: flag indicating whether or not the roll engines are turned on
USED IN:     GP, RECLP
UNITS:       none
RANGE:       [0:roll_engines_are_off, 1:roll_engines_are_on]
DATA TYPE:   logical-1
ATTRIBUTE:   data
DATA STORE:  GUIDANCE_STATE
ACCURACY:    N/A
```

RECLP_GS_IN (data flow) =

```
RE_SWITCH
+ THETA
```

RECLP_GS_OUT (data flow) =

```
RE_STATUS
+ THETA
```

RECLP_RP_IN (data flow) =

```
DELTA_T
+ P1
+ P2
+ P3
+ P4
+ THETA1
+ THETA2
```

RUN_PARAMETERS (store) =

```
*-----------------------------------------------------------------------
 Implementaion of the data elements in the data store RUN_PARAMETERS
 must be arranged in the order specified below.
 ----------------------------------------------------------------------*


A_BIAS
+ A_GAIN_0
+ A_SCALE
+ ALPHA_MATRIX
+ AR_FREQUENCY
+ COMM_SYNC_PATTERN
+ CONTOUR_ALTITUDE
+ CONTOUR_VELOCITY
+ DELTA_T
+ DROP_HEIGHT
+ DROP_SPEED
+ ENGINES_ON_ALTITUDE
+ FULL_UP_TIME
+ G1
+ G2
+ G3
+ G4
+ G_GAIN_0
+ G_OFFSET
+ GA
+ GAX
+ GP1
+ GP2
+ GPY
+ GQ
+ GR
+ GRAVITY
+ GV
+ GVE
+ GVEI
+ GVI
+ GW
+ GWI
+ M1
+ M2
+ M3
+ M4
+ MAX_NORMAL_VELOCITY
+ OMEGA
+ P1
+ P2
+ P3
+ P4
+ PE_MAX
+ PE_MIN
+ T1
+ T2
+ T3
+ T4
```

```
+ TDLR_ANGLES
+ TDLR_GAIN
+ TDLR_LOCK_TIME
+ TDLR_OFFSET
+ TE_DROP
+ TE_INIT
+ TE_MAX
+ TE_MIN
+ THETA1
+ THETA2
+ YE_MAX
+ YE_MIN
```

SENSOR_OUTPUT (store) =

```
*-----------------------------------------------------------------------
 Implementaion of the data elements in the data store SENSOR_OUTPUT
 must be arranged in the order specified below.
 -----------------------------------------------------------------------*


A_ACCELERATION
+ AR_ALTITUDE
+ ATMOSPHERIC_TEMP
+ G_ROTATION
+ TD_SENSED
+ TDLR_VELOCITY
```

SS_TEMP (data flow) =

```
*SS_TEMP*


---------

DESCRIPTION: solid state temperature data
USED IN:     TSP
UNITS:       none
RANGE:       [0, (2^15)-1]
DATA TYPE:   integer-2
ATTRIBUTE:   data
DATA STORE:  EXTERNAL
ACCURACY:    N/A
```

SUBFRAME_COUNTER (control flow, del) =

```
[ "1"
  | "2"
  | "3"
]
*SUBFRAME_COUNTER*

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

DESCRIPTION: Counter containing the number of the present subframe.
USED IN:     CP
```

UNITS: none
RANGE: [1,3]
DATA TYPE: Integer-2
ATTRIBUTE: data
DATA STORE: EXTERNAL
ACCURACY: N/A

T1 (data flow) =

T1

- - DESCRIPTION : lower ambient temperature calibration point for solid state temperature sensor USED IN : TSP degrees C RANGE : [- 50 , 250] DATA TYPE : real-8 ATTRIBUTE : data DATA STORE : RUN_PARAMETERS ACCURACY : N / A

T2 (data flow) =

T2

- - DESCRIPTION : upper ambient temperature calibration point for solid state temperature sensor USED IN : TSP degrees C RANGE : [- 50 , 250] DATA TYPE : real-8 ATTRIBUTE : data DATA STORE : RUN_PARAMETERS ACCURACY : N / A

T3 (data flow) =

T3

- - DESCRIPTION : lower amibent temperature calibration point for thermocouple pair temperature sensor USED IN UNITS : degrees C RANGE : [- 0 , 50] DATA TYPE : real-8 ATTRIBUTE : data DATA STORE : RUN_PARAMETERS ACCURACY : N / A

T4 (data flow) =

T4

- - DESCRIPTION : upper ambient temperature calibration point for thermocouple pair temperature sensor USED IN UNITS : degrees C RANGE : [- 0 , 50] DATA TYPE : real-8 ATTRIBUTE : data DATA STORE : RUN_PARAMETERS ACCURACY : N / A

TD_COUNTER (data flow) =

TD_COUNTER

DESCRIPTION: value returned by touch down sensor
USED IN: TDSP
UNITS: none
RANGE: [(-2^15), (2^15)-1]
DATA TYPE: integer-2
ATTRIBUTE: data
DATA STORE: EXTERNAL
ACCURACY: N/A

TD_SENSED (data flow) =

```
[ "0"
  | "1"
]
*TD_SENSED*

-----------

DESCRIPTION: flag indicating whether or not touch down has been sensed
USED IN:     CP, GP, TDSP
UNITS:       none
RANGE:       [0:touch_down_not_sensed, 1:touch_down_sensed]
DATA TYPE:   logical-1
ATTRIBUTE:   data
DATA STORE:  SENSOR_OUTPUT
ACCURACY:    N/A
```

TDLR_ANGLES (data flow) =

```
*TDLR_ANGLES*

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

DESCRIPTION: vector of doppler radar beam offset angles (i.e.,alpha ,beta ,gamma)
USED IN:     TDLRSP
UNITS:       radians
RANGE:       [0, PI/2) where PI = 3.141592653589793
DATA TYPE:   array (1..3) of real-8
ATTRIBUTE:   data
DATA STORE:  RUN_PARAMETERS
ACCURACY:    N/A
```

TDLR_COUNTER (data flow) =

```
*TDLR_COUNTER*

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

DESCRIPTION: value returned by doppler radar
USED IN:     TDLRSP
UNITS:       none
RANGE:       [0, (2^15)-1]
DATA TYPE:   array (1..4) of integer-2
ATTRIBUTE:   data
DATA STORE:  EXTERNAL
ACCURACY:    N/A
```

TDLR_GAIN (data flow) =

```
*TDLR_GAIN*
```

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

DESCRIPTION: gain in doppler radar beam
USED IN:     TDLRSP
UNITS:       meters/sec
RANGE:       [-1, 1]
DATA TYPE:   real-8
ATTRIBUTE:   data
DATA STORE:  RUN_PARAMETERS
ACCURACY:    N/A
```

TDLR_LOCK_TIME (data flow) =

```
*TDLR_LOCK_TIME*


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

DESCRIPTION: locking time of doppler radar beam
USED IN:     TDLRSP
UNITS:       seconds
RANGE:       [0, 60]
DATA TYPE:   real-8
ATTRIBUTE:   data
DATA STORE:  RUN_PARAMETERS
ACCURACY:    N/A
```

TDLR_OFFSET (data flow) =

```
*TDLR_OFFSET*


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

DESCRIPTION: offset in doppler radar beam
USED IN:     TDLRSP
UNITS:       meters/sec
RANGE:       [-100, 0]
DATA TYPE:   real-8
ATTRIBUTE:   data
DATA STORE:  RUN_PARAMETERS
ACCURACY:    N/A
```

TDLR_STATE (data flow) =

```
[ "0"
  | "1"
]
*TDLR_STATE*

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

DESCRIPTION: state of the touch down landing radar beams
USED IN:     CP, TDLRSP
```

```
UNITS:       none
RANGE:       [0:beam_unlocked, 1:beam_locked]
DATA TYPE:   array (1..4) logical-1
ATTRIBUTE:   data
DATA STORE:  GUIDANCE_STATE
ACCURACY:    N/A
```

TDLR_STATUS (data flow) =

```
[ "0"
  | "1"
]
*TDLR_STATUS*

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

DESCRIPTION: status of the doppler radar
USED IN:     CP, TDLRSP
UNITS:       none
RANGE:       [0:healthy, 1:failed]
DATA TYPE:   array (1..4) of logical-1
ATTRIBUTE:   data
DATA STORE:  GUIDANCE_STATE
ACCURACY:    N/A
```

TDLR_VELOCITY (data flow) =

```
*TDLR_VELOCITY*

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

DESCRIPTION: velocity as computed by the touch down landing radar
USED IN:     CP, GP, TDLRSP
UNITS:       meters/sec
RANGE:       [-100, 100]
DATA TYPE:   array (1..3, 0..4) of real-8
ATTRIBUTE:   data
DATA STORE:  SENSOR_OUTPUT
ACCURACY:    TBD
```

TDLRSP_EXT_IN (data flow) =

```
FRAME_COUNTER
+ TDLR_COUNTER
```

TDLRSP_GS_IN (data flow) =

```
FRAME_BEAM_UNLOCKED
+ K_MATRIX
+ TDLR_STATE
```

TDLRSP_GS_OUT (data flow) =

```
FRAME_BEAM_UNLOCKED
+ K_MATRIX
+ TDLR_STATE
+ TDLR_STATUS
```

TDLRSP_RP_IN (data flow) =

```
DELTA_T
+ TDLR_ANGLES
+ TDLR_GAIN
+ TDLR_LOCK_TIME
+ TDLR_OFFSET
```

TDS_STATUS (data flow) =

```
[ "0"
  | "1"
]
*TDS_STATUS*

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

DESCRIPTION: status of the touch down sensor
USED IN:     CP, GP, TDSP
UNITS:       none
RANGE:       [0:healthy, 1:failed]
DATA TYPE:   logical-1
ATTRIBUTE:   data
DATA STORE:  GUIDANCE_STATE
ACCURACY:    N/A
```

TE_DROP (data flow) =

```
*TE_DROP*


---------

DESCRIPTION: the axial thrust error when axial engines are warm and the
             velocity altitude contour has not been intersected.
USED IN:     AECLP
UNITS:       none
RANGE:       [-2, 2]
DATA TYPE:   real-8
ATTRIBUTE:   data
DATA STORE:  RUN_PARAMETERS
ACCURACY:    N/A
```

TE_INIT (data flow) =

```
*TE_INIT*


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

```
DESCRIPTION: the axial thrust error when the axial engines are cold
USED IN:     AECLP
UNITS:       none
RANGE:       [-2, 2]
DATA TYPE:   real-8
ATTRIBUTE:   data
DATA STORE:  RUN_PARAMETERS
ACCURACY:    N/A
```

TE_INTEGRAL (data flow) =

```
*TE_INTEGRAL*


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

DESCRIPTION: integral portion of thrust error equation
USED IN:     AECLP, CP, GP
UNITS:       meters
RANGE:       [-100, 100]
DATA TYPE:   real-8
ATTRIBUTE:   data
DATA STORE:  GUIDANCE_STATE
ACCURACY:    TBD
```

TE_LIMIT (data flow) =

```
*TE_LIMIT*


----------

DESCRIPTION: limiting thrust error
USED IN:     AECLP
UNITS:       none
RANGE:       [-100, 100]
DATA TYPE:   real-8
ATTRIBUTE:   data
DATA STORE:  GUIDANCE_STATE
ACCURACY:    TBD
```

TE_MAX (data flow) =

```
*TE_MAX*


--------

DESCRIPTION: maximum thrust error tolerable
USED IN:     AECLP
UNITS:       none
RANGE:       [-2, 2]
DATA TYPE:   array (1..2) of real-8
ATTRIBUTE:   data
DATA STORE:  RUN_PARAMETERS
ACCURACY:    N/A
```

TE_MIN (data flow) =

TE_MIN

```
DESCRIPTION: minimum thrust error tolerable
USED IN:     AECLP
UNITS:       none
RANGE:       [-2, 2]
DATA TYPE:   array (1..2) of real-8
ATTRIBUTE:   data
DATA STORE:  RUN_PARAMETERS
ACCURACY:    N/A
```

THERMO_TEMP (data flow) =

THERMO_TEMP

```
DESCRIPTION: thermocouple pair temperature
USED IN:     TSP
UNITS:       none
RANGE:       [0,(2^15)-1]
DATA TYPE:   integer-2
ATTRIBUTE:   data
DATA STORE:  EXTERNAL
ACCURACY:    N/A
```

THETA (data flow) =

THETA

- - - - DESCRIPTION : roll angle USED IN : RECLP UNITS : radians RANGE : [- I , PI
] where PI = 3 .141592653589793 DATA
real-8 ATTRIBUTE : data DATA STORE : GUIDANCE_STATE ACCURACY : TBD

THETA1 (data flow) =

THETA1

```
DESCRIPTION: pulse angle boundary
USED IN:     RECLP
UNITS:       radians
RANGE:       [0, 0.05]
DATA TYPE:   real-8
ATTRIBUTE:   data
DATA STORE:  RUN_PARAMETERS
ACCURACY:    N/A
```

THETA2 (data flow) =

```
*THETA2*


--------

DESCRIPTION: pulse angle boundary
USED IN:     RECLP
UNITS:       radians
RANGE:       [0, 0.05]
DATA TYPE:   real-8
ATTRIBUTE:   data
DATA STORE:  RUN_PARAMETERS
ACCURACY:    N/A
```

TS_STATUS (data flow) =

```
[ "0"
  | "1"
]
*TS_STATUS*

-----------

DESCRIPTION: status of the temperature sensors in solid state, then thermocouple pair order
USED IN:     CP, TSP
UNITS:       none
RANGE:       [0:healthy, 1:failed]
DATA TYPE:   array (1..2) of logical-1
ATTRIBUTE:   data
DATA STORE:  GUIDANCE_STATE
ACCURACY:    N/A
```

TSP_EXT_IN (data flow) =

```
SS_TEMP
+ THERMO_TEMP
```

TSP_RP_IN (data flow) =

```
M1
+ M2
+ M3
+ M4
+ T1
+ T2
+ T3
+ T4
```

VELOCITY_ERROR (data flow) =

```
*VELOCITY_ERROR*
```

DESCRIPTION: distance from velocity-altitude contour (difference in velocities from actual to desired on contour)
USED IN: AECLP, CP, GP
UNITS: meters/sec
RANGE: [-300, 20]
DATA TYPE: real-8
ATTRIBUTE: data
DATA STORE: GUIDANCE_STATE
ACCURACY: TBD

YE_INTEGRAL (data flow) =

YE_INTEGRAL

DESCRIPTION: integral portion of yaw error equation
USED IN: AECLP, CP
UNITS: meters
RANGE: [-100, 100]
DATA TYPE: real-8
ATTRIBUTE: data
DATA STORE: GUIDANCE_STATE
ACCURACY: TBD

YE_MAX (data flow) =

YE_MAX

DESCRIPTION: maximum yaw error tolerable
USED IN: AECLP
UNITS: none
RANGE: [-1, 1]
DATA TYPE: array (1..2) of real-8
ATTRIBUTE: data
DATA STORE: RUN_PARAMETERS
ACCURACY: N/A

YE_MIN (data flow) =

YE_MIN

DESCRIPTION: minimum yaw error tolerable
USED IN: AECLP
UNITS: none

RANGE:	[-1, 1]
DATA TYPE:	array (1..2) of real-8
ATTRIBUTE:	data
DATA STORE:	RUN_PARAMETERS
ACCURACY:	N/A

Appendix C: Source Code for the Pluto Implementation of the Guidance and Control Software

Author: Philip Morris, Lockheed Martin Engineering and Sciences Corp.

This document was produced as part of Guidance and Control Software (GCS) Project conducted at NASA Langley Research Center. Although some of the requirements for the Guidance and Control Software application were derived from the NASA Viking Mission to Mars, this document does not contain data from an actual NASA mission.

```
***********************************************************************
*  Module:        AECLP.FOR
*  Facility:      Pluto
*  P-Spec:        3.2
*  Abstract:
*         This module contains the implementation of the functional
*         requirements for AECLP.
*
*  List of Routines:
*         subroutine AECLP
***********************************************************************

***********************************************************************
*  Title: AECLP
*  Facility:      Pluto
*  Abstract:
*         1) determine the current operational status of the axial engines.
*         2) generate the appropriate axial engine commands.
*
*  Arguments:   None
*
*  Revision History:
*         v0    15-sep-1994  Rob Angellatta (RKA) Original.
*      v1    30-Nov-1994  Philip Morris (PEM)
***********************************************************************

      subroutine AECLP

      implicit none

*** include the global common stores ***

      include 'external.for'
      include 'guidance_state.for'
      include 'sensor_output.for'
      include 'run_parameters.for'

*** include constant definitions ***

      include 'constants.for'

*** declare local variables ***

      real*8              q_over_omega
      real*8              pitch_error
      real*8              pitch_error_limit
      real*8              yaw_error
      real*8              yaw_error_limit
      real*8              thrust_error

      integer*4   i
```

```
***********************************************************************
*  1) Determine the current operational status of the axial engines.
***********************************************************************

      AE_STATUS = K$HEALTHY

***********************************************************************
*  2) Generate the appropriate axial engine commands.
*
*  Determine if the axial engines are on.  If the axial engines
*  are "off" (value 0) then the axial engine commands are "0".
*  Otherwise, further processing is required in order to determine
*  the appropriate axial engine commands.
************************************************************************

      if (AE_SWITCH .EQ. K$AXIAL_ENGINES_ARE_OFF) then
        AE_CMD(1) = 0
        AE_CMD(2) = 0
        AE_CMD(3) = 0
      else

***********************************************************************
* The axial engines are "on" so further processing is required.
*
* 2A) determine the axial engine temperature.
***********************************************************************

*** range check the current altitude ***

        call RANGE_CHECK(GP_ALTITUDE(0), K$GP_ALTITUDE$LB,
     &                  K$GP_ALTITUDE$UB, 'AECLP', K$GP_ALTITUDE$NAME)

***********************************************************************
* The three possible engine temperature states are: "Cold" (value 0),
* "Warming up" (value 1), and "Hot" (value 2).  The current temperature
* of the axial engines is stored in the data element AE_TEMP.
************************************************************************

        if (GP_ALTITUDE(0) .LE. ENGINES_ON_ALTITUDE) then

          if (AE_TEMP .EQ. K$COLD) then

            if ((FRAME_COUNTER - FRAME_ENGINES_IGNITED) *
     &           DELTA_T .LT. FULL_UP_TIME) then
              AE_TEMP = K$WARMING_UP
            end if

          else if (AE_TEMP .EQ. K$WARMING_UP) then
            if ((FRAME_COUNTER - FRAME_ENGINES_IGNITED) *
     &           DELTA_T .GE. FULL_UP_TIME) then
```

```
            AE_TEMP = K$HOT
          end if
        end if
      end if

*********************************************************************
* 2B) Compute the pitch error limit.
*********************************************************************

*** range check the pitch error integral ***

      call RANGE_CHECK(PE_INTEGRAL, K$PE_INTEGRAL$LB,
   &                 K$PE_INTEGRAL$UB, 'AECLP', K$PE_INTEGRAL$NAME)

*** range check the x-axis roll rate ***

      call RANGE_CHECK(GP_VELOCITY(1, 0), K$GP_VELOCITY$LB,
   &                 K$GP_VELOCITY$UB, 'AECLP', K$GP_VELOCITY$NAME)

*** range check the z-axis roll rate ***

      call RANGE_CHECK(GP_VELOCITY(3, 0), K$GP_VELOCITY$LB,
   &            K$GP_VELOCITY$UB, 'AECLP', K$GP_VELOCITY$NAME)

*** check for potential divide by zero condition ***

      call ZERO_CHECK(GP_VELOCITY(1, 0), 'AECLP')

*** compute the current value for PE_INTEGRAL ***

      PE_INTEGRAL = PE_INTEGRAL +
   &       (GP_VELOCITY(3, 0) / ABS(GP_VELOCITY(1, 0))) * DELTA_T

*** range check the pitch error integral (again) ***

      call RANGE_CHECK(PE_INTEGRAL, K$PE_INTEGRAL$LB,
   &       K$PE_INTEGRAL$UB, 'AECLP', K$PE_INTEGRAL$NAME)

*** range check the pitch rotational displacement ***

      call RANGE_CHECK(GP_ROTATION(3, 1), K$GP_ROTATION$LB,
   &                 K$GP_ROTATION$UB, 'AECLP', K$GP_ROTATION$NAME)

*** compute the pitch error limit ***

      pitch_error_limit = GQ(CL) * GP_ROTATION(3, 1) +
   &        GW(CL) * (GP_VELOCITY(3, 0) / ABS(GP_VELOCITY(1, 0))) +
   &        GWI(CL) * PE_INTEGRAL

      if (pitch_error_limit .LT. PE_MIN(CL)) then
        pitch_error_limit = PE_MIN(CL)
```

```
          else if (pitch_error_limit .GT. PE_MAX(CL)) then
            pitch_error_limit = PE_MAX(CL)

          end if

*****************************************************************
* 2C) Compute the yaw error limit.
*****************************************************************

*** range check the yaw error integral ***

          call RANGE_CHECK(YE_INTEGRAL, K$YE_INTEGRAL$LB,
     &                K$YE_INTEGRAL$UB, 'AECLP', K$YE_INTEGRAL$NAME)

*** range check the y-axis roll rate ***

          call RANGE_CHECK(GP_VELOCITY(2, 0), K$GP_VELOCITY$LB,
     &                K$GP_VELOCITY$UB, 'AECLP', K$GP_VELOCITY$NAME)

*** check for potential divide by zero condition ***

          call ZERO_CHECK(GP_VELOCITY(1, 0), 'AECLP')

*** Compute the current value for YE_INTEGRAL ***

          YE_INTEGRAL = YE_INTEGRAL +
     &         (GP_VELOCITY(2, 0) / ABS(GP_VELOCITY(1, 0))) * DELTA_T

*** range check the yaw error integral (again) ***

          call RANGE_CHECK(YE_INTEGRAL, K$YE_INTEGRAL$LB,
     &                K$YE_INTEGRAL$UB, 'AECLP', K$YE_INTEGRAL$NAME)

*** range check the yaw rotational displacement ***

          call RANGE_CHECK(GP_ROTATION(1, 2), K$GP_ROTATION$LB,
     &                K$GP_ROTATION$UB, 'AECLP', K$GP_ROTATION$NAME)

*** compute the yaw error limit ***

          yaw_error_limit = -GR(CL) * GP_ROTATION(1, 2) +
     &        GV(CL) * (GP_VELOCITY(2, 0) / ABS(GP_VELOCITY(1, 0))) +
     &        GVI(CL) * YE_INTEGRAL

          if (yaw_error_limit .LT. YE_MIN(CL)) then
            yaw_error_limit = YE_MIN(CL)

          else if (yaw_error_limit .GT. YE_MAX(CL)) then
            yaw_error_limit = YE_MAX(CL)
```

```
      end if

***********************************************************************
* 2D) Compute the thrust limiting error.
***********************************************************************

      if (CONTOUR_CROSSED .EQ. K$CONTOUR_CROSSED) then

*** range check the thrust error integral ***

        call RANGE_CHECK(TE_INTEGRAL, K$TE_INTEGRAL$LB,
     &               K$TE_INTEGRAL$UB, 'AECLP', K$TE_INTEGRAL$NAME)

*** range check the velocity error ***

        call RANGE_CHECK(VELOCITY_ERROR, K$VELOCITY_ERROR$LB,
     &               K$VELOCITY_ERROR$UB, 'AECLP', K$VELOCITY_ERROR$NAME)

*** Compute the current value for TE_INTEGRAL ***

        TE_INTEGRAL = TE_INTEGRAL + VELOCITY_ERROR * DELTA_T

*** range check the thrust error integral (again) ***

        call RANGE_CHECK(TE_INTEGRAL, K$TE_INTEGRAL$LB,
     &               K$TE_INTEGRAL$UB, 'AECLP', K$TE_INTEGRAL$NAME)

*** range check the attitude component ***

        call RANGE_CHECK(GP_ATTITUDE(1, 3, 0), K$GP_ATTITUDE$LB,
     &               K$GP_ATTITUDE$UB, 'AECLP', K$GP_ATTITUDE$NAME)

*** range check the x-axis acceleration ***

        call RANGE_CHECK(A_ACCELERATION(1, 0),
     &          K$A_ACCELERATION$LB,
     &               K$A_ACCELERATION$UB, 'AECLP', K$A_ACCELERATION$NAME)

*** range check the thrust error limit ***

        call RANGE_CHECK(TE_LIMIT, K$TE_LIMIT$LB,
     &               K$TE_LIMIT$UB, 'AECLP', K$TE_LIMIT$NAME)

***
*  v1 Changes for AR#23. Item 8. Added check for zero.
***
       call ZERO_CHECK(OMEGA, 'AECLP')
           q_over_omega = (GA * (-GAX * (A_ACCELERATION(1,0) +
     &             GRAVITY * GP_ATTITUDE(1,3,0)) + GVE *
     &             VELOCITY_ERROR + GVEI(CL) * TE_INTEGRAL)) /
     &             OMEGA
```

```
***
*  v1 Changes for AR#23. End Change.
***
          TE_LIMIT = q_over_omega +
    &           (TE_LIMIT - q_over_omega) * EXP(-OMEGA * DELTA_T)

*** range check the current value of for the thrust error limit ***

          call RANGE_CHECK(TE_LIMIT, K$TE_LIMIT$LB,
    &               K$TE_LIMIT$UB, 'AECLP', K$TE_LIMIT$NAME)

          if (TE_LIMIT .LT. TE_MIN(CL)) then
            TE_LIMIT= TE_MIN(CL)

          else if (TE_LIMIT .GT. TE_MAX(CL)) then
            TE_LIMIT = TE_MAX(CL)
          end if
        end if

************************************************************************
* 2E) Compute the pitch, yaw and thrust errors.
************************************************************************

*** Note, to get here (AE_SWITCH = K$AXIAL_ENGINES_ON) ***

        if (CHUTE_RELEASED .EQ. K$CHUTE_RELEASED) then

          if (CONTOUR_CROSSED .EQ. K$CONTOUR_NOT_CROSSED) then
            pitch_error  = pitch_error_limit
            yaw_error    = yaw_error_limit
            thrust_error =  TE_DROP

          else
            pitch_error  = pitch_error_limit
            yaw_error    = yaw_error_limit
            thrust_error =  TE_LIMIT
          end if

        else
***                                  "Chute Attached" ***
          pitch_error  =  GQ(CL) * GP_ROTATION(3, 1)
          yaw_error    = -GR(CL) * GP_ROTATION(1, 2)
          thrust_error =  TE_INIT

        end if

************************************************************************
* 2F) Compute the axial engine value settings.
************************************************************************
```

```
      INTERNAL_CMD(1) = GP1 * pitch_error + thrust_error
      INTERNAL_CMD(2) = GP2 * pitch_error -
   &                    GPY * yaw_error + thrust_error
      INTERNAL_CMD(3) = GP2 * pitch_error +
   &                    GPY * yaw_error + thrust_error

***********************************************************************
* 2G) Convert the axial engine value settings to engine commands.
***********************************************************************

*** range check the internal command ***

      call RANGE_CHECK(INTERNAL_CMD(1), K$INTERNAL_CMD$LB,
   &              K$INTERNAL_CMD$UB, 'AECLP', K$INTERNAL_CMD$NAME)

      call RANGE_CHECK(INTERNAL_CMD(2), K$INTERNAL_CMD$LB,
   &              K$INTERNAL_CMD$UB, 'AECLP', K$INTERNAL_CMD$NAME)

      call RANGE_CHECK(INTERNAL_CMD(3), K$INTERNAL_CMD$LB,
   &              K$INTERNAL_CMD$UB, 'AECLP', K$INTERNAL_CMD$NAME)

*** do the convertion for each engine ***

      do i = 1,3

        if (INTERNAL_CMD(i) .LT. 0) then
          AE_CMD(i) = 0

***
*  v1 Changes for AR#23. Item 2. Added D0 to 0.5
***
        else if (INTERNAL_CMD(i) .LE. 1) then
          AE_CMD(i) = INT(127 * INTERNAL_CMD(i) + 0.5D0)

***
*  v1 Changes for AR#23. End Change.
***
        else
          AE_CMD(i) = 127

        end if
      end do

     end if

     return
     end

***** end of module AECLP.FOR *******************************************
```

```
**********************************************************************
*  Module:      ARSP.FOR
*  Facility:    Pluto
*  P-Spec:      1.2
*  Abstract:
*        This module contains the implementation of the functional
*        requirements for ARSP.
*
*  List of Routines:
*        subroutine ARSP
**********************************************************************

**********************************************************************
*  Title: ARSP
*  Facility:    Pluto
*  Abstract:
*        1) maintain the history of the altitude and altimeter sensor data
*           elements
*        2) determine the operational status of the altimeter radar sensor
*        3) Report the current altitude.
*
*  Arguments:   None
*
*  Revision History:
*        v0   15-sep-1994  Rob Angellatta (RKA) Original.
*        v1   10-JAN-1995  Philip Morris (PEM)
**********************************************************************
      subroutine ARSP

      implicit none

*** include the global common stores ***

      include 'external.for'
      include 'guidance_state.for'
      include 'sensor_output.for'
      include 'run_parameters.for'

*** include constant definitions ***

      include 'constants.for'

************************************************************************
*  1) Maintain the history of the altitude and the sensor status by
*  "rotating variables."
************************************************************************
      AR_ALTITUDE(4) = AR_ALTITUDE(3)
      AR_ALTITUDE(3) = AR_ALTITUDE(2)
      AR_ALTITUDE(2) = AR_ALTITUDE(1)
      AR_ALTITUDE(1) = AR_ALTITUDE(0)
```

```
      AR_STATUS(4) = AR_STATUS(3)
      AR_STATUS(3) = AR_STATUS(2)
      AR_STATUS(2) = AR_STATUS(1)
      AR_STATUS(1) = AR_STATUS(0)

      K_ALT(4) = K_ALT(3)
      K_ALT(3) = K_ALT(2)
      K_ALT(2) = K_ALT(1)
      K_ALT(1) = K_ALT(0)

************************************************************************
*  2) determine the operational status of the altimeter radar sensor.
*
*  3) There are three methods for determining the altitude.
*
*  A) compute altitude from the sensor measurement.
*
*  B) estimate altitude by fitting a third-order polynomial to the
*     altitude history data values.
*
*  C) report the altitude as the most recently reported altitude.
************************************************************************

************************************************************************
* If an echo has been received, then the lower order fifteen bits of
* AR_COUNTER contain the raw sensor measurement, and the upper bit of
* AR_COUNTER will be clear (ie: 0).  When an echo has not been received,
* the AR_COUNTER will contain 16 set bits (ie: 0xFFFF).
*
* The data type of AR_COUNTER is integer*2 and the valid value range
* is specified as (-1, 32767). This implementation assumes that integer
* values are represented by twos complement.  Thus, when an echo has not
* been recieved, the AR_COUNTER will contain the value of -1.  Similarly,
* when an echo has been received, AR_COUNTER will contain a non-negative
* value.
************************************************************************

      if  (AR_COUNTER .NE. -1) then
        AR_STATUS(0) = K$HEALTHY
        K_ALT(0)         = 1

*** A) compute the altitude from the sensor measurement ***
***
*         v1 Changes for PR#24. Item 8. Changed 3E08 to 3D08.
***
*        AR_ALTITUDE(0) = (AR_COUNTER * 3E08) / (2.0 * AR_FREQUENCY)
        AR_ALTITUDE(0) = (AR_COUNTER * 3D08) / (2.0 * AR_FREQUENCY)
***
*         v1 Changes for PR#24. End Change.
***
```

```
      else
*                                      no echo received
         AR_STATUS(0) = K$FAILED

*** if at least one of the history sensor status is "failed" ***

         if ((AR_STATUS(1) .EQ. K$FAILED) .OR.
   &         (AR_STATUS(2) .EQ. K$FAILED) .OR.
   &         (AR_STATUS(3) .EQ. K$FAILED) .OR.
   &         (AR_STATUS(4) .EQ. K$FAILED))     then

            K_ALT(0)       = 0

*** C) return previously computed value ***

*** range check the altitude ****

            call RANGE_CHECK(AR_ALTITUDE(1),K$AR_ALTITUDE$LB,
   &                 K$AR_ALTITUDE$UB,'ARSP', K$AR_ALTITUDE$NAME)

***          the value stored in AR_ALTITUDE(1) is aready
***          stored in AR_ALTITUDE(0)!

         else
*                            all sensor status histories are "healthy"

*** B) extrapolate the altitude ***

            K_ALT(0) = 1

*** range check the altitude ****

***
*         v1 Changes for PR#24. Extra. Changed 'ASP' to 'ARSP'.
***
            call RANGE_CHECK(AR_ALTITUDE(1),K$AR_ALTITUDE$LB,
   &                         K$AR_ALTITUDE$UB,'ARSP', K$AR_ALTITUDE$NAME)

            call RANGE_CHECK(AR_ALTITUDE(2),K$AR_ALTITUDE$LB,
   &                         K$AR_ALTITUDE$UB,'ARSP', K$AR_ALTITUDE$NAME)

            call RANGE_CHECK(AR_ALTITUDE(3),K$AR_ALTITUDE$LB,
   &                         K$AR_ALTITUDE$UB,'ARSP', K$AR_ALTITUDE$NAME)

            call RANGE_CHECK(AR_ALTITUDE(4),K$AR_ALTITUDE$LB,
   &                         K$AR_ALTITUDE$UB,'ARSP', K$AR_ALTITUDE$NAME)

            AR_ALTITUDE(0) = 4*AR_ALTITUDE(1) - 6*AR_ALTITUDE(2) +
   &                 4*AR_ALTITUDE(3) - AR_ALTITUDE(4)

***
```

```
*         v1 Changes for PR#24. End Change.
***
        end if
      end if

      return
      end

***** end of module arsp.for ********************************************
```

```
**********************************************************************
*  Module:        ASP.FOR
*  Facility:      Pluto
*  P-Spec:     1.3
*  Abstract:
*           This module contains the implementation of the functional
*           requirements for ASP.
*
*  List of Routines:
*           subroutine ASP
**********************************************************************

**********************************************************************
*  Title: ASP
*  Facility:      Pluto
*  Abstract:
*           1) maintaining the history of the accelerations and accelerometer
*             sensor statuses
*           2) determining the operational status of the accelerometer sensors
*           3) Reporting the current vehicle accelerations along each of the
*             vehicle's three axes.
*
*  Arguments:   None
*
*  Revision History:
*           v0    15-sep-1994  Rob Angellatta (RKA) Original.
*           v1    16-MAR-1995  Philip Morris (PEM)
**********************************************************************

      subroutine ASP

      implicit none

*** define local constants ***

*** include the global common stores ***

      include 'external.for'
      include 'guidance_state.for'
      include 'sensor_output.for'
      include 'run_parameters.for'

*** include constant definitions ***

      include 'constants.for'

*** declare local variables ***

      real*8              temp
      real*8              accel_m(3)
```

```
      integer*4   i

      real*8            mean
      real*8            sd

***********************************************************************
*  1) Maintain the history of the vehicle accelerations and
*  accelerometer sensor status by "rotating variables."
***********************************************************************

      A_ACCELERATION(1, 4) = A_ACCELERATION(1, 3)
      A_ACCELERATION(1, 3) = A_ACCELERATION(1, 2)
      A_ACCELERATION(1, 2) = A_ACCELERATION(1, 1)
      A_ACCELERATION(1, 1) = A_ACCELERATION(1, 0)

      A_ACCELERATION(2, 4) = A_ACCELERATION(2, 3)
      A_ACCELERATION(2, 3) = A_ACCELERATION(2, 2)
      A_ACCELERATION(2, 2) = A_ACCELERATION(2, 1)
      A_ACCELERATION(2, 1) = A_ACCELERATION(2, 0)

      A_ACCELERATION(3, 4) = A_ACCELERATION(3, 3)
      A_ACCELERATION(3, 3) = A_ACCELERATION(3, 2)
      A_ACCELERATION(3, 2) = A_ACCELERATION(3, 1)
      A_ACCELERATION(3, 1) = A_ACCELERATION(3, 0)

      A_STATUS(1, 3) = A_STATUS(1, 2)
      A_STATUS(1, 2) = A_STATUS(1, 1)
      A_STATUS(1, 1) = A_STATUS(1, 0)

      A_STATUS(2, 3) = A_STATUS(2, 2)
      A_STATUS(2, 2) = A_STATUS(2, 1)
      A_STATUS(2, 1) = A_STATUS(2, 0)

      A_STATUS(3, 3) = A_STATUS(3, 2)
      A_STATUS(3, 2) = A_STATUS(3, 1)
      A_STATUS(3, 1) = A_STATUS(3, 0)

***********************************************************************
*  2) and 3), determine the operational status and the vehicle
*  accelerations for each axis.
***********************************************************************

*** range check the atmospheric temperature ****

      call RANGE_CHECK(ATMOSPHERIC_TEMP,K$ATMOSPHERIC_TEMP$LB,
     &            K$ATMOSPHERIC_TEMP$UB,'ASP', K$ATMOSPHERIC_TEMP$NAME)

*** compute the preliminary value for the accelerations ***

      temp = (G1 * ATMOSPHERIC_TEMP) + (G2 * ATMOSPHERIC_TEMP**2)
```

```
      accel_m(1) = A_BIAS(1)+ (A_GAIN_0(1) + temp) * A_COUNTER(1)
      accel_m(2) = A_BIAS(2)+ (A_GAIN_0(2) + temp) * A_COUNTER(2)
      accel_m(3) = A_BIAS(3)+ (A_GAIN_0(3) + temp) * A_COUNTER(3)

      A_ACCELERATION(1, 0) = ALPHA_MATRIX(1, 1) * accel_m(1) +
     &                       ALPHA_MATRIX(1, 2) * accel_m(2) +
     &                       ALPHA_MATRIX(1, 3) * accel_m(3)

      A_ACCELERATION(2, 0) = ALPHA_MATRIX(2, 1) * accel_m(1) +
     &                       ALPHA_MATRIX(2, 2) * accel_m(2) +
     &                       ALPHA_MATRIX(2, 3) * accel_m(3)

      A_ACCELERATION(3, 0) = ALPHA_MATRIX(3, 1) * accel_m(1) +
     &                       ALPHA_MATRIX(3, 2) * accel_m(2) +
     &                       ALPHA_MATRIX(3, 3) * accel_m(3)

*************************************************************************
*  Determine whether or not the preliminary values for the
*  accelerations are reasonable. The preliminary value for an
*  acceleration is deemed reasonable: 1) if it differs from the mean
*  of the previous three measurements by not more than A_SCALE
*  standard deviations; 2) when any of the three accelerometer
*  history statuses is "unhealthy" (value 1). If a preliminary
*  acceleration value is found to be reasonable,
*  then it is reported as the acceleration for it's axis.  If a
*  preliminary value is not found to be reasonable, then the
*  mean of the previous three measurements is reported as the
*  acceleration for that axis.
*
*  The current value for the sensor status is determined directly
*  from the reasonableness of the value of the preliminary
*  accleration.  If the preliminary acceleration is reasonable, the
*  sensor status is deemed "healthy " (value 0).  If the preliminary
*  acceleration is not reasonable, the sensor status is deemed
*  "unhealthy."
*************************************************************************

      do i=1,3

***
*   v1 PR#27 Item 1.  Adjust mean calculation.
***
*         if ((A_STATUS(i, 1) .EQ. K$UNHEALTHY) .OR.
*    &         (A_STATUS(i, 2) .EQ. K$UNHEALTHY) .OR.
*    &         (A_STATUS(i, 3) .EQ. K$UNHEALTHY)) then
*
*** one or more history statuses are "unhealthy" ***
*
*            A_STATUS(i, 0) = K$HEALTHY
*
*         else
```

```
*                                    all history status are "healthy"
          A_STATUS(i, 0) = K$HEALTHY

          if ((A_STATUS(i, 1) .EQ. K$HEALTHY) .AND.
    &        (A_STATUS(i, 2) .EQ. K$HEALTHY) .AND.
    &        (A_STATUS(i, 3) .EQ. K$HEALTHY)) then

            if ((A_ACCELERATION(i,1) .NE. A_ACCELERATION(i,2)) .OR.
    &           (A_ACCELERATION(i,1) .NE. A_ACCELERATION(i,3))) then
***
*   v1 PR#27 End Change.
***

*** compute the mean of the previous three values ***

*** range check the acceleration values ****

             call RANGE_CHECK(A_ACCELERATION(i, 1),K$A_ACCELERATION$LB,
    &                K$A_ACCELERATION$UB,'ASP', K$A_ACCELERATION$NAME)

             call RANGE_CHECK(A_ACCELERATION(i, 2),K$A_ACCELERATION$LB,
    &                K$A_ACCELERATION$UB,'ASP', K$A_ACCELERATION$NAME)

             call RANGE_CHECK(A_ACCELERATION(i, 3),K$A_ACCELERATION$LB,
    &                K$A_ACCELERATION$UB,'ASP', K$A_ACCELERATION$NAME)

***
*   v1 PR#27 Item 2.  Adjust Standard deviation calculation.
***

*** compute the standard deviation ***

             mean = (A_ACCELERATION(i, 1) +
    &               A_ACCELERATION(i, 2) +
    &               A_ACCELERATION(i, 3)) / 3.0D0

             temp = ((A_ACCELERATION(i,1) - mean)**2 +
    &               (A_ACCELERATION(i,2) - mean)**2 +
    &               (A_ACCELERATION(i,3) - mean)**2   )
    &              / 3.0D0

             sd = SQRT(temp)

             temp = ABS(mean - A_ACCELERATION(i, 0))
             if (temp .GT. A_SCALE * sd) then
               A_ACCELERATION(i, 0) = mean
               A_STATUS(i, 0) = K$UNHEALTHY
             end if

            end if     ! close if block for A_ACCELERATION
          end if    ! close if block for A_STATUS
```

```
      end do

**********************************************************************
*  Note, numerical inaccuracies are inherent in the digital
*  representation of real numbers. Computing the variance can
*  potentially result in a small negative value, when the previous
*  accelerations have identical values. Therefore, the following
*  algorithm specifies a seperate case for computing the standard
*  deviation when the previous accelerations have identical values.
**********************************************************************
*
*          if ((A_ACCELERATION(i,1) .EQ. A_ACCELERATION(i,2)) .AND.
*   &          (A_ACCELERATION(i,1) .EQ. A_ACCELERATION(i,3))) then
*            sd = 0.0
*          else
*            temp = ((A_ACCELERATION(i,1)**2 +
*   &                 A_ACCELERATION(i,2)**2 +
*   &                 A_ACCELERATION(i,3)**2) / 3.0) - mean**2
*
*            call NEG_VALUE_CHECK(temp, 'ASP')
*
*            sd = SQRT(temp)
*
*          end if
*
*          if (ABS(mean - A_ACCELERATION(i, 0)) .GT.
*   &                              A_SCALE * sd) then
*            A_ACCELERATION(i, 0) = mean
*            A_STATUS(i, 0) = K$UNHEALTHY
*          else
*            A_STATUS(i, 0) = K$HEALTHY
*          end if
*
*        end if
*
*     end do
***
*  v1 PR#27 End Change.
***
      return
      end

***** end of module asp.for *******************************************
```

```
************************************************************************
*  Module:      CLPSF.FOR
*  Facility:    Pluto
*  P-Spec:      3.0
*  Abstract:
*        This module contains the entry for the control law processing
*        subframe.
*
*  List of Routines:
*        subroutine CLPSF
************************************************************************

************************************************************************
*  Title: CLPSF
*  Facility:    Pluto
*  Abstract:
*        This routine provides control of the Control Law Processing
*        SubFrame processing.
*
*  Arguments:   None
*
*  Revision History:
*        v0   15-sep-1994  Rob Angellatta (RKA) Original.
*        v1   15-Feb-1995  Philip Morris (PEM)
************************************************************************

      subroutine CLPSF

      implicit none

*** execution begins here ***

      call GCS_SIM_RENDEZVOUS
***
*  v1 Changes for AR#26. Item 1. Correct Spelling
***
*      call AELCP
      call AECLP
***
*  v1 Changes for AR#26. Item 1. End Cahnge.
***
      call RECLP
      call CRCP
      call CP

      return
      end

***** end of module CLPSF.FOR *****************************************
```

```
***************************************************************************
*  Module:        CONSTANTS.FOR
*  Facility:      Pluto
*  Abstract:
*          This module contains the constants used in Pluto.  The constants
*          consist of values for enumerated types, upper and lower
*          bounds, and error reporting text.
*
*  Revision History:
*          v0    15-sep-1994  Rob Angellatta (RKA) Original.
*       v1    30-Nov-1994  Philip Morris (PEM)
*       v2    10-Jan-1995  Philip Morris (PEM)
***************************************************************************

*** define constant values (for enumerated types) **********************

*** AE_TEMP values ***

***
*  v1 Changes for AR#23. Item 4. Changed logical*1 to integer*2
***
      integer*2   K$COLD
      parameter  (K$COLD                  = 0)

      integer*2   K$WARMING_UP
      parameter  (K$WARMING_UP     = 1)

      integer*2   K$HOT
      parameter  (K$HOT                   = 2)

***
*  v1 Changes for AR#23. End Change.
***

*** AE_SWITCH values ***

      logical*1   K$AXIAL_ENGINES_ARE_OFF
      parameter  (K$AXIAL_ENGINES_ARE_OFF= 0)

      logical*1   K$AXIAL_ENGINES_ARE_ON
      parameter  (K$AXIAL_ENGINES_ARE_ON         = 1)

*** CHUTE_RELEASED values ***

      logical*1   K$CHUTE_ATTACHED
      parameter  (K$CHUTE_ATTACHED        = 0)

      logical*1   K$CHUTE_RELEASED
      parameter  (K$CHUTE_RELEASED         = 1)

*** CL values ***
```

```
      integer*2   K$FIRST
      parameter  (K$FIRST        = 1)

      integer*2   K$SECOND
      parameter  (K$SECOND    = 2)

*** CONTOUR_CROSSED values ***

      logical*1   K$CONTOUR_NOT_CROSSED
      parameter  (K$CONTOUR_NOT_CROSSED         = 0)

      logical*1   K$CONTOUR_CROSSED
      parameter  (K$CONTOUR_CROSSED       = 1)

*** RE_CMD values ***

      integer*2   K$CCW
      parameter  (K$CCW                    = 0)

      integer*2   K$CW
      parameter  (K$CW            = 1)

      integer*2   K$OFF
      parameter  (K$OFF                    = 0)

      integer*2   K$MINIMUM
      parameter  (K$MINIMUM = 2)

      integer*2   K$INTERMEDIATE
      parameter  (K$INTERMEDIATE   = 4)

      integer*2   K$MAXIMUM
      parameter  (K$MAXIMUM           = 6)

*** RE_SWITCH values ***

      logical*1   K$ROLL_ENGINES_ARE_OFF
      parameter  (K$ROLL_ENGINES_ARE_OFF          = 0)

      logical*1   K$ROLL_ENGINES_ARE_ON
      parameter  (K$ROLL_ENGINES_ARE_ON           = 1)

*** Sensor statuses ***

      logical*1   K$HEALTHY
      parameter  (K$HEALTHY = 0)

      logical*1   K$UNHEALTHY
      parameter  (K$UNHEALTHY        = 1)
```

```
      logical*1   K$FAILED
      parameter  (K$FAILED      = 1)

*** TD_SENSED values ***

      logical*1   K$TOUCH_DOWN_NOT_SENSED
      parameter  (K$TOUCH_DOWN_NOT_SENSED = 0)

      logical*1   K$TOUCH_DOWN_SENSED
      parameter  (K$TOUCH_DOWN_SENSED = 1)

*** TDLR_STATE values ***

      logical*1   K$BEAM_UNLOCKED
      parameter  (K$BEAM_UNLOCKED= 0)

      logical*1   K$BEAM_LOCKED
      parameter  (K$BEAM_LOCKED   = 1)

*** define  Range checking constants ***********************************

*** upper and lower bounds ***

***
*  v1 Changes for AR#23. Item 3. Added D0 to some reals.
***
      real*8                   K$A_ACCELERATION$LB
      parameter  (K$A_ACCELERATION$LB   =  -20.0)

      real*8                   K$A_ACCELERATION$UB
      parameter  (K$A_ACCELERATION$UB   =    5.0)

      real*8                   K$AR_ALTITUDE$LB
      parameter  (K$AR_ALTITUDE$LB          =    0.0)

      real*8                   K$AR_ALTITUDE$UB
      parameter  (K$AR_ALTITUDE$UB          = 2000.0)

      real*8                   K$ATMOSPHERIC_TEMP$LB
      parameter  (K$ATMOSPHERIC_TEMP$LB          = -200.0)

      real*8                   K$ATMOSPHERIC_TEMP$UB
      parameter  (K$ATMOSPHERIC_TEMP$UB          =   25.0)

      real*8                   K$G_ROTATION$LB
      parameter  (K$G_ROTATION$LB =   -1.0)

      real*8                   K$G_ROTATION$UB
      parameter  (K$G_ROTATION$UB=    1.0)

      real*8                   K$GP_ALTITUDE$LB
```

```
parameter  (K$GP_ALTITUDE$LB          =    0.0)

real*8                  K$GP_ALTITUDE$UB
parameter  (K$GP_ALTITUDE$UB          = 2000.0)

real*8                  K$GP_ATTITUDE$LB
parameter  (K$GP_ATTITUDE$LB          =   -1.0)

real*8                  K$GP_ATTITUDE$UB
parameter  (K$GP_ATTITUDE$UB          =    1.0)

real*8                  K$GP_ROTATION$LB
parameter  (K$GP_ROTATION$LB          =   -1.0)

real*8                  K$GP_ROTATION$UB
parameter  (K$GP_ROTATION$UB          =    1.0)

real*8                  K$GP_VELOCITY$LB
parameter  (K$GP_VELOCITY$LB          = -100.0)

real*8                  K$GP_VELOCITY$UB
parameter  (K$GP_VELOCITY$UB          =  100.0)

real*8                  K$INTERNAL_CMD$LB
parameter  (K$INTERNAL_CMD$LB         =  -0.7D0)

real*8                  K$INTERNAL_CMD$UB
parameter  (K$INTERNAL_CMD$UB         =   1.7D0)

real*8                  K$PE_INTEGRAL$LB
parameter  (K$PE_INTEGRAL$LB          = -100.0)

real*8                  K$PE_INTEGRAL$UB
parameter  (K$PE_INTEGRAL$UB          = 100.0)

real*8                  K$TDLR_VELOCITY$LB
parameter  (K$TDLR_VELOCITY$LB        = -100.0)

real*8                  K$TDLR_VELOCITY$UB
parameter  (K$TDLR_VELOCITY$UB        =  100.0)

real*8                  K$TE_INTEGRAL$LB
parameter  (K$TE_INTEGRAL$LB          = -100.0)

real*8                  K$TE_INTEGRAL$UB
parameter  (K$TE_INTEGRAL$UB          = 100.0)

real*8                  K$TE_LIMIT$LB
parameter  (K$TE_LIMIT$LB             = -100.0)

real*8                  K$TE_LIMIT$UB
```

```
      parameter  (K$TE_LIMIT$UB               = 100.0)
***
*  v2 Changes for AR#24. Item 5. Changed signs.
***
      real*8                 K$THETA$UB
      parameter  (K$THETA$UB          = 3.141592653589793)

      real*8                 K$THETA$LB
      parameter  (K$THETA$LB          = -3.141592653589793)

*       real*8               K$THETA$UB
*       parameter(K$THETA$UB          = -3.141592653589793)
*
*       real*8               K$THETA$LB
*       parameter(K$THETA$LB          =  3.141592653589793)
***
*  v2 Changes for AR#24. End Change.
***

      real*8                 K$VELOCITY_ERROR$LB
      parameter  (K$VELOCITY_ERROR$LB   = -300.0)

      real*8                 K$VELOCITY_ERROR$UB
      parameter  (K$VELOCITY_ERROR$UB   =   20.0)

      real*8                 K$YE_INTEGRAL$LB
      parameter  (K$YE_INTEGRAL$LB          = -100.0)

      real*8                 K$YE_INTEGRAL$UB
      parameter  (K$YE_INTEGRAL$UB          =  100.0)

***
*  v1 Changes for AR#23. End Change.
***

*** define constants for data element names used in error messages ******

      character*(*)          K$A_ACCELERATION$NAME
      parameter  (K$A_ACCELERATION$NAME         = 'A_ACCELERATION')

      character*(*)          K$AR_ALTITUDE$NAME
      parameter  (K$AR_ALTITUDE$NAME     = 'AR_ALTITUDE')

      character*(*)          K$ATMOSPHERIC_TEMP$NAME
      parameter  (K$ATMOSPHERIC_TEMP$NAME= 'ATMOSPHERIC_TEMP')

      character*(*)          K$G_ROTATION$NAME
      parameter  (K$G_ROTATION$NAME       = 'G_ROTATION')

      character*(*)          K$GP_ALTITUDE$NAME
      parameter  (K$GP_ALTITUDE$NAME     = 'GP_ALTITUDE')
```

```
character*(*)        K$GP_ATTITUDE$NAME
parameter  (K$GP_ATTITUDE$NAME     = 'GP_ATTITUDE')

character*(*)        K$GP_ROTATION$NAME
parameter  (K$GP_ROTATION$NAME     = 'GP_ROTATION')

character*(*)        K$GP_VELOCITY$NAME
parameter  (K$GP_VELOCITY$NAME     = 'GP_VELOCITY')

character*(*)        K$INTERNAL_CMD$NAME
parameter  (K$INTERNAL_CMD$NAME = 'INTERNAL_CMD')

character*(*)        K$PE_INTEGRAL$NAME
parameter  (K$PE_INTEGRAL$NAME     = 'PE_INTEGRAL')

character*(*)        K$TDLR_VELOCITY$NAME
parameter  (K$TDLR_VELOCITY$NAME           = 'TDLR_VELOCITY')

character*(*)        K$TE_INTEGRAL$NAME
parameter  (K$TE_INTEGRAL$NAME     = 'TE_INTEGRAL')

character*(*)        K$TE_LIMIT$NAME
parameter  (K$TE_LIMIT$NAME = 'TE_LIMIT')

character*(*)        K$THETA$NAME
parameter  (K$THETA$NAME                = 'THETA')

character*(*)        K$VELOCITY_ERROR$NAME
parameter  (K$VELOCITY_ERROR$NAME         = 'VELOCITY_ERROR')

character*(*)        K$YE_INTEGRAL$NAME
parameter  (K$YE_INTEGRAL$NAME     = 'YE_INTEGRAL')

***** end of module CONSTANTS.FOR *************************************
```

```
**********************************************************************
*  Module:        CP.FOR
*  Facility:      Pluto
*  P-Spec:        2.3
*  Abstract:
*          This module contains the implementation of the functional
*          requirements for CP.
*
*  List of Routines:
*          subroutine CP
*          function   CRC16
**********************************************************************

**********************************************************************
*  Title: CP
*  Facility:      Pluto
*  Abstract:
*          1) determine the current operational status of the communicator.
*          2) construct a telemetry data packet.
*
*  Arguments:   None
*
*  Revision History:
*          v0    15-sep-1994  Rob Angellatta (RKA) Original.
*          v1    13-jan-1995  Philip Morris (PEM)
**********************************************************************

      subroutine CP

      implicit none

*** include the global common stores ***

      include 'external.for'
      include 'guidance_state.for'
      include 'sensor_output.for'
      include 'run_parameters.for'

*** include constant definitions ***

      include 'constants.for'

*** define local constants ***

*** the byte count (size in bytes) of the three packets ***

      integer*4   K$SP_SIZE
      parameter  (K$SP_SIZE = 129)

      integer*4   K$GP_SIZE
```

```
      parameter  (K$GP_SIZE = 173)

      integer*4   K$CLP_SIZE
      parameter  (K$CLP_SIZE = 45)

*** declare local functions ***

      integer*2   CRC16

*** declare local variables ***
          integer*2         seq_temp
          logical*1         seq_temp_char(2)
          equivalence       (seq_temp,seq_temp_char(1))

*************************************************************************
*  1) Determine the current operational status of the communicator.
*************************************************************************

      C_STATUS = 0

*************************************************************************
*  2) Construct a telemetry data packet.
*************************************************************************

*************************************************************************
*  2A) Get synchronization pattern.
*************************************************************************

      PACKET.sync_pattern = COMM_SYNC_PATTERN

*************************************************************************
*  2B) Determine the sequence number.
*************************************************************************

      seq_temp = MOD(3*(FRAME_COUNTER-1)+
     &                      (SUBFRAME_COUNTER-1), 256)
          PACKET.seq_number = seq_temp_char(1)

*************************************************************************
*  2C) Prepare the data mask,
*  2D) Prepare the data, and
*  2E) Compute the checksum.
*
*  The 'PACKET' data structure is defined in module EXTERNAL.FOR
*************************************************************************

      if  (SUBFRAME_COUNTER .EQ. 1) then
        PACKET.DATA_MASK = '1F20F1F4'X

        PACKET.sp.ar_altitude      = AR_ALTITUDE(0)
        PACKET.sp.ar_status        = AR_STATUS(0)
```

```
          PACKET.sp.atmospheric_temp = ATMOSPHERIC_TEMP
          PACKET.sp.a_acceleration(1)= A_ACCELERATION(1,0)
          PACKET.sp.a_acceleration(2)= A_ACCELERATION(2,0)
          PACKET.sp.a_acceleration(3)= A_ACCELERATION(3,0)
          PACKET.sp.a_status(1)      = A_STATUS(1,0)
          PACKET.sp.A_STATUS(2)      = A_STATUS(2,0)
          PACKET.sp.a_status(3)      = A_STATUS(3,0)
          PACKET.sp.c_status         = C_STATUS
          PACKET.sp.g_rotation(1)    = G_ROTATION(1,0)
          PACKET.sp.g_rotation(2)    = G_ROTATION(2,0)
          PACKET.sp.g_rotation(3)    = G_ROTATION(3,0)
          PACKET.sp.g_status         = G_STATUS
          PACKET.sp.k_alt            = K_ALT(0)
          PACKET.sp.k_matrix(1)      = K_MATRIX(1,1,0)
          PACKET.sp.k_matrix(2)      = K_MATRIX(2,2,0)
          PACKET.sp.k_matrix(3)      = K_MATRIX(3,3,0)
          PACKET.sp.tdlr_state(1)    = TDLR_STATE(1)
          PACKET.sp.tdlr_state(2)    = TDLR_STATE(2)
          PACKET.sp.tdlr_state(3)    = TDLR_STATE(3)
          PACKET.sp.tdlr_state(4)    = TDLR_STATE(4)
          PACKET.sp.tdlr_status(1)   = TDLR_STATUS(1)
          PACKET.sp.tdlr_status(2)   = TDLR_STATUS(2)
          PACKET.sp.tdlr_status(3)   = TDLR_STATUS(3)
          PACKET.sp.tdlr_status(4)   = TDLR_STATUS(4)
          PACKET.sp.tdlr_velocity(1) = TDLR_VELOCITY(1,0)
          PACKET.sp.tdlr_velocity(2) = TDLR_VELOCITY(2,0)
          PACKET.sp.tdlr_velocity(3) = TDLR_VELOCITY(3,0)
          PACKET.sp.tds_status       = TDS_STATUS
          PACKET.sp.td_sensed        = TD_SENSED
          PACKET.sp.ts_status(1)     = TS_STATUS(1)
          PACKET.sp.ts_status(2)     = TS_STATUS(2)
***
*   v1 PR#25 Item 1.  Send whole packet.
***
*         PACKET.sp.checksum         = CRC16(PACKET.sp, K$SP_SIZE)
          PACKET.sp.checksum         = CRC16(PACKET.PACKET, K$SP_SIZE)
***
*   v1 PR#25 End Change.
***

        else if (SUBFRAME_COUNTER .EQ. 2) then
          PACKET.data_mask = '007F0002'X

          PACKET.gp.contour_crossed  = CONTOUR_CROSSED
          PACKET.gp.c_status         = C_STATUS
          PACKET.gp.gp_altitude      = GP_ALTITUDE(0)

*** first element of array changes most rapidly ***

          PACKET.gp.gp_attitude(1)   = GP_ATTITUDE(1, 1, 0)
          PACKET.gp.gp_attitude(2)   = GP_ATTITUDE(2, 1, 0)
```

```
         PACKET.gp.gp_attitude(3)   = GP_ATTITUDE(3, 1, 0)
         PACKET.gp.gp_attitude(4)   = GP_ATTITUDE(1, 2, 0)
         PACKET.gp.gp_attitude(5)   = GP_ATTITUDE(2, 2, 0)
         PACKET.gp.gp_attitude(6)   = GP_ATTITUDE(3, 2, 0)
         PACKET.gp.gp_attitude(7)   = GP_ATTITUDE(1, 3, 0)
         PACKET.gp.gp_attitude(8)   = GP_ATTITUDE(2, 3, 0)
         PACKET.gp.gp_attitude(9)   = GP_ATTITUDE(3, 3, 0)

         PACKET.gp.gp_phase         = GP_PHASE

         PACKET.gp.gp_rotation(1)   = GP_ROTATION(2, 1)
         PACKET.gp.gp_rotation(2)   = GP_ROTATION(3, 1)
         PACKET.gp.gp_rotation(3)   = GP_ROTATION(1, 2)
         PACKET.gp.gp_rotation(4)   = GP_ROTATION(3, 2)
         PACKET.gp.gp_rotation(5)   = GP_ROTATION(1, 3)
         PACKET.gp.gp_rotation(6)   = GP_ROTATION(2, 3)

         PACKET.gp.gp_velocity(1)   =  GP_VELOCITY(1, 0)
         PACKET.gp.gp_velocity(2)   =  GP_VELOCITY(2, 0)
         PACKET.gp.gp_velocity(3)   =  GP_VELOCITY(3, 0)

         PACKET.gp.velocity_error   = VELOCITY_ERROR
***
*   v1 PR#25 Item 1.  Send whole packet.
***
*         PACKET.gp.checksum        = CRC16(PACKET.gp, K$GP_SIZE)
         PACKET.gp.checksum         = CRC16(PACKET.PACKET, K$GP_SIZE)
***
*   v1 PR#25 End Change.
***

       else
         PACKET.data_mask = 'E0A00E09'X

         PACKET.clp.ae_cmd(1)       = AE_CMD(1)
         PACKET.clp.ae_cmd(2)       = AE_CMD(2)
         PACKET.clp.ae_cmd(3)       = AE_CMD(3)
         PACKET.clp.ae_status       = AE_STATUS
         PACKET.clp.ae_temp         = AE_TEMP
         PACKET.clp.chute_released  = CHUTE_RELEASED
         PACKET.clp.c_status        = C_STATUS
         PACKET.clp.pe_integral     = PE_INTEGRAL
         PACKET.clp.re_cmd          = RE_CMD
         PACKET.clp.re_status       = RE_STATUS
         PACKET.clp.te_integral     = TE_INTEGRAL
         PACKET.clp.ye_integral     = YE_INTEGRAL
***
*   v1 PR#25 Item 1.  Send whole packet.
***
*         PACKET.clp.checksum       = CRC16(PACKET.clp, K$CLP_SIZE)
         PACKET.clp.checksum        = CRC16(PACKET.PACKET, K$CLP_SIZE)
```

```
***
*   v1 PR#25 End Change.
***

        end if

        return
        end

***** end of subroutine CP *******************************************

**********************************************************************
*  Title: CRC16
*  Facility:       Pluto
*  Abstract:
*     Compute the Cyclic Redundancy Code of the specified buffer using
*     CRC-16 as the generator polynomial.
*
*  Arguments:
*          character*(*)   message   - address of first byte of message.
*      integer*4  bytecount - count of bytes in message.
*
*  Returns:
*          integer*2           crc16     - the bit checksum of the specified message.
*
*  Revision History:
*          v0    15-sep-1994  Rob Angellatta (RKA) Original.
**********************************************************************

        integer*2 function CRC16(message, bytecount)

        implicit none

*** declare the arguments ***

        integer*4   bytecount
        byte                    message(bytecount)

*** declare local variables ***

        integer*4   i
        logical*2   index
        integer*2   temp

*** The "signature" table for the CRC-16 generator polynomial 0xA001 ***

        integer*2   crc16_table(0:255)

        data (crc16_table(i),i=  0,  7)
     &                /'0000'X, 'c0c1'X, 'c181'X, '0140'X,
     &                 'c301'X, '03c0'X, '0280'X, 'c241'X/
```

```
  data (crc16_table(i),i=  8, 15)
&                 /'c601'X, '06c0'X, '0780'X, 'c741'X,
&                  '0500'X, 'c5c1'X, 'c481'X, '0440'X/
  data (crc16_table(i),i= 16, 23)
&                 /'cc01'X, '0cc0'X, '0d80'X, 'cd41'X,
&                  '0f00'X, 'cfc1'X, 'ce81'X, '0e40'X/
  data (crc16_table(i),i= 24, 31)
&                 /'0a00'X, 'cac1'X, 'cb81'X, '0b40'X,
&                  'c901'X, '09c0'X, '0880'X, 'c841'X/
  data (crc16_table(i),i= 32, 39)
&                 /'d801'X, '18c0'X, '1980'X, 'd941'X,
&                  '1b00'X, 'dbc1'X, 'da81'X, '1a40'X/
  data (crc16_table(i),i= 40, 47)
&                 /'1e00'X, 'dec1'X, 'df81'X, '1f40'X,
&                  'dd01'X, '1dc0'X, '1c80'X, 'dc41'X/
  data (crc16_table(i),i= 48, 55)
&                 /'1400'X, 'd4c1'X, 'd581'X, '1540'X,
&                  'd701'X, '17c0'X, '1680'X, 'd641'X/
  data (crc16_table(i),i= 56, 63)
&                 /'d201'X, '12c0'X, '1380'X, 'd341'X,
&                  '1100'X, 'd1c1'X, 'd081'X, '1040'X/
  data (crc16_table(i),i= 64, 71)
&                 /'f001'X, '30c0'X, '3180'X, 'f141'X,
&                  '3300'X, 'f3c1'X, 'f281'X, '3240'X/
  data (crc16_table(i),i= 72, 79)
&                 /'3600'X, 'f6c1'X, 'f781'X, '3740'X,
&                  'f501'X, '35c0'X, '3480'X, 'f441'X/
  data (crc16_table(i),i= 80, 87)
&                 /'3c00'X, 'fcc1'X, 'fd81'X, '3d40'X,
&                  'ff01'X, '3fc0'X, '3e80'X, 'fe41'X/
  data (crc16_table(i),i= 88, 95)
&                 /'fa01'X, '3ac0'X, '3b80'X, 'fb41'X,
&                  '3900'X, 'f9c1'X, 'f881'X, '3840'X/
  data (crc16_table(i),i= 96,103)
&                 /'2800'X, 'e8c1'X, 'e981'X, '2940'X,
&                  'eb01'X, '2bc0'X, '2a80'X, 'ea41'X/
  data (crc16_table(i),i=104,111)
&                 /'ee01'X, '2ec0'X, '2f80'X, 'ef41'X,
&                  '2d00'X, 'edc1'X, 'ec81'X, '2c40'X/
  data (crc16_table(i),i=112,119)
&                 /'e401'X, '24c0'X, '2580'X, 'e541'X,
&                  '2700'X, 'e7c1'X, 'e681'X, '2640'X/
  data (crc16_table(i),i=120,127)
&                 /'2200'X, 'e2c1'X, 'e381'X, '2340'X,
&                  'e101'X, '21c0'X, '2080'X, 'e041'X/
  data (crc16_table(i),i=128,135)
&                 /'a001'X, '60c0'X, '6180'X, 'a141'X,
&                  '6300'X, 'a3c1'X, 'a281'X, '6240'X/
  data (crc16_table(i),i=136,143)
&                 /'6600'X, 'a6c1'X, 'a781'X, '6740'X,
&                  'a501'X, '65c0'X, '6480'X, 'a441'X/
```

```
      data (crc16_table(i),i=144,151)
     &              /'6c00'X, 'acc1'X, 'ad81'X, '6d40'X,
     &               'af01'X, '6fc0'X, '6e80'X, 'ae41'X/
      data (crc16_table(i),i=152,159)
     &              /'aa01'X, '6ac0'X, '6b80'X, 'ab41'X,
     &               '6900'X, 'a9c1'X, 'a881'X, '6840'X/
      data (crc16_table(i),i=160,167)
     &              /'7800'X, 'b8c1'X, 'b981'X, '7940'X,
     &               'bb01'X, '7bc0'X, '7a80'X, 'ba41'X/
      data (crc16_table(i),i=168,175)
     &              /'be01'X, '7ec0'X, '7f80'X, 'bf41'X,
     &               '7d00'X, 'bdc1'X, 'bc81'X, '7c40'X/
      data (crc16_table(i),i=176,183)
     &              /'b401'X, '74c0'X, '7580'X, 'b541'X,
     &               '7700'X, 'b7c1'X, 'b681'X, '7640'X/
      data (crc16_table(i),i=184,191)
     &              /'7200'X, 'b2c1'X, 'b381'X, '7340'X,
     &               'b101'X, '71c0'X, '7080'X, 'b041'X/
      data (crc16_table(i),i=192,199)
     &              /'5000'X, '90c1'X, '9181'X, '5140'X,
     &               '9301'X, '53c0'X, '5280'X, '9241'X/
      data (crc16_table(i),i=200,207)
     &              /'9601'X, '56c0'X, '5780'X, '9741'X,
     &               '5500'X, '95c1'X, '9481'X, '5440'X/
      data (crc16_table(i),i=208,215)
     &              /'9c01'X, '5cc0'X, '5d80'X, '9d41'X,
     &               '5f00'X, '9fc1'X, '9e81'X, '5e40'X/
      data (crc16_table(i),i=216,223)
     &              /'5a00'X, '9ac1'X, '9b81'X, '5b40'X,
     &               '9901'X, '59c0'X, '5880'X, '9841'X/
      data (crc16_table(i),i=224,231)
     &              /'8801'X, '48c0'X, '4980'X, '8941'X,
     &               '4b00'X, '8bc1'X, '8a81'X, '4a40'X/
      data (crc16_table(i),i=232,239)
     &              /'4e00'X, '8ec1'X, '8f81'X, '4f40'X,
     &               '8d01'X, '4dc0'X, '4c80'X, '8c41'X/
      data (crc16_table(i),i=240,247)
     &              /'4400'X, '84c1'X, '8581'X, '4540'X,
     &               '8701'X, '47c0'X, '4680'X, '8641'X/
      data (crc16_table(i),i=248,255)
     &              /'8201'X, '42c0'X, '4380'X, '8341'X,
     &               '4100'X, '81c1'X, '8081'X, '4040'X/

*** crc is a 16-bit unsigned integer value ***

      CRC16 = 0

*** process every byte in the message ***

      do i = 1,bytecount
         temp  = message(i)
```

```
   index = IEOR(CRC16, temp)       ! bitwise xOR lower 8 bits of crc
      index = IAND(index,'00FF'X) ! clear top byte of word
   CRC16 = ISHFT(CRC16, -8)       ! bitwise right shift crc 8 times
   CRC16 = IEOR(CRC16,crc16_table(index)) ! bitwise XOR 16 bits
 end do

 return
 end

***** end of function CRC16 **********************************************

***** end of module CP ****************************************************
```

```
***********************************************************************
*  Module:        CRCP.FOR
*  Facilitiy:     Pluto
*  P-Spec:        3.3
*  Abstract:
*         This module contains the implementation of the functional
*         requirements for CRCP.
*
*  List of Routines:
*         subroutine CRCP
***********************************************************************
***********************************************************************
*  Title: CRCP
*  Facility:      Pluto
*  Abstract:
*         1) Determine whether or not to release the parachute.
*
*  Arguments:     None.
*
*  Revision History:
*         v0    15-sep-1994  Rob Angellatta (RKA) Original.
***********************************************************************
      subroutine CRCP

      implicit none

*** include the global common stores ***

      include 'guidance_state.for'
      include 'external.for'

*** include constant definitions ***

      include 'constants.for'

***********************************************************************
*  The parachute is to be released during the same frame in which the
*  axial engine temperature becomes "HOT" (2). Valid states for
*  CHUTE_RELEASED are "Chute Attached" (0) and "Chute Released" (1).
***********************************************************************
***   1) Determine whether or not to release the parachute. ***

      if (CHUTE_RELEASED .eq. K$CHUTE_ATTACHED) then

        if (AE_TEMP .eq. K$HOT) then
          CHUTE_RELEASED = K$CHUTE_RELEASED
        end if

      end if

      return
```

```
      end
***** end of module crcp.for *********************************************
```

```
**********************************************************************
*  Module:        EXTERNAL.FOR
*  Facility:      Pluto
*  Abstract:
*         This module contains the data definitions for the
*         global common data store named EXTERNAL.
*
*  Revision History:
*         v0    15-sep-1994  Rob Angellatta (RKA) Original.
*      v1    30-Nov-1994  Philip Morris (PEM)
**********************************************************************

*** COMMON block definition ***

      COMMON           /EXTERNAL/
    &              A_COUNTER,
    &              AE_CMD,
    &              AR_COUNTER,
    &              CHUTE_RELEASED,
    &              FRAME_COUNTER,
    &              G_COUNTER,
    &        PACKET,
    &              RE_CMD,
    &              SS_TEMP,
    &              SUBFRAME_COUNTER,
    &              TD_COUNTER,
    &              TDLR_COUNTER,
    &              THERMO_TEMP

*** data type declarations ***

      integer*2   A_COUNTER(1:3)
      integer*2   AE_CMD(1:3)
      integer*2   AR_COUNTER
      logical*1   CHUTE_RELEASED
      integer*4   FRAME_COUNTER
      integer*2   G_COUNTER(1:3)
      integer*2   RE_CMD
      integer*2   SS_TEMP
      integer*2   SUBFRAME_COUNTER
      integer*2   TD_COUNTER
      integer*2   TDLR_COUNTER(1:4)
      integer*2   THERMO_TEMP

**********************************************************************
*  Although the specifications define 'packet' as an array of
*  integer*2's, the functional unit CP treats 'packet' as a
*  variant record.  The definitions below reserve an array
*  of 256 integer*2 data and overlay the area with a variant
*  record structure.
**********************************************************************
```

```
*** the Sensor Processing subframe data field and checksum ***

        structure /sp_data_t/
           real*8     ar_altitude
           logical*1 ar_status
           real*8     atmospheric_temp
           real*8     a_acceleration(1:3)
           logical*1 a_status(1:3)
           logical*1 c_status
           real*8     g_rotation(1:3)
           logical*1 g_status
           integer*4k_alt
           integer*4k_matrix(1:3)
           logical*1 tdlr_state(1:4)
           logical*1 tdlr_status(1:4)
           real*8     tdlr_velocity(1:3)
           logical*1 tds_status
           logical*1 td_sensed
           logical*1 ts_status(2)
           integer*2checksum
        end structure

*** the Guidance Processing subframe data field and checksum ***

        structure /gp_data_t/
           logical*1 contour_crossed
           logical*1 c_status
           real*8     gp_altitude
           real*8     gp_attitude(1:9)
           integer*4gp_phase
           real*8     gp_rotation(1:6)
           real*8     gp_velocity(1:3)
           real*8     velocity_error
           integer*2checksum
        end structure

*** the Control Law Processing subframe data field and checksum ***

***
*  v1 Changes for AR#23. Item 5. ae_temp was changed from logical*1 to integer*2
***
        structure /clp_data_t/
           integer*2ae_cmd(1:3)
           logical*1 ae_status
           integer*2ae_temp
           logical*1 chute_released
           logical*1 c_status
           real*8     pe_integral
           integer*2re_cmd
           logical*1 re_status
```

```
      real*8     te_integral
      real*8     ye_integral
      integer*2checksum
    end structure

***
*  v1 Changes for AR#23. End Change.
***
*** the data packet structure ***

    structure /data_packet_t/
      union
        map
          integer*2       PACKET(1:256)
        end map
        map
          integer*2       sync_pattern
          logical*1       seq_number
          integer*4       data_mask
          union
            map
              record /sp_data_t/ sp
            end map
            map
              record /gp_data_t/ gp
            end map
            map
              record /clp_data_t/ clp
            end map
          end union
        end map
      end union
    end structure

*** declare a variable of type PACKET ***

    record /data_packet_t/ PACKET

***** end of module EXTERNAL.FOR ****************************************
```

```
***********************************************************************
* Module:       GP.FOR
* Facility:     Pluto
* P-Spec:       2.2
* Abstract:
*         This module contains the implementation of the functional
*         requirements for GP.
*
* List of Routines:
*         subroutine GP
*         subroutine DERIV_ALT
*         subroutine DERIV_ATT
*         subroutine DERIV_VEL
*         subroutine MULT_ATT
*         subroutine MULT_VEL
*       subroutine AVG_ATT
*       subroutine AVG_VEL
***********************************************************************

***********************************************************************
* Title: GP
* Facility:     Pluto
* Abstract:
*         1) Maintain the history of the vehicle's altitude, velocities,
*            and attitude.
*         2) Compute the current vehicle altitude, velocities and attitude.
*         3) Determine if the engines should be switched on or off.
*         4) Compute the current velocity error.
*         5) Determe if the predetermined velocity-altitude contour has
*            been crossed.
*         6) Determine the current guidance phase.
*         7) Determine the appropriate axial engine control law parameters.
*
* Arguments:   None
*
* Revision History:
*         v0    15-sep-1994  Rob Angellatta (RKA) Original.
*       v1    01-Dec-1994  Philip Morris (PEM)
*       v2    10-JAN-1995  Philip Morris (PEM)
*       v3    16-MAR-1995  Philip Morris (PEM)
***********************************************************************

      subroutine GP

      implicit none

*** include the global common stores ***

      include 'external.for'
      include 'guidance_state.for'
      include 'sensor_output.for'
```

```
      include 'run_parameters.for'

*** include constant definitions ***

      include 'constants.for'

*** declare local variables ***

      real*8          temp
      real*8          cur_altitude
      real*8          optimal_velocity

      real*8          att_k1(3,3), att_k2(3,3)
      real*8    att_k3(3,3), att_k4(3,3)
      real*8          att_tmp(3,3)

      real*8          vel_k1(3), vel_k2(3), vel_k3(3), vel_k4(3)
      real*8    vel_tmp(3)
      real*8          alt_k1, alt_k2, alt_k3, alt_k4

      real*8          step_size

      integer*4   i, j

************************************************************************
* 1) Maintain the history of the vehicle altitude, velocities,
*    and attitude by "rotating variables."
************************************************************************

      GP_ALTITUDE(4) = GP_ALTITUDE(3)
      GP_ALTITUDE(3) = GP_ALTITUDE(2)
      GP_ALTITUDE(2) = GP_ALTITUDE(1)
      GP_ALTITUDE(1) = GP_ALTITUDE(0)

      GP_VELOCITY(1, 4) = GP_VELOCITY(1, 3)
      GP_VELOCITY(1, 3) = GP_VELOCITY(1, 2)
      GP_VELOCITY(1, 2) = GP_VELOCITY(1, 1)
      GP_VELOCITY(1, 1) = GP_VELOCITY(1, 0)

      GP_VELOCITY(2, 4) = GP_VELOCITY(2, 3)
      GP_VELOCITY(2, 3) = GP_VELOCITY(2, 2)
      GP_VELOCITY(2, 2) = GP_VELOCITY(2, 1)
      GP_VELOCITY(2, 1) = GP_VELOCITY(2, 0)

      GP_VELOCITY(3, 4) = GP_VELOCITY(3, 3)
      GP_VELOCITY(3, 3) = GP_VELOCITY(3, 2)
      GP_VELOCITY(3, 2) = GP_VELOCITY(3, 1)
      GP_VELOCITY(3, 1) = GP_VELOCITY(3, 0)

      GP_ATTITUDE(1, 1, 4) = GP_ATTITUDE(1, 1, 3)
      GP_ATTITUDE(1, 2, 4) = GP_ATTITUDE(1, 2, 3)
```

```
      GP_ATTITUDE(1, 3, 4) = GP_ATTITUDE(1, 3, 3)
      GP_ATTITUDE(2, 1, 4) = GP_ATTITUDE(2, 1, 3)
      GP_ATTITUDE(2, 2, 4) = GP_ATTITUDE(2, 2, 3)
      GP_ATTITUDE(2, 3, 4) = GP_ATTITUDE(2, 3, 3)
      GP_ATTITUDE(3, 1, 4) = GP_ATTITUDE(3, 1, 3)
      GP_ATTITUDE(3, 2, 4) = GP_ATTITUDE(3, 2, 3)
      GP_ATTITUDE(3, 3, 4) = GP_ATTITUDE(3, 3, 3)

      GP_ATTITUDE(1, 1, 3) = GP_ATTITUDE(1, 1, 2)
      GP_ATTITUDE(1, 2, 3) = GP_ATTITUDE(1, 2, 2)
      GP_ATTITUDE(1, 3, 3) = GP_ATTITUDE(1, 3, 2)
      GP_ATTITUDE(2, 1, 3) = GP_ATTITUDE(2, 1, 2)
      GP_ATTITUDE(2, 2, 3) = GP_ATTITUDE(2, 2, 2)
      GP_ATTITUDE(2, 3, 3) = GP_ATTITUDE(2, 3, 2)
      GP_ATTITUDE(3, 1, 3) = GP_ATTITUDE(3, 1, 2)
      GP_ATTITUDE(3, 2, 3) = GP_ATTITUDE(3, 2, 2)
      GP_ATTITUDE(3, 3, 3) = GP_ATTITUDE(3, 3, 2)

      GP_ATTITUDE(1, 1, 2) = GP_ATTITUDE(1, 1, 1)
      GP_ATTITUDE(1, 2, 2) = GP_ATTITUDE(1, 2, 1)
      GP_ATTITUDE(1, 3, 2) = GP_ATTITUDE(1, 3, 1)
      GP_ATTITUDE(2, 1, 2) = GP_ATTITUDE(2, 1, 1)
      GP_ATTITUDE(2, 2, 2) = GP_ATTITUDE(2, 2, 1)
      GP_ATTITUDE(2, 3, 2) = GP_ATTITUDE(2, 3, 1)
      GP_ATTITUDE(3, 1, 2) = GP_ATTITUDE(3, 1, 1)
      GP_ATTITUDE(3, 2, 2) = GP_ATTITUDE(3, 2, 1)
      GP_ATTITUDE(3, 3, 2) = GP_ATTITUDE(3, 3, 1)

      GP_ATTITUDE(1, 1, 1) = GP_ATTITUDE(1, 1, 0)
      GP_ATTITUDE(1, 2, 1) = GP_ATTITUDE(1, 2, 0)
      GP_ATTITUDE(1, 3, 1) = GP_ATTITUDE(1, 3, 0)
      GP_ATTITUDE(2, 1, 1) = GP_ATTITUDE(2, 1, 0)
      GP_ATTITUDE(2, 2, 1) = GP_ATTITUDE(2, 2, 0)
      GP_ATTITUDE(2, 3, 1) = GP_ATTITUDE(2, 3, 0)
      GP_ATTITUDE(3, 1, 1) = GP_ATTITUDE(3, 1, 0)
      GP_ATTITUDE(3, 2, 1) = GP_ATTITUDE(3, 2, 0)
      GP_ATTITUDE(3, 3, 1) = GP_ATTITUDE(3, 3, 0)

************************************************************************
*  2) Compute the current vehicle altitude, velocities and attitude.
************************************************************************

*** range check the following data elements ***

      call RANGE_CHECK(GP_ATTITUDE(1, 1, 2), K$GP_ATTITUDE$LB,
     &                 K$GP_ATTITUDE$UB, 'GP', K$GP_ATTITUDE$NAME)
      call RANGE_CHECK(GP_ATTITUDE(1, 2, 2), K$GP_ATTITUDE$LB,
     &                 K$GP_ATTITUDE$UB, 'GP', K$GP_ATTITUDE$NAME)
      call RANGE_CHECK(GP_ATTITUDE(1, 3, 2), K$GP_ATTITUDE$LB,
     &                 K$GP_ATTITUDE$UB, 'GP', K$GP_ATTITUDE$NAME)
      call RANGE_CHECK(GP_ATTITUDE(2, 1, 2), K$GP_ATTITUDE$LB,
```

```
&                   K$GP_ATTITUDE$UB, 'GP', K$GP_ATTITUDE$NAME)
  call RANGE_CHECK(GP_ATTITUDE(2, 2, 2), K$GP_ATTITUDE$LB,
&                   K$GP_ATTITUDE$UB, 'GP', K$GP_ATTITUDE$NAME)
  call RANGE_CHECK(GP_ATTITUDE(2, 3, 2), K$GP_ATTITUDE$LB,
&                   K$GP_ATTITUDE$UB, 'GP', K$GP_ATTITUDE$NAME)
  call RANGE_CHECK(GP_ATTITUDE(3, 1, 2), K$GP_ATTITUDE$LB,
&                   K$GP_ATTITUDE$UB, 'GP', K$GP_ATTITUDE$NAME)
  call RANGE_CHECK(GP_ATTITUDE(3, 2, 2), K$GP_ATTITUDE$LB,
&                   K$GP_ATTITUDE$UB, 'GP', K$GP_ATTITUDE$NAME)
  call RANGE_CHECK(GP_ATTITUDE(3, 3, 2), K$GP_ATTITUDE$LB,
&                   K$GP_ATTITUDE$UB, 'GP', K$GP_ATTITUDE$NAME)

  call RANGE_CHECK(GP_VELOCITY(1, 2), K$GP_VELOCITY$LB,
&                   K$GP_VELOCITY$UB, 'GP',  K$GP_VELOCITY$NAME)
  call RANGE_CHECK(GP_VELOCITY(2, 2), K$GP_VELOCITY$LB,
&                   K$GP_VELOCITY$UB, 'GP', K$GP_VELOCITY$NAME)
  call RANGE_CHECK(GP_VELOCITY(3, 2), K$GP_VELOCITY$LB,
&                   K$GP_VELOCITY$UB, 'GP', K$GP_VELOCITY$NAME)

  call RANGE_CHECK(GP_ALTITUDE(2), K$GP_ALTITUDE$LB,
&                   K$GP_ALTITUDE$UB, 'GP', K$GP_ALTITUDE$NAME)

*** sensor data ***

  call RANGE_CHECK(G_ROTATION(1, 0), K$G_ROTATION$LB,
&                   K$G_ROTATION$UB, 'GP', K$G_ROTATION$NAME)
  call RANGE_CHECK(G_ROTATION(2, 0), K$G_ROTATION$LB,
&                   K$G_ROTATION$UB, 'GP', K$G_ROTATION$NAME)
  call RANGE_CHECK(G_ROTATION(3, 0), K$G_ROTATION$LB,
&                   K$G_ROTATION$UB, 'GP', K$G_ROTATION$NAME)
  call RANGE_CHECK(G_ROTATION(1, 1), K$G_ROTATION$LB,
&                   K$G_ROTATION$UB, 'GP', K$G_ROTATION$NAME)
  call RANGE_CHECK(G_ROTATION(2, 1), K$G_ROTATION$LB,
&                   K$G_ROTATION$UB, 'GP', K$G_ROTATION$NAME)
  call RANGE_CHECK(G_ROTATION(3, 1), K$G_ROTATION$LB,
&                   K$G_ROTATION$UB, 'GP', K$G_ROTATION$NAME)
  call RANGE_CHECK(G_ROTATION(1, 2), K$G_ROTATION$LB,
&                   K$G_ROTATION$UB, 'GP', K$G_ROTATION$NAME)
  call RANGE_CHECK(G_ROTATION(2, 2), K$G_ROTATION$LB,
&                   K$G_ROTATION$UB, 'GP', K$G_ROTATION$NAME)
  call RANGE_CHECK(G_ROTATION(3, 2), K$G_ROTATION$LB,
&                   K$G_ROTATION$UB, 'GP', K$G_ROTATION$NAME)

  call RANGE_CHECK(A_ACCELERATION(1, 0), K$A_ACCELERATION$LB,
&                   K$A_ACCELERATION$UB, 'GP', K$A_ACCELERATION$NAME)
  call RANGE_CHECK(A_ACCELERATION(2, 0), K$A_ACCELERATION$LB,
&                   K$A_ACCELERATION$UB, 'GP', K$A_ACCELERATION$NAME)
  call RANGE_CHECK(A_ACCELERATION(3, 0), K$A_ACCELERATION$LB,
&                   K$A_ACCELERATION$UB, 'GP', K$A_ACCELERATION$NAME)
  call RANGE_CHECK(A_ACCELERATION(1, 1), K$A_ACCELERATION$LB,
&                   K$A_ACCELERATION$UB, 'GP', K$A_ACCELERATION$NAME)
```

```
      call RANGE_CHECK(A_ACCELERATION(2, 1), K$A_ACCELERATION$LB,
     &                K$A_ACCELERATION$UB, 'GP', K$A_ACCELERATION$NAME)
      call RANGE_CHECK(A_ACCELERATION(3, 1), K$A_ACCELERATION$LB,
     &                K$A_ACCELERATION$UB, 'GP', K$A_ACCELERATION$NAME)
      call RANGE_CHECK(A_ACCELERATION(1, 2), K$A_ACCELERATION$LB,
     &                K$A_ACCELERATION$UB, 'GP', K$A_ACCELERATION$NAME)
      call RANGE_CHECK(A_ACCELERATION(2, 2), K$A_ACCELERATION$LB,
     &                K$A_ACCELERATION$UB, 'GP', K$A_ACCELERATION$NAME)
      call RANGE_CHECK(A_ACCELERATION(3, 2), K$A_ACCELERATION$LB,
     &                K$A_ACCELERATION$UB, 'GP', K$A_ACCELERATION$NAME)

      call RANGE_CHECK(TDLR_VELOCITY(1, 0), K$TDLR_VELOCITY$LB,
     &                K$TDLR_VELOCITY$UB, 'GP', K$TDLR_VELOCITY$NAME)
      call RANGE_CHECK(TDLR_VELOCITY(2, 0), K$TDLR_VELOCITY$LB,
     &                K$TDLR_VELOCITY$UB, 'GP', K$TDLR_VELOCITY$NAME)
      call RANGE_CHECK(TDLR_VELOCITY(3, 0), K$TDLR_VELOCITY$LB,
     &                K$TDLR_VELOCITY$UB, 'GP', K$TDLR_VELOCITY$NAME)
      call RANGE_CHECK(TDLR_VELOCITY(1, 1), K$TDLR_VELOCITY$LB,
     &                K$TDLR_VELOCITY$UB, 'GP', K$TDLR_VELOCITY$NAME)
      call RANGE_CHECK(TDLR_VELOCITY(2, 1), K$TDLR_VELOCITY$LB,
     &                K$TDLR_VELOCITY$UB, 'GP', K$TDLR_VELOCITY$NAME)
      call RANGE_CHECK(TDLR_VELOCITY(3, 1), K$TDLR_VELOCITY$LB,
     &                K$TDLR_VELOCITY$UB, 'GP', K$TDLR_VELOCITY$NAME)
      call RANGE_CHECK(TDLR_VELOCITY(1, 2), K$TDLR_VELOCITY$LB,
     &                K$TDLR_VELOCITY$UB, 'GP', K$TDLR_VELOCITY$NAME)
      call RANGE_CHECK(TDLR_VELOCITY(2, 2), K$TDLR_VELOCITY$LB,
     &                K$TDLR_VELOCITY$UB, 'GP', K$TDLR_VELOCITY$NAME)
      call RANGE_CHECK(TDLR_VELOCITY(3, 2), K$TDLR_VELOCITY$LB,
     &                K$TDLR_VELOCITY$UB, 'GP', K$TDLR_VELOCITY$NAME)

      call RANGE_CHECK(AR_ALTITUDE(0), K$AR_ALTITUDE$LB,
     &                K$AR_ALTITUDE$UB, 'GP', K$AR_ALTITUDE$NAME)
      call RANGE_CHECK(AR_ALTITUDE(1), K$AR_ALTITUDE$LB,
     &                K$AR_ALTITUDE$UB, 'GP', K$AR_ALTITUDE$NAME)
      call RANGE_CHECK(AR_ALTITUDE(2), K$AR_ALTITUDE$LB,
     &                K$AR_ALTITUDE$UB, 'GP', K$AR_ALTITUDE$NAME)

************************************************************************
*   A five step implementation of the RK method.  The functions
*   deriv_att(), deriv_vel(), and deriv_alt() are described below.
*
*  The interval begins at the current frame minus 2 frames.
*
*  1.  Compute the first estimate of the incremental value for
*      GP_ATTITUDE, GP_VELOCITY, and GP_ALTITUDE based upon the
*      rate of change at the beginning of the interval
*      (2 frames ago):
*
*      estimate    = rate_of_change * step_size
************************************************************************
```

```
      step_size   = 2 * DELTA_T

      call deriv_att(att_k1, GP_ATTITUDE(1,1,2), 2)
      call mult_att(att_k1, step_size)

***
*  v1 Changes for AR#23. Item 13. "att_k1" changed to "vel_k1"
***
      call deriv_vel(vel_k1, GP_VELOCITY(1,2), GP_ATTITUDE(1,1,2), 2)
      call mult_vel(vel_k1, step_size)

***
*  v1 Changes for AR#23. End Change.
***
      call deriv_alt(alt_k1, GP_ALTITUDE(2),GP_VELOCITY(1,2),
     &               GP_ATTITUDE(1,1,2), 2)
      alt_k1 = alt_k1 * step_size

**********************************************************************
*  2.  Compute the second estimate of the incremental value for
*      GP_ATTITUDE, GP_VELOCITY, and GP_ALTITUDE based upon the
*      rate of change at the midpoint of the first estimate k1:
**********************************************************************

      call avg_att(att_tmp, GP_ATTITUDE(1,1,2), att_k1)
      call deriv_att(att_k2, att_tmp, 1)
      call mult_att(att_k2, step_size)

      call avg_vel(vel_tmp, GP_VELOCITY(1,2), vel_k1)
      call deriv_vel(vel_k2, vel_tmp, att_tmp, 1)
***
*  v2 Changes for PR#24. Item 2. Changed division placement.
***
*      call mult_vel(att_k2, step_size)
      call mult_vel(vel_k2, step_size)

*      call deriv_alt(alt_k2, (GP_ALTITUDE(2) + alt_k1)/2,
*    &               vel_tmp, att_tmp, 1)
      call deriv_alt(alt_k2, (GP_ALTITUDE(2) + alt_k1/2),
     &               vel_tmp, att_tmp, 1)
      alt_k2 = alt_k2 * STEP_SIZE
***
*  v2 Changes for AR#24. End Change.
***

**********************************************************************
*  3.  Compute the third estimate of the incremental value for
*      GP_ATTITUDE, GP_VELOCITY, and GP_ALTITUDE based upon the
*      rate of change at the midpoint of the second estimate k2:
**********************************************************************
```

```
      call avg_att(att_tmp, GP_ATTITUDE(1,1,2), att_k2)
      call deriv_att(att_k3, att_tmp, 1)
      call mult_att(att_k3, step_size)

      call avg_vel(vel_tmp, GP_VELOCITY(1,2), vel_k2)
      call deriv_vel(vel_k3, vel_tmp, att_tmp, 1)
      call mult_vel(vel_k3, step_size)

***
*  v2 Changes for PR#24. Item 2. Changed division placement.
***
*      call deriv_alt(alt_k3, (GP_ALTITUDE(2) + alt_k2)/2,
*    &                 vel_tmp, att_tmp, 1)
      call deriv_alt(alt_k3, (GP_ALTITUDE(2) + alt_k2/2),
     &                 vel_tmp, att_tmp, 1)
***
*  v2 Changes for AR#24. End Change.
***
      alt_k3 = alt_k3 * STEP_SIZE

************************************************************************
*  4.  Compute the fourth estimate of the incremental value for
*      GP_ATTITUDE, GP_VELOCITY, and GP_ALTITUDE based upon the
*      the rate-of-change at the third estimate k3:
************************************************************************

      do i = 1,3
        do j = 1,3
          att_tmp(i,j) = GP_ATTITUDE(i, j, 2) + att_k3(i, j)
        end do
      end do

      call deriv_att(att_k4, att_tmp, 0)
      call mult_att(att_k4, step_size)

      do i = 1,3
        vel_tmp(i) = GP_VELOCITY(i,2) + vel_k3(i)
      end do

***
*  v1 Changes for AR#23. Item 15. 4th parameter changed to "0"
***
      call deriv_vel(vel_k4, vel_tmp, att_tmp, 0)
***
*  v2 Changes for PR#24. Item 4. Changed att_k to vel_k4.
***
*      call mult_vel(att_k4, step_size)
      call mult_vel(vel_k4, step_size)
***
*  v2 Changes for AR#24. End Change.
***
```

```
***
* v1 Changes for AR#23. End Change.
***
      call deriv_alt(alt_k4, (GP_ALTITUDE(2) + alt_k3),
     &              vel_tmp, att_tmp, 0)
      alt_k4 = alt_k4 * STEP_SIZE

************************************************************************
* 5.  Perform a weighted average of the four previously computed
*     estimates of the new value for  GP_ATTITUDE, GP_VELOCITY, and
*     GP_ALTITUDE.
*
* Note, the syntax (*, x) and (*, *, x) represent the xth history of
* the data element.
************************************************************************

************************************************************************
*       GP_ATTITUDE(*, *, 0) = GP_ATTITUDE(*, *, 2) +
*    &              (1/6)(att_k1 + 2*(att_k2 + att_k3) + att_k4)
************************************************************************

      do i = 1,3
        do j = 1,3
          att_tmp(i,j) = (att_k2(i,j) + att_k3(i,j)) * 2.0
          att_tmp(i,j) = (att_tmp(i,j) + att_k1(i,j) +
     &                    att_k4(i,j)) / 6.0
          GP_ATTITUDE(i, j, 0) = GP_ATTITUDE(i, j, 2) + att_tmp(i,j)
        end do
      end do

************************************************************************
*       GP_VELOCITY(*, 0) = GP_VELOCITY(*, 2) +
*    &              (1/6)(vel_k1 + 2*(vel_k2 + vel_k3) + vel_k4)
************************************************************************

      do i = 1,3
        vel_tmp(i) = (vel_k2(i) + vel_k3(i)) * 2.0
        vel_tmp(i) = (vel_tmp(i) + vel_k1(i) + vel_k4(i)) / 6.0
        GP_VELOCITY(i, 0) = GP_VELOCITY(i, 2) + vel_tmp(i)
      end do

************************************************************************
*       GP_ALTITUDE(0) = GP_ALTITUDE(2) +
*    &              (1/6)(alt_k1 + 2*(alt_k2 + alt_k3) + alt_k4)
************************************************************************

      GP_ALTITUDE(0) = GP_ALTITUDE(2) +
     &              (alt_k1 + 2.0*(alt_k2 + alt_k3) + alt_k4) / 6.0

*** establish the "final" rotation matrix ***
```

```
      GP_ROTATION(1, 1) =  0
      GP_ROTATION(1, 2) =  G_ROTATION(3, 0)
      GP_ROTATION(1, 3) = -G_ROTATION(2, 0)
      GP_ROTATION(2, 1) = -G_ROTATION(3, 0)
      GP_ROTATION(2, 2) =  0
      GP_ROTATION(2, 3) =  G_ROTATION(1, 0)
      GP_ROTATION(3, 1) =  G_ROTATION(2, 0)
      GP_ROTATION(3, 2) = -G_ROTATION(1, 0)
      GP_ROTATION(3, 3) =  0

**********************************************************************
*  3) Determine if the engines should be switched on or off.
**********************************************************************

*** range check the current altitude ****

      call RANGE_CHECK(GP_ALTITUDE(0), K$GP_ALTITUDE$LB,
     &                 K$GP_ALTITUDE$UB, 'GP', K$GP_ALTITUDE$NAME)

*** range check the current x-axis vehicle velocity ****

      call RANGE_CHECK(GP_VELOCITY(1, 0), K$GP_VELOCITY$LB,
     &                 K$GP_VELOCITY$UB, 'GP', K$GP_VELOCITY$NAME)

*** ***

      if (AE_SWITCH .EQ. K$AXIAL_ENGINES_ARE_OFF) then
        if (RE_SWITCH .EQ. K$ROLL_ENGINES_ARE_ON) then
          if (TD_SENSED .EQ. K$TOUCH_DOWN_NOT_SENSED) then
            if (GP_ALTITUDE(0) .LE. ENGINES_ON_ALTITUDE) then
              AE_SWITCH = K$AXIAL_ENGINES_ARE_ON
              FRAME_ENGINES_IGNITED = FRAME_COUNTER
            end if
          end if
        end if
      else
***                              the axial engines are "on" ***

        if (TD_SENSED .EQ. K$TOUCH_DOWN_SENSED) then
          AE_SWITCH = K$AXIAL_ENGINES_ARE_OFF
          RE_SWITCH = K$ROLL_ENGINES_ARE_OFF
        else
***                              touch down "not sensed" ***

***
*   v1 PR#27 Item 3.  Included a MAX function.
***
*         if (GP_ALTITUDE(0) .LE. DROP_HEIGHT) then
*           temp = 2*GRAVITY*GP_ALTITUDE(0)
*           call NEG_VALUE_CHECK(temp, 'GP')
```

```
          if (GP_ALTITUDE(0) .LE. DROP_HEIGHT) then
            temp = 2 * GRAVITY * MAX(0.0D0, GP_ALTITUDE(0))
***
*   v1 PR#27 End Change
***

            if (sqrt(temp)+GP_VELOCITY(1,0) .LE.
     &                   MAX_NORMAL_VELOCITY) then
              AE_SWITCH = K$AXIAL_ENGINES_ARE_OFF
              RE_SWITCH = K$ROLL_ENGINES_ARE_OFF
            end if
          end if
        end if
      end if

*************************************************************************
*  4) Compute the current velocity error.
*************************************************************************

*** compute the optimal velocity ***

*** convert GP_ALTITUDE from meters to kilometers ***

      cur_altitude = GP_ALTITUDE(0) / 1000.0

      do i = 1, 100
        if (CONTOUR_ALTITUDE(i) .EQ. cur_altitude) then

*** found an exact match in the table ***

          optimal_velocity = CONTOUR_VELOCITY(i)

          goto 100                         ! early exit

        else if (CONTOUR_ALTITUDE(i) .GT. cur_altitude) then
          if (i .GT. 1) then

*** interpolate between i-1 and i ***

*** check for potential divide by zero condition ***

            call ZERO_CHECK(CONTOUR_ALTITUDE(i)-
     &           CONTOUR_ALTITUDE(i-1), 'GP')

*** interpolation formula ***

            optimal_velocity = CONTOUR_VELOCITY(i-1) +
     &          ((CONTOUR_VELOCITY(i) - CONTOUR_VELOCITY(i-1)) /
     &           (CONTOUR_ALTITUDE(i) - CONTOUR_ALTITUDE(i-1)) *
     &           (cur_altitude - CONTOUR_ALTITUDE(i-1)))
```

```
            goto 100                  ! early exit

          else

*** Extrapolate for altitude < smallest value in table entries ***

*** check for potential divide by zero condition ***

            call ZERO_CHECK(CONTOUR_ALTITUDE(2) -
     &           CONTOUR_ALTITUDE(1), 'GP')

*** Extrapolation formula ***

            optimal_velocity = CONTOUR_VELOCITY(1) -
     &           ((CONTOUR_VELOCITY(2) - CONTOUR_VELOCITY(1)) /
     &            (CONTOUR_ALTITUDE(2) - CONTOUR_ALTITUDE(1)) ) *
     &            (CONTOUR_ALTITUDE(1) - cur_altitude)

            goto 100                          ! early exit

          end if

        else
***                        CONTOUR_ALTITUDE(i) < cur_altitude

          if ((CONTOUR_ALTITUDE(i) .EQ. 0) .OR. (i .EQ. 100)) then

*** Extrapolate for altitude > largest value in table entries ***
*** note, i points to first (lowest) "0" entry in the table ***

*** check for potential divide by zero condition ***

            call ZERO_CHECK(CONTOUR_ALTITUDE(i-1) -
     &           CONTOUR_ALTITUDE(i-2), 'GP')

*** Extrapolation formula ***

            optimal_velocity = CONTOUR_VELOCITY(i-1) +
     &        ((CONTOUR_VELOCITY(i-1) - CONTOUR_VELOCITY(i-2)) /
     &         (CONTOUR_ALTITUDE(i-1) - CONTOUR_ALTITUDE(i-2)) ) *
     &         (cur_altitude - CONTOUR_ALTITUDE(i-1))

            goto 100                          ! early exit

          end if
        end if
      end do

 100  continue
```

```
*** convert optimal_velocity from km/sec to m/sec ***

      optimal_velocity = optimal_velocity * 1000.0

*** compute the velocity error ***

      VELOCITY_ERROR = GP_VELOCITY(1, 0) - optimal_velocity

****************************************************************************
*  5) Determine if the predetermined velocity-altitude contour has
*     been crossed.
****************************************************************************

***
*  v1 Changes for AR#23. Item 17. >= changed to <=
***
      if (GP_ALTITUDE(0) .LE. ENGINES_ON_ALTITUDE) then
        if (CONTOUR_CROSSED .EQ. K$CONTOUR_NOT_CROSSED) then
          if (VELOCITY_ERROR .GE. 0) then
            CONTOUR_CROSSED = K$CONTOUR_CROSSED
          end if
        end if
      end if

***
*  v1 Changes for AR#23. End Change.
***
****************************************************************************
*  6) Determine the current guidance phase.
****************************************************************************

      go to (1010, 1020, 1030, 1040, 2000), GP_PHASE

***
*  v1 Changes for AR#23. Item 33. Added default goto
***
      go to 2000
***
*  v1 Changes for AR#23. End Change.
***

*** GP_PHASE == 1 **************************

 1010   continue

*** trans from 1 to 2 when the "engines on altitude" is reached ***

      if (GP_ALTITUDE(0) .LE. ENGINES_ON_ALTITUDE) then
        GP_PHASE = 2
      end if
```

```
      goto 2000

*** GP_PHASE == 2 **************************

 1020   continue

*** trans from 2 to 5 when touch down is sensed ***

      if (TD_SENSED .EQ. K$TOUCH_DOWN_SENSED) then
        GP_PHASE = 5
      else

*** trans from 2 to 3 when the engines are hot and the chute is released ***

        if (AE_TEMP .EQ. K$HOT) then
          if (CHUTE_RELEASED .EQ. K$CHUTE_RELEASED) then
            GP_PHASE = 3
          end if
        end if
      end if

      goto 2000

*** GP_PHASE == 3 **************************

 1030   continue

*** trans from 3 to 5 when touch down is sensed ***

      if (TD_SENSED .EQ. K$TOUCH_DOWN_SENSED) then
        GP_PHASE = 5
      else

*** trans from 3 to 5 when the TD sensor fails and altitude too low ***
*** trans from 3 to 4 when the TD sensor healthy and altitude too low ***

        if (GP_ALTITUDE(0) .LE. DROP_HEIGHT) then

          if (TDS_STATUS .EQ. K$FAILED) then
            GP_PHASE = 5
          else
***
*   v1 PR#27 Item 3.  Included a MAX function.
***
*             temp = 2 * GRAVITY * GP_ALTITUDE(0)
*             call NEG_VALUE_CHECK(temp, 'GP')
            temp = 2 * GRAVITY * MAX(0.0D0, GP_ALTITUDE(0))
***
*   v1 PR#27 End Change
***
```

```
          if (sqrt(temp)+GP_VELOCITY(1,0) .LE.
     &                 MAX_NORMAL_VELOCITY) then
            GP_PHASE = 4
          end if
        end if
      end if
    end if

    goto 2000

*** GP_PHASE == 4 **************************

 1040   continue

*** trans from 4 to 5 when touch down is sensed ***

    if (TD_SENSED .EQ. K$TOUCH_DOWN_SENSED) then
      GP_PHASE = 5
    else

*** trans from 4 to 5 when the TD sensor fails ***

      if (TDS_STATUS .EQ. K$FAILED) then
        GP_PHASE = 5
      end if
    end if

 2000   continue

************************************************************************
*  7) Determine the appropriate axial engine control law parameter index.
************************************************************************

*** Note, the optimal_velocity is computed above during the computing of
***     the current velocity error.

    if (CL .EQ. K$FIRST) then
      if (optimal_velocity .EQ. DROP_SPEED) then
        if (GP_VELOCITY(1, 0) .LT. DROP_SPEED) then
          CL = K$SECOND
          TE_INTEGRAL = 0
        end if
      end if
    end if

   return
   end

*** end of subroutine GP ***********************************************
```

```
************************************************************************
*  Title: DERIV_ATT
*  Facility:       Pluto
*  Abstract:
*         Compute the derivative of the vehicle attitude.
*         rate-of-change = deriv_att(attitude, i)
*
*  Arguments:
*         result   array (1..3, 1..3) of real*8  The computed derivative.
*         att      array (1..3, 1..3) of real*8  The attitude structure
*      index    integer*4                  The history index
*
*  Revision History:
*         v0    15-sep-1994  Rob Angellatta (RKA) Original.
*      v1    01-Dec-1994  Philip Morris (PEM)
************************************************************************

      subroutine DERIV_ATT(result, att, index)

      implicit none

*** define arguments ***

      real*8                 att(1:3,1:3)
      real*8                 result(1:3,1:3)

      integer*4   index

*** include the global common stores ***

      include 'sensor_output.for'

***
*  v1 Changes for AR#23. Item 18. Replaced pv, qv, rv with rotation variables
***

      result(1,1) = G_ROTATION(3,index) * att(2,1) +
     &    (-G_ROTATION(2,index) * att(3,1))
      result(1,2) = G_ROTATION(3,index) * att(2,2) +
     &    (-G_ROTATION(2,index) * att(3,2))
      result(1,3) = G_ROTATION(3,index) * att(2,3) +
     &    (-G_ROTATION(2,index) * att(3,3))

      result(2,1) = (-G_ROTATION(3,index) * att(1,1)) +
     &    (G_ROTATION(1,index) * att(3,1))
      result(2,2) = (-G_ROTATION(3,index) * att(1,2)) +
     &    (G_ROTATION(1,index) * att(3,2))
      result(2,3) = (-G_ROTATION(3,index) * att(1,3)) +
     &    (G_ROTATION(1,index) * att(3,3))
```

```
      result(3,1) = G_ROTATION(2,index) * att(1,1) +
     &    (-G_ROTATION(1,index) * att(2,1))
      result(3,2) = G_ROTATION(2,index) * att(1,2) +
     &    (-G_ROTATION(1,index) * att(2,2))
      result(3,3) = G_ROTATION(2,index) * att(1,3) +
     &    (-G_ROTATION(1,index) * att(2,3))

***
*  v1 Changes for AR#23. End Change.
***
      return
      end

***** end of subroutine DERIV_ATT ***************************************

*************************************************************************
*  Title:  DERIV_VEL
*  Facility:     Pluto
*  Abstract:
*        Compute the derivative of the vehicle velocity.
*        rate-of-change = deriv_vel(velocity, attitude, i)
*
*  Arguments:
*        result   array (1..3) of real*8        The computed derivative.
*        vel      array (1..3) of real*8        The velocity vector
*        att      array (1..3, 1..3) of real*8  The attitude structure
*     index    integer*4                  The history index
*
*  Revision History:
*        v0    15-sep-1994  Rob Angellatta (RKA) Original.
*     v1    01-Dec-1994  Philip Morris (PEM)
*************************************************************************

      subroutine DERIV_VEL(result,vel,att,index)

      implicit none

*** define arguments ***

      real*8              att(1:3,1:3)
      real*8              result(1:3)
      real*8              vel(1:3)
      integer*4   index

*** include the global common stores ***

      include 'guidance_state.for'
      include 'sensor_output.for'
      include 'run_parameters.for'

***
```

```
*  v1 Changes for AR#23. Item 19. Relpaced pv, qv, rv with rotation variables
***
*** declare local variables ***

      real*8                    temp(3)

*** execution begins here ***

***
*  v1 Changes for AR#23. Item 20. Changed index values for temp
***
      temp(1) = TDLR_VELOCITY(1,index) - vel(1)
      temp(2) = TDLR_VELOCITY(2,index) - vel(2)
      temp(3) = TDLR_VELOCITY(3,index) - vel(3)

      result(1) =  G_ROTATION(3,index)*vel(2) +
     &            (-G_ROTATION(2,index)*vel(3)) +
     &            GRAVITY * att(1,3) + A_ACCELERATION(1,index) +
     &            K_MATRIX(1,1,index) * temp(1) +
     &            K_MATRIX(1,2,index) * temp(2) +
     &            K_MATRIX(1,3,index) * temp(3)


      result(2) = -G_ROTATION(3,index)*vel(1) +
     &            G_ROTATION(1,index)*vel(3) +
     &            GRAVITY * att(2,3) + A_ACCELERATION(2,index) +
     &            K_MATRIX(2,1,index) * temp(1) +
     &            K_MATRIX(2,2,index) * temp(2) +
     &            K_MATRIX(2,3,index) * temp(3)


      result(3) =  G_ROTATION(2,index)*vel(1) +
     &            (-G_ROTATION(1,index)*vel(2)) +
     &            GRAVITY * att(3,3) + A_ACCELERATION(3,index) +
     &            K_MATRIX(3,1,index) * temp(1) +
     &            K_MATRIX(3,2,index) * temp(2) +
     &            K_MATRIX(3,3,index) * temp(3)


***
*  v1 Changes for AR#23. End Change.
***
      return
      end

***** end of subroutine DERIV_VEL **************************************

************************************************************************
*  Title: DERIV_ALT
*  Facility:      Pluto
*  Abstract:
```

```
*         Compute the derivative of the vehicle altitude.
*         rate-of-change = deriv_att(attitude, index)
*
*  Arguments:
*         result  real*8                          The computed derivative.
*         alt        real*8                       The altitude
*         vel     array (1..3) of real*8       The velocity vector
*         att     array (1..3, 1..3) of real*8  The attitude structure
*      index   integer*4                  The history index
*
*  Revision History:
*         v0    15-sep-1994  Rob Angellatta (RKA) Original.
**********************************************************************

      subroutine DERIV_ALT(result,alt,vel,att,index)

      implicit none

*** define arguments ***

      real*8                 alt
      real*8                 att(1:3,1:3)
      real*8                 result
      real*8                 vel(1:3)
      integer*4   index

*** include the global common stores ***

      include 'guidance_state.for'
      include 'sensor_output.for'

*** execution begins here ***

      result =  -att(1,3)*vel(1) + (-att(2,3)*vel(2)) +
    &          (-att(3,3)*vel(3)) +
    &          K_ALT(index)*(AR_ALTITUDE(index) - alt)

      return
      end

***** end of subroutine DERIV_ALT **************************************

**********************************************************************
*  Title: MULT_ATT
*  Facility:        Pluto
*  Abstract:
*         Multiply a 3x3 array by a scaler, result -> 3x3 array.
*      mat = mat * scaler
*
*  Arguments:
*         att     array (1..3, 1..3) of real*8  The attitude structure
```

```
*       factor       real*8                              The scalar multiplier
*
*  Revision History:
*           v0    15-sep-1994  Rob Angellatta (RKA) Original.
*       v1    01-Dec-1994  Philip Morris (PEM)
***********************************************************************

       subroutine MULT_ATT(att,factor)

       implicit none

*** define arguments ***

       real*8                    att(1:3,1:3)
       real*8                    factor

*** execution begins here ***

***
*  v1 Changes for AR#23. Item 22. Changed att index values
***
       att(1,1)      = att(1,1) * factor
       att(1,2)      = att(1,2) * factor
       att(1,3)      = att(1,3) * factor

       att(2,1)      = att(2,1) * factor
       att(2,2)      = att(2,2) * factor
       att(2,3)      = att(2,3) * factor

       att(3,1)      = att(3,1) * factor
       att(3,2)      = att(3,2) * factor
       att(3,3)      = att(3,3) * factor

       return
       end

***
*  v1 Changes for AR#23. End Change.
***
***** end of subroutine MULT_ATT ***************************************

***********************************************************************
*  Title: MULT_VEL
*  Facility:        Pluto
*  Abstract:
*           Multiply a 1x3 vector by a scaler, result -> vector
*       vector = vector * scaler
*
*  Arguments:
*           att        array (1..3) of real*8                The velocity structure
*       factor       real*8                                  The scalar multiplier
```

```
*
*  Revision History:
*         v0    15-sep-1994  Rob Angellatta (RKA) Original.
***********************************************************************

      subroutine MULT_VEL(vel, factor)

      implicit none

*** define arguments ***

      real*8                  vel(1:3)
      real*8                  factor

*** execution begins here ***

      vel(1)        = vel(1) * factor
      vel(2)        = vel(2) * factor
      vel(3)        = vel(3) * factor

      return
      end

***** end of subroutine MULT_VEL ***************************************

***********************************************************************
*  Title: AVG_ATT
*  Facility:        Pluto
*  Abstract:
*         Add two 3x3 array's
*      result = (mat1 + mat2 / 2)
*
*  Arguments:
*         result    array (1..3, 1..3) of real*8  the result attitude structure
*         att_1   array (1..3, 1..3) of real*8  an attitude structure
*         att_2   array (1..3, 1..3) of real*8  an attitude structure
*
*  Revision History:
*         v0    15-sep-1994  Rob Angellatta (RKA) Original.
*      v1    01-Dec-1994  Philip Morris (PEM)
***********************************************************************

      subroutine AVG_ATT(result, att_1, att_2)

      implicit none

*** define arguments ***

      real*8                  result(1:3, 1:3)
      real*8                  att_1(1:3, 1:3),  att_2(1:3, 1:3)
```

```
*** execution begins here ***

***
*  v1 Changes for PR#23. Item 14 & 21. Function no longer averages but
*                         divides one element by 2
***
***
*  v2 Changes for PR#24. Item 1. Changed index values.
***
*      result(1,1)         = att_1(1,1) + att_2(1,1) / 2.0
*      result(1,2)         = att_1(1,1) + att_2(1,1) / 2.0
*      result(1,3)         = att_1(1,1) + att_2(1,1) / 2.0
*
*      result(2,1)         = att_1(2,1) + att_2(2,1) / 2.0
*      result(2,2)         = att_1(2,1) + att_2(2,1) / 2.0
*      result(2,3)         = att_1(2,1) + att_2(2,1) / 2.0
*
*      result(3,1)         = att_1(3,1) + att_2(3,1) / 2.0
*      result(3,2)         = att_1(3,1) + att_2(3,1) / 2.0
*      result(3,3)         = att_1(3,1) + att_2(3,1) / 2.0
      result(1,1)  = att_1(1,1) + att_2(1,1) / 2.0
      result(1,2)  = att_1(1,2) + att_2(1,2) / 2.0
      result(1,3)  = att_1(1,3) + att_2(1,3) / 2.0

      result(2,1)  = att_1(2,1) + att_2(2,1) / 2.0
      result(2,2)  = att_1(2,2) + att_2(2,2) / 2.0
      result(2,3)  = att_1(2,3) + att_2(2,3) / 2.0

      result(3,1)  = att_1(3,1) + att_2(3,1) / 2.0
      result(3,2)  = att_1(3,2) + att_2(3,2) / 2.0
      result(3,3)  = att_1(3,3) + att_2(3,3) / 2.0

***
*  v2 Changes for PR#24. End Change.
***
***
*  v1 Changes for PR#23. End Change.
***
      return
      end

***** end of subroutine AVG_ATT ****************************************

**********************************************************************
*  Title: AVG_VEL
*  Facility:      Pluto
*  Abstract:
*        Add two 1x3 vector's
*      result = (vec1 + vec2 / 2)
*
*  Arguments:
```

```
*          result    array (1..3) of real*8  the result velocity structure
*          vel_1   array (1..3) of real*8  an velocity structure
*          vel_2   array (1..3) of real*8  an velocity structure
*
*  Revision History:
*          v0    15-sep-1994  Rob Angellatta (RKA) Original.
*      v1    01-Dec-1994  Philip Morris (PEM)
**********************************************************************

       subroutine AVG_VEL(result, vel_1, vel_2)

       implicit none

*** define arguments ***

       real*8                  result(1:3)
       real*8                  vel_1(1:3),  vel_2(1:3)

*** execution begins here ***

***
*  v1 Changes for AR#23. Item 14 & 21. Function no longer averages but
*                              divides one element by 2
***
       result(1)    = vel_1(1) + vel_2(1) / 2.0
       result(2)    = vel_1(2) + vel_2(2) / 2.0
       result(3)    = vel_1(3) + vel_2(3) / 2.0

***
*  v1 Changes for AR#23. End Change.
***
       return
       end

***** end of subroutine AVG_VEL *****************************************

***** end of module GP ***************************************************
```

```
***********************************************************************
* Module:       GPSF.FOR
* Facility:     Pluto
* P-Spec:       2
* Abstract:
*         This module contains the entry for the guidance processing
*         subframe.
*
* List of Routines:
*         subroutine GPSF
***********************************************************************

***********************************************************************
* Title: GPSF
* Facility:     Pluto
* Abstract:
*         This routine provides control of the Guidance Processing SubFrame
*         processing.
*
* Arguments:    None
*
* Revision History:
*         v0    15-sep-1994  Rob Angellatta (RKA) Original.
***********************************************************************

      subroutine GPSF

      implicit none

*** execution begins here ***

      call GCS_SIM_RENDEZVOUS
      call GP
      call CP

      return
      end

***** end of module GPSF.FOR ********************************************
```

```
**********************************************************************
*  Module:        GSP.FOR
*  Facility:      Pluto
*  P-Spec:        1.4
*  Abstract:
*         This module contains the implementation of the functional
*         requirements for GSP.
*
*  List of Routines:
*         subroutine GSP
**********************************************************************

**********************************************************************
*  Title: GSP
*  Facility:      Pluto
*  Abstract:
*         1) maintain the history of the vehicle rotation rates
*         2) determine the operational status of the gyroscope sensors
*         3) Report the current vehicle rotation rates along each of the
*            vehicle's three axes.
*
*  Arguments:   None
*  Revision History:
*         v0    15-sep-1994  Rob Angellatta (RKA) Original.
*      v1    30-Nov-1994  Philip Morris (PEM)
**********************************************************************

      subroutine GSP

      implicit none

*** include the global common stores ***

      include 'external.for'
      include 'guidance_state.for'
      include 'sensor_output.for'
      include 'run_parameters.for'

*** include constant definitions ***

      include 'constants.for'

*** declare local variables ***

***
*  v1 Changes for AR#23. Item 23. counter type changed from real*8 to integer*2
***
      integer*2   counter
      real*8              temp

      integer*4   i
```

```
***
*  v1 Changes for AR#23. End Change.
***
************************************************************************
*  1) Maintain the history of the vehicle rotation rates by "rotating
*  variables."
************************************************************************

         G_ROTATION(1, 4) = G_ROTATION(1, 3)
      G_ROTATION(1, 3) = G_ROTATION(1, 2)
      G_ROTATION(1, 2) = G_ROTATION(1, 1)
      G_ROTATION(1, 1) = G_ROTATION(1, 0)

      G_ROTATION(2, 4) = G_ROTATION(2, 3)
      G_ROTATION(2, 3) = G_ROTATION(2, 2)
      G_ROTATION(2, 2) = G_ROTATION(2, 1)
      G_ROTATION(2, 1) = G_ROTATION(2, 0)

      G_ROTATION(3, 4) = G_ROTATION(3, 3)
      G_ROTATION(3, 3) = G_ROTATION(3, 2)
      G_ROTATION(3, 2) = G_ROTATION(3, 1)
      G_ROTATION(3, 1) = G_ROTATION(3, 0)

************************************************************************
*  2) determine the operational status of the gyroscope sensors.
************************************************************************

      G_STATUS = K$HEALTHY

************************************************************************
*  3) Report the current vehicle rotation rates along each of the
*  vehicle's three axes.
************************************************************************

*** range check the atmospheric temperature ****

      call RANGE_CHECK(ATMOSPHERIC_TEMP,K$ATMOSPHERIC_TEMP$LB,
    &              K$ATMOSPHERIC_TEMP$UB,'GSP', K$ATMOSPHERIC_TEMP$NAME)

************************************************************************
*  The raw sensor data stored in G_COUNTER represents the vehicle rate
*  of rotation about a specific axis.  The sensor data is
*  stored in a modified sign magnitude format.  The lower 14-bits
*  represent the magnitude of the rotation and the most significant
*  bit (bit 15) represents the sign.  Bit 14 is not used.  A
*  positive value of G_COUNTER indicates a positive rotation about
*  the vehicle axis consistent with a right handed coordinate system,
*  while a negative value indicates a negative rotation consistent
*  with a right handed coordinate system.
************************************************************************
```

```
      temp = (G3 * ATMOSPHERIC_TEMP) + (G4 * ATMOSPHERIC_TEMP**2)

      do 100 i = 1, 3

*** convert the raw sensor ***

************************************************************************
*  Convert the raw sensor data from the modified sign magnitude
*  format into an appropriate format for use by the target CPU, in
*  this case two's complement.  Positive values are represented in
*  the same fashion in sign magnitude and two's complement, however,
*  negative sensor values must be massaged.
*
*  Transfer the magitude of the rotation from G_COUNTER to the local
*  data element named counter by masking bits 14 and 15 from
*  G_COUNTER. If G_COUNTER bit 15 is clear, the data element counter
*  now contains the properly converted value. If G_COUNTER bit 15 is
*  set, the value of data element counter must be negated.
************************************************************************

*** clear the two most significant bits (bits 15,14) ***

        counter = IAND(G_COUNTER(i), '3FFF'X)

*** if the bit was set, then convert value to two's complement ***

        if (BTEST(G_COUNTER(i), 15) .EQ. .TRUE.) then
          counter = 0 - counter
        end if

*** now, compute the vehicle rotation from the sensor data ***

        G_ROTATION(i, 0) = G_OFFSET(i) +
     &               (G_GAIN_0(i) + temp) * counter

 100    continue

      return
      end

***** end of module GSP.FOR ********************************************
```

```
**********************************************************************
* Module:       GUIDANCE_STATE.FOR
* Facility:     Pluto
* Abstract:
*       This module contains the data definitions for the
*       global common data store named GUIDANCE_STATE.
*
* Revision History:
*       v0   15-sep-1994  Rob Angellatta (RKA) Original.
**********************************************************************

**** COMMON block definition ***

      COMMON /GUIDANCE_STATE/
     &          A_STATUS,
     &          AE_STATUS,
     &          AE_SWITCH,
     &          AE_TEMP,
     &          AR_STATUS,
     &          C_STATUS,
     &          CL,
     &          CONTOUR_CROSSED,
     &          FRAME_BEAM_UNLOCKED,
     &          FRAME_ENGINES_IGNITED,
     &          G_STATUS,
     &          GP_ALTITUDE,
     &          GP_ATTITUDE,
     &          GP_PHASE,
     &          GP_ROTATION,
     &          GP_VELOCITY,
     &          INTERNAL_CMD,
     &          K_ALT,
     &          K_MATRIX,
     &          PE_INTEGRAL,
     &          RE_STATUS,
     &          RE_SWITCH,
     &          TDLR_STATE,
     &          TDLR_STATUS,
     &          TDS_STATUS,
     &          TE_INTEGRAL,
     &          TE_LIMIT,
     &          THETA,
     &          TS_STATUS,
     &          VELOCITY_ERROR,
     &          YE_INTEGRAL

*** data type declarations ***

      logical*1  A_STATUS(1:3, 0:3)
      logical*1  AE_STATUS
      logical*1  AE_SWITCH
```

```
integer*2   AE_TEMP
logical*1   AR_STATUS(0:4)
logical*1   C_STATUS
integer*2   CL
logical*1   CONTOUR_CROSSED
integer*4   FRAME_BEAM_UNLOCKED(1:4)
integer*4   FRAME_ENGINES_IGNITED
logical*1   G_STATUS
real*8              GP_ALTITUDE(0:4)
real*8              GP_ATTITUDE(1:3, 1:3, 0:4)
integer*4   GP_PHASE
real*8              GP_ROTATION(1:3, 1:3)
real*8              GP_VELOCITY(1:3, 0:4)
real*8              INTERNAL_CMD(1:3)
integer*4   K_ALT(0:4)
integer*4   K_MATRIX(1:3, 1:3, 0:4)
real*8              PE_INTEGRAL
logical*1   RE_STATUS
logical*1   RE_SWITCH
logical*1   TDLR_STATE(1:4)
logical*1   TDLR_STATUS(1:4)
logical*1   TDS_STATUS
real*8              TE_INTEGRAL
real*8              TE_LIMIT
real*8              THETA
logical*1   TS_STATUS(1:2)
real*8              VELOCITY_ERROR
real*8              YE_INTEGRAL
```

***** end of module GUIDANCE_STATE.FOR **********************************

```
**********************************************************************
*  Module:        PLUTO.FOR
*  Facility:      Pluto
*  P-Spec:        0
*  Abstract:
*          This module contains the main routine for the Pluto
*          implementation.
*
*  List of Routines:
*          program Pluto
**********************************************************************

**********************************************************************
*  Title: PLUTO
*  Facility:      Pluto
*  Abstract:
*          This is the main routine for the Pluto implementation.
*
*  Arguments:   None
*
*  Revision History:
*          v0    15-sep-1994  Rob Angellatta (RKA) Original.
*      v1    30-Nov-1994  Philip Morris (PEM)
*      v2    15-Feb-1995  Philip Morris (PEM)
**********************************************************************

      program PLUTO

      implicit none

*** include the global common stores ***

      include 'guidance_state.for'

*** execution begins here ***

***********************************************************************
*  Simply loop through the three subframes until done
***********************************************************************

***
*  v2 Changes for AR#26. Item 2. Remove dead code.
***
* 100   continue
***
*  v2 Changes for AR#26. End Change.
***

***
*  v1 Changes for AR#23. Item 6&7. Added DO WHILE loop to remove gotos
***
```

```
*** stop when gp_phase = 5 ***

      do while (GP_PHASE .NE. 5)

*** execute the sensor processing subframe ***

          call SPSF

*** execute the guidance processing subframe ***

          call GPSF

*** execute the control law processing subframe ***

          call CLPSF

      end do

***
*  v1 Changes for AR#23. End Change.
***
       stop
       end

***** end of module PLUTO.FOR ******************************************
```

```
***********************************************************************
*  Module:       RECLP.FOR
*  Facility:     Pluto
*  P-Spec:       3.4
*  Abstract:
*         This module contains the implementation of the functional
*         requirements for RECLP.
*
*  List of Routines:
*         subroutine RECLP
***********************************************************************

***********************************************************************
*  Title: RECLP
*  Facility:     Pluto
*  Abstract:
*         1) determine the current operational status of the roll engines.
*         2) generate the appropriate roll engine command.
*
*  Arguments:   None
*
*  Revision History:
*         v0    15-sep-1994  Rob Angellatta (RKA) Original.
***********************************************************************

      subroutine RECLP

      implicit none

*** include the global common stores ***

      include 'external.for'
      include 'guidance_state.for'
      include 'sensor_output.for'
      include 'run_parameters.for'

*** include constant definitions ***

      include 'constants.for'

************************************************************************
*  1) Determine the current operational status of the roll engines.
************************************************************************

      RE_STATUS = K$HEALTHY

************************************************************************
*  2) Generate the appropriate roll engine command.
************************************************************************

      if (RE_SWITCH .EQ. K$ROLL_ENGINES_ARE_OFF) then
```

```
          RE_CMD = K$OFF + K$CW
        else

*** range check the x-axis vehicle rotation rate ***

          call RANGE_CHECK(G_ROTATION(1, 0), K$G_ROTATION$LB,
     &                K$G_ROTATION$UB, 'RECLP', K$G_ROTATION$NAME)

*** range check the x-axis vehicle rotation displacement ***

          call RANGE_CHECK(THETA, K$THETA$LB,
     &                K$THETA$UB, 'RECLP', K$THETA$NAME)

*********************************************************************
* The roll engine command consists of two components: an
* intensity, and a direction.  Taking into account the command data
* encoding, the possible intensities are:  Off (0), Minimum (2),
* Intermediate (4), and Maximum (6), and the possible directions
* are CounterClockwise (0) and Clockwise (1).
*
* Both roll engine command components are determined from the
* current value of the vehicle's roll rate and rotational
* displacement about the x-axis.
*
* Employing Euler's method for differential equations, compute the
* current x-axis angular displacement, theta.
*********************************************************************

          THETA  = THETA + G_ROTATION(1, 0) * DELTA_T

*** range check the theta again before use ***

          call RANGE_CHECK(THETA, K$THETA$LB,
     &                K$THETA$UB, 'RECLP', K$THETA$NAME)

*********************************************************************
* From figure 5.2 "Graph for Deriving Roll Engine Commands" of the
* GCS development specifications, determine the appropriate roll
* engine intensity and direction.
*********************************************************************

*** check case when theta = 0 ***

          if (THETA .EQ. 0) then
            if (G_ROTATION(1, 0) .GT. P4) then
               RE_CMD = K$MAXIMUM + K$CW
             else if (G_ROTATION(1, 0) .LT. -P4) then
               RE_CMD = K$MAXIMUM + K$CCW
             else
               RE_CMD = K$OFF + K$CW
            end if
```

```
*** check first and fourth quadrants ***

      else if (THETA .GT. 0) then
        if (THETA .LE. THETA1) then

          if (G_ROTATION(1, 0) .GT. P2) then
            RE_CMD = K$MAXIMUM + K$CW
          else if (G_ROTATION(1, 0) .GT. P1) then
            RE_CMD = K$INTERMEDIATE + K$CW
          else if (G_ROTATION(1, 0) .GE. -P4) then
            RE_CMD = K$OFF + K$CW
          else
            RE_CMD = K$MAXIMUM + K$CCW
          end if

        else if (THETA .LE. THETA2) then

          if (G_ROTATION(1, 0) .GT. P2) then
            RE_CMD = K$MAXIMUM + K$CW
          else if (G_ROTATION(1, 0) .GT. P1) then
            RE_CMD = K$INTERMEDIATE + K$CW
          else if (G_ROTATION(1, 0) .GT. 0.0) then
            RE_CMD = K$MINIMUM + K$CW
          else if (G_ROTATION(1, 0) .GE. -P4) then
            RE_CMD = K$OFF + K$CW
          else
            RE_CMD = K$MAXIMUM + K$CCW
          end if

        else
*                                THETA > THETA2

          if (G_ROTATION(1, 0) .GT. -P3) then
            RE_CMD = K$MAXIMUM + K$CW
          else if (G_ROTATION(1, 0) .GE. -P4) then
            RE_CMD = K$OFF + K$CW
          else
            RE_CMD = K$MAXIMUM + K$CCW
          end if

        end if

*** check second and third quadrants ***

      else
*                                THETA .LT. 0
        if (THETA .GE. -THETA1) then

          if (G_ROTATION(1, 0) .GT. p4) then
            RE_CMD = K$MAXIMUM + K$CW
```

```
            else if (G_ROTATION(1, 0) .GE. -P1) then
              RE_CMD = K$OFF + K$CW
            else if (G_ROTATION(1, 0) .GE. -P2) then
              RE_CMD = K$INTERMEDIATE + K$CCW
            else
              RE_CMD =          K$MAXIMUM + K$CCW
            end if

          else if (THETA .GE. -THETA2) then

            if (G_ROTATION(1, 0) .GT. P4) then
              RE_CMD = K$MAXIMUM + K$CW
            else if (G_ROTATION(1, 0) .GE. 0.0) then
              RE_CMD =          K$OFF + K$CW
            else if (G_ROTATION(1, 0) .GE. -P1) then
              RE_CMD = K$MINIMUM + K$CCW
            else if (G_ROTATION(1, 0) .GE. -P2) then
              RE_CMD = K$INTERMEDIATE + K$CCW
            else
              RE_CMD = K$MAXIMUM + K$CCW
            end if

          else
*                                  THETA < -THETA2

            if (G_ROTATION(1, 0) .GT. P4) then
              RE_CMD = K$MAXIMUM + K$CW
            else if (G_ROTATION(1, 0) .GE. P3) then
              RE_CMD = K$OFF + K$CW
            else
              RE_CMD =          K$MAXIMUM + K$CCW
            end if

          end if
        end if
      end if

      return

      end

***** end of module RECLP.FOR *******************************************
```

```
***********************************************************************
*  Module:       RUN_PARAMETERS.FOR
*  Facility:     Pluto
*  Abstract:
*        This module contains the data definitions for the
*        global common data store named RUN_PARAMETERS.
*
*  Revision History:
*        v0    15-sep-1994  Rob Angellatta (RKA) Original.
***********************************************************************

*** COMMON block definition ***

      COMMON          /RUN_PARAMETERS/
     &          A_BIAS,
     &          A_GAIN_0,
     &          A_SCALE,
     &          ALPHA_MATRIX,
     &          AR_FREQUENCY,
     &          COMM_SYNC_PATTERN,
     &          CONTOUR_ALTITUDE,
     &          CONTOUR_VELOCITY,
     &          DELTA_T,
     &          DROP_HEIGHT,
     &      DROP_SPEED,
     &          ENGINES_ON_ALTITUDE,
     &          FULL_UP_TIME,
     &          G1,
     &          G2,
     &          G3,
     &          G4,
     &          G_GAIN_0,
     &          G_OFFSET,
     &          GA,
     &          GAX,
     &          GP1,
     &          GP2,
     &          GPY,
     &          GQ,
     &          GR,
     &          GRAVITY,
     &          GV,
     &          GVE,
     &          GVEI,
     &          GVI,
     &          GW,
     &          GWI,
     &          M1,
     &          M2,
     &          M3,
     &          M4,
```

```
     &        MAX_NORMAL_VELOCITY,
     &             OMEGA,
     &             P1,
     &             P2,
     &             P3,
     &             P4,
     &             PE_MAX,
     &             PE_MIN,
     &             T1,
     &             T2,
     &             T3,
     &             T4,
     &             TDLR_ANGLES,
     &             TDLR_GAIN,
     &             TDLR_LOCK_TIME,
     &             TDLR_OFFSET,
     &             TE_DROP,
     &             TE_INIT,
     &             TE_MAX,
     &             TE_MIN,
     &             THETA1,
     &             THETA2,
     &             YE_MAX,
     &             YE_MIN

*** data type declarations ***

      real*8         A_BIAS(1:3)
      real*8         A_GAIN_0(1:3)
      integer*4   A_SCALE
      real*8         ALPHA_MATRIX(1:3, 1:3)
      real*8         AR_FREQUENCY
      integer*2   COMM_SYNC_PATTERN
      real*8         CONTOUR_ALTITUDE(1:100)
      real*8         CONTOUR_VELOCITY(1:100)
      real*8         DELTA_T
      real*8         DROP_HEIGHT
      real*8      DROP_SPEED
      real*8         ENGINES_ON_ALTITUDE
      real*8         FULL_UP_TIME
      real*8         G1
      real*8         G2
      real*8         G3
      real*8         G4
      real*8         G_GAIN_0(1:3)
      real*8         G_OFFSET(1:3)
      real*8         GA
      real*8         GAX
      real*8         GP1
      real*8         GP2
      real*8         GPY
```

```
real*8       GQ(1:2)
real*8       GR(1:2)
real*8       GRAVITY
real*8       GV(1:2)
real*8       GVE
real*8       GVEI(1:2)
real*8       GVI(1:2)
real*8       GW(1:2)
real*8       GWI(1:2)
integer*2  M1
integer*2  M2
integer*2  M3
integer*2  M4
real*8    MAX_NORMAL_VELOCITY
real*8       OMEGA
real*8       P1
real*8       P2
real*8       P3
real*8       P4
real*8       PE_MAX(1:2)
real*8       PE_MIN(1:2)
real*8       T1
real*8       T2
real*8       T3
real*8       T4
real*8       TDLR_ANGLES(1:3)
real*8       TDLR_GAIN
real*8       TDLR_LOCK_TIME
real*8       TDLR_OFFSET
real*8       TE_DROP
real*8       TE_INIT
real*8       TE_MAX(1:2)
real*8       TE_MIN(1:2)
real*8       THETA1
real*8       THETA2
real*8       YE_MAX(1:2)
real*8       YE_MIN(1:2)

***** end of module RUN_PARAMETERS.FOR *********************************
```

```
*************************************************************************
*  Module:        SENSOR_OUTPUT.FOR
*  Facility:      Pluto
*  Abstract:
*         This module contains the data definitions for the
*         global common data store named EXTERNAL.
*
*  Revision History:
*         v0    15-sep-1994  Rob Angellatta (RKA) Original.
*************************************************************************

*** COMMON block definition ***

      COMMON /SENSOR_OUTPUT/
     &          A_ACCELERATION,
     &          AR_ALTITUDE,
     &          ATMOSPHERIC_TEMP,
     &          G_ROTATION,
     &          TD_SENSED,
     &          TDLR_VELOCITY

*** data type declarations ***

      real*8          A_ACCELERATION(1:3, 0:4)
      real*8          AR_ALTITUDE(0:4)
      real*8          ATMOSPHERIC_TEMP
      real*8          G_ROTATION(1:3, 0:4)
      logical*1   TD_SENSED
      real*8          TDLR_VELOCITY(1:3, 0:4)

***** end of module SENSOR_OUTPUT.FOR **********************************
```

```
***********************************************************************
* Module:       SPSF.FOR
* Facility:     Pluto
* Abstract:
*         This module contains the entry for the sensor processing
*         subframe.
*
* List of Routines:
*         subroutine SPSF
***********************************************************************

***********************************************************************
* Title: SPSF
* Facility:     Pluto
* Abstract:
*         This routine provides control of the Sensor Processing SubFrame
*         processing.
*
* Arguments:    None
*
* Revision History:
*         v0    15-sep-1994  Rob Angellatta (RKA) Original.
***********************************************************************

      subroutine SPSF

      implicit none

*** execution begins here ***

      call GCS_SIM_RENDEZVOUS
      call TSP
      call ARSP
      call ASP
      call GSP
      call TDLRSP
      call TDSP
      call CP

      return
      end

***** end of module SPSF.FOR *******************************************
```

```
**********************************************************************
* Module:       TDLRSP.FOR
* Facility:     Pluto
* P-Spec:       1.5
* Abstract:
*         This module contains the implementation of the functional
*         requirements for TDLRSP.
*
* List of Routines:
*         subroutine TDLRSP
**********************************************************************

**********************************************************************
* Title: TDLRSP
* Facility:     Pluto
* Abstract:
*         1) Maintain the history of the vehicle velocities and the
*            velocity computation indicator
*         2) Determine the operational status of touch down landing radar
*            sensor
*         3) Report the current vehicle velocities along each of the
*            vehicle's three axes
*         4) Report the velocity computation indicators.
*
* Arguments:   None
* Revision History:
*         v0    15-sep-1994  Rob Angellatta (RKA) Original.
*      v1    30-Nov-1994  Philip Morris (PEM)
*      v2    10-JAN-1995  Philip Morris (PEM)
**********************************************************************

      subroutine TDLRSP

      implicit none

*** include the global common stores ***

      include 'external.for'
      include 'guidance_state.for'
      include 'sensor_output.for'
      include 'run_parameters.for'

*** include constant definitions ***

      include 'constants.for'

*** declare local variables ***

      integer*4   i

      real*8              b(4)
```

```
      real*8                pbvX
      real*8                pbvY
      real*8                pbvZ

      real*8                elapsed_time

************************************************************************
*  1) Maintain the history of the vehicle velocities and the
*  velocity computation indicator by "rotating variables."  The data
************************************************************************

      TDLR_VELOCITY(1, 4) = TDLR_VELOCITY(1, 3)
      TDLR_VELOCITY(1, 3) = TDLR_VELOCITY(1, 2)
      TDLR_VELOCITY(1, 2) = TDLR_VELOCITY(1, 1)
      TDLR_VELOCITY(1, 1) = TDLR_VELOCITY(1, 0)

      TDLR_VELOCITY(2, 4) = TDLR_VELOCITY(2, 3)
      TDLR_VELOCITY(2, 3) = TDLR_VELOCITY(2, 2)
      TDLR_VELOCITY(2, 2) = TDLR_VELOCITY(2, 1)
      TDLR_VELOCITY(2, 1) = TDLR_VELOCITY(2, 0)

      TDLR_VELOCITY(3, 4) = TDLR_VELOCITY(3, 3)
      TDLR_VELOCITY(3, 3) = TDLR_VELOCITY(3, 2)
      TDLR_VELOCITY(3, 2) = TDLR_VELOCITY(3, 1)
      TDLR_VELOCITY(3, 1) = TDLR_VELOCITY(3, 0)

      K_MATRIX(1, 1, 4)   = K_MATRIX(1, 1, 3)
      K_MATRIX(1, 2, 4)   = K_MATRIX(1, 2, 3)
      K_MATRIX(1, 3, 4)   = K_MATRIX(1, 3, 3)
      K_MATRIX(2, 1, 4)   = K_MATRIX(2, 1, 3)
      K_MATRIX(2, 2, 4)   = K_MATRIX(2, 2, 3)
      K_MATRIX(2, 3, 4)   = K_MATRIX(2, 3, 3)
      K_MATRIX(3, 1, 4)   = K_MATRIX(3, 1, 3)
      K_MATRIX(3, 2, 4)   = K_MATRIX(3, 2, 3)
      K_MATRIX(3, 3, 4)   = K_MATRIX(3, 3, 3)

      K_MATRIX(1, 1, 3)   = K_MATRIX(1, 1, 2)
      K_MATRIX(1, 2, 3)   = K_MATRIX(1, 2, 2)
      K_MATRIX(1, 3, 3)   = K_MATRIX(1, 3, 2)
      K_MATRIX(2, 1, 3)   = K_MATRIX(2, 1, 2)
      K_MATRIX(2, 2, 3)   = K_MATRIX(2, 2, 2)
      K_MATRIX(2, 3, 3)   = K_MATRIX(2, 3, 2)
      K_MATRIX(3, 1, 3)   = K_MATRIX(3, 1, 2)
      K_MATRIX(3, 2, 3)   = K_MATRIX(3, 2, 2)
      K_MATRIX(3, 3, 3)   = K_MATRIX(3, 3, 2)

      K_MATRIX(1, 1, 2)   = K_MATRIX(1, 1, 1)
      K_MATRIX(1, 2, 2)   = K_MATRIX(1, 2, 1)
      K_MATRIX(1, 3, 2)   = K_MATRIX(1, 3, 1)
      K_MATRIX(2, 1, 2)   = K_MATRIX(2, 1, 1)
      K_MATRIX(2, 2, 2)   = K_MATRIX(2, 2, 1)
```

```
      K_MATRIX(2, 3, 2)   = K_MATRIX(2, 3, 1)
      K_MATRIX(3, 1, 2)   = K_MATRIX(3, 1, 1)
      K_MATRIX(3, 2, 2)   = K_MATRIX(3, 2, 1)
      K_MATRIX(3, 3, 2)   = K_MATRIX(3, 3, 1)

      K_MATRIX(1, 1, 1)   = K_MATRIX(1, 1, 0)
      K_MATRIX(1, 2, 1)   = K_MATRIX(1, 2, 0)
      K_MATRIX(1, 3, 1)   = K_MATRIX(1, 3, 0)
      K_MATRIX(2, 1, 1)   = K_MATRIX(2, 1, 0)
      K_MATRIX(2, 2, 1)   = K_MATRIX(2, 2, 0)
      K_MATRIX(2, 3, 1)   = K_MATRIX(2, 3, 0)
      K_MATRIX(3, 1, 1)   = K_MATRIX(3, 1, 0)
      K_MATRIX(3, 2, 1)   = K_MATRIX(3, 2, 0)
      K_MATRIX(3, 3, 1)   = K_MATRIX(3, 3, 0)

************************************************************************
*  2) Determine the operational status of touch down landing radar sensor.
************************************************************************

      TDLR_STATUS(1) = K$HEALTHY
      TDLR_STATUS(2) = K$HEALTHY
      TDLR_STATUS(3) = K$HEALTHY
      TDLR_STATUS(4) = K$HEALTHY

************************************************************************
*  3) Reporting the current vehicle velocities along each of the
*  vehicle's three axes and reporting the velocity computation
*  indicators.
************************************************************************

************************************************************************
*  3A)  Determine the state of the four radar beams.
*
* The data element TDLR_STATE contains the state of the radar
* beams.
*
* Valid radar beam states are "locked" (value 1) and "unlocked"
* (value 0). The present state of a radar beam is determined from
* the current value of the sensor data and the previous state of
* the radar beam. A sensor measurement of zero indicates that the
* radar beam echo was not received and the radar beam is considered
* to be "unlocked."  A non-zero sensor measurement indicates that a
* radar beam echo was received, but does not imply a radar beam
* state of "locked."  Because, once a radar beam is declared
* "unlocked," it is rendered unusable (remains "unlocked"
* regardless of the sensor data value) for a specified period of
* time.  This waiting period must be implemented in the software.
*
* A beam is deemed "locked" when 1) the current sensor value
* contains a non-zero value and the beam's previous state was
* "locked"; or 2) the current sensor value contains a non-zero
```

```
* value and the beam's previous state was "unlocked" and the
* elapsed time since the beam was determined "unlocked" is greater
* than or equal to the sensor recovery period.
*
* The data element TDLR_LOCK_TIME specifies the unlocked sensor
* recovery (waiting) period. The data element FRAME_BEAM_UNLOCKED
* is updated with the value of the FRAME_COUNTER during the frame
* in which  a radar beam state is first determined as "unlocked."
* The data element DELTA_T specifies in seconds the duration of a
* single frame.  Thus the elapsed time since a radar beam was
* declared "unlocked" can be determined by subtracting the present
* value of FRAME_COUNTER from the value of FRAME_BEAM_UNLOCKED and
* multipling the result by the value of DELTA_T.
*********************************************************************

**** process each radar beam ***

      do 100 i=1,4

        if (TDLR_COUNTER(i) .EQ. 0) then

          if (TDLR_STATE(i) .EQ. K$BEAM_LOCKED) then
            TDLR_STATE(i) = K$BEAM_UNLOCKED
               FRAME_BEAM_UNLOCKED(i) = FRAME_COUNTER

***
*  v2 Changes for AR#24. Item 7. Added else if.
***
*           else
             elseif (TDLR_STATE(i) .EQ. K$BEAM_UNLOCKED) then
***
*  v2 Changes for AR#24. End Change.
***
*                                the beam was unlocked
            elapsed_time = DELTA_T *
     &              (FRAME_COUNTER - FRAME_BEAM_UNLOCKED(i))

            if (elapsed_time .GE. TDLR_LOCK_TIME) then
              FRAME_BEAM_UNLOCKED(i) = FRAME_COUNTER
            end if
          end if

        else
*                            the sensor measurement != 0

          if (TDLR_STATE(i) .EQ. K$BEAM_UNLOCKED) then
            elapsed_time = DELTA_T *
     &              (FRAME_COUNTER - FRAME_BEAM_UNLOCKED(i))

            if (elapsed_time .GE. TDLR_LOCK_TIME) then
              TDLR_STATE(i) = K$BEAM_LOCKED
```

```
          end if
        end if
      end if
 100  continue

************************************************************************
*  3B)  Determine the beam velocities.
************************************************************************

      do 200 i=1,4
        b(i) = TDLR_OFFSET + TDLR_GAIN * TDLR_COUNTER(i)
 200  continue

************************************************************************
*  3C)  Determine the "processed" beam velocities, and
*   4)  Determine the velocity computation indicators.
************************************************************************
************************************************************************
* Compute a "processed" beam velocity for each of the three axes as
* specified by the following table:
*
*  Beams  |          PROCESSED  BEAM  VELOCITIES          | K-MATRIX | Case
* in lock |      pbvX         pbvY          pbvZ     | X   Y   Z | Number
* --------|---------------|---------------|---------------|----------------
*   none  |      0      |      0      |      0      | 0 | 0 | 0 |  0
*    1    |      0      |      0      |      0      | 0 | 0 | 0 |  1
*    2    |      0      |      0      |      0      | 0 | 0 | 0 |  2
*    3    |      0      |      0      |      0      | 0 | 0 | 0 |  4
*    4    |      0      |      0      |      0      | 0 | 0 | 0 |  8
* --------|---------------|---------------|---------------|----------------
*   1,2   |      0      | (b(1)-b(2))/2 |      0      | 0 | 1 | 0 |  3
*   1,3   | (b(1)+b(3))/2 |      0      |      0      | 1 | 0 | 0 |  5
*   1,4   |      0      |      0      | (b(1)-b(4))/2 | 0 | 0 | 1 |  9
*   2,3   |      0      |      0      | (b(2)-b(3))/2 | 0 | 0 | 1 |  6
*   2,4   | (b(2)+b(4))/2 |      0      |      0      | 1 | 0 | 0 | 10
*   3,4   |      0      | (b(4)-b(3))/2 |      0      | 0 | 1 | 0 | 12
* --------|---------------|---------------|---------------|----------------
*  1,2,3  | (b(1)+b(3))/2 | (b(1)-b(2))/2 | (b(2)-b(3))/2 | 1 | 1 | 1 |  7
*  1,2,4  | (b(2)+b(4))/2 | (b(1)-b(2))/2 | (b(1)-b(4))/2 | 1 | 1 | 1 | 11
*  1,3,4  | (b(1)+b(3))/2 | (b(4)-b(3))/2 | (b(1)-b(4))/2 | 1 | 1 | 1 | 13
*  2,3,4  | (b(2)+b(4))/2 | (b(4)-b(3))/2 | (b(2)-b(3))/2 | 1 | 1 | 1 | 14
* --------|---------------|---------------|---------------|----------------
* 1,2,3,4 |      a      |      b      |      c      | 1 | 1 | 1 | 15
*
* a) (b(1)+b(2)+b(3)+b(4))/4
* b) (b(1)-b(2)-b(3)+b(4))/4
* c) (b(1)+b(2)-b(3)-b(4))/4
*
* Each of the 16 possible cases has been assigned a case number to
* facilitate the description of the necessary processing.  The case
* number is found in the column labled "Case Number" in the table
```

```
* above.
*
* Determine the case number value for the current processing.
* Each of the four radar beams' state has been assigned a weight
* value:  beam 1: 1, beam 2: 2, beam 3: 4, beam 4: 8.  The "case
* number" is computed by  summing the radar beams multiplied by their
* their weight factors.
*********************************************************************

***
*  v1 Changes for AR#23. Item 24. Default goto 2000 added.
***
      go to (1000,1000,1000,1010,1000,1020,1040,1070,
     &        1000,1030,1050,1080,1060,1090,1100,1110),
     &       TDLR_STATE(1) + 2*TDLR_STATE(2) +
     &       4*TDLR_STATE(3) + 8*TDLR_STATE(4) + 1
          go to 2000
***
*  v1 Changes for AR#23. End Change.
***
*** cases 0, 1, 2, 4, 8 ***

 1000     pbvX = 0.0
        pbvY = 0.0
        pbvZ = 0.0

        K_MATRIX(1, 1, 0) = 0
        K_MATRIX(2, 2, 0) = 0
        K_MATRIX(3, 3, 0) = 0
        go to 2000

*** case 3 ***

 1010     pbvX = 0.0
        pbvY = (b(1) - b(2)) / 2.0
        pbvZ = 0.0

        K_MATRIX(1, 1, 0) = 0
        K_MATRIX(2, 2, 0) = 1
        K_MATRIX(3, 3, 0) = 0
      go to 2000

*** case 5 ***

 1020     pbvX = (b(1) + b(3)) / 2.0
        pbvY = 0.0
        pbvZ = 0.0

        K_MATRIX(1, 1, 0) = 1
        K_MATRIX(2, 2, 0) = 0
        K_MATRIX(3, 3, 0) = 0
```

```
      go to 2000

*** case 9 ***

 1030     pbvX = 0.0
        pbvY = 0.0
        pbvZ = (b(1) - b(4)) / 2.0

        K_MATRIX(1, 1, 0) = 0
        K_MATRIX(2, 2, 0) = 0
        K_MATRIX(3, 3, 0) = 1
      go to 2000

*** case 6 ***

***
*  v1 Changes for AR#23. Item 25. Goto 2000 added to finish the case properly
***
 1040     pbvX = 0.0
        pbvY = 0.0
        pbvZ = (b(2) - b(3)) / 2.0

        K_MATRIX(1, 1, 0) = 0
        K_MATRIX(2, 2, 0) = 0
        K_MATRIX(3, 3, 0) = 1
      go to 2000
***
*  v1 Changes for AR#23. End Change.
***
*** case 10 ***

 1050     pbvX = (b(2) + b(4)) / 2.0
        pbvY = 0.0
        pbvZ = 0.0

        K_MATRIX(1, 1, 0) = 1
        K_MATRIX(2, 2, 0) = 0
        K_MATRIX(3, 3, 0) = 0
      go to 2000

*** case 12 ***

 1060     pbvX = 0.0
        pbvY = (b(4) - b(3)) / 2.0
        pbvZ = 0.0

        K_MATRIX(1, 1, 0) = 0
        K_MATRIX(2, 2, 0) = 1
        K_MATRIX(3, 3, 0) = 0
      go to 2000
```

```
*** case 7 ***

 1070      pbvX = (b(1) + b(3)) / 2.0
         pbvY = (b(1) - b(2)) / 2.0
         pbvZ = (b(2) - b(3)) / 2.0

         K_MATRIX(1, 1, 0) = 1
         K_MATRIX(2, 2, 0) = 1
         K_MATRIX(3, 3, 0) = 1
       go to 2000

*** case 11 ***

 1080      pbvX = (b(2) + b(4)) / 2.0
         pbvY = (b(1) - b(2)) / 2.0
         pbvZ = (b(1) - b(4)) / 2.0

         K_MATRIX(1, 1, 0) = 1
         K_MATRIX(2, 2, 0) = 1
         K_MATRIX(3, 3, 0) = 1
       go to 2000

*** case 13 ***

 1090      pbvX = (b(1) + b(3)) / 2.0
         pbvY = (b(4) - b(3)) / 2.0
         pbvZ = (b(1) - b(4)) / 2.0

         K_MATRIX(1, 1, 0) = 1
         K_MATRIX(2, 2, 0) = 1
         K_MATRIX(3, 3, 0) = 1
       go to 2000

*** case 14 ***

 1100      pbvX = (b(2) + b(4)) / 2.0
         pbvY = (b(4) - b(3)) / 2.0
         pbvZ = (b(2) - b(3)) / 2.0

         K_MATRIX(1, 1, 0) = 1
         K_MATRIX(2, 2, 0) = 1
         K_MATRIX(3, 3, 0) = 1
       go to 2000

*** case 15 ***

 1110      pbvX = (b(1) + b(2) + b(3) + b(4)) / 4.0
         pbvY = (b(1) - b(2) - b(3) + b(4)) / 4.0
         pbvZ = (b(1) + b(2) - b(3) - b(4)) / 4.0

         K_MATRIX(1, 1, 0) = 1
```

```
          K_MATRIX(2, 2, 0) = 1
          K_MATRIX(3, 3, 0) = 1

 2000   continue

************************************************************************
*  3D)  Convert "processed" beam velocities into body velocites.
************************************************************************

        TDLR_VELOCITY(1, 0) = pbvX / COS(TDLR_ANGLES(1))
        TDLR_VELOCITY(2, 0) = pbvY / COS(TDLR_ANGLES(2))
        TDLR_VELOCITY(3, 0) = pbvZ / COS(TDLR_ANGLES(3))

        return
        end

***** end of module tdlrsp.for ******************************************
```

```
**********************************************************************
*  Module:        TDSP.FOR
*  Facility:      Pluto
*  P-Spec:        1.6
*  Abstract:
*         This module contains the implementation of the functional
*         requirements for CRCP.
*
*  List of Routines:
*         subroutine TDSP
**********************************************************************

**********************************************************************
*  Title: TDSP
*  Facility:      Pluto
*  Abstract:
*         1) Determine the operational status of the touch down sensor
*         2) determine if touch down has been sensed.
*
*  Arguments:   None
*
*  Revision History:
*         v0    15-sep-1994  Rob Angellatta (RKA) Original.
**********************************************************************

      subroutine TDSP

      implicit none

*** include the global common stores ***

      include 'external.for'
      include 'guidance_state.for'
      include 'sensor_output.for'

*** include constant definitions ***

      include 'constants.for'

**********************************************************************
*  1) Determine the operational status of the touch down sensor.
*  2) determine if touch down has been sensed.
*
*  The data element TD_COUNTER represents the sensor's measurement.
*  There are only two valid sensor measurements: A) all bits set
*  (value -1) which indicates touch down is sensed, and B) all bits
*  clear (value 0) which indicates touch down is not sensed.  If a valid
*  sensor value exists, then the operation status of the touch down sensor
*  is reported as "healthy" (value 0). Any other value of  TD_COUNTER
*  indicates a faulty sensor in which case the touch down sensor
*  status is reported as "failed" (value 1).
```

```
*
*  Note, once the touch down sensor has been determined to be
*  faulty, it is considered to be failed for the duration of the
*  mission -- no processing occurs once the sensor has failed.
************************************************************************

      if (TDS_STATUS .EQ. K$HEALTHY) then

        if (TD_COUNTER .EQ. 0) then
          TD_SENSED = K$TOUCH_DOWN_NOT_SENSED

        else if (TD_COUNTER .EQ. -1) then
          TD_SENSED = K$TOUCH_DOWN_SENSED
*                                       faulty sensor
        else
          TD_SENSED  = K$TOUCH_DOWN_NOT_SENSED
          TDS_STATUS = K$FAILED

        end if

      end if

      return
      end

***** end of module TDSP.FOR ******************************************
```

```
*********************************************************************
*  Module:        TSP.FOR
*  Facility:      Pluto
*  P-Spec:        1.7
*  Abstract:
*         This module contains the implementation of the functional
*         requirements for TSP.
*
*  List of Routines:
*         subroutine TSP
*         function   LOWER_PARABOLIC_FUNCTION
*         function   UPPER_PARABOLIC_FUNCTION
*********************************************************************

*********************************************************************
*  Title: TSP
*  Facility:      Pluto
*  Abstract:
*
*  Purpose:
*         1) Ascertain the operational status of the temperature sensors.
*         2) Determine the current atmospheric temperature based on the
*        measurements provided by two on-board temperature sensors.
*
*  Arguments:   None
*  Revision History:
*         v0    15-sep-1994  Rob Angellatta (RKA) Original.
*         v1    30-Nov-1994  Philip Morris (PEM)
*         v2    10-JAN-1995  Philip Morris (PEM)
*********************************************************************

      subroutine TSP

      implicit none

*** include the global common stores ***

      include 'external.for'
      include 'guidance_state.for'
      include 'sensor_output.for'
      include 'run_parameters.for'

*** include constant definitions ***

      include 'constants.for'

*** declare local functions ***

      real*8       LOWER_PARABOLIC_FUNCTION
      real*8       UPPER_PARABOLIC_FUNCTION
```

```
*** declare local variables ***

      real*8       slope
      real*8  solid_state_temp
      real*8  lower_parabolic_temp_limit
      real*8  upper_parabolic_temp_limit
          real*8  para_M2_M1
          real*8  REAL_THERMO_TEMP

************************************************************************
* 1) Determine the operational status of the temperature sensors
************************************************************************

      TS_STATUS(1) = K$HEALTHY
      TS_STATUS(2) = K$HEALTHY

************************************************************************
* 2A) Compute the temperature based on the solid state sensor
************************************************************************

***
*  v1 Changes for AR#23. Item 26. Added variable to cast M2-M1 to a real
***
          para_M2_M1 = M2-m1
      call ZERO_CHECK(para_M2_M1, 'TSP')

***
*  v1 Changes for AR#23. End Change.
***
      slope        = (T2 - T1)/(M2 - M1)

      solid_state_temp = slope * SS_TEMP + T1 - slope * M1

************************************************************************
* 2B) Determine if the temperature is within the valid range of the
*    TC sensor;
************************************************************************

************************************************************************
* Once the function describing the parabola has been determined, the
* temperature representing the lower limit of the parabolic region can
* be determined.  The lower limit of the lower parabolic region is
* specified as 15% of the difference of the two calibration
* measurements less than the lower calibration point.
************************************************************************

***
*  v1 Changes for AR#23. Item 2. "D0" added to 0.15
***
      lower_parabolic_temp_limit =
     &              LOWER_PARABOLIC_FUNCTION(M3 - 0.15D0*(M4 - M3))
```

```
***
*  v1 Changes for AR#23. End Change.
***
************************************************************************
* Once the function describing the parabola has been determined, the
* temperature representing the upper limit of the parabolic region can
* be determined.  The upper limit of the upper parabolic region is
* specified as 15% of the difference of the two calibration
* measurements greater than the upper calibration point.
************************************************************************

***
*  v1 Changes for AR#23. Item 2. "D0" added to 0.15
***
      upper_parabolic_temp_limit =
   &            UPPER_PARABOLIC_FUNCTION(M4 + 0.15D0*(M4 - M3))

***
*  v1 Changes for AR#23. End Change.
***
************************************************************************
* Now determine sensor temperature measurement to report
************************************************************************

      if ((solid_state_temp .LT. lower_parabolic_temp_limit) .OR.
   &     (solid_state_temp .GT. upper_parabolic_temp_limit)) then

*** the atmospheric temp is beyond the valid range of the TC sensor ***
***     so return the solid_state_temp                                ***

        ATMOSPHERIC_TEMP = solid_state_temp
      else

************************************************************************
* 2C) Compute the temperature based on the TC sensor
************************************************************************

        if (THERMO_TEMP .LT. M3) then

*** the atmospheric temp resides within the TC lower parabolic region ***

***
*  v2 Changes for AR#24. Item 6. Added variable to cast to a real
***
                  REAL_THERMO_TEMP=THERMO_TEMP

        ATMOSPHERIC_TEMP =
LOWER_PARABOLIC_FUNCTION(REAL_THERMO_TEMP)
*          ATMOSPHERIC_TEMP = LOWER_PARABOLIC_FUNCTION(THERMO_TEMP)
```

```
          else if (THERMO_TEMP .GT. M4) then

*** the atmospheric temp resides within the TC upper parabolic region ***

                     REAL_THERMO_TEMP=THERMO_TEMP

         ATMOSPHERIC_TEMP =
UPPER_PARABOLIC_FUNCTION(REAL_THERMO_TEMP)
*            ATMOSPHERIC_TEMP =  UPPER_PARABOLIC_FUNCTION(THERMO_TEMP)
***
*  v2 Changes for AR#24. End Change.
***

          else

*** The temperature resides within the TC sensor linear region ***
*** compute the temperature from the TC linear region ***

            slope    = (T4 - T3)/(M4 - M3)

            ATMOSPHERIC_TEMP =
   &                 slope * THERMO_TEMP + T3 - slope * M3
          end if
        end if

        return
        end

***** end of subroutine TSP *******************************************

*********************************************************************
*  Title: LOWER_PARABOLIC_FUNCTION
*  Facility:       Pluto
*  Abstract:
*          This routine represents the function of the lower parabolic
*          curve of the TC temperature sensor.  Given an 'X' value,
*          return the corresponding 'Y' value.
*
*  Arguments:
*          real*8 x  -- the 'X' value of interest
*
*  Revision History:
*          v0    15-sep-1994  Rob Angellatta (RKA) Original.
*       v1    30-Nov-1994  Philip Morris (PEM)
*********************************************************************

        real*8 function LOWER_PARABOLIC_FUNCTION(x)

        implicit none

*** define the arguments ***
```

```
      real*8  x

*** include the global common stores ***

      include 'run_parameters.for'

*** local variables ***

      real*8        half_slope

*** execution begins here ***

      half_slope = ((T4 - T3)/(M4 - M3)) / 2.0

***
*  v1 Changes for AR#23. Item 27. "M3 + half" changed "M3 - half"
***
      LOWER_PARABOLIC_FUNCTION =
     &        -(x - M3 - half_slope)**2 + T3 + half_slope**2

***
*  v1 Changes for AR#23. End Change.
***
      return
      end

***** end of function LOWER_PARABOLIC_FUNCTION ************************

**********************************************************************
*  Title: UPPER_PARABOLIC_FUNCTION
*  Facility:        Pluto
*  Abstract:
*         This routine represents the function of the upper parabolic
*         curve of the TC temperature sensor.  Given an 'X' value,
*         return the corresponding 'Y' value.
*
*  Arguments:
*         real*8 x  -- the 'X' value of interest
*
*  Revision History:
*         v0    15-sep-1994  Rob Angellatta (RKA) Originial.
*      v1    30-Nov-1994  Philip Morris (PEM)
**********************************************************************

      real*8 function UPPER_PARABOLIC_FUNCTION(x)

      implicit none

*** define the arguments ***
```

```
      real*8  x

*** include the global common stores ***

      include 'run_parameters.for'

*** local variables ***

      real*8        half_slope

*** execution begins here ***

      half_slope = ((T4 - T3)/(M4 - M3)) / 2.0

***
*  v1 Changes for AR#23. Item 28. Algebra Problem fixed.
***
      UPPER_PARABOLIC_FUNCTION =
     &       (x - M4 + half_slope)**2 + T4 - half_slope**2

***
*  v1 Changes for AR#23. End Change.
***
      return
      end

***** end of function UPPER_PARABOLIC_FUNCTION *************************

***** end of module TSP.FOR *******************************************
```

```
*********************************************************************
*  Module:        UTILITY.FOR
*  Facility:      Pluto
*  Abstract:
*         A collection of utility routines for Pluto.
*
*  List of Routines:
*         subroutine RANGE_CHECK
*         subroutine NEG_VALUE_CHECK
*         subroutine ZERO_CHECK
*********************************************************************

*********************************************************************
*  Title: RANGE_CHECK
*  Facility:      Pluto
*  Abstract:
*         Given a real*8 data element and it's lower and upper bounds,
*         determine if the data element exceeds the lower or upper
*         bound. If the element exceeds one of the bounds, then display
*         an error message.
*
*  Arguments:
*         source        real*8              The value to check.
*         lower_bound   real*8              The lower bound
*         upper_bound   real*8              The upper bound
*         module_text   character*(*)       The module name for error msg
*         variable_text character*(*)       The data name for error msg
*
*  Notes:
*         The upper bound >= lower_bound
*
*  Revision History:
*         v0    15-sep-1994  Rob Angellatta (RKA) Original.
*      v1    30-Nov-1994  Philip Morris (PEM)
*********************************************************************

      subroutine RANGE_CHECK(source, lower_bound, upper_bound,
     &                       module_text,variable_text)

      implicit none

*** define the subroutine arguments ***

      real*8               source
      real*8               lower_bound
      real*8               upper_bound
      character*(*)        module_text
      character*(*)        variable_text

*** include a global common store ***
```

```
      include 'external.for'

*** format statements ***

***
*  v1 Changes for AR#23. Item 29. "x" added before "I4".
***
 10   format (x,'%EXCEPTIONAL-CONDITION-GCS-LOWER_LIMIT_EXCEEDED')
 20   format (x,'%EXCEPTIONAL-CONDITION-GCS-UPPER_LIMIT_EXCEEDED')
 30   format (x, A6, X, A32, x,I4)
 40   format (x, A32, E23.14)

***
*  v1 Changes for AR#23. End Change.
***
*** execution begins here ***

      if (source .LT. lower_bound) then
        write (6, 10)
        write (6, 30) module_text, module_text, FRAME_COUNTER
        write (6, 40) variable_text, source
      else if (source .GT. upper_bound) then
        write (6, 20)
        write (6, 30) module_text, module_text, FRAME_COUNTER
        write (6, 40) variable_text, source
      end if

      return
      end

*** end of RANGE_CHECK ***********************************************

***********************************************************************
*  Title: NEG_VALUE_CHECK
*  Facility:      Pluto
*  Abstract:
*        Given a real*8 data element determine if the data element
*        has a value of less then zero.  If the value is less than zero,
*        then display an error message.
*
*  Arguments:
*        source        real*8              The value to check.
*        module_text   character*(*)       The module name for error msg
*
*  Revision History:
*        v0    15-sep-1994  Rob Angellatta (RKA) Original.
*      v1    30-Nov-1994  Philip Morris (PEM)
***********************************************************************

      subroutine NEG_VALUE_CHECK(source, module_text)
```

```
      implicit none

*** define the subroutine arguments ***

      real*8              source
      character*(*)       module_text

*** include a global common store ***

      include 'external.for'

*** format statements ***

***
*  v1 Changes for AR#23. Item 29. "x" added before "I4".
***
 10   format (' ','%EXCEPTIONAL-CONDITION-GCS-NEGATIVE_SQUARE_ROOT')
 30   format (x, A6, X, A32, x,I4)
 40   format (x, E23.14)

***
*  v1 Changes for AR#23. End Change.
***
*** execution begins here ***

      if (source .LT. 0) then
        write (6, 10)
        write (6, 30) module_text, module_text, FRAME_COUNTER
        write (6, 40) source
      end if

      return
      end

*** end of NEG_VALUE_CHECK *********************************************

**************************************************************************
*  Title: ZERO_CHECK
*  Facility:     Pluto
*  Abstract:
*        Given a real*8 data element determine if the data element
*        has a value of zero.  If the value is zero, then display
*        an error message.
*
*  Arguments:
*        source      real*8              The value to check.
*        module_text   character*(*)     The module name for error msg
*
*  Revision History:
*        v0    15-sep-1994  Rob Angellatta (RKA) Original.
*     v1    30-Nov-1994  Philip Morris (PEM)
```

```
*************************************************************************

        subroutine ZERO_CHECK(source, module_text)

        implicit none

*** define the subroutine arguments ***

        real*8              source
        character*(*)       module_text

*** include a global common store ***

        include 'external.for'

*** format statements ***

***
*  v1 Changes for AR#23. Item 29. "x" added before "I4".
***
  10    format (x,'%EXCEPTIONAL-CONDITION-GCS-DIVIDE-BY-ZERO')
  30    format (x, A6, X, A32, x,I4)

***
*  v1 Changes for AR#23. End Change.
***
*** execution begins here ***

        if (source .EQ. 0) then
          write (6, 10)
          write (6, 30) module_text, module_text, FRAME_COUNTER
        end if

        return
        end

*** end of ZERO_CHECK ***************************************************

***** end of module UTILITY.FOR ******************************************
```

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE *(DD-MM-YYYY)* 01- 12 - 2008	**2. REPORT TYPE** Technical Memorandum	**3. DATES COVERED** *(From - To)*

4. TITLE AND SUBTITLE Guidance and Control Software Project Data - Volume 2: Development Documents	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Hayhurst, Kelly J. (Editor)	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER 457280.02.07.07.06.02
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) NASA Langley Research Center Hampton, VA 23681-2199	**8. PERFORMING ORGANIZATION REPORT NUMBER** L-19549
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Aeronautics and Space Administration Washington, DC 20546-0001	**10. SPONSOR/MONITOR'S ACRONYM(S)** NASA
	11. SPONSOR/MONITOR'S REPORT NUMBER(S) NASA/TM-2008-215551

12. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified - Unlimited
Subject Category 61
Availability: NASA CASI (301) 621-0390

13. SUPPLEMENTARY NOTES

14. ABSTRACT
The Guidance and Control Software (GCS) project was the last in a series of software reliability studies conducted at Langley Research Center between 1977 and 1994. The technical results of the GCS project were recorded after the experiment was completed. Some of the support documentation produced as part of the experiment, however, is serving an unexpected role far beyond its original project context. Some of the software used as part of the GCS project was developed to conform to the RTCA/DO-178B software standard, "Software Considerations in Airborne Systems and Equipment Certification," used in the civil aviation industry. That standard requires extensive documentation throughout the software development life cycle, including plans, software requirements, design and source code, verification cases and results, and configuration management and quality control data. The project documentation that includes this information is open for public scrutiny without the legal or safety implications associated with comparable data from an avionics manufacturer. This public availability has afforded an opportunity to use the GCS project documents for DO-178B training. This report provides a brief overview of the GCS project, describes the 4-volume set of documents and the role they are playing in training, and includes the development documents from the GCS project.

15. SUBJECT TERMS
Software engineering; Computer programming; Software reliability; DO-178B

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			STI Help Desk (email: help@sti.nasa.gov)
					19b. TELEPHONE NUMBER ***(Include area code)***
U	U	U	UU	380	(301) 621-0390