An Object–Oriented Method of Mission Profile Input for Aircraft Design

by

Francisco Rivera Jr.

Thesis submitted to the Faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of Master of Science in Mechanical Engineering

APPROVED

Sankar Jayaram, Chairman

A. Myklebust

J. R. Mahan

April 30, 1993
Blacksburg, Virginia
An Object–Oriented Method of Mission Profile Input
for Aircraft Design

by
Francisco Rivera Jr.
Sankar Jayaram, Chairman
Mechanical Engineering

(ABSTRACT)

This thesis discusses the creation of an object–oriented method to facilitate the creation and specification of aircraft mission profiles. Mission profiles are detailed descriptions of an aircraft's flight path and its inflight mission activities. They are a vital aspect of the conceptual design process of an aircraft. The Mission Profile Input System (MPIS) created is general in nature and can be customized to be compatible with existing aircraft CAD systems. All data associated with the mission—phase parameters, phases, and mission parameters, are defined to be objects. Each data type can therefore be customized individually to meet any requirements which may be necessary to make the MPIS compatible with a host system. Customization of the MPIS is further enhanced by the nature of the design upon which it is based. An object-oriented design provides the system with a high degree of extendibility. The encapsulation and inheritance features of object-oriented design allow new types of phases and phase parameters to be simply "plugged" into the existing system. The MPIS provides the user with an interactive, Motif-like interface which is conducive to manipulating the large quantities of data inherent in specifying mission profiles. The system is based on the ISO graphics standard, PHIGS, and hence is device-independent. Moreover, the system has been implemented using the hybrid object-oriented language, C++, which is supported by a large number of computer systems.
Acknowledgments

Were it not for the continuous, unconditional, support of my parents I doubt I could've managed to pull this one out—they have been my source of inspiration. Though, I can never hope to fully repay the debt I owe them, I dedicate this thesis as a small, symbolic token of my utmost appreciation for what they’ve provided. Para mi papa y mama, Francisco y Marcelina Rivera. Los mejor parientes que un hijo puede tener... Gracias por todo...

I would like to extend my appreciation to Paul Gelhausen and Gary Hill for their numerous suggestions. Their input contributed significantly to the development of this thesis.

I thank my advisor, Dr. Sankar Jayaram, whose “I promise you you’ll graduate” assurances gave me the impetus to continue on with the endeavor. Also I would like to thank Dr. A. Myklebust and Dr. J. R. Mahan for serving on my committee.
The following is a list of people who, through some measure, "own" a piece of my thesis for their encouragement and help.

Trisha Tran - the greatest friend a person could ever hope to have—thanks for everything Trish. I’ll see ya in England!!!
Laura Palacios - for being there when it counted the most—best of luck in Loyola Law School
Steven Jump - for making sure I'd risk my life every time we went hiking—I'll "pay" you back if you ever get nominated for the Supreme Court.
Frank Sager - who told me "people like you belong in graduate school." Frank, it's "nearly as much fun as a sharp stick in the eye" :-(
Greg Simpson - well, Homer, it's time that I join you in the real world . . .
Dom Dal Bello - who because he went to graduate school, I had to go . . . "Thanks"
Joseph Wood - the 'tons of money" you promised me better be on their way. I hate to think I did this for nothing
John Merrill - who assured me "college is nothing but fun and games"
Douglas T. Shafer - my roomi who told me, "Go for it, it's only ONE year!"—you "deserve" to be a lawyer :-(
David Coe - for all the support and "clear insight" he offered
Andreas Steude - whose help was immeasurable—thanks for helping me escape
John Kelly - whose Wayne's World humor ensured that I kept my sanity—better luck in cards next time, dude!!!!
Srinivas Dhulipala - whose whipping I'd inflict in racquetball made me look forward to the next day—thanks for everything—best of luck in Detroit
The "P-man" - for being a great office "pal dude"—we're outta here—"COOLness!"
Scott Woyak - object-oriented programming personified — thanks for all the help
Jim Pascoe - who introduced me to Coconut—the only female in my life this year. :-( Merci.

Colin Heichman - Owner of Coconut
Vellaurit Rivera - thanks for all the support sis!—we'll celebrate your graduation soon enough . . .
Nereyda Rivera - for reminding me that life is more than coding
Jim Veny - for providing some seriously needed direction
The SLAC crowd - for all their engineering advice
The FMC crowd - for all their non-engineering advice
AND FINALLY
Kathy Ireland - Nope, I don't know her personally. But she provided some "divine" inspiration, nonetheless. :-(

Acknowledgements
# Table Of Contents

1.0 INTRODUCTION .................................................. 1

2.0 LITERATURE REVIEW ........................................... 3
   Overview .................................................................... 3
   Conceptual Aircraft Design ........................................ 3
   Program Design ...................................................... 5
   Object-Oriented Design ........................................... 6
   Mission Profiles .................................................... 7

3.0 THESIS OBJECTIVES ............................................. 9

4.0 AIRCRAFT DESIGN AND MISSION PROFILE INPUT .......... 10
   The Aircraft Design Process ........................................ 10
   Tools For Aircraft Conceptual Design ............................ 12
      ACSYNT ................................................................ 12
      Interactive CAD Version Of ACSYNT ......................... 14
      ACSYNT's Trajectory Module .................................... 15

5.0 REQUIREMENTS .................................................... 18
   Overview .................................................................... 18
   Explanation Of Requirements ....................................... 20

6.0 DESIGN CONSIDERATIONS ..................................... 24
   Overview .................................................................... 24
   Program Design Considerations .................................... 26
      Object-Oriented Design And Language Selection .......... 26
      Object Relationships ............................................ 27
      Extendibility And Maintainability ............................... 29
      Data Handling ..................................................... 30
      Interface Layout .................................................. 30

7.0 THE MISSION PROFILE INPUT SYSTEM ...................... 32
   Class Structure Layout ............................................. 32
   The PHIGS-Based, Motif-Like Interface ......................... 37
8.0 CLASS DESCRIPTIONS

Overview .................................................................. 43
Parameters .................................................................. 43
  Data Members-Parameters class .......................... 44
  Functions Of Parameters .................................. 44
Phases ..................................................................... 46
  Data Members-Phases class ............................... 46
  Functions Of Phases ........................................ 47
Mission_Parameters ............................................ 49
  Data Members-Mission_Parameters Class ........... 49
  Functions Of Mission_Parameters .................... 50
Missions ............................................................... 51
  Data Members-Missions Class .......................... 51
  Functions Of Missions .................................... 52
Phase_Diagram_Window ...................................... 56
  Data Members-Phase_Diagram_Window Class ........ 56
  Functions Of Phase_Diagram_Window ................. 56
Mission_Window .................................................. 58
  Data Members-Mission_Window Class ............... 58
  Functions Of Mission_Window ......................... 58

9.0 SYSTEM CUSTOMIZATION .................................... 63

Overview ................................................................ 63
Trajectory Data Files ........................................ 64
Phase Defaults File ............................................ 66
Parameters .......................................................... 68
  The set_value_to function ............................... 70
  The check function .......................................... 71
Phases ............................................................... 73
  The calculate function .................................... 74
  The geo_segment function ............................... 77
Miscellaneous ..................................................... 78

10.0 IMPLEMENTATION AND EXAMPLES OF RESULTS ....... 79
The Parameters Class ............................................................... 133
Functions Of The Parameters Class ......................................... 135
The Phases Class ................................................................. 139
Functions Of The Phases Class ................................................ 141
The Mission_Parameters Class .................................................. 145
Functions Of The Mission_Parameters Class ............................... 147
The Missions Class ............................................................... 150
Functions Of The Missions Class .............................................. 153
The Phase_Diagram_Window Class ............................................ 163
Functions Of The Phase_Diagram_Window Class .......................... 165
The Mission_Window Class ...................................................... 167
Functions Of The Mission_Window Class .................................... 174
Additional Functions ............................................................ 185
Other Classes ........................................................................ 186
VITA ....................................................................................... 187
List of Illustrations

Figure 1. The Trajectory Input Interface for ACSYNT ............................................. 16
Figure 2. ACYNT’s Trajectory Module Input Method [Tayl88] .................................. 17
Figure 3. Approach to the Development of the MPIS ............................................. 25
Figure 4. Real-Life Object Analogy ........................................................................ 28
Figure 5. Class Relationships .............................................................................. 33
Figure 6. Other Classes Created by Mission_Window ............................................ 34
Figure 7. Class Organization for PHIGS Motif-like Framework [Woya92] ............... 38
Figure 8. Internal Data Relationships .................................................................. 40
Figure 9. Trajectory Data File ............................................................................ 65
Figure 10. The Phase.dfl.miss File ...................................................................... 67
Figure 11(a). Basic Parameter Definition .............................................................. 69
Figure 11(b). Modification Required For Parameter Definition ......................... 69
Figure 12(a). Basic Phase Definition ................................................................ 75
Figure 12(b). Required Modification for Phase Definition ................................ 75
Figure 13. Creation of Mission (Pseudo-Code) .................................................... 84
Figure 14. Parameter Modification (Pseudo-Code) ............................................. 87
Figure 15. The MPIS With the Phase Diagram Window Activated ....................... 89
Figure 16. The MPIS With the Mission Parameters Menu Activated ................. 90
Figure 17. The Phase Menu ............................................................................... 91
Figure 18. The Defaults Menu .......................................................................... 92
Figure 19. The Phase Options Menu .................................................................. 93
Figure 20. The Quick Input Menu ..................................................................... 94
Figure 21. The Parameter Menu ....................................................................... 95
Figure 22. The Move Parameter Menu ................................................................ 96
Figure 23. The Select Other Variables Menu ................................................... 97
Figure 24. The Other Variables Menu ............................................................... 98
Figure 25. The Row/Column Menu ......................................................... 99
Figure 26. Menu Listing Available Files ............................................... 100
Figure 27. The Message Menu ............................................................. 101
Figure 28. The Confirm Menu ............................................................... 102
Figure 29. The Filename Menu .............................................................. 103
Figure 30. Sample Usage of the Quick Input Menu ................................. 126
1.0 INTRODUCTION

The design of an aircraft requires collaboration from a variety of disciplines. Conflicting design proposals often result from each group trying to optimize a different aspect of the aircraft design. To reconcile such differences, a set of objectives which the final design must satisfy is instituted before the design process begins. These objectives are referred to as the basic requirements of an aircraft—a set of design and performance specifications the final product must satisfy. Once these requirements are established, the various groups must compromise and reconcile their differences in order to achieve these basic requirements. A significant aspect of the basic requirements for an aircraft is the specification of its mission profiles. Mission profiles are detailed descriptions outlining the anticipated flight activities of the aircraft during a typical flight. These profiles are broken into more specialized segments known as phases which focus on specific flight operations. Each operation, in turn, is defined by a set of parameters which describe the various aircraft conditions. Examples of typical phases include cruise, loiter, and acceleration, whereas examples of typical parameters include time, speed, and altitude. The list of possible mission profiles is inexhaustible. The types of phases employed by a mission depend strongly on the type of aircraft under consideration. A combat phase, for example, used extensively in designing military aircraft, has no application in the design of commercial aircraft.

Mission profiles are inherently data intensive. The number of phases and parameters that comprise a typical mission is usually quite large. The creation and manipulation of such a
A large amount of data can prove to be overwhelming if the proper tools are not provided. This thesis addresses the creation of an object-oriented method of specifying mission profiles for aircraft design. The design, development, and use of this system is discussed in detail. The system created provides a good graphical user interface which displays the vast amounts of mission data in a friendly and consistent fashion. This system is general in nature and can be customized to be compatible with a wide variety of existing aircraft CAD software.

The system, referred to as the Mission Profile Input System (MPIS), was developed at Virginia Polytechnic Institute and State University. It is part of an ongoing endeavor to develop better methods by which to improve the usefulness of current interactive CAD systems.
2.0 LITERATURE REVIEW

Overview

The creation of the Mission Profile Input System relied upon ideas borrowed from the disciplines of aircraft and software design. The amount of literature which is available in these two areas is immense. The following sections do not reflect a comprehensive list of the literature which is available. Rather, they are intended to give a small, but representative, sample of what may be obtained. The books referenced in this section are included because the methods discussed within them were used extensively in the development of the MPIS. The literature survey is divided into four parts: literature on aircraft design, software design, object-oriented design and mission profiles. The third section describes a specialized method of software design which was used extensively in the creation of the Mission Profile Input System. The final section discusses some of the available literature pertaining to mission profiles and their role in the aircraft design process.

Conceptual Aircraft Design

As in the design of other products, iteration is an inherent element of aircraft design. Each iteration “fine tunes” the results from the preceding one to enhance and optimize the configuration of the aircraft. The process is repeated until the design satisfies a desired set of requirements and specifications. Leland M. Nicolai guides the reader through one
complete iteration of the aircraft design process in his book "Fundamentals of Aircraft Design" [Nico84]. The topics covered range from a concise review of aerodynamics to a detailed discussion of environmental concerns. The book outlines analytical methods for making and assessing certain design decisions. The design considerations covered include engine sizing and selection, material selection, sizing of the vertical and horizontal tails, sizing of the fuselage, and static stability and control.

A more comprehensive look at aircraft design is given by Daniel P. Raymer in his book "Aircraft Design: A Conceptual Approach" [Raym89]. The book gives equal treatment to the two major aspects of aircraft design: design layout and design analysis. Special emphasis is given to aircraft configuration layout. The reader is guided through the complicated procedure of drafting aircraft from analysis results. In addition to the topics discussed by Nicolai, Raymer addresses other aspects of design such as cost analysis and sizing and trade studies.

An eight-book series designed to familiarize engineering students with the methodology and decision making involved in airplane design is given by Jan Roskam [Rosk89]. Each book in the series covers a different aspect of the design process. The topics include preliminary sizing, preliminary configuration, layout design of cockpit, layout design of landing gear, weight estimation, aerodynamic calculations, determination of stability, and cost estimation. Roskam presents the student with detailed examples of the considerations that go into designing an aircraft. Requirements and specifications that need to be considered are also detailed.

The three aforementioned books focus primarily on the traditional design process—very little, if any, attention is given to the role of the computer in modern-day design. In his book "Development and Application of Computer-Based System for Conceptual Aircraft
Design," Cornelis Bil focuses on the impact of computer-aided engineering (CAE) techniques on the design of aircraft [Corn88]. In particular, he focuses on the development and application of the Aircraft Design and Analysis System (ADAS) and its graphical interface, MEDUSA. The various modules of the ADAS program are discussed in detail, including how to utilize them. Emphasis is given to illustrating how to read the information provided by the system. Finally, to demonstrate the system, a design optimization study is performed on a short-haul passenger airliner.

**Program Design**

A comprehensive look at software design is given by Ray Turner in his book "Software Engineering Methodology" [Turn84]. Every aspect of program development, from conceptual design to code validation, is discussed in great detail. Emphasis is given to developing proper design principles including modularity, hiding, understandability, and uniformity. Coding and debugging techniques designed to facilitate development of good software are also offered. In addition to focusing on the specifics of individual design projects, Turner also offers advice and examples on how to manage software development projects which involve a group of people. Examples of small and major projects and discussions of project phases such as planning, motivation, control, and new product training, are included.

In order for a software system to be useful and successful it must have a well-designed user interface. Judith R. Brown and Steve Cunningham detail the process of developing a good interface in their book "Programming The User Interface" [Brow89]. Techniques and examples to help in the development of interactive program design features such as input, output, screen layout, and error handling, are discussed in detail. Topics covered range from simple input and menu layout configurations to complicated window and user
interface managers. For the interested reader, discussion on how to develop interfaces for people with disabilities is also included.

A complete look at the many aspects of software development is provided by Robert C. Tausworthe in his book “Standardized Development of Computer Software” [Taus77]. The topics covered range from discussions on the need for software standards to the assessment of program correctness. Before delving into specificities of specialized program development, Tausworthe presents general topics. This includes discussion on the fundamental principles of software development, specification of program behavior, and program design.

**Object-Oriented Design**

A practical introduction to software engineering and object-oriented design is given by Darrel Ince in his book “Object-Oriented Engineering With C++” [Ince91]. As indicated by the title, the language used to illustrate object-oriented programming (OOP) techniques is C++, one of the most widely used languages in OOP implementation. As in most other books related to this topic, important OOP concepts such as polymorphism, inheritance, and encapsulation are described in great detail. Emphasis is given to proper object-oriented design, with focus on the aspects of object identification, object implementation, specification of object functionality, and object linkage. Formal software methodologies are also discussed, such as the three types of software prototyping: throw-away, incremental, and evolutionary.

The recent attraction to OOP techniques comes from the fact that they offer a revolutionary way to look at programming. It offers not only a new way to write programs, but also a new way to think about how programs interact with the world. It is the latter concept which
the book "The Tao Of Objects," written by Gary Entsminger [Ents90], tries to get across. Through the use of numerous examples, this book explores how OOP techniques and real-world problems are related. Emphasis is placed on encouraging the reader to think in terms of objects and methods to define them. Discussion on the determination of their makeup, boundaries, and generality is given. Real-world examples which illustrate good and bad object definition are given to demonstrate the advantages of a good design. These advantages include efficiency, maintainability, and readability.

Mission Profiles

At present, a general mission profile input system or methodology do not exist. Each aircraft design system has input methods which are system-specific. Systems such as ACSYNT [Wamp88a & Wamp88b], STOP [Stei67], and CASTOR [Simo86], utilize individual input techniques to funnel information into their mission performance analysis routines. Of these methods, only the ACSYNT Trajectory Input Module [Tayl88] provides a graphical user interface to facilitate the creation of the mission profile.

The amount of research devoted to developing better methods for the analysis of mission profiles demonstrates their importance to the overall aircraft design process. Various techniques to improve the analysis have been explored over the years. Specialized techniques were investigated by Rutowski using energy methods [Ruto54]. These methods analyzed the mission performance by minimizing the fuel and time paths of the mission. More general methods were explored by Stein [Stei67], Bryson [Brys68], and Shultz [Shul70]. These three methods optimized the complete path of the trajectory—i.e. the climb-cruise-descent phases were all taken into account. Stein utilized steepest descent methods whereas Bryson and Shultz obtained the trajectory solutions by applying optimization principles to the energy state equations. In 1984 Simos and Jenkinson coupled
the flight profile analysis algorithm with a multivariate optimization (MVO) routine to determine the optimum mission profile for short-haul routes [Simo84]. This technique permitted the entire flight to be treated as a single complete problem and allowed physical, air-traffic control, and environmental limitations to be incorporated more easily. More importantly, however, the problems associated with the mathematical modeling of the thrust characteristics of propeller-driven aircraft were eliminated. Up to this point such modeling had been restricted to simplified thrust models in order to avoid the mathematical complexities. By 1986 a program combining the previously separate aspects of preliminary design optimization and flight-profile optimization was created. Dimitri Simos utilized this program, referred to as CASTOR, to optimize both aspects of the aircraft design for a short-haul aircraft [Simo86].

As the aircraft design process is rearranged from a sequential procedure into a nonhierarchical decomposition of the various design aspects, the analysis of the mission profile has become an even more integral part of the aircraft design process. The work started by [Simo86] is being expanded to provide for more sophisticated simultaneous optimization of the various design disciplines. Jaroslaw Sobieszczanski-Sobieski discusses some of these multidisciplinary analysis techniques [Sobi88]. Such techniques generate better results than those obtained by the traditional sequential design process. More importantly, these techniques eliminate the notion that each design aspect may be thought of as a separate process, independent from the rest. Henceforth the entire design process will be thought of as a single problem with each aspect within it influencing the others.
3.0 THESIS OBJECTIVES

Motivation for the creation of the Mission Profile Input System (MPIS) was to provide a user interface by which large volumes of trajectory data could be easily entered, manipulated, and evaluated. The module should be flexible enough to be compatible with a wide variety of existing aircraft CAD systems. To create a system that would satisfy these requirements, the following objectives were established:

- Create methods by which a mission profile can be easily created and modified.
- Design and develop the system such that it can be customized to be compatible with existing aircraft CAD systems.
- Create a user interface which meets the demands of both novice and experienced users by providing a high degree of user-friendliness along with a variety of efficient data-entry procedures.
- Design a system conducive to future enhancements and extensions.
- Test the system by integration with an existing aircraft CAD system—ACSYNT.

Satisfying these requirements would provide a method which facilitates the often arduous task of creating a mission profile—a critical aspect in the design of an aircraft.
4.0 AIRCRAFT DESIGN AND MISSION PROFILE INPUT

The Aircraft Design Process

The successful design of an aircraft requires a concerted effort from a multitude of disciplines. Engineers specializing in the areas of structures, flight control, propulsion, aerodynamics, performance, and weights, must work together to produce the optimum aircraft design. The optimum design from one specialized area often conflicts with the design of another area. Consequently, the final design is usually the result of multiple compromises made by the various design groups. In addition to engineering considerations, the economic and manufacturing aspects of the aircraft must be considered—design feasibility implies not only whether a product can be produced, but whether it can be produced at a reasonable cost and within a reasonable period of time.

Before an aircraft is designed, its mission requirements must be established. These requirements identify the criteria to which the final design must adhere. Among other things, the mission requirements often identify the purpose, payload, speed, range, endurance, cost, and maintainability of the aircraft [Nico84]. In general, the mission requirements are established by the supplier (e.g. Boeing Airplane Company, Piper Aircraft Corporation, etc.) for commercial aircraft, whereas the user (e.g. U.S. government) establishes the mission requirements for military endeavors.
The aircraft design process has been traditionally divided into three major phases: conceptual design, preliminary design, and detail design [Nico84].

In the initial design phase certain general design constraints are imposed. The overall size, configuration, and inboard profile of the airplane are determined. Parametric trade studies are used to converge upon the best wing loading, wing sweep, aspect ratio, thickness ratio and general wing-body-tail configuration [Nico84]. These design parameters, however, are but best approximations—based upon various design assumptions which are subject to future changes.

In the second phase the design begins to exhibit more detail. The engine is selected, the structural integrity of the aircraft is analyzed, refined weight and aerodynamic analysis are performed, and the dynamic stability and control influences on the control system are determined. More importantly, aerodynamic and structural tests are done on scaled models to verify the analytical conclusions.

The final design phase consists of readying the design for production. All design parameters are "frozen," detailed component and assembly drawings are created, and the necessary tooling for the manufacturing process is developed. Finally, a prototype is built to test and prove the design.

Presently, many aerospace companies are shifting from the traditional design process to concurrent engineering methods. Concurrent engineering reflects a growing trend by companies to move away from the vertical, hierarchical organizational structure to a flat and lean one. To expedite and decentralize the decision making process (thereby enhancing organizational flexibility) the number of job categories at each level of the hierarchy is reduced and the responsibility associated with each job is broadened [Dert89]. In virtually all cases, the fewer layers of hierarchy and the greater functional integration has resulted in
faster and cheaper product development. Certain companies, such as The Ford Motor Company, have successfully applied the concept. In developing the Ford Taurus, Ford created product-development teams which consisted of representatives from planning, design, engineering, manufacturing, and marketing. By working simultaneously rather than serially, these teams managed to successfully integrate the various phases of product development and produce a better product. Boeing Corporation is now attempting to duplicate Ford’s success. Multifunctional “design/build teams” are now in charge of developing both the product and the production process. The net effect is that two processes which were previously separated—design and production—now completely overlap.

**Tools For Aircraft Conceptual Design**

ACSNT

ACSNT (AirCraft SYNThesis) was developed by NASA Ames Research Center for conceptual design studies of advanced aircraft in the early ‘70’s. It is a highly flexible program which allows for the study of a broad range of aircraft. ACSNT consists of ten separate modules each dedicated to analyzing a separate aspect of an aircraft. The modules are as follows [Jaya92]:

**GEOMETRY:** calculates the surface areas and volumes

**TRAJECTORY:** determines the mission performance of the aircraft configuration

**AERODYNAMICS:** calculates the minimum drag, the lift, and the induced drag
PROPULSION: computes the design and off-design performance of the engine (turbojet, turbofan, turboprop, or propeller)

STABILITY AND CONTROL: determines the center of gravity, the size of the horizontal control surface, the pitching-moment curve slope, and the static margin

WEIGHTS: assigns initial weight values for computation and gives weight component multiplying factors

SUPersonic AERO: analyzes the supersonic characteristics of the aircraft configuration

COST: determines the manufacturing costs, direct and indirect operating costs, and the manufacturing and airline return on investment

TAKEOFF: predicts takeoff data which varies according to the type of aircraft

BALANCE: separate program used to balance the aircraft configuration

Additionally, ACSYNT contains analysis functions which facilitate the design process. The routines included may be used to optimize the design parameters, determine the gross weight of the aircraft, and/or calculate the sensitivity of the overall design to a specified design variable. The variety and number of options available to the user of ACSYNT are too numerous to be listed in this thesis. For further information on ACSYNT, the reader should refer to the ACSYNT Manual.
INTERACTIVE CAD VERSION OF ACSYNT

Despite its great power and flexibility, ACSYNT lacked a friendly user interface. The analytical results consisted of little more than row upon row of numbers. Much time and effort was devoted to deciphering what the numerical output implied. A greater limitation, however, was that the effects on the overall aircraft design resulting from "tweaking" the design parameters were not quickly, nor easily, apparent. In order to address this problem, Virginia Polytechnic Institute and State University (VPI) and NASA Ames Research Center began work in 1986 to create a computer aided design version of ACSYNT [Wamp88a & Wamp88b]. Today, many of the initial goals established have been realized. ACSYNT is now highly interactive—design parameters may be quickly edited and their effects immediately observed on the aircraft geometry. Shaded and hidden surface views give designers the ability to quickly analyze and assess multiple aircraft configurations. More importantly, ACSYNT allows extraction of dimensional geometric parameters from its B-Spline surface models [Jaya91 & Jaya92a]. This feature allows the smooth transfer of data between the preliminary and conceptual stages of aircraft design.

In 1990 several aerospace companies and government agencies along with VPI formed the ACSYNT Institute at the VPI CAD laboratory—an innovative program of joint R & D between government, academia and industry to continue sponsorship of the endeavor. Faster and better algorithms for creating, manipulating, and displaying aircraft geometry are being developed. Various methods for facilitating the entry and manipulation of data are also being explored.
ACSNT's Trajectory Module

One example of a current mission profile input program is ACSNT's trajectory module. Like other trajectory modules, the ACSNT Trajectory Module is exclusively tailored to be compatible only with its host program—ACSNT. Although the program is a separate module, it cannot be easily modified and made compatible with other CAD systems.

Figure 1 illustrates the layout of ACSNT's Trajectory Module. Interactive icons and numerous help screens make it ideal for users unfamiliar with the program. However, for more experienced users, the rigidity introduced by the additional levels of interactivity detract from the efficient construction of a mission. This problem demonstrates the need to balance the user-friendly and entry-efficient demands of the interface layout.

The greatest drawback to the interactive input methods associated with ACSNT's Trajectory Module, however, is that it fails to provide the end-user the means by which to make a quick assessment of the overall mission. Mission data is not readily available to the user for inspection. Rather it is hidden from view by three hierarchical editing levels [Tayl88]. Modification of each parameter requires the user to respond to three separate inquiries: specification of a parameter category, specification of a parameter within the category, and finally, specification of the parameter value (see Figure 2). This tedious process for modifying each parameter needlessly renders the creation of a mission profile an inefficient and arduous task. Of greater consequence, the three-level hierarchical editing scheme allows for the inspection of only one parameter at a time. Hence, the end user cannot easily make comparisons between the various parameters. The final result of such restrictions and limitations is that the cumbersome task of entering data into the module often nullifies the positive aspects (e.g. layout simplicity, consistency, etc.) of the module.
Figure 1. The Trajectory Input Interface for ACSYNT; [Tayl88]
Figure 2. ACYNT's Trajectory Module Input Method [Tayl88]
5.0 REQUIREMENTS

Overview

In order to satisfy the thesis objectives, a set of basic requirements was identified. The process of indentifying such requirements included requested feedback from Boeing, NASA, and Lockheed. The list of requirements developed was effectively divided into two parts—the user and the software requirements. The user requirements focused on such matters as program functionality and the friendliness of the interface. Software requirements focused on providing program flexibility and extendibility. This section describes the requirements which needed to be addressed:

User Requirements

- The user interface must have a consistent look and feel to eliminate unnecessary confusion to the end-user.
  - The entry of all data should be done directly onto similar data input areas.
  - The various pop-up menus and their menu items should appear and operate similarly.
- The system must be user-friendly to reduce the learning curve for new users.
  - The purpose of each pop-up menu and its menu items should be made self-evident to the user.
• The module must exhibit a high degree of functionality to increase its usefulness.
  
• Pop-up menus must be draggable to eliminate visual interference with the background.
  
• Windows should be draggable and resizable to maximize the efficient use of the screen.
  
• The effects of a modification should be reflected immediately on the overall mission.
  
• The windows must provide automatic centering and zooming capabilities.
  
• The interface should display sufficient data to allow for a quick assessment of the overall mission. It should allow this data to be partitioned into smaller groupings such as phase and parameter data.
  
• A separate window displaying a graphical representation of the overall mission should be provided. The user should be able to select a certain phase by selecting it directly on the graph.
  
• The user should be able to edit phase data and delete, add, change and reorder mission phases.

Software Requirements

• The program must exhibit good extendibility and flexibility features to allow for future enhancements and/or modifications to the code.
  
• The types of acceptable phase parameters, phases, and mission parameters should be easily modifiable.
  
• The design of the MPIS should allow for the easy addition of new classes (i.e. data types).

• Sufficient program utilities must be provided to facilitate customization of the system.
  
• Routines to facilitate the definition of unique properties for newly created data types should be provided.
  
• The module must be portable to maximize its usability.
Explanation Of Requirements

Look and Feel

The computer industry has devoted much time and effort to research and develop graphical user interfaces (GUI's) which present a consistent "look" and "feel" to the end-user. The term "look and feel" refers to the method in which the program interacts with the user—both visually and physically. Certain aspects of the look and feel are application-dependent. For example, it is best to manipulate and display data differently in a spreadsheet than from how it is done in a word processor. However, many other aspects are not program-dependent and may be similar across a variety of applications. Certain features, such as the pull-down menus, popularized by the Apple Macintosh, have become widely accepted as the "standard" and have been adopted by competing GUI's (e.g. Microsoft Windows, OS/2, Motif, etc.).

The guiding motivation for the Mission Profile Input System's look and feel is to facilitate the creation and modification of a mission. Large volumes of data are inherent in describing aircraft missions. The creation of a single mission can easily become a long and tedious chore if the user interface is not designed correctly. The purpose of the Mission Profile Input System should be to provide a user interface by which a user can quickly and easily enter, display, manipulate, and modify the large quantity of data. The objective should be to present the user with the proper amount of data—enough to allow complete grasp of the mission at a glance, yet not so much as to make it overwhelming.

User Friendliness

User friendliness describes how easy and accommodating an application interface is. User friendly applications offer clear, consistent options to the user and, for the sake of
efficiency, do not make unnecessary demands on the user. The "friendliness" of an application's interface greatly dictates how successful or useful the application will be. Although a universal standard on user friendliness does not exist and the topic remains somewhat subjective, many practices have been widely accepted and implemented (e.g. asking for user confirmation to validate critical commands, having a logically consistent menu hierarchy, etc.).

Much emphasis must be placed on making the Mission Profile Input System user friendly. The data should be displayed in a complete, logical fashion to enhance its usefulness. To aid in the quick evaluation of the mission, the data should also be displayed graphically. Care must be taken to ensure that no data can be altered without first having to explicitly select a menu item that fully explains the ramifications of the action. For commands that greatly alter or destroy data, user verification of the command should be requested prior to command execution.

**Functionality**

The functionality of a program describes its capabilities. Although the number of user and software requirements which must be satisfied to create an acceptable version of an application is finite, the number of enhancements that can be implemented is inexhaustible. The more features a program is able to support (i.e. the greater its functionality) the more useful it will prove to the end-user.

The most important functionality the Mission Profile Input System should exhibit is the proper maintenance and processing of the trajectory data. The system should also be flexible to allow the definition of new phases and parameters so that the functionality of the module can be quickly customized to adhere to existing aircraft design systems. Moreover, the MPIS should allow for the definition of a set of governing rules for each parameter and
phase which will support the complex handling of the data. This feature will greatly increase the functionality of the system.

**Extendibility and Flexibility**

Program extendibility refers to how easily the functionality of a program can be extended. Programs are no longer designed to meet a limited set of objectives without regard to future, unforeseen requirements. Having to recode or modify entire programs because they lack the flexibility to satisfy a special case can prove to be an expensive and time-consuming proposition. Well-designed programs are structured such that very little, if any, modifications are required when new code is added.

To prepare for unforeseen requirements, programs often exhibit a large degree of modularity—a characteristic which usually lends itself to good extendibility and flexibility traits. Because the Mission Profile Input System is intended to be used by a variety of design software, it must be highly flexible in the phases and parameters it can accept. Therefore, the MPIS should be modular in nature—phases and parameters should be definable independent of the existing code. The newly defined entities should then be able to quickly and easily “plug” themselves into the system.

It is impossible to envision beforehand all the possible demands that may be put upon an application after it has been created. Undoubtedly, demands will arise which cannot be implemented without a major code revision. However, a well designed program should be flexible enough to accommodate reasonable changes and expansions. The MPIS should meet this criterion.
Utilities

Program utilities often refer to routines which are provided to facilitate program-user interaction. It can refer to interactive tools which allow the user to customize the layout of the interface, or it can refer to program functions which facilitate additions to the code. Regardless of the level of their implementation, utilities provide a simple alternative to what would otherwise require more complicated procedures by the user.

The Mission Profile Input System should provide utilities which simplify program customization. It should provide routines to facilitate the creation of new phases and parameters. Functions which assist the user in defining set rules unique to individual phases and parameters should also be provided.

Portability

Program portability refers to the ability of a program to perform correctly in different environments. A highly portable program, for example, may run on an IBM RISC/6000, a Silicon Graphics, and a Hewlett Packard machine. Since the number of machines on which a program will be useful is directly related to the portability a program displays, high portability is quickly becoming a major design objective in modern programs.

Since the Mission Profile Input System is intended to be utilized by a number of different CAD systems, it is likely that it will be implemented in different types of environments. To support such demands, the MPIS should be highly portable (i.e. it should be machine-independent and graphics device-independent).
6.0 DESIGN CONSIDERATIONS

Overview

In order to satisfy the objectives and requirements for the thesis, a good program design for the Mission Profile Input System was required. The flexibility, modularity, and extendibility required of the program made an object-oriented language the obvious choice. Other considerations, however, were not so obvious. Among the design aspects considered were the following:

- Language selection
- Object relationships
- Program maintainability
- Data handling
- Interface layout

Before coding was initiated, an outline of the systematic approach to be used in the creation of the system was developed. A graphical representation of the procedure followed in the creation of the system is given in figure 3. The user requirements were
Figure 3. Approach to the Development of the MPIS
reviewed extensively to ensure that the final product met not only the end user's needs, but also the needs of those wishing to make future extensions and improvements to the code. Recommendations from companies and government agencies, such as Lockheed and NASA, helped guide the final configuration of the interface.

To satisfy these requirements much emphasis was placed on the object-oriented design of the system. Attention was given to such matters as inter-object relationships, object manipulation routines, and the modularity of the overall program.

**Program Design Considerations**

**OBJECT-ORIENTED DESIGN AND LANGUAGE SELECTION**

The compatibility and extendibility requirements of the Mission Profile Input System made an object-oriented language a natural selection. C++, being one of the most accessible and widely used object-oriented language, was selected as the programming language. C++ is a superset of the C programming language. Thus, not only does it provide the powerful features which make object-oriented programming appealing, it also supports the vast library of functions associated with the C programming language.

C++, like other object-oriented programming languages, supports polymorphism, function overloading, inheritance, and data/function encapsulation. These features allow for more efficient, flexible code than would otherwise be possible. In the case of the MPIS, the inheritance and overloading features are especially useful. Inheritance allows the reusability of code—the general features of an object can be specified in a single base class which can then be inherited by derived classes exhibiting these features. Overloading is useful in
writing compact and readable code. Functions with the same name can refer to different routines based on the arguments used when calling them. Thus, functions which perform similar operations, but which must perform them differently for different argument types, can be overloaded. This eliminates the need to employ a different function name for each version of the routine.

The numerous benefits object-oriented programming provides, however, prove even more valuable in developing maintainable code. Class encapsulation allows for the simplification of code debugging, editing, and expansion.

For a complete discussion of object-oriented design the reader should refer to [Booc91].

**OBJECT RELATIONSHIPS**

The key to a successful object-oriented program design is the development of proper relationships between the various classes (i.e. objects). The design of the class hierarchy greatly influences the flexibility and robustness the final code will exhibit. The motivation for object-oriented design is an attempt to create class relationships which mimic the relationships found in the real world [Ents90]. The objective is to identify the most basic objects (base classes) that may be defined and have them serve as the building blocks for more complex ones (derived classes). Basic objects usually reflect the commonalities exhibited by the more complicated objects. One way to look at class hierarchy is by envisioning the derived classes as being more specific instances of the base class. Figure 4 presents a real-life analogy. The illustration depicts a cow as being a more specific instance of a herbivore which, in turn, is a more specific instance of an animal. Thus, if these examples are thought of as being classes, "animals" serves as a base class to its derived class "herbivores" and "herbivores" serves as a base class to its derived class "cow".
Figure 4. Real-Life Object Analogy
EXTENDIBILITY AND MAINTAINABILITY

Extendibility refers to how easily existing code can be modified to accommodate new requirements. As programs become more complex—performing multiple tasks, operating across platforms, supporting active links with other applications, etc.—extendibility has developed into a vital consideration in program design. The key to designing extendible code is instilling it with a high degree of modularity. In this manner, whenever additions to the program are made, a completely separate module can be created independent of the existing code and simply be "plugged" into the program. With a proper program design, the modifications required for the "plugging" process is usually minimal.

Classes, in object-oriented programming, are modular by their very nature. Consequently, languages such as C++ are ideal for creating extendible and maintainable code. Object-oriented programming, however, offers an even more powerful concept for maintainability—encapsulation. Data and functions within classes may be "encapsulated" from the rest of the program such that they are limited in definition to the class in which they are defined. Access to these functions from the outside is restricted to public functions which the programmer defines. The inner working of such classes, in essence, take on the nature of "black boxes" to other programmers wishing to use them. Programmers utilize these classes through the public functions provided without needing to understand the details of how the classes work. Moreover, because the data and functions are fully encapsulated, newly created code is guaranteed to never affect the code already in existence—an extremely powerful feature for program extendibility.
DATA HANDLING

The method by which data is handled in a program greatly influences the efficiency and performance of a program. Speed of program execution is usually the primary casualty in a poorly designed code and proves to be detrimental in the effective use of the program. Proper data handling, however, entails more than the efficiency of how data is processed. It also includes efficient memory management. The dynamic allocation of memory as it is required keeps the amount of memory the computer must devote to the application to a minimum. Moreover, such allocated memory can be freed once it is no longer needed. Keeping memory requirements to a minimum is becoming increasingly important to today's applications. Although memory and processing speeds of modern computers is progressively increasing, the days when an entire machine was devoted to running a single application are quickly disappearing. Machines today often run multiple processes simultaneously. Care must be taken to ensure that an application does not hoard unnecessary resources from the rest of the system.

INTERFACE LAYOUT

The proper design of the user interface is perhaps the most important phase in the development of an interactive program. The user interface acts as the communication bridge between the user and the program. Thus, regardless of how powerful or efficient a program may be, a poorly designed interface will render it useless if the user is unable to utilize it efficiently and correctly.

In designing a good interface, certain considerations must be maintained. The demands made upon the user should be kept to a minimum. Sufficient interaction and communication
should be provided to help the user effectively use the program. However, any unnecessary interactions serve only to detract from the efficient use of the program and should therefore be avoided. To avoid confusion and errors all options and requests made by the interface should be made self-evident to the user. If it is unable to do so, the interface should provide a safety mechanism, such as a request for confirmation, in order to ensure that the user is fully aware of the ramifications of the action to be performed. Finally, a consistent look and feel should be provided to the user. Menus, buttons, messages, requests, etc., should be consistent in their appearance and in the actions they perform. Items which perform unexpectedly should be fully explained and a request for command confirmation should be issued.
7.0 THE MISSION PROFILE INPUT SYSTEM

Class Structure Layout

Figure 5 and Figure 6 illustrate the class structure of the Mission Profile Input System. The structure reflects a balance between inheritance and instancing of classes. As a general rule, inheritance is used if the "derived object type [is] inherently similar to the base type" [Ince91]. If an object "contains" another object, then instancing is used [Booc91]. As an example, every parameter defined by the user is "inherently similar" to the class Parameters, therefore each one of them inherits Parameters. These parameters, in turn, are "contained" by the user-defined phases. Therefore, each phase instances the parameter classes. The following paragraphs clarify the relationship among the classes by explaining Figure 5 in words.

Symbols:

A  \rightarrow  B  
Class A inherits class B.

A  \rightarrow  B  
Class B creates class A.

A  \rightarrow  B  
Class B creates one instance or more of class A.

A  \rightarrow  B  
Class B creates one instance of class A.
Figure 5. Class Relationships
Figure 6. Other Classes Created by Mission_Window
Classes:

**Parameters Class:** This class contains all the functions common to any parameter that may be created. Consequently, this class must be inherited by all user-defined parameters. It must be emphasized that for purposes of the MPIS the term "parameters" implies much more than mere storage containers for values (i.e. variables). Parameters are defined to be classes in order to allow the association of unique rules with each type. For example, each parameter may contain functions which allow it to automatically calculate itself, impose upper and lower limits on its value, etc.

**User-Defined Parameters:** A valid parameter may be defined by the creation of its own class. The class contains all the information pertaining to the parameter, such as name, value, default value, and unique rules. The class *Parameters* contains all the functions common to any parameter that may be defined. Consequently, every newly defined parameter class must inherit *Parameters*.

**Phases Class:** This class contains all the functions common to any phase that may be created. Consequently, this class must be inherited by all user-defined phases.

**User-Defined Phases:** Similar to user-defined parameters, a valid phase may be defined by the creation of its own class. The class contains all the information which is relevant to the corresponding phase: its name, a linked list of the phase parameters, and all the rules which are unique to the phase. The class *Phases* contains all the functions common to any phase that may be defined. Consequently, every newly defined phase class must inherit the class *Phases*. Moreover, because each phase must create its own linked list of parameters, every phase class instances all of the parameter classes.
Mission_Parameters Class: This class contains all the information which pertains to the mission parameters. These parameters are strictly dependent on the requirements of the CAD system onto which the Mission Profile Input System is mounted.

Missions Class: This class provides for the manipulation of mission data. Functions to create, modify, and destroy data are contained within this class. A mission is comprised of a linked-list of phases and a linked-list of mission parameters. Thus the classes corresponding to these objects are instanced within the Missions class. Note that the Missions class inherits the Mission_Parameters class and appears to violate the general rule for inheritance. A mission is not "inherently" similar to a mission parameter. The reason for inheriting the Mission_Parameters class is to provide the Missions class direct access to the Mission_Parameters functions. Although it has no notable affect on the efficiency of mission parameter manipulation, it does make the code more readable. Functions of the Missions class used in creating the Mission_Parameters linked list need not worry about passing mission parameter information between them. This information becomes globally known within the Missions class.

Phase_Diagram_Window Class: This class creates the graphical representation of the mission data (i.e. the phase diagram). To create the window in which the phase diagram is displayed the class inherits the class Scroll_Window developed by Andreas Steude at Virginia Tech [Steu93]. The class is instanced by the Mission_Window Class whenever it is requested by the user.

Mission_Window Class: This class acts as the central coordinator of all the other classes in the Mission Profile Input System. It coordinates the creation and control of the user-interface with the proper processing of the mission data. To create the Motif-like window in which the data are displayed, the Mission_Window Class inherits the
Scroll_Window Class. Other features, such as the pop-up menus, text input areas, and menu items are created by instancing classes created by Scott Woyak, also at Virginia Tech [Woya92 & Woya93]. To keep track of the data, the Mission_Window Class first inherits and then instances the Missions Class. By inheriting the Missions Class all the routines required for manipulation of the mission data become readily available to the Mission_Window Class.

THE PHIGS-BASED, MOTIF-LIKE INTERFACE

As mentioned above, the development of the interface for the Mission Profile Input Mission relied heavily on a PHIGS-based, Motif-like framework developed by Scott Woyak at Virginia Tech [Woya92, Woya93, & Mykl93]. For the sake of completeness, a brief description of this framework is provided here.

The framework consists of five major groups of classes: Windows, Interface Managers, Menu Managers, Menu Items, and Menu Item Managers. The Interface Manager is the central processing class—it manages the various windows which a user creates. A variety of windows have been defined, including geometry managers (windows that display and maintain a PHIGS view), pop-up menus (windows which display menu items), and dialogue managers (windows which are used to exchange information with the user). Through inheritance, all these windows inherit the “Windows” base class which enables them to be managed identically by the Interface Manager. Similarly, menu items inherit the “Menu Item” class which enables them to be managed by the Menu Manager. Figure 7 outlines the class organization of the framework.
Figure 7. Class Organization for PHIGS Motif-like Framework [Woya92]
**Framework For Handling Data**

Sound data handling techniques are a major prerequisite to a successful program. Inefficient routines often result in poor or unreliable program performance. To address the concerns of proper data handling, all data handled by the MPIS is performed through the use of dynamic linked lists. Figure 8 illustrates the basic data configuration of the module. As shown by the figure, all mission data is handled by three types of linked lists: the parameter linked list, the phase linked list, and the Mission_Parameters linked list. Only one phase and Mission_Parameters linked list exists per mission. However, the number of parameter linked lists is equal to the number of phases in the mission—one for each phase.

Employing linked lists to manipulate object data proves to be very efficient. The swapping, moving, and inserting of objects can be performed much quicker through pointer manipulation than could otherwise be possible with direct object manipulation. Moreover, pointers allow for the dynamic allocation of memory. The module's memory requirements can be limited to what is absolutely essential. More importantly, memory can be freed once an object is deleted—a frequent occurrence during mission modifications.

The Mission Profile Input System is highly flexible in the types of input it can accept. Unfortunately, however, this flexibility introduces a fair amount of complexity as to how data must be handled. User input must be processed through numerous functions to ensure the validity of the entry. Moreover, functions are needed to ensure that the entire mission reflects the effects of the input. For an explanation of how the MPIS handles its data, refer to the section “Implementation and Examples of Results.”
Figure 8. Internal Data Relationships
The User Interface

The primary purpose for developing the Mission Profile Input System is to facilitate the creation of aircraft missions. With this in mind, the user interface for the system is designed to accelerate the entry and manipulation of large amounts of data. The layout of the interface is intended to give the end-user the ability to make an overall assessment of a mission very quickly. The proper balance for displaying a complete representation of the mission without cluttering up the screen is achieved by making extensive use of pop-up menus. The pop-up menus employed ensure that data and options not immediately relevant to the manipulation of the mission profile are hidden from the viewer until they are explicitly requested.

To aid in the quick evaluation of the entire mission, all phase and parameter data are displayed to the user simultaneously. Editing, however, cannot be performed directly on the data on the screen. For purposes of editing, data are presented to the user one phase (or parameter) at a time. By clicking on the value on a screen, an editing pop-up menu displaying the data to be modified is activated. This approach is deemed desirable since it focuses attention on the data most closely associated with the parameter to be altered. It should be emphasized that data entry efficiency is not sacrificed by using this approach. The number of steps required by the user to enter and modify the data are exactly the same as it would be by using the direct entry approach.

The Mission Profile Input System is intended for experienced and novice users. A simple consistency to the system is maintained by keeping the methods by which data are displayed highly structured. Except in cases where it is beneficial to display more than one pop-up menu, the number of pop-up menus on the screen is limited to one. Recognizing
that such a highly structured layout may serve as a hindrance to more experienced users, multiple methods for manipulating data have been provided. For example, a novice user can modify each parameter individually. This method provides immediate feedback on the effects the modification has on the rest of the mission profile. The more experienced user can manipulate data faster by modifying parameters at the phase level. At this level all the parameters for a single phase may be modified simultaneously. Although this method is less user friendly than modifying parameters individually, a person who is familiar and confident with the changes to be made can create and modify a mission much quicker in this manner.

The look and feel of the interface was developed using a Motif-like, object-oriented, interface framework [Woya92 & Woya93]. The framework was developed at Virginia Polytechnic Institute and State University and is based upon the three-dimensional international graphics standard, PHIGS. It allows for the creation of Motif-like windows, buttons, menus, etc., with Motif-like characteristic (e.g. resizeable, draggable, etc.). Because it is based on PHIGS, the interface is machine-independent, thus supporting the portability requirements of the system.
8.0 CLASS DESCRIPTIONS

Overview

The Mission Profile Input System consists of six (6) major classes: Parameters, Phases, Missions, Global_Variables, Mission_Window, and Phase_Diagram_Window. The following sections describe the purpose of each class and give detailed descriptions of the important data and functions they contain. For the exact protocol for invoking each function the reader should refer to Appendix B.

Parameters

This class contains functions which are common to every user-defined parameter. Consequently, every parameter which is defined for the system must inherit this class. The Parameters class contains functions used in positioning it within the parameter linked list. Other functions of the class are intended to facilitate the manipulation of data at the parameter level (i.e. modifications are limited in scope to the parameter). They include some powerful virtual functions which offer greater flexibility in the definition of new parameters. Detailed descriptions on how to use and implement these functions are given in the section System Customization.
DATA MEMBERS-PARAMETERS CLASS

The Parameters class contains all the data pertaining to a parameter. It stores its value, default value, and name. Since keywords (e.g. LAST, SAME, etc.) are valid entries which represent a numeric parameter value, the value of a parameter must be stored in its two formats: character and float. In cases of keywords, the relationship between the character and float values is determined by the pre-defined rules governing the keyword. Otherwise, the character value is simply the character representation of the numeric value.

The variables used to store parameter information are:

- cvalue: Stores the character value of the parameter.
- fvalue: Stores the float value.
- cdefault: Stores the default character value.
- fdefault: Stores the default float value.
- param_title: Stores the name of the parameter. The name of the parameter must be a single word. However, underscores (_) can be used to represent white spaces. These will be removed whenever the name is displayed on the screen.

In addition, the class also contains the variable next—a pointer to the next parameter within the parameter linked list. In this fashion, the position of the parameter within its linked list is maintained.

FUNCTIONS OF PARAMETERS

The primary reason for defining each type of parameter to be its own class is that it offers greater extendibility and flexibility. Of interest is the ability to define unique rules governing the values of each parameter. One example is the definition of keywords.
Keywords (and their corresponding set of rules) may be defined to govern the numerical value of the parameter. Often times, keywords are used to describe a dependency of the current parameter to the value of another parameter.

To facilitate the creation of a governing set of rules for a parameter, a series of functions is provided by the Parameters class. These functions are used to return a pointer to other parameters in the current or previous phase. By obtaining a pointer to a certain parameter, its value can be easily retrieved. The functions to retrieve a pointer are:

\begin{itemize}
  \item \texttt{prev} - Returns a pointer to a specified parameter of the \texttt{previous} phase.
  \item \texttt{cur} - Returns a pointer to a specified parameter of the \texttt{current} phase.
  \item \texttt{nxt} - Returns a pointer to a specified parameter of the \texttt{next} phase.
\end{itemize}

The rest of the functions contained by the Parameters class are used to set and retrieve the data of the parameter. Some functions, such as the \texttt{get_value} function are overloaded to allow for the two types of value formats. Other functions, such as the \texttt{set_value_to} and \texttt{check} functions are declared to be virtual so that they may be redefined. By overloading these functions a unique set of rules can be defined for the parameter. Some of the more important functions follow:

\begin{itemize}
  \item \texttt{put_next} - Sets a pointer to the next parameter in the linked list.
  \item \texttt{set_param_title} - Sets the name of the parameter.
  \item \texttt{set_value} - Sets the value.
  \item \texttt{set_default_value} - Sets the default value.
  \item \texttt{get_next} - Returns a pointer to the next parameter in the linked list.
  \item \texttt{get_param_title} - Returns the name of the parameter.
  \item \texttt{get_value} - Returns the value of the parameter.
  \item \texttt{get_default_value} - Returns the default value.
  \item \texttt{check} - Checks the validity of the current value of the parameter
\end{itemize}
Phases

This class contains functions which are common to every user-defined phase. Consequently, every phase which is defined for the system must inherit this class. The constructor of the class (an initialization routine which is launched automatically when the object is created) creates the parameter linked list associated with the phase. In addition to functions used to create the parameter linked list, the class also contains functions which allow for the manipulation of data at the phase level. Phase level manipulation allows for the modification of an entire phase or of one or more of its parameters. The Phases class also contains certain private and virtual functions designed to facilitate the definition of new phases and the rules which may be imposed upon its parameters.

DATA MEMBERS-PHASES CLASS

A large portion of the information attributable to a phase is contained within its parameter linked list. The parameter linked list is self-contained—i.e. the parameter data is encapsulated within the parameter object and is not part of the phase object. Consequently, the amount of data stored by the Phases class is quite small. Only three variables are required. They are as follows:

phase_title - Stores the name of the phase. The same naming convention applies for naming a phase as does for naming a parameter.
first_param - A pointer to the first parameter of the parameter linked list. Only the location of the first parameter is required since each parameter will point to the next one in the list.
next - A pointer to the next phase in the phase linked list.
FUNCTIONS OF PHASES

Like parameters, phases are defined to be their own objects in order to enhance their extendibility and flexibility. Because modifications are done at the phase level, comparisons can easily be made between the various parameters of a phase. This allows for the implementation of unique dependencies among the parameters—a powerful feature in expediting the creation of a mission profile.

As mentioned, the creation of parameter dependencies relies heavily on comparing parameter values. To facilitate the task, a set of functions is provided by the class Phases which return the value of a specified parameter in either the current or previous phase. These functions are follow:

- `number` - Returns the float value of the specified parameter in the current phase.
- `prev_number` - Returns the float value of the specified parameter in the previous phase.
- `word` - Returns the character value of the specified parameter in the current phase.
- `set` - Sets the specified parameter to the specified value. The function is overloaded to provide for character and float values.

The majority of the remaining functions contained by the Phases class may be divided into two types: functions to create the parameter linked list, and functions to manipulate the data within the parameter linked list.

The task of creating the parameter linked list is primarily handled by two functions. The first, the `get_proper_param` function, returns a pointer to a parameter object which it creates. The type of object it creates is determined by the parameter name which it receives.
as an argument. The second function is the load_param_values function which creates the linked list from the information it receives. To assist it in creating the linked list, this function makes repeated calls to the get_proper_param function.

The second type of function provides for the setting and retrieving of parameter data. The parameter to be modified may be specified by indicating its position within the parameter linked list or by specifying the parameter by name. Some of the functions provided by the class are as follows:

set_param_char_value - Sets the value of the specified (by name) parameter to the character value.
set_param_float_value - Sets the value of the specified (by name) parameter to the float value.
set_first_param - Sets the parameter specified (by pointer address) to be the first parameter of the parameter linked list
get_param_pointer - Returns a pointer to the specified (by position) parameter.
get_float_param_value - Retrieves the value of the specified (by name) parameter.
get_first - Returns a pointer to the first parameter of the linked list

In addition to the aforementioned functions the class Phases has two very important virtual functions. The first is the check function which executes the various rules which have been defined for the phase. The function contains all the rules defined by the user and therefore must be edited when wishing to alter the rules for the phase. The second virtual function is the geo_segment function. This routine creates the graphical representation of the phase. The base definition for the function creates a straight line for every type of phase. To create more complicated representations, this function for the particular phase must be redefined.
**Mission_Parameters**

The Mission_Parameters class is the object data type of the mission parameters for the Mission Profile Input System. The purpose for defining each mission parameter to be a distinct object is to provide for system extendibility. Aircraft CAD systems require various information for their particular analysis programs. Some systems require additional variable data, such as data type and data format. To accommodate additional requirements imposed by the various systems, the data type (i.e. class) may be quickly and easily modified.

**DATA MEMBERS-MISSION_PARAMETERS CLASS**

The Mission_Parameters class of the stand-alone version of the MPIS provides for four pieces of information for each mission parameter: the name, the value, a code indicating whether the variable should be displayed, and a comment about the variable.

<table>
<thead>
<tr>
<th>field</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>-The name of the variable.</td>
</tr>
<tr>
<td>value</td>
<td>-The value of the variable</td>
</tr>
<tr>
<td>display</td>
<td>-A code indicating whether the variable should be displayed on the Other_Variables Menu (See the User Guide in Appendix B).</td>
</tr>
<tr>
<td>comment</td>
<td>-Storage space for miscellaneous information about the variable. This variable is included to allow for one additional piece of information without the need to physically modify the code.</td>
</tr>
</tbody>
</table>

Like parameters and phases, mission parameters are maintained and manipulated in a linked list. Each mission parameter contains the variable **next** which points to the next mission parameter in the linked list.
FUNCTIONS OF MISSION_PARAMETERS

The functions of the class Mission_Parameters are limited to setting and retrieving the various data of each variable. For every variable in the class, there exists one function to set it and one to retrieve it. In adapting the MPIS to different CAD systems it may become necessary to add more complex routines. The functions defined for the class in the stand-alone version of the MPIS are:

- `put_next`: Sets a pointer to the next mission parameter of the linked list.
- `set_mp_name_to`: Sets the name of the mission parameter to whatever is specified.
- `set_mp_value_to`: Sets the value of the variable.
- `set_mp_display_to`: Sets the display code.
- `set_mp_comment_to`: Sets the miscellaneous piece of information.
- `get_next`: Returns a pointer to the next mission parameter of the linked list.
- `get_mp_name`: Returns the name of the current variable.
- `get_mp_value`: Returns the value of the current variable.
- `get_mp_display`: Returns the display code.
- `get_mp_comment`: Returns the miscellaneous comment.
Missions

This class is the object data type of a mission. It contains all the data and functions necessary to create, manipulate, and modify a mission profile. It creates a mission interactively or by reading information from a file. Either way, it creates the appropriate phase and mission parameter objects from the information it receives. The Missions class coordinates the plethora of data by grouping it into two distinct linked lists: the phase linked list and the Mission_Parameters linked list. Linked lists are utilized because they increase the efficiency by which the data can be manipulated. It is more efficient to manipulate a list of objects through the modification of their pointers, than it is to manipulate the objects themselves.

DATA MEMBERS-MISSIONS CLASS

The majority of the information a mission contains is stored within the phase and Mission_Parameters linked lists. Therefore, the amount of data stored by the mission class itself is quite small. It is limited to the name of the mission, the list of parameter names which the mission contains, and a pointer to the first phase of the mission. A variable also exists to store the names of available phases which can be dynamically added to the mission. The actual variable names used by the mission are as follow:

mission_name
- Stores the name of the mission. The same naming convention applies for the naming of a mission as does for the naming of a phase or parameter.

param_names
- An array which stores the name of the parameters contained within the mission. The order in which the names appear in the array is the order by which they appear on the screen.
names_of_avail_phases - An array which stores the names of phases which can be added interactively by the end user during a session.

FUNCTIONS OF MISSIONS

The Missions class contains a wide variety of functions designed to manipulate mission data. The functions perform manipulation at the mission level which implies that they have open and direct access to all data. This allows for the creation of sophisticated relations between any number of mission variables. For example, the value of a parameter within a certain type of phase can be linked to the value of a specific mission parameter. To facilitate the creation of such relations, functions to set, retrieve, and navigate through data are provided. An attempt was made to make the list of functions comprehensive. If more complex or specialized functions are required in the future, they will no doubt be definable by the combination of two or more of the following functions. The functions are grouped into four distinct types: functions to create a mission, functions to manipulate parameters, functions to manipulate phases, and functions to manipulate mission parameters.

Functions to Create Mission:

get_proper_phase - This function returns a pointer to a phase object which it creates. The type of object it creates is determined by the phase name it receives as an argument. It also receives a list of values which it assigns to the linked list of parameters which is created when the phase is instanced.

create_mission - The function which creates a mission profile by reading data from a file. From the data read, it creates the linked list of the phases by making successive calls to the get_proper_phase function. Once it completes the phase linked list, it creates the linked list of mission parameters by instancing the Mission_Parameters class.
save_miss

-This function archives the current mission. It stores the mission data in a recognizable format to a file specified by the user.

Functions to Manipulate Parameters:

get_num_of_params

-Returns the number of parameters in the parameter linked lists. Since every phase must contain all the parameters available, every parameter linked list necessarily has the same number of components.

update_element

-Changes the value of the parameter to the newly specified one. The parameter to be modified is specified by position—the position of the phase it belongs to in the phase linked list, plus the position of the parameter within its parameter linked list. The function is overloaded to allow for character and float value types.

reset_to_default_value

-Resets the value of a parameter to its default value. The parameter is specified by position.

retrieve_float_value

-Retrieves the float value of the specified (by position) parameter.

retrieve_char_value

-Retrieves the character value of the specified (by position) parameter.

move_param

-Moves a parameter specified by name from its current location within the parameter linked list to the newly specified one.

retrieve_param_title

-Retrieves the name of the parameter specified by position. Since the order in which parameters appear is the same for every phase, only the location of the parameter within the parameter linked list needs to be specified.

get_max_param_value

-Retrieves the maximum value of a parameter. The parameter must be specified by name. The routine traverses the phase linked list of the mission and returns the maximum value it encounters for the parameter in question.

get_min_param_value

-Retrieves the minimum value of a parameter. The parameter must be specified by name. The routine traverses the phase linked list of the mission and returns the minimum value it encounters for the parameter in question.
copy_element - The function copies the value of one parameter onto a second one. Both parameters are specified by position. Note that the usefulness of this function is somewhat limited since only one parameter can be copied at a time. However, the function can serve as a building block to more complex copying routines.

Unlike parameters, phase object types are not unique within their respective linked list. More than one CLIMB phase, for example, can be contained within a mission. Consequently, when modifying a phase, it should never be specified by name. Instead, a phase should always be specified by its position within the phase linked list.

Functions to Manipulate Phases:

get_phase_pointer  - Returns a pointer to the specified phase.
get_num_of_phases  - Returns the number of phases contained by the current mission.
insert_phase       - Inserts a new phase into the specified position. The new phase is created by calling the get_proper_phase function which creates the phase object from the phase name it receives.
add_phase          - Works identically to the insert_phase function except that instead of inserting the new phase in the phase link list it adds it to the end of the list.
delete_phase       - Removes the specified phase from the phase link list. The object is deleted to free up memory.
default_phase      - Works identically to the get_proper_phase function except that it only takes one argument—the name of the phase. It reads the names and values for its parameter linked list from a file which contains their default values. This is used to quickly add entire phases to the mission profile.
reset_phase_defaults - Resets all the parameters within the phase to their default values. This is done by making successive calls to the reset_to_default_value function.
move_phase         - Moves the specified phase to a new position.
retrieve_phase_title - Returns the name of the specified phase.
assign_values_to_phase - Assigns a list of values to the parameters of the phase. The values are assigned sequentially—the first value is assigned to the first parameter, the second value to the second, etc. Assignment of values continues until the list of values is exhausted or the number of parameters is exceeded.

Functions to Manipulate Mission_Parameters:

load_mission_parameters
- This function is called by the create_mission function to create the Mission_Parameters linked list. It reads information from a specified file and properly loads it into the Mission_Parameters object is creates.

write_mission_parameters
- This function is called by the save_mission function to store the mission parameters in their proper format. The file to which it writes is received as an argument.

get_mp_pointer
- This function returns a pointer to the specified mission parameter. The function is overloaded to allow specification of the variable by either name or position.

set_mp_value_to
- This function is also overloaded. It sets the specified (by name) mission parameter a new value. The function is overloaded to provide for both character and float values.

get_float_mp_value
- This function retrieves the float value of the specified mission parameter. The variable must be specified by name. The class Mission_Parameters makes no provisions for storing mission parameters in both character and float formats. Thus the function tests whether the character value can be converted to a float type. If it cannot, it returns an error message.

get_char_mp_value
- This function returns the character value of the mission parameter.
**Phase_Diagram_Window**

The Phase_Diagram_Window class creates a graphical representation (i.e. phase diagram) of the mission data to aid the user in a quick assessment of the mission. The class is comprised of mostly private variables and functions which enable it to display the diagram. Except for the Motif-like qualities of the window, which allow the user to resize and drag the window, the class makes no provisions for interactive modification of how the diagram is displayed. The user can, however, select a particular phase and the corresponding phase editing menu in the main window (i.e. the window containing the data) will be activated and will be ready for input.

**DATA MEMBERS-PHASE_DIAGRAM_WINDOW CLASS**

As previously mentioned, the data contained within this class is used mostly to allow the class to properly display the phase diagram. Variables are defined to store information such as the width, height, and scaling factors of the diagram. One important variable that needs mention, however, is the *mission_window* variable. This variable stores a pointer to the main window (i.e. the window which displays the data). This pointer is used to return control of the program to the main window whenever the user clicks on a leg of the diagram.

**FUNCTIONS OF PHASE_DIAGRAM_WINDOW**

As is the case with the data pertaining to this class, the functions defined are mostly for internal use. The functions automatically retrieve data from the mission and properly scale it
to create the phase diagram. No input from the user is required. A brief explanation on the purpose and functionality of the more important functions follows:

`set_proper_scale` - This function calculates the correct scaling factors in the x and y directions to properly display the graph. The x scaling factor is always set equal to one and the y value is calculated accordingly. The y value is calculated using the following formula:

$$\frac{|\text{maximum}_x - \text{minimum}_x|}{(6*\text{maximum}_y)}$$

It was determined, through the process of trial-and-error, that this formula generates the desired visual results.

`set_proper_zoom` - This function retrieves the maximum and minimum x and y values from the mission and sets the correct view magnification to display the scaled diagram properly. The required maximum and minimum values are retrieved by using the `get_max_param_value` and `get_min_param_value` functions respectively. These functions are defined for the class `Missions`. Before the magnification of the window is set, a call to the `set_proper_scale` function is made to determine the actual size of the scaled diagram.

`create_axis` - This function creates the axis for the diagram. Since the magnification of the display window is continually changing, the size of the axis must also be continually updated to retain a fixed appearance on the screen.

`create_geometry` - This function creates the phase diagram. It creates a PHIGS structure and inserts each leg of the phase diagram by calling the `geo_segment` function of each phase.

`display_error_message` - This routine is used to display an error message in the phase diagram window whenever the graph can not be constructed.
Mission_Window

The Mission_Window class serves as the coordinator for all other classes in the Mission Profile Input System. The class controls all aspects of the program—from the interactive display on the screen to the “behind the scene” synchronized execution of data manipulation routines. The data manipulation functions are directly accessible to the Mission_Window class through the inheritance of the class Missions. To assist it in the creation of the user-interface, classes created at Virginia Tech by Scott Woyak [Woya92 & Woya93] and Andreas Steude [Steu93] are used. These classes create many of the objects which make up the graphical user interface. The names of the particular classes employed are given in Appendix B under “Other Classes.”

DATA MEMBERS-MISSION_WINDOW CLASS

All the data contained within the Mission_Window class is for internal use only. Pointers and flags for object types such as menus, menu items, labels, and color groups are defined and used to properly coordinate the various interactive features of the user interface. None of these variables has significance beyond being holders for information required by the class.

FUNCTIONS OF MISSION_WINDOW

The functions of the Mission_Window class can be generally divided into two types: functions which create objects to be displayed on the screen and functions which govern
and process the internal control of the program. The majority of the functions are used to create the various interactive menus of the user interface and to transfer program control between them.

Functions Which Create Objects:

The stand-alone version of the Mission Profile Input System contains sixteen separate menus. These menus allow the user to manipulate the data and the way in which it is displayed. Although separate functions exist for each type of menu, all the functions are similar in structure. The pop_up menu and its menu items are created by instancing the appropriate classes [Woya92] and control is maintained by executing a control loop until some valid input is received from a logical input device. The sixteen functions which create the different menus are as follow:

- **filename_menu** - Creates a menu which prompts the user for the name of a file.
- **mission_parameter_menu** - Creates the only menu which does not demand complete program control. The control can be toggled between this menu and other windows. The menu displays a selected list of mission parameters.
- **phase_options_menu** - Creates a menu which displays the various options available for manipulating phases.
- **options_menu** - Creates a menu which displays the general options available to the user. Routines that can be executed from this menu affect the entire mission. Examples are: Save Mission, Retrieve Mission, Modify Display, etc.
- **add_phase_menu** - Creates a menu which displays the phases available for addition to the mission.
- **phase_menu** - Creates a menu which lists all the parameter values of a particular phase. This menu is the primary method by which parameter values can be modified.
parameter_menu - Creates a menu which lists all the values of a particular parameter (i.e. lists the value of a particular parameter for all the phases). This menu is an auxiliary method by which parameter values can be modified.

move_parameter_menu - Creates a menu which allows the user to move a particular parameter.

confirm_menu - Creates a menu which requests confirmation to the previous command.

message_menu - Creates a menu which displays a message to the user. The menu waits for acknowledgment from the user.

apply_defaults_menu - Creates a menu which allows the user to reset a particular parameter to its default value. Of the sixteen menus, this is the only one which never removes the previous menu from the screen. Consequently, more than one menu is displayed whenever this menu is activated. However, although multiple menus appear, only this one accepts input.

row_column_menu - Creates a menu which allow the user to vary the column and row spacing of the display.

quick_input_menu - Creates a menu which allows the user to modify all of the parameters of a particular phase simultaneously. This method is not as user-friendly as the method offered by the parameter_menu, but it is much more efficient for the experienced user.

retr_del_file_menu - Creates a menu which lists all the files containing mission data. The routine looks for all files with a "miss.data" extension in a specified directory.

other_variables_menu - Creates a menu which lists all mission parameters specified as "viewable" in the select_variables_menu.

select_variables_menu - Creates a menu by which the user can select the variables to be displayed by the other_variables_menu.

The user-interface consists of objects aside from menus. The most significant of these are the five push-buttons which appear directly above the main window (the window which displays the data). The buttons are created using the create_additional_components
function. This function is available to the Missions class from inheritance of the Scroll_Window class [Steu93].

Functions Which Govern Control:

As mentioned previously, the functions which create the menus contain control loops which maintain control within their corresponding menus. However, this control is only local—the menu, and its control loop, are destroyed when the menu disappears from the screen. General control is primarily enforced by three functions. These functions process input from the logical input devices and act accordingly. All three functions are defined to be virtual in the base class Scroll_Window [Steu93] and are redefined by the Mission_Window class to reflect its particular needs. The functions are as follows:

process_additional_components

-This function is used to process information from the additional components defined in the Mission_Window class (objects which are defined in the derived class, Mission_Window, but not in the base class Scroll_Window). The Mission_Window class has five such components—the five push buttons which appear at the top. Thus, this function processes information whenever one of these buttons is selected.

process_geometry_view

-This function processes information whenever a logical input is detected inside the geometry view (i.e. inside the window in which the data is displayed). The function returns a variety of information concerning the location and description of the item selected. However, only the logical pick information is of interest for the purposes of the class.
process_from_mouse  -This function processes information whenever the computer mouse button is pressed. Because the Mission Profile Input System relies heavily on input from the mouse, this function is perhaps the most important of the control processing functions. When input from the mouse is detected, the function determines what item on the screen was selected and routes control appropriately. One of the functions it calls is the process_additional_components routine which tests whether one of the components defined within Mission_Window was selected.

The Mission_Window class contains additional functions to those listed so far. However, they are mostly intended to perform functions that are frequently required by the routines just described. Some of the more important ones are listed.

initialize_colors  -Resets the PHIGS color table to ensure that certain color indices reflect certain colors.
refresh_window  -Updates the window to reflect the latest changes.
toggle_window  -Toggles the window between its current size and the specified one.
scan_for_pick  -This function is used extensively by functions which create the menus. It takes a pick id and determines what type of data was selected. It routes control accordingly.
9.0 SYSTEM CUSTOMIZATION

Overview

The Mission Profile Input System is intended for use by a variety of CAD systems. Recognizing that requirements will vary for different programs, the module was designed to be customizable. Parameters, phases, and mission parameters can be easily created, modified, or deleted, to accommodate the various requirements. Much thought was given to whether such flexibility should be given at the implementation or end-user level. Finally it was decided that this flexibility should be introduced at the implementation level. It is foreseen that the programs onto which the module will be implemented will have a fixed set of requirements (i.e. the types of parameters, phases, and mission parameters the CAD system will accept). In order for the module to function correctly, such requirements must be addressed upon the module’s implementation. Once the module is implemented, the user will have no need for the ability to define new object types, since the host CAD system is unable to process them.

The following sections describe in detail the steps required to customize the module. They describe how to create, modify and delete parameters, phases, and mission parameters. However, to accomplish this requires some knowledge of the data files which the MPIS creates for archiving purposes. For this reason, the explanation begins with a description of these files.
Trajectory Data Files

Figure 9 lists a typical trajectory data file. The first two lines of the file contain the name of the mission and the number of mission phases, respectively. The third line gives a listing of all the parameters. Note that the entire list of parameters MUST be contained within one line and that entries MUST be separated by at least one tab character. Beginning on line four, the phases contained by the mission are listed. Each line consists of the phase name plus the values of its parameters. The list of values correspond in order to the list of parameters given in line three. Tabs are used to separate the entries within each phase and end-of-line characters are used to separate the phases.

All defined parameters must be listed for each phase. A distinction should be drawn between the list of parameters found on the second line of the file and the list of phases which subsequently follows. The list of parameters lists ALL the parameters which have been defined for the module. Since multiple instances of the same parameter are not allowed for any given phase, each entry in the list must be unique. The list of phases, on the other hand, lists the phases which the mission contains and do not necessarily reflect all the phases which are available. Moreover, since each mission may contain multiple occurrences of the same phase, each phase entry need not be unique.

After the list of phases contained by the mission is exhausted, a listing of the mission parameters begins. Each line thereafter contains the data for one mission parameter: the name, the value, a code denoting whether it should be displayed, and a brief associated comment. Like previous entries, each of these entries must be separated by at least one tab character.
| line 1 | Mission_Name |
| line 2 | 4            |
| line 3 | PARAM1  PARAM2  PARAM4  PARAM3  PARAM5 |
| line 4 | PHASE1  value1-1  value1-2  value1-3  value1-4  value1-5 |
| line 5 | PHASE2  value2-1  value2-2  value2-3  value2-4  value2-5 |
| line 6 | PHASE3  value3-1  value3-2  value3-3  value3-4  value3-5 |
| line 7 | PHASE4  value4-1  value4-2  value4-3  value4-4  value4-5 |
| line 8 | gv1_name  gv1_value  gv1_display_code  gv1_comment |
| line 9 | gv2_name  gv2_value  gv2_display_code  gv2_comment |

additional global variables

Figure 9. Trajectory Data File
Phase Defaults File

To facilitate and expedite the creation of trajectory missions by the end user, entire phases may be added with the click of a button. The file phase.dfl.miss is found in the same directory as the trajectory data files and contains a listing of all the phases which are available to the MPIS. Figure 10 shows the file.

Line one of this file gives a listing of all the parameters. The order of the list is arbitrary. However, once defined, it governs the order in which the parameter values must be listed in the phase definitions which follow. The rest of the file consists of information for the default phases. As in the Trajectory Data Files, phases are defined within a single line. The line is comprised of the phase name and the default values of the phase parameters. As in previous files, all entries within a line must be separated by at least one tab character.
<table>
<thead>
<tr>
<th>line</th>
<th>PHASE</th>
<th>value</th>
<th>value</th>
<th>value</th>
<th>value</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>PHASE1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>PHASE2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>PHASE3</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>PHASE4</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

additional phases which are defined

Figure 10. The Phase.dfl.miss File
Parameters

Creation

Figures 11(a) and 11(b) give a typical class definition for a parameter. A separate class must be defined for each type of parameter. Note that each parameter class which is created must inherit the class Parameters which contains all the functions associated with the object. New parameters can be easily defined by adhering to the following procedure:

1. Copy the code given in Figure 11(a).

2. Change the name of the class to reflect the new parameter. Ensure that the name change is reflected in the constructor of the class.

3. Assign the class a new reference name. This is done by changing the name inside quotes in the set_param_title_to function call. The name given to the parameter must be a single word and is the EXACT title by which the parameter must be referred when referenced by name from any other part of the code. Although the name is restricted to being a single word, underscores may be used in its definition. These underscores will be treated as blank spaces when the title is displayed on the screen.

4. Add a new conditional statement to the function Phases::get_proper_param. Figure 11(b) illustrates the format of the conditional statement. Note that the name of the new parameter must be placed inside the quotes found in the conditional test and that the newly defined parameter class must be reflected in the call to the new data type.

5. Modify the phase.defaults.miss to reflect the newly created parameter.
class parameter_name : public Parameters
{
    public:
        parameter_name (char value [])
        {
            set_param_title_to("NAME");
            // give name of parameter
            if (format_type (value))
                set_value_to (value);
            else
                set_value_to (atoi (value));
            set_default_values (value);
            next = NULL;
        }

        arg3 = list of acceptable input
        void check () { checker (arg1, arg2, arg3); }  // optional: used to list acceptable entries
        void check () { checker (arg1, arg2, arg3); }  // optional: redefine function.
        // see following sections

    }

Figure 11(a). Basic Parameter Definition

Parameter *Phases::get_proper_param (char* name, char* value)
{
    .
    .
    .
    else if (! strcmp ("NAME", name))
    {
        param = new parameter_name (value);
        return param;
    }
    .
    .
    .

Figure 11(b). Modification Required For Parameter Definition
Modification

The primary reason for defining each type of parameter to be a separate class is that it offers the flexibility of defining set rules which are unique to each one. This is done through the use of virtual functions. Virtual functions are functions which are defined in the base class but may be redefined by derived classes. The power of such functions is that each derived class can make identical function calls without necessarily referring to the same routine. Virtual functions which are not redefined by the derived class default to the base class definition.

The Parameters class offers two virtual functions: the set_value_to and the check functions (Appendix B describes the exact function protocol).

THE SET_VALUE_TO FUNCTION

The first function, set_value_to, is overloaded. The term "overloaded" implies that separate routines are called depending on the types of arguments used in calling the function. The reason for overloading this function is that the procedure required for setting the value of the parameter is contingent upon the type of the value. A value of type float is treated differently than a value of type character.

In its base class definition, the set_value_to function simply assigns the argument as the value of the parameter. However, because the function is virtual, it may be redefined to perform more sophisticated assignments. For example, bounds may be imposed on the value of the parameter by performing conditional tests. If the value is above (or below) a certain value, the value can be reset to the proper bound, an error message can be
displayed, or the previous value of the parameter can be restored. In other instances, where only a certain type of data is allowable, this function can be redefined to ensure that the data meets the proper criteria. Regardless of its sophistication, however, it should always be kept in mind that the assignment occurs at the parameter level. Consequently, access to data is restricted in scope to the current parameter. The implication of such a restriction is that the parameter value cannot be compared to values outside of it—comparisons to other parameters or mission parameters are not allowed. Because this assignment is done at the lowest (i.e. parameter) level, caution should be used in defining rules. This set of rules will supersede all other rules which may be imposed at the phase and mission levels.

THE CHECK FUNCTION

In manipulating vast amounts of data, it is often convenient if keywords can be assigned to variables which imply certain dependencies. Not only does it make the data more readable, it often reduces the work required in making modifications to it. For example, if the final speed of a phase is the same as its initial speed, it is much more readable, and convenient, to assign the parameter the keyword "SAME" than it would be to assign it the numerical value of the initial speed. Moreover, if the initial speed is altered, the keyword "SAME" will still be valid whereas the numerical value will no longer be.

The virtual function check is used to process such keywords. It calls the function checker(arg1, arg2, arg3) whose arguments dictate how the keywords are processed. The first argument must be the name of the parameter from the current phase whose value will be substituted for the keyword "SAME". The second argument is the name of the parameter from the previous phase whose value will be substituted for the keyword "LAST". The last parameter is a list of words that will be considered acceptable input by the parameter.
To illustrate the function's usage, suppose that the parameter FINAL_SPEED is being defined. In this case the keyword "SAME" should refer to the value of the parameter "INITIAL_SPEED" of the current phase. The final speed of the phase depends on conditions of the current phase, not those of the previous one. Therefore the keyword "LAST" should have no meaning. Finally, suppose that a maximum speed is defined. The keyword "MAX" should be allowed to reflect this upper limit. Thus a list containing this keyword should be used as the third argument. The process of redefining the virtual function is as follows:

1. Define the list of acceptable keywords:
   ```c
   char list [] [] = { "MAX", NULL };
   ```

2. Redefine the virtual function:
   ```c
   check () { checker ("INITIAL_SPEED",NULL,list) };
   ```

Note that the first two arguments have routines associated with them. The third argument, however, only makes the entries in the list acceptable—it does not describe how these words will be processed. In the example given, the `set_value_to` function of the parameter must be redefined to check for the keyword "MAX". If the value matches the keyword, the maximum value should then be attributed to the parameter.

Example: redefining the `set_value_to` function:

```c
parameter_name::set_value_to (char *value)
{
   ...
   ...
   if (! (strcmp ("MAX", value)))
      fvalue = 1000.00;
   ...
   ...
}
```
In summary, the parameter `FINAL_SPEED`, in this case, would accept all numeric entries, plus two keywords: "SAME" and "MAX". It should be noted that in its default definition check function provides only for numeric entries. Any non-numeric entries are treated as invalid entries.

**Deleting**

To delete a parameter which has previously been defined, the following procedure must be followed:

1. Delete its class definition.
2. Delete the conditional statement in the function `get_proper_params`.
3. Eliminate its name and default values in the phase defaults file.

**Phases**

**Creation**

Figures 12(a) and 12(b) give a typical class definition for a phase. A separate class must be defined for each type of phase. Note that each phase class inherits the class phases which contains all the functions associated with the object. New phases can be easily defined by adhering to the following procedure:

1. Copy the code given in figure 12(a).
2. Change the name of the class to reflect the new phase. Ensure that the name change is reflected in the destructor of the class.
3. Assign the class a new reference name. This is done by changing the name inside quotes in the `set_phase_title` function call. The name given to the phase must be a single word and is the name by which the phase must be referred to exactly when referring to it by name from any other part of the code. Although the name is restricted to being a single
word, underscores may be used in its definition. These underscores will be treated as blank spaces when the title is displayed on the screen.

4. Add a new conditional statement to the function Missions::get_proper_phase. Figure 12(b) illustrates the format of the conditional statement. Note that the name of the new phase must be placed inside the quotes found in the conditional test and that the newly defined phase class must be reflected in the call to the new data type.

5. Modify the phase.defaults.miss file to ensure that the newly created phase is reflected.

Modification

As is the case with parameters, defining each type of phase as a separate class allows for the definition of set rules unique to each one. As explained earlier, this is accomplished via use of virtual functions.

Like the parameters class, the phases class provides two virtual functions. They are the calculate and geo_segment functions.

THE CALCULATE FUNCTION

The calculate function is similar to the check function described earlier in that it is used to impose user-defined constraints on parameter data. A significant difference, however, is
class phase_name : public Phases // assign name
{
public:
    ~phase_name () { delete_phase_variables (); } // reflect phase name

    phase_name (char *param_titles [], char *input) // reflect phase name
    {
        set_phase_title_to ("NAME"); // give name of phase

        load_param_values (param_titles, input)

        next = NULL;
    }

    void calculate (); // optional: Include to redefine the
        // function. See following sections
}

Figure 12(a). Basic Phase Definition

Phases *Missions::get_proper_phase (char* title, char* data [])
{
    .
    .
    .
    else if (! (strcmp (title, "NAME"))) // add new conditional statement
    {
        phase = new phase_name (value); // give object type
        return phase;
    }
    .
    .
    .
}

Figure 12(b). Required Modification for Phase Definition
that this function is performed at the phase level. Comparisons between parameters belonging to the same or the previous phase can be made. Simple to complicated parameter interdependencies can be quickly constructed. These parameter interdependencies are unique to the phase in which they are defined.

To simplify the creation of such dependencies, four private functions are currently provided to return character or float information from a specified parameter. They are as follow:

**prev_number (name):** This function returns the numeric value of the specified parameter from the previous phase. To return the value of the parameter FINAL_SPEED in the previous phase, the following function call is used:

```plaintext
prev_number ("FINAL_SPEED");
```

**number (name):** This function returns the numeric value of the specified parameter from the current phase. To return the value of the parameter FINAL_SPEED in the current phase, the following function call is used:

```plaintext
number ("FINAL_SPEED");
```

**word (name):** This function returns the character value of the specified parameter from the current phase. To return the value of the parameter FINAL_SPEED in the current phase, the following function call is used:

```plaintext
word ("FINAL_SPEED");
```

**set (name, value):** This function is overloaded to handle an argument of type float or of type character. It sets the value of the specified parameter to the specified value. To set the value of the parameter FINAL_SPEED to the keyword "SAME" the following function call is made:

```plaintext
set("FINAL_SPEED", "SAME");
```
or, to set the value to 10.0, use:

    set ("FINAL_SPEED", 10.0);

As an example of redefining the calculate function, the parameter "DISTANCE"—the
distance traveled during the current phase—can be defined simply as the following:

    calculate () {
        float time = number ("TIME_ELAPSED");
        float final_speed = number ("FINAL_SPEED");
        float init_speed = number ("INITIAL_SPEED");
        float new_value = int_speed + ((final_speed - init_speed) / 2.0) * time ** 2.0;
        set ("DISTANCE", new_value);
    }

Other, more complicated, interdependencies can similarly be created.

**THE GEO_SEGMENT FUNCTION**

The *geo_segment* function is used by the class *Phase_Diagram_Window* to represent the
mission data in graphical form (i.e. the phase diagram). The function constructs the
graphical representation of its corresponding phase. The default function definition creates
a straight line proportional to the distance covered by the phase. This function may be
redefined to create more sophisticated graphical representations. In redefining the function,
caution should be used to ensure that the newly defined representations correctly reflect
scaling.
Miscellaneous

In addition to imposing rules on the parameter and phase levels, rules may also be imposed on the mission level. In general, rules at this level can affect data throughout the mission. Such rules, for example, may include routines to alter various parameters whose values are contingent on the values of some specified mission parameter. Regardless of how simple or sophisticated the testing routines may be, it is suggested that they be defined within functions which are made part of the Missions class. These functions should then be called from the appropriate location within the refresh function.
10.0 IMPLEMENTATION AND EXAMPLES OF RESULTS

Overview

The design and creation of the Mission Profile Input System (MPIS) is now complete—the design and user requirements outlined in the “Requirements” section have been implemented. The result has been the stand-alone version of the MPIS. Testing of the system has demonstrated that the objectives initially set forth have been satisfied. The system offers a friendly, interactive method by which a mission profile can be created, manipulated, and modified. The structured and forthright approach to the input of data offered by the system is of great benefit to the novice user. Equally important, however, the system also offers various methods by which the experienced user can circumvent the rigidity of such an approach to accelerate the process of creating and manipulating a mission profile.

The resulting system has also proven to contain good extendibility features. To test the customization traits of the system, the stand-alone version of the MPIS has been customized to be compatible with the aircraft CAD system, ACSYNT (see the section “Integration With ACSYNT”). The design structure of the program has proven sufficiently robust and flexible such that no major modifications or extensions to the code were required in the adaptation of the system.
The Stand-Alone Version Of The MPIS

The stand-alone version of the MPIS is intended to demonstrate the functionality of a general, customizable system which can be used to create and manipulate mission profiles. Since existing aircraft CAD systems have a variety of phase and parameter requirements, no attempt was made to include a comprehensive definition of phases and parameters with the stand-alone version. Rather, the phases and parameters defined within this version of the MPIS are intended to demonstrate the flexibility of the system—serving as templates for the definition of new ones.

The following parameters and phases have been implemented to prove the design of the system. However, it should be emphasized that they are merely intended to prove the concept and will thus most likely have to be modified or replaced upon adaptation of the MPIS to a CAD system.

Parameters defined:

| Class Name: | initial_speed |
| Parameter Name: | INIT_SPD |
| Notes: | The following keywords are defined: LAST -Sets the float value of the parameter equal to the float value of the parameter FINAL_SPD from the previous phase. |

| Class Name: | final_speed |
| Parameter Name: | FINAL_SPD |
| Notes: | The following keywords are defined: SAME -Sets the float value of the parameter equal to the float value of the parameter INIT_SPD from the same phase. |
Class Name: initial_altitude
Parameter Name: INIT_ALT
Notes: The following keywords are defined:
LAST - Sets the float value of the parameter equal to the float value of the parameter FINAL_ALT from the previous phase.

Class Name: final_altitude
Parameter Name: FINAL_ALT
Notes: The following keywords are defined:
SAME - Sets the float value of the parameter equal to the float value of the parameter INIT_ALT from the same phase.

Class Name: time
Parameter Name: TIME
Notes: None

Class Name: distance
Parameter Name: DISTANCE
Notes: This parameter is calculated automatically. It is defined as:
\[
INIT_{SPD} + \frac{1}{2}\left(\frac{FINAL_{SPD} - INIT_{SPD}}{2}\right)TIME^2
\]

Class Name: total_distance
Parameter Name: TOTAL_DIST
Notes: This parameter is calculated automatically. It is defined as:
TOTAL_DIST(from previous phase) + DISTANCE

Class Name: direction
Parameter Name: DIRECTION
Notes: The following character entries are permitted:
F - indicates forward travel
B - indicates backward travel
Used by the Phase_Diagram_Window to draw distance in the negative direction.
Class Name: drop_bomb
Parameter Name: DROP_BOMB
Notes: The following character entries are permitted:
Y  - bomb is dropped
N  - bomb is not dropped

Phases Defined:

Class Name: acceleration
Phase Name: ACCEL
Notes: The DIST and TOTAL_DIST parameters are calculated from their dependencies.

Class Name: cruise
Phase Name: CRUISE
Notes: The DIST and TOTAL_DIST parameters are calculated from their dependencies.

Class Name: loiter
Phase Name: LOITER
Notes: The DIST and TOTAL_DIST parameters are calculated from their dependencies.

Class Name: climb
Phase Name: CLIMB
Notes: The DIST and TOTAL_DIST parameters are calculated from their dependencies.
Methods For Handling Data

As explained in the section “The Mission Profile Input Module,” the ability of the system to accept diverse types of input results in a fair amount of complexity in how the data must be handled. The input must be processed through numerous functions to ensure its validity. To fully appreciate how the MPIS handles data, it is necessary to understand two major procedures: how a mission is created, and how the modification of a parameter value is resolved. The sequence of function calls for these two procedures fully illustrate the methods by which objects are created, input values validated, and dependent parameters updated. By comprehending these particular methods, other procedures, such as phase insertion and phase deletion, can be easily understood.

CREATION OF A MISSION

Figure 13 describes the control path of the program when a mission is first created. Data to create the proper mission objects is read from a file specified by the user by the function Missions::create_mission. The values are assigned to the newly created corresponding objects. To ensure that the data is valid and that dependencies between the data are properly maintained, comprehensive tests are performed on the input data after it has been assigned.

The first information read from the file is the mission name, the number of phases and the names of the parameters contained by the mission. The last two values are used to properly read the rest of the data. After reading this information the linked list of phases is created. This is done by making successive calls to the Missions::get_proper_phase function. This function takes the name of the phase and the list of the values for its parameters as
Missions::create_mission (filename)
{
  temp = # of phases
  param_names [] = list of parameters
  for (# of phases)
  {
    Phases::get_proper_phase (word (name), list)
    add phase to the phase linked list
  }
  for (# of mission parameters)
  {
    create mission parameter
    assign values to mission parameter
    add mission parameter to Mission_Parameters linked list
  }
}

phase constructor
{
  assign phase attributes
  Phases::load_param_values (param_names, values)
}

Phases::load_param_values (param_names, values)
{
  for (# of parameters)
  {
    Phases::get_proper_param (name, value)
    add parameter to parameter linked list
  }
}

parameter constructor
{
  assign parameter attributes
  Parameters::set_value_to (word (value))
}

Parameters::set_value_to (value)
{
  assign value
  Parameters::check ()
}

Parameters::check ()
{
  if (value of param is keyword)
    launch appropriate routine
  else if (entry is acceptable)
    return
  else if (format (value of param))
    display error message
  // If it is a keyword, launch proper routine
  // launch appropriate routine
  // else if the entry is defined to be acceptable
  // if yes, leave value in tack
  // else check what the format of the value is
  // If it is a "word" display error message
  // else end check. Entry is a numeric value
}

Figure 13. Creation of Mission (Pseudo-Code)
arguments. By comparing the name it receives to the names of the available phases, the function determines which phase object to create. The constructor of the phase created, in turn, creates the parameter link list. This is done by calling the Phases::load_param_values function which takes a the list of available parameters and the list of their corresponding values as arguments. This function traverses the list of parameters and matches each parameter name with it’s respective value. The Phases::load_param_values function makes successive calls the Phases::get_proper_param function, sending one pair of values as arguments for each call. The Phases::get_proper_param function determines the proper parameter object to create by comparing the names of the available parameters to the parameter name which it receives. The constructor of the parameter created requires one argument—its value. The constructor assigns the value it receives to itself. Assignment of the parameter value is done by calling the Parameters::set_value_to function. The Parameters::set_value_to function is a virtual function and thus may be redefined to perform differently. However, in its base definition it assigns the value to the proper parameter variable and then calls the Parameters::check function. Like the Parameters::set_value_to function, the Parameters::check function is virtual. Its definition can be altered to adhere to different rules. However, it’s main purpose should always be to check the validity of the input. Conditional tests should be used to check whether the entry is valid. If not, the proper actions should be taken.

Once the linked list of phases is complete, the linked list of mission parameters is constructed. This is done by creating objects of type Mission_Parameters, assigning them the values read from the file, and linking them in a linked list. Upon completing this linked list, the creation of the mission is complete.
MODIFICATION OF A PARAMETER

Figure 14 illustrates the control path of the Mission Profile Input System whenever the value of a parameter is modified. The modification process is complicated by the need to verify the validity of the input and the need to determine whether any special processes (e.g. process keyword entries) need to be executed.

The value of a parameter is modified by calling the Missions::update_element function. This function takes three arguments. The first two arguments determine the parameter to be updated by specifying the exact location of the parameter within the various linked lists. The third argument is the new value to which the parameter is to be set. Setting the parameter value is performed by calling the Parameters::set_value_to function. This function assigns the value and then calls the Parameters::check function, which verifies the validity of the input and executes any special processes.

Once the value of the parameter is updated, all the data within the mission must be updated to reflect the effects on dependent parameters. This is done by executing the Missions::refresh function. The Missions::refresh function updates the parameters with keyword entries to ensure that their values are properly reflected. Moreover, the function also ensures that all the dependencies between parameters are also updated.

To ensure that the mission is updated correctly, the Missions::refresh function executes a thorough scan of all the mission data. Each phase of the mission is scanned for keywords. The entire linked list of parameters is updated and the set rules for the phase are executed.
process for making modification to a parameter
{
    Missions::update_element (phase_num, param_num, new_value)  // assign the new_value to the a parameter
    refresh ()  // update all the data to reflect the change
}

Missions::update_element (phase_num, param_num, new_value)
{
    phase_pt = Missions::get_phase_pointer (phase_num)  // get pointer to the correct phase
    param_pt = phase_pt->Phases::get_param_pointer (param_num)  // use phase pointer to get parameter pointer
    param_pt->Parameters::set_value_to (word (value))  // set parameter value; ensure proper format
}

Parameters::set_value_to (value)
{
    assign value
    Parameters::check ()  // check input for validity
}

Parameters::check ()
{
    if (value of param is keyword)  // If it is a keyword, launch proper routine
        launch appropriate routine
    else if (entry is acceptable)  // else if the entry is defined to be acceptable
        return
    else if (format (value of param))  // else check what the format of the value is
        display error message
        // If it is a "word" display error message
        // else end check. Entry is a numeric value
}

Missions::refresh ()
{
    for (# of phases)  // repeat for the number of phases
    {
        for (# of parameters)  // repeat for the number of parameters
        {
            if (parameter value is a keyword)  // if keyword do the following
            {
                for (# of parameters)  // rescan the entire current phase
                {
                    if (parameter value is a keyword)  // act only on parameters which have a keyword
                    {
                        curr_param->Parameters::set_value_to (key_word)  // set current parameter value—this function will again
                        // call the check function
                    }
                }
            }
        }
    }
    current_phase->Phases::calculate ()  // update the entire current phase
}

Phases::calculate ()  // updates entire phase to reflect parameter dependencies
{
    perform user-defined calculations on phase parameters
}

Figure 14. Parameter Modification (Pseudo-Code)
for each keyword encountered. Thus, for a phase containing five keywords, five iterative updates of the phase are performed. In this manner, the data is always guaranteed to reflect the proper values regardless of the order in which the parameter modifications were made.

For the interested reader, the exact protocol for the functions (denoted by italicized names) may be found in Appendix B.

**The User-Interface**

Figures 15 through 29 illustrate the interface for the stand-alone version of the MPIS. Figure 15 shows the system with the phase diagram activate, and Figure 16 shows the system with the Mission Parameters menu activate. The remaining figures illustrate the various interactive pop-up menus offered by the system. Explanations for the purpose of each of the menus, along with instructions on how to use them, may be found in Appendix A.
<table>
<thead>
<tr>
<th>TYPE</th>
<th>INIT SPD</th>
<th>FINAL SPD</th>
<th>INIT ALT</th>
<th>FINAL ALT</th>
<th>TIME</th>
<th>DISTANCE</th>
<th>TOTAL RST</th>
<th>DIRECTION</th>
<th>DROP الوقود</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMZ</td>
<td>0.9</td>
<td>355.8</td>
<td>9.2</td>
<td>646.8</td>
<td>11.6</td>
<td>16.4</td>
<td>F</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>ACCEL</td>
<td>LAST</td>
<td>996.8</td>
<td>LAST</td>
<td>666.8</td>
<td>41.6</td>
<td>58.3</td>
<td>F</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>LOITER</td>
<td>LAST</td>
<td>198.8</td>
<td>LAST</td>
<td>468.8</td>
<td>37.5</td>
<td>38.0</td>
<td>F</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>CRUISE</td>
<td>LAST</td>
<td>130.8</td>
<td>LAST</td>
<td>986.8</td>
<td>27.5</td>
<td>122.3</td>
<td>F</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>ACCEL</td>
<td>LAST</td>
<td>996.8</td>
<td>LAST</td>
<td>666.8</td>
<td>21.6</td>
<td>38.0</td>
<td>F</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>CRUISE</td>
<td>LAST</td>
<td>996.8</td>
<td>LAST</td>
<td>666.8</td>
<td>21.6</td>
<td>38.0</td>
<td>F</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>ACCEL</td>
<td>LAST</td>
<td>996.8</td>
<td>LAST</td>
<td>666.8</td>
<td>21.6</td>
<td>38.0</td>
<td>F</td>
<td>N</td>
<td></td>
</tr>
</tbody>
</table>

Figure 15. The MPIS With the Phase Diagram Window Activated
Figure 16. The MPIS With the Mission Parameters Menu Activated
Figure 17. The Phase Menu
Figure 18. The Defaults Menu
Figure 19. The Phase Options Menu
Figure 20. The Quick Input Menu
Figure 21. The Parameter Menu
Figure 22. The Move Parameter Menu
Figure 23. The Select Other Variables Menu
Figure 24. The Other Variables Menu
Figure 25. The Row/Column Menu
Figure 26. Menu Listing Available Files
Figure 27. The Message Menu
Figure 28. The Confirm Menu
Figure 29. The Filename Menu
11.0 INTEGRATION WITH ACSYNT

Overview

As proof of its flexible features, the Mission Profile Input System was integrated into the interactive CAD version of ACSYNT. The amount of code modification required to make the stand-alone version of the system compatible with ACSYNT was carefully noted. Upon completion, ACSYNT integration proved the robustness and flexibility of the data handling routines provided by the MPIS. All of ACSYNT’s data manipulation requirements were satisfied without the need to modify or append the utilities offered by the system. However, certain characteristics of ACSYNT did require unexpected modifications to other aspects of the stand-alone MPIS. Most noticeable was the separation of the analysis and interactive portions of ACSYNT—data entered by the user cannot immediately be processed by the ACSYNT analysis modules. Since the affects of modifications made to the mission profile are not immediately available, the graph generated by the ACSYNT-integrated version of the MPIS is not drawn to scale.

Integration

Integration of the module to ACSYNT required approximately three man-days. Most of the effort was dedicated to writing translators which convert data from ACSYNT to MPIS
format. The following sections describe the modifications made to the stand-alone version of the Mission Profile Input System to make it compatible with ACSYNT.

**Transferring Data**

Data transferred between the MPIS and ACSYNT by translating data from the *TRIN.MOD* file of ACSYNT into the MPIS format (see the section “Trajectory Data Files”). The *TRIN.MOD* file is used by the ACSYNT spreadsheet utility for temporary storage of the modifications made by the user (changes are not considered permanent until they are stored as “ACS input” [ACSY93]). The decision to transfer data using the TRIN.MOD file was made in an effort to adapt the MPIS system without disabling the ACSYNT spreadsheet. By writing to the same file, both systems have access to the same data and can therefore both be utilized.

When the MPIS is initiated, it translates the information from the *TRIN.MOD* file into MPIS format—stored in a scratch file called *trin.mod*—and used as the default mission by the system. When the user exits from the MPIS, the system translates the mission information back into ACSYNT format and overwrites the *TRIN.MOD* file.

**Saving Mission Information**

To save a mission in MPIS format the user must explicitly save it using the “SAVE MISSION AS...” option under the Options Menu (see Appendix B). Mission information can also be saved by saving it in ACSYNT format. This is done by using the “SAVE ACS INPUT” option under the FILES menu (see [ACSY93]). This option, however, saves all information pertaining to the current aircraft configuration. Moreover, it makes any modifications made to the current aircraft permanent. For this reason, it is sometimes more advisable to save the mission information under MPIS format.
**Modifications to the MPIS**

The number and types of parameters and phases the MPIS accepts were modified to reflect the requirements of ACSYNT. The modifications were made following the guidelines described in the section “System Customization”. Following is a list of the new data types (i.e. phase and parameter classes) defined. Explanations are provided on the rules implemented for each data type.

**Parameters defined:**

<table>
<thead>
<tr>
<th>Class Name:</th>
<th>Parameter Name:</th>
<th>Special Notes:</th>
</tr>
</thead>
<tbody>
<tr>
<td>start_mach</td>
<td>MSTART</td>
<td>The following keywords are defined:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PREV -Represents code -1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OPT -Represents code 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The parameter value is restricted to be less than 10.0.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class Name:</th>
<th>Parameter Name:</th>
<th>Special Notes:</th>
</tr>
</thead>
<tbody>
<tr>
<td>end_mach</td>
<td>MEND</td>
<td>The parameter value is restricted to be less than 10.0.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class Name:</th>
<th>Parameter Name:</th>
<th>Special Notes:</th>
</tr>
</thead>
<tbody>
<tr>
<td>start_alt</td>
<td>HSTART</td>
<td>The following keywords are defined:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PREV -Represents code -1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OPT -Represents code 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The parameter value is restricted to be less than 100,000.</td>
</tr>
</tbody>
</table>
Class Name: end_alt
Parameter Name: HEND
Special Notes: The following keywords are defined:
NEXT - Represents code -1
OPT - Represents code 0
NOCLIMB - Represents code -1

The parameter value is restricted to be less than 100,000.

Class Name: distance
Parameter Name: DIST
Special Notes: The following keywords are defined:
RANGE - Represents code -10

The parameter value is restricted to be less than 100,000.0.

Class Name: time
Parameter Name: TIME
Special Notes: The parameter value is restricted to be less than 10,000.0.

Class Name: turns
Parameter Name: NTURNS
Special Notes: The parameter value is restricted to be less than 1,000.0.

Class Name: vind
Parameter Name: VIND
Special Notes: The parameter value is restricted to be less than 10,000.

Class Name: fuel_factor
Parameter Name: WKFUEL
Special Notes: The parameter value is restricted to be less than 100.0.

Class Name: mparam
Parameter Name: M
Special Notes: The parameter is always set equal to the mission parameter MMPROP.
Class Name: power_setting
Parameter Name: IP
Special Notes: The following keywords are defined:
   ALTWAFT  - Represents code -3
   ALTWOAFT - Represents code -2
   TOFFWAFT - Represents code -1
   TOFFWOAFT - Represents code 0
   MAXAFT   - Represents code 1
   MAXTOFF  - Represents code 2
   MAXCONT  - Represents code 3
   DRAG     - Represents code 4
   IDLE     - Represents code 5

The parameter must be one of the above values. The default value is IDLE.

Class Name: range_indicator
Parameter Name: IX
Special Notes: The following keywords are defined:
   NEITHER - Represents code 0
   ADD      - Represents code 1
   SUB      - Represents code -1

Class Name: missile_indicator
Parameter Name: W
Special Notes: The following keywords are defined:
   DROP     - Represents code 1
   NODROP   - Represents code 0

The parameter must be one of the above values. The default value is NODROP.

Class Name: bomb_indicator
Parameter Name: B
Special Notes: The following keywords are defined:
   DROP     - Represents code 1
   NODROP   - Represents code 0

The parameter must be one of the above values. The default value is NODROP.
Class Name: ammo_indicator
Parameter Name: A
Special Notes: The following keywords are defined:
DROP - Represents code 1
NODROP - Represents code 0

The parameter must be one of the above values. The default value is NODROP.

Class Name: print_indicator
Parameter Name: P
Special Notes: The following keywords are defined:
PRINT - Represents code 1
NOPRINT - Represents code 0

The parameter must be one of the above values. The default value is NOPRINT.

Phases Defined:

Class Name: climb
Phase Name: CLIMB
Special Notes: The following parameters do not apply to the phase:
NTURNS
TIME
MEND
DIST

Class Name: acceleration
Phase Name: ACCEL
Special Notes: The following parameters do not apply to the phase:
VIND
NTURNS
DIST

Class Name: cruise
Phase Name: CRUISE
Special Notes: The following parameters do not apply to the phase:
IX
VIND
NTURNS
Class Name: loiter
Phase Name: LOITER
Special Notes: The following parameters do not apply to the phase:
IX
VIND
NTURNS

Class Name: combat
Phase Name: COMBAT
Special Notes: The following parameters do not apply to the phase:
IX
DIST

Class Name: descent
Phase Name: DESCENT
Special Notes: The following parameters do not apply to the phase:
IX
VIND
NTURNS
TIME
DIST

Class Name: hover
Phase Name: HOVER
Special Notes: The following parameters do not apply to the phase:
IX
VIND
NTURNS
DIST

 Modifications Made to the Mission_Parameters Class

In addition to the mission parameter information required by the stand-alone version of the MPIS, ACSYNT requires two additional pieces of information to correctly keep track of its mission parameters. It requires that the type and format of the variable be specified. Consequently, the Mission_Parameters class was modified to accommodate the additional information requirements. The corresponding functions to set and retrieve this information
were also implemented. ACSYNT makes no provisions for keeping track of the variable information which determines whether the variable is to be shown by the Other Variables Menu. Consequently, every time the data are transferred from ACSYNT, the value of the variables is reset to its default value of "NO" (i.e. the variables is not shown by default).

**Using The System From Within ACSYNT**

The MPIS can be initiated from within ACSYNT by using the "Trajectory Module" option under the "Trajectory" menu. The system will utilize a different window (PHIGS workstation) from the one in which ACSYNT appears. However, although both the MPIS and ACSYNT appear simultaneously on the screen, only the MPIS will accept user input. Control will not return to ACSYNT until the MPIS is exited.
12.0 CONCLUSION AND RECOMMENDATIONS

The stand-alone version of the Mission Profile Input System has satisfied the objectives initially set forth. The system offers a friendly, interactive method by which a mission profile can be created, manipulated, and modified. It offers a structured and forthright approach to the input of data and yet retains the flexibility necessary to enhance efficiency. The system has also proven to contain good extendibility features. Adaptation to the aircraft conceptual design system, ACSYN, was performed without the need to implement major modifications or extensions to the code.

Development of the MPIS has also shown that object-oriented design and programming lend themselves well to programs of its nature. As mentioned earlier, object-oriented programs attempt to mimic relationships found in the real-world. The relationship between parameters, phases, and missions, where one is contained within another, can easily be mimicked using the instancing of objects. The missions instance the phases, which, in turn, instance the parameters.

The current version of the Mission Profile Input System should be thought of as the first complete iteration of its design—it is the framework upon which future versions can be improve. As is the tendency of code development, better and more efficient algorithms for a better system were uncovered even as the first design was being implemented.

Suggested areas of improvement are in the definition of phases and parameters. Although the current method is highly flexible and straightforward, it could be made even less
tedious if one generic class was defined to represent any type of phase or parameter. Functions to define the properties of the particular type could be used to customize the individual phases and parameters. Problems arising from trying to store such information will need to be resolved.

Another area which can be improved is the dynamic definition of rules by the user for the various phases and parameters. Currently, such rules must be implemented at the code level. By giving the user the ability to modify these rules interactively, the functionality of the program will be greatly increased. Methods by which to determine invalid or inconsistent rules will have to be considered. Also provisions will have to be made for storing such information.

The power and usefulness of a program is determined by how effectively the end-user is able to maximize its potential. Ultimately, however, only the end-user can determine what is the most efficient interface layout for personal maximization of the potential of the program. Usually, this cannot be determined until a layout configuration has been implemented and its limitations are discovered through daily usage. For this reason, it would be desirable to give end-users the ability to fully customize the interface to suit their individual needs. The types of menus, their menu items, and when and how the menus appear would be completely dictated by the user. Such a system would no doubt require a major endeavor—well beyond the scope of a single thesis. However, such a system would provide much insight and take full advantage of the power of object-oriented programming.
13.0 REFERENCES


APPENDIX A: User Guide
OVERVIEW

Throughout the design process, the concerns of the end-user were consistently kept in mind. The interface was configured to give a complete representation of the trajectory mission in a straightforward, consistent manner. To prevent the module's display from getting cluttered with unnecessary options—and to avoid an unnecessarily deep menu structure—pop-up menus were used extensively. These menus can be launched by selecting the data directly or by selecting one of the five push buttons situated above the main window. These menus and their options were grouped in their most logical order and every effort was made in making them self-explanatory. The following sections describe in detail each menu and its options. The menus are divided into two major types: menus launched by selecting mission data, and menus launched by selecting one of the five push buttons. Illustrations of the various menus are given in the section “Implementation & Examples of Results.”

SELECTING DATA

The “Phase” Menu

Selecting any parameter value on the screen with the mouse highlights the entire phase to which the parameter belongs and launches the Phase Menu. This menu is the primary method by which the values of the parameters can be modified. The heading of the menu gives the number and name of the phase selected. The names and values of the parameters belonging to the phase are listed in order. Parameters which are not applicable to the phase—denoted by the value "N/A"—are not listed. To minimize the number of steps
required to modify a parameter, the value of the parameter selected is automatically highlighted. Typing in a new value and hitting ENTER on the keyboard will replace the old value. Note that ENTER must be pressed in order to register the new value. Tabbing out of the input area or selecting the DONE menu option will NOT register the change. To make additional changes, or to transfer control to a different menu item, the tab key may be used. Control may also be transferred by directly selecting the desired menu item.

The Phase Menu will remain on the screen until the DONE option is selected or the mouse button is pressed outside the menu area. If another mission data item is selected the menu corresponding to the new item selected will be launched. Otherwise, the menu will simply be removed from the screen.

The rightmost side of the Phase Menu displays a column of push buttons. These buttons launch the Defaults Menu for the parameter selected.

The Phase Menu may also be launched by selecting any leg in the phase diagram. By default, the first parameter of the phase will be highlighted when it is launched by this method.

THE "DEFAUlTS" MENU

The default value of a parameter is the value originally assigned to it. If the mission was read from a file, the default value is the parameter value read. After modifications have been made to a mission, it is sometimes desirable to reassign the original value to a parameter. The Defaults Menu is used for this purpose. As mentioned earlier, the Defaults Menu is launched by selecting one of the push buttons in the Phase Menu. Unlike most other pop-up menus, the Defaults Menu requires that a menu item be selected before it be removed from the screen. It consists of four push buttons and an input area. The input area
displays the current default value of the parameter. This can be changed by entering a new value and pressing the ENTER key. If ENTER is not pressed after changing the value, the change will NOT be registered. The four push buttons dictate the action to be performed before exiting the menu. The menu will be removed as soon as an option has been selected.

**APPLY Menu Option:** This option takes the current default value and assigns it to the parameter. If the default value has been altered (i.e. a new value has been registered in the input area), the new value will be applied, but the default value of the parameter will remain unchanged.

**REDEFINE Menu Option:** This menu item defines the default value of a parameter to be the value registered in the input area (CAUTION: The ENTER key must be pressed to register a new value). Note, however, that the new default value is not applied to the parameter.

**REDEFINE & APPLY Menu Item:** This menu item combines the actions of the two previous ones. It defines the default value to be the value registered in the input area and applies it to the parameter.

**CANCEL:** This menu item exits the menu without performing any tasks.

**The “Phase Options” Menu**

Selecting a phase title will highlight the entire phase and launch the Phase Options Menu. This menu offers editing operations which are performed at the phase level. The menu options are as follow:
Delete current phase: Selecting this option will delete the currently highlighted phase. The menu will remain on the screen with the next available phase becoming active. In the interest of efficiency, no confirmation is required in issuing this command. Therefore caution should be exercised in its usage.

Add new phase before current one: This option will activate a menu by which phases can be added. The menu lists the phases available for addition. Selecting a phase from this menu will insert the phase BEFORE the currently active one.

Add new phase after current one: Same as the previous option, except that the phase will be inserted AFTER the currently active one.

Set phase to default values: This option sets all the parameters of the active phase to their default values (see the “Defaults Menu” section for the definition of a default value). To safeguard against accidental changes, command confirmation is required in issuing this command.

Quick input: This option launches the Quick Input Menu. See the “Quick Input Menu” section.

Done: This option quits the Phase Options Menu. Control may also be released by pressing the mouse button while the cursor is anywhere outside the menu.
THE “QUICK INPUT” MENU

Although the Phase Menu offers a convenient method by which to modify the parameters, it limits changes to one parameter at a time. This restriction can prove quite burdensome to more experienced users. The Quick Input Menu circumvents this dilemma. It allows users to modify the parameters for an entire phase simultaneously, albeit in a less friendly fashion.

The Quick Input Menu contains two input areas which the user can use to enter the list of modifications. Although entries can be made in either input area, it is recommended that the second area be used only if room for entries is exhausted in the first one. The two push buttons below the input areas are used to process the list. The APPLY button applies the list of modifications to the currently active phase. The DONE button exits the menu without performing the modifications.

Each entry in the entry list must be separated by at least one blank character. The list of modifications corresponds directly to the parameters of the active phase (i.e. the first entry affects the first parameter, the second entry the second parameter, etc.). The list need not have the same number of entries as there are number of parameters. Once the list of modifications has been exhausted the remaining parameters will remain unchanged. To prevent a parameter, which lies inside the range of the modification list, from being modified, the special symbol "*" may be used. This symbol skips over the parameter without altering it. Figure 30 gives an example of using the Quick Input Menu.
Figure 30. Sample Usage of the Quick Input Menu
The “Parameter” Menu

This menu offers an additional method by which parameters can be modified. It is identical to the Phase Menu except that all the values for a single parameter are displayed instead of the values of all the parameters for a single phase. Due to the size limitation of the screen, the menu is disabled once the number of phases in the mission exceeds twenty-two. Unlike the Phase Menu, all values are displayed—including those designated as not applicable (i.e. "N/A"). The Parameter Menu is launched in a unique fashion. To launch it, the middle mouse button must be used to select a parameter value while a pop-up menu is active. If no pop-up menu is active and the procedure is followed, the Phase Menu will be launched.

The “Move Parameter” Menu

Selecting the title of a parameter highlights the parameter and launches the Move Parameter Menu. This menu allows the order of the parameters to be changed. To move the parameter, select the number box until it displays the desired new position. Select the APPLY push button to move the currently active parameter to the new position. The DONE push button exits the menu without applying changes.

THE PUSH BUTTONS

Quit

This button exits the Mission Profile Input System. To avoid the accidental termination of the system, confirmation of the command is required.
Mission Parameters Toggle

This button is used to toggle the Mission_Parameters Menu on and off. The Mission_Parameters Menu displays some of the more important mission parameters. These and other mission parameters may also be displayed in the Other Variables Menu. This menu, unlike the others, is not a pop-up menu. Control of the program can be toggled between this menu and other windows on the screen. Thus, this menu can be displayed and used without the need to sacrifice control from other parts of the program.

Phase Diagram Toggle

This button toggles the phase diagram of the mission on and off.

Options

This button displays all the general options available. They include options to save and retrieve files, modify the screen layout, and display and manipulate mission parameters. A brief description on each option is given below.

Save Mission: This option saves the mission under its current name.

Save Mission As: This option saves the mission under a name specified by the user. Names should be void of blank spaces and/or punctuation marks.

Retrieve Mission: This option displays a list of the currently available missions. Clicking on the desired mission will retrieve it. Loading a new mission will redefine the current one. For this reason, confirmation of the command is required.

Select Viewable Variables: Launches the Select Other Variables Menu.
**View Other Variable:** Launches the Other Variables Menu.

**Row/Column Display:** Launches the Row/Column Menu

**Number Display:** Launches the Number Display Menu.

---

**THE “SELECT OTHER VARIABLES” MENU**

This menu is used to designate which mission parameters will be displayed by the Other Variables Menu. Variables can be toggled to be viewable or non-viewable by selecting their checkboxes. A lowered checkbox indicates that the variable is viewable. The reason for this option is that it allows the user to limit the variables displayed in the Other Variables Menu to those deemed most important. Variables that are less important, or which are not frequently modified, can be hidden from view.

---

**THE “OTHER VARIABLES” MENU**

This menu allows the modification of mission parameters which have been designated as viewable by the Select Other Variables Menu. It is important to note that no type or data checking is performed on any of the input values. Thus, it is up to the user to ensure that the data entered is indeed valid.
THE "ROW/COLUMN" MENU

This menu allows the user to customize the layout of the display. The left and right number boxes reflect the current column and row spacing, respectively. To change the spacing, select the number boxes until they reflect the desired new dimensions and select the APPLY push button. The DONE push button exits the menu without applying any changes.

THE "NUMBER DISPLAY" MENU

Like the previous menu, the Number Display Menu allows the user to customize the display layout. The number box displays the precision to which numeric values are displayed. The precision ranges from zero (0) to nine (9). To change the current precision click on the number box until it reflects the desired new numerical precision. The "Toggle Alignment" option toggles the alignment of the data between left and center alignment. The DONE option exits the menu.

100 % (Fit-to-Screen)

This push button scales the display to fit the window. It is primarily used to adjust the display after new phases have been added to the mission or frequent zooms have been performed.
WINDOW BASICS

The window layout of the Mission Profile Input System has been modeled to mimic a Motif-based window environment. As in the Motif interface, the window can be resized by dragging its border and it can be moved by dragging its title bar. A toggle button on the upper-right corner toggles the size of the window between its current and maximum dimensions.

In addition to the Motif-like functions, the windows provide zooming capabilities. These buttons are situated on the lower right hand corner of the windows. The left button makes the display larger, whereas the right button makes it smaller.
APPENDIX B: Detailed Class Description

Following is a detailed description of all the classes used by the Mission Profile Input System. Note that only classes created by the author are described in detail. Classes which are used, but not created by the author, are listed under "Other Classes" and are only included for the sake of completeness.
THE PARAMETERS CLASS

Class Name: Parameters

Location: parameters.h

Inheritance: None

Description:
This class provides the functions necessary to manipulate data at the parameter level. Every parameter which is defined for the mission must inherit this class. This class is instanced in the class Phases.

Private Variables:
- char * cvalue: stores the character value of the parameter
- float fvalue: stores the float value of the parameter
- char * cdefault: stores the character default value of the parameter
- float fdefault: stores the float default value of the parameter

Protected Variables:
- Parameters * next: points to the next parameter in the linked list

Public Variables:
- None

Private Functions:
- Parameters * prev: returns pointer to a specified parameter in the previous phase
- Parameters * cur: returns pointer to a specified parameter in the current phase
- Parameters * nxt: returns pointer to a specified parameter in the next phase
Protected Functions:

void delete_parameter_variables
frees up memory taken up by the class

Public Functions:

void set_param_title_to
sets the name of the parameter to a specified title

cchar * get_value
returns character value of parameter

float get_value
returns float value of parameter

char * get_default_value
returns character default value of parameter

float get_default_value
returns float default value of parameter

void set_default_values
sets the default value of the parameter

char * get_param_title
returns the name of the parameter

Parameters * get_next
returns a pointer to the next parameter in the linked list

void put_next
sets a pointer to the next parameter in the linked list

void checker
checks and process keywords

Virtual Functions:

void set_value_to
assigns the value to the parameter

void check
specifies how the keywords are treated
FUNCTIONS OF THE PARAMETERS CLASS

Location: parameters.C

Function: private
Parameters * prev (char *name)
returns a pointer to parameter name in the previous phase

Argument Description:
char * name name of a parameter in the previous phase

Function: private
Parameters * cur (char *name)
returns a pointer to parameter name in the current phase

Argument Description:
char * name name of a parameter in the current phase

Function: private
Parameters * nxt (char *name)
returns a pointer to parameter name in the next phase

Argument Description:
char * name name of a parameter in the next phase

Function: protected
void delete_parameter_variables ()
used to free up memory when class is deleted

Argument Description:
None
Function: public

void set_param_title_to (char *name)

sets the title of the parameter to name

Argument Description:
char * name name of a parameter

Function: public

char * get_value (char *type)

returns the character value of the parameter

Argument Description:
char * type type = CHAR = "1"

Function: public

float get_value (float type)

returns the float value of the parameter

Argument Description:
float type type = REAL = 1.0

Function: public

char * get_default_value (char *type)

returns the default character value of the parameter

Argument Description:
char * type type = CHAR = "1"

Function: public

float get_default_value (float type)

returns the default float value of the parameter

Argument Description:
float type type = REAL = 1.0
Function: public

    void set_default_values (char *entry)
    
    sets the default value of the parameter to entry

Argument Description:

    char * entry  
    the default value of the parameter

Function: public

    char * get_param_title ()
    
    returns the name of the parameter

Argument Description:

    None

Function: public

    Parameters * get_next ()
    
    returns a pointer to the next parameter in the linked list

Argument Description:

    None

Function: public

    void put_next (Parameters *next)
    
    sets the parameter next to be the next parameter in the parameter linked list

Argument Description:

    Parameters * next  
    pointer to the next parameter in the linked list
Function: public

void checker (char * name1, char * name2, char ** list)

dictates how keywords in the program will be processed

Argument Description:

char * name1
name of a parameter in the current phase whose value
will be substituted for the keyword "SAME"

char * name2
name of a parameter in the previous phase whose
value will be substituted for the keyword "LAST"

char ** list
list of entries which will be considered acceptable by
the parameter

Function: virtual, overloaded

void set_value_to (char *value)
void set_value_to (float value)

sets the value of the parameter to value—can be redefined to define new
requirements for parameter value assignment

Argument Description:

char * value
the value of the parameter

float value
the value of the parameter

Function: virtual

void check ()

used to specify how the keywords in the program are to be treated—can be
redefined to specify new keywords

Argument Description:

None
THE PHASES CLASS

Class Name:
    Phases

Location:
    phases.h

Description:
    This class provides the necessary function to manipulate phases and their parameter data. Every phase which is defined for the mission must inherit this class. A linked list of parameters is created by each phase through instancing of the Parameters Class.

Inheritance:
    None

Private Variables:
    char * phase_title            name of the phase

Protected Variables:
    Phases * next                 pointer to the next phase in the linked list
    Parameters * first_param      pointer to the first parameter of the phase

Public Variables:
    None

Private Functions:
    None
Protected Functions:

float prev_number returns float value of a specified parameter in the previous phase
float number returns float value of a specified parameter in the current phase
char * word returns character value of a specified parameter in the current phase
void set sets a specified parameter in the current phase to a specified value
void delete_phase_variables frees up memory taken up by the phase

Public Functions:

Parameters * get_proper_param returns a pointer to a new parameter of a specified type
void load_param_values assigns the parameter values to the phase
void set_phase_title_to assigns the name of the phase
char * get_phase_title returns the name of the phase
Parameters * get_param_pointer returns a pointer to the specified parameter
float get_float_param_value returns the float value of the specified parameter

Phases * get_next returns a pointer to the next phase in the linked list
void put_next sets a pointer to the next phase in the linked list
Parameters * get_first returns a pointer to the first parameter of the phase
void set_first_param_to sets the first pointer of the phase to be the specified phase

Virtual Functions:

void calculates updates all the parameters in the phase
void geo_segment creates the graphical representation of the phase
FUNCTIONS OF THE PHASES CLASS

Location: phases.C

Function: protected
float prev_number (char *name)
returns the float value of the parameter name in the previous phase

Argument Description:
char * name the name of the parameter

Function: protected
float number (char *name)
returns the float value of the parameter name in the current phase

Argument Description:
char * name the name of the parameter

Function: protected
char * word (char *name)
returns the character value of the parameter name in the current phase

Argument Description:
char * name the name of the parameter

Function: protected, overloaded
void set (char *name, float value)
void set (char * name, char *value)
sets the value of parameter name in the current phase to value

Argument Description:
char * name the name of the parameter
float value the value of the parameter
char * value the value of the parameter
Function: protected

void delete_phase_variables()
    used to free up memory when the phase is deleted

Argument Description:
    None

Function: public

Parameters * get_proper_param (char *name, char *value)
    returns a pointer to a newly created parameter of type name and assigns the value
    value to it

Argument Description:
    char * name      the name of the parameter—used to create the right
                     type of parameter object
    char * value     the value to be assigned to the newly created
                     parameter

Function: public

void load_param_values (char **parameter_list, char **values_list)
    assigns all the parameters in the parameter_list to a phase and assigns the values
    taken from the values_list

Argument Description:
    char ** parameter_list the list of the names of the parameters to be assigned
                            to the phase
    char ** values_list    a list from which the values to be assigned to the
                            parameters are taken

Function: public

void set_phase_title_to (char *name)
    sets the name of the phase to name

Argument Description:
    char * name    the name of the phase
Function: public
char * get_phase_title ()
returns the name of the phase

Argument Description:
None

Function: public
Parameters * get_param_pointer (int location)
returns a pointer to the parameter occupying the position location in the linked list

Argument Description:
int location the position the parameter occupies within the parameter linked list

Function: public
float get_float_param_value (char *name)
returns the float value of the phase of name name

Argument Description:
char * name the name of the phase

Function: public
Phases * get_next ()
returns a pointer to the next phase in the phase linked list

Argument Description:
None

Function: public
void put_next (Phases *next)
sets the phase next to be the next phase in the phase linked list

Argument Description:
Phases * next pointer to the next phase in the phase linked list
Function: public

Parameters * get_first ()
returns a pointer to the first parameter of the phase

Argument Description:
None

Function: public

void set_first_param_to (Parameters *first_parameter)
sets the first parameter of the phase to be first_parameter

Argument Description:
Parameters * first_parameter pointer to the what will be defined as the first	parameter in the parameter linked list

Function: virtual

void calculate ()
updates parameter values to reflect dependencies—can be redefined to define new
parameter dependencies

Argument Description:
None

Function: virtual

void geo_segment ()
draws the graphical representation of the phase—can be redefined to allow for more
elaborate graphical representations

Argument Description:
None
THE MISSION_PARAMETERS CLASS

Class Name:
Mission_Parameters

Location:
Mission_Parameters.h

Description:
This class provides all the functions necessary to assign and retrieve mission parameter data. These class is inherited by the class Missions.

Inheritance:
None

Private Variables:
char * name the name of the mission parameter
char * value the value of the mission parameter
char * comment general comment about the mission parameter
cchar * display code to determine whether to display variable = ON/OFF

Mission_Parameters *
next points to the next mission parameter in the mission parameter linked list

Protected Variables:
None

Public Variables:
None

Private Functions:
None

Protected Functions:
None
Public Functions:

void put_next  
Mission_Parameters * get_next  
void set_mp_name_to  
char * get_mp_name  
void set_mp_value_to  
char * get_mp_value  
void set_mp_display_to  
char * get_mp_display  
void set_mp_comment_to  
char * get_mp_comment

assigns a specified mission parameter to be the next one in the Mission_Parameters linked list
returns a pointer to the next mission parameter in the Mission_Parameters linked list
sets the name of the mission parameter
returns the name of the mission parameter
sets the value of the mission parameter
returns the value of the mission parameter
sets the variable to either displayable or non-displayable
returns the code which indicates whether a mission parameter is displayable
sets a comment to the mission parameter
returns the comment of the mission parameter

Virtual Functions:

None
FUNCTIONS OF THE MISSION_PARAMETERS CLASS

Location: Mission_Parameters.C

Function: public

void put_next (Mission_Parameters *next)

assigns the mission parameter next to be the next item in the Mission_Parameters linked list

Argument Description:

Mission_Parameters *
next

pointer to the variable which will be assigned as the next mission parameter

Function: public

Mission_Parameters * get_next ()

returns a pointer to the next item in the Mission_Parameters linked list

Argument Description:

None

Function: public

void set_mp_name_to (char *name)

sets the name of the mission parameter to name

Argument Description:

char *
name

the name of the mission parameter

Function: public

char * get_mp_name ()

returns the name of the mission parameter

Argument Description:

None
Function: public

void set_mp_value_to (char *value)

sets the value of the mission parameter to value

Argument Description:

char * value the value of the mission parameter

Function: public

char * get_mp_value ()

returns the value of the mission parameter

Argument Description:

None

Function: public

void set_mp_display_to (char *code)

sets the display of the mission parameter to code

Argument Description:

char * code code = "ON" or code = "OFF" — describes whether a mission parameter should be displayed

Function: public

char * get_mp_display ()

returns the display code of the mission parameter

Argument Description:

None
Function: public

void set_mp_comment_to (char *comment)

sets the comment of the mission parameter to comment

Argument Description:
char * comment comment which is assigned to the mission parameter

Function: public

char * get_mp_comment ()

returns the comment of the mission parameter

Argument Description:
None
THE MISSIONS CLASS

Class Name:
Missions

Location:
missions.h

Inheritance:
Mission_Parameters Class

Description:
This class serves as the central coordinator for all mission data manipulation. It creates the phase linked lists from information read from a file by instancing the class phases. From the information in the file, it also creates a linked list of mission parameters. The Mission_Parameters functions are directly accessible through inheritance. Routines to create, modify, and destroy any part of the mission are available. This class is inherited by the class Mission_Window

Private Variables:
char * mission_name name of the mission

Protected Variables:
Mission_Parameters *
first_mission_parameter
pointer to the first mission parameter in the Mission_Parameters linked list
char * param_names ordered list of the parameter names
char * names_of_avail_phases list of phases available to the mission

Public Variables:
None

Private Functions:
None
Protected Functions:
void delete_mission_variables
frees up memory taken up by the Missions class

Public Functions:
int check_file_validity
checks to see whether specified file exists
void create_mission
creates mission by loading information from a file
Phases * get_proper_phase
returns pointer to a newly created phase
void save_miss
saves mission information to a file
void set_mission_name_to
sets the name of mission
char * get_mission_name
returns the name of the mission
Phases * get_phase_pointer
returns a pointer to a specified phase
int get_num_of_phases
returns the number of phases in the mission
int get_num_of_params
returns the number of parameters in the mission
void get_avail_phases
retrieves the available phases from a file and stores
them in the protected variable
names_of_avail_phases
Phases * default_phase
returns a pointer to a newly created phase—with
default parameter values
void insert_phase
inserts a phase into the phase linked list
void add_new_phase
adds a new phase to the phase linked list
void delete_phase
removes a phase from the phase linked list
void refresh
updates the entire mission to reflect dependencies
void update_element
updates a specified parameter to reflect a specified
value
void reset_to_default_value
resets a specified parameter to its default value
void reset_phase_defaults
resets all the parameters of the phase to their default
values
void move_phase
moves a specified phase within the phase linked list
to the specified location
void move_param
moves a specified parameter within the parameter
linked list to the specified location
float retrieve_float_value
returns the float value of the specified parameter
char * retrieve_char_value
returns the character value of the specified parameter

char * retrieve_param_title
returns the name of the specified parameter

char * retrieve_phase_title
returns the name of the specified phase

float get_max_param_value
returns the maximum value in the mission for a specified parameter

float get_min_param_value
returns the minimum value in the mission for a specified parameter

void copy_element
copies a specified element onto a second specified element

void assign_values_to_phase
assigns values to a list of parameters for the specified phase

void load_mission_parameters
creates the mission parameters by reading information from a file

void write_mission_parameters
writes the mission parameters to a file

Mission_Parameters *
get_mp_pointer
returns a pointer to a specified mission parameter

void set_mp_value_to
assigns a specified value to the mission parameter

float get_float_mp_value
returns the float value of the mission parameter

char * get_char_mp_value
returns the character value of the mission parameter

void write_to_acsynt_file
writes the mission information to ACSYNT format

Virtual Functions:
None
FUNCTIONS OF THE MISSIONS CLASS

Location: missions.C

Function: protected

void delete_mission_variables ()
frees up memory allocated to the mission

Argument Description:

None

Function: public

int check_file_validity (char *filename )
checks to see whether file filename exist—returns 1 if YES; 0 if NO

Argument Description:

char * filename name of the file

Function: public

void create_mission (char *filename)
creates the mission by reading information from the file filename

Argument Description:

char * filename name of the file

Function: public

Phases * get_proper_phase (char *name, char **values)
returns a pointer to a new phase whose data type is determined by name and assigns
the values of the parameters from values

Argument Description:

char * name name of the phase
char ** values list of values from which the parameter values are assigned
Function: public
void           save_mission (char *filename)
saves the mission information to the file filename

Argument Description:
char * filename  name of the file

Function: public
void           set_mission_name_to (char *name)
sets the name of the mission to name

Argument Description:
char * name      name of the mission

Function: public
char *           get_mission_name ()
returns the name of the mission

Argument Description:
None

Function: public
Phases *          get_phase_pointer (int location)
returns a pointer to the phase specified by location

Argument Description:
int location      position of the phase within the phase linked list

Function: public
int               get_num_of_phases ()
returns the number of phases within the mission

Argument Description:
None
Function: public

int get_num_of_params ()
returns the number of parameters within the mission

Argument Description:
None

Function: public

int get_avail_phases ()
retrieves the names of the available phases from a file and stores them in the
variable names_of_avail_phases

Argument Description:
None

Function: public

Phases * default_phase (char *name)
returns a pointer to a new phase whose data type is determined by name—the
parameter values are taken from a file

Argument Description:
char * name name of the phase

Function: public

void insert_phase (char *name, int location)
inserts a new phase of name name in the location location of the phase linked list

Argument Description:
char * name name of the phase
int location position of the phase within the phase linked list
Function: public

void add_new_phase (char *name)

adds a new phase of name name at the end of the phase linked list

Argument Description:

char* name name of the phase

Function: public

void delete_phase (int location)

removes the phase located at position location within the phase linked list

Argument Description:

int location position of the phase within the phase linked list

Function: public

void refresh ()

updates the entire mission to ensure that all values reflect all data interdependencies

Argument Description:

None

Function: public, overloaded

void update_element (int phase_number, int parameter_number, char * value)
void update_element (int phase_number, int parameter_number, float value)

assigns the value value to the parameter determined by phase_number and
parameter_number

Argument Description:

int phase_number the position of the phase within the phase linked list—phase to which the parameter in question belongs
int parameter_number the position of the parameter within the parameter linked list
char* value the new character value of the parameter
float value the new float value of the parameter
Function: public

void reset_to_default_value (int phase_number, int parameter_number)
resets the parameter determined by phase_number and parameter_number to its
default value

Argument Description:

int phase_number the position of the phase within the phase linked list—phase to which the parameter in question belongs

int parameter_number the position of the parameter within the parameter linked list

Function: public

void reset_phase_defaults (int location)
resets all the parameters specified by location to their default values

Argument Description:

int location position of the phase within the phase linked list

Function: public

void move_phase (int location1, int location2)
moves phase from location1 to location2

Argument Description:

int location1 old location of the phase

int location2 new location of the phase
Function: public
void move_param (char *name, int location)
       moves the parameter specified by name to the new position location within the
       parameter linked list

Argument Description:
char * name  name of the parameter
int location new position of the parameter within the parameter
              linked list

Function: public
float retrieve_float_value (int phase_number, int parameter_number)
       returns the float value of the parameter specified by phase_number and
       parameter_number

Argument Description:
int phase_number  the position of the phase within the phase linked
                  list—phase to which the parameter in question belongs
int parameter_number the position of the parameter within the parameter
                       linked list

Function: public
char * retrieve_char_value (int phase_number, int parameter_number)
       returns the character value of the parameter specified by phase_number and
       parameter_number

Argument Description:
int phase_number  the position of the phase within the phase linked
                  list—phase to which the parameter in question belongs
int parameter_number  the position of the parameter within the parameter
                       linked list
Function: public
    char * retrieve_param_title (int location)
    returns the name of the parameter located at location within the parameter linked list

Argument Description:
    int location                  position of the parameter within the parameter linked list

Function: public
    char * retrieve_phase_title (int location)
    returns the name of the phase located at location within the phase linked list

Argument Description:
    int location                  position of the phase within the phase linked list

Function: public
    float get_max_param_value (char * name)
    returns the maximum value of the parameter of name name found within the mission

Argument Description:
    char * name                    name of the parameter

Function: public
    float get_min_param_value (char * name)
    returns the minimum value of the parameter of name name found within the mission

Argument Description:
    char * name                    name of the parameter
Function: public

void *copy_element (int old_phase_pos, int old_param_pos, int new_phase_pos, new_param_pos)
copies the value of a parameter from one position to another

Argument Description:

int old_phase_position position of the phase within the phase linked list to which the parameter which is to be copied from belongs

int old_param_position position of the parameter to be copied from within the parameter linked list

int new_phase_position position of the phase within the phase linked list to which the parameter to be copied to belongs

int new_param_position position of the parameter to be copied to within the parameter linked list

Function: public

void *assign_values_to_phase (char *list, int phase_number)
assigns values to the parameters belonging to the phase specified by the
phase_number—the values are taken from list, a string of entries

Argument Description:

char * list a list of values to be assigned to the parameter—entries within the list are separated by at least one blank character

int phase_number the location of the phase within the phase linked list
Function: public
void load_mission_parameters (FILE *filename)
creates the Mission_Parameters linked list by reading information from the file
filename

Argument Description:
FILE * filename pointer to the file from which to read

Function: public
void write_mission_parameters (FILE *filename)
saves the mission parameter information to the file filename

Argument Description:
FILE * filename pointer to the file to which to write

Function: public, overloaded
Mission_Parameters * get_mp_pointer (int location)
Mission_Parameters * get_mp_pointer (char *name)
returns a pointer to the mission parameter specified by location or name

Argument Description:
int location location of the mission parameter within the
Mission_Parameters linked list
char * name name of the mission parameter

Function: public, overloaded
void set_mp_value_to (char *name, float value)
void set_mp_value_to (char *name, char *value)
sets the mission parameter of name name to the value value

Argument Description:
char * name name of the mission parameter
float value float value of the mission parameter
char * value character value of the mission parameter
Function: public
    float     get_float_mp_value (char * name)
          returns the float value of the mission parameter name

Argument Description:
    char *   name       name of the mission parameter

Function: public
    char *    get_char_mp_value (char * name)
          returns the character value of the mission parameter name

Argument Description:
    char *   name       name of the mission parameter

Function: public
    void     write_to_acsynt_file ()
          saves the mission information to an ACSYNT file

Argument Description:
    None
THE PHASE_DIAGRAM_WINDOW CLASS

Class Name:
Phase_Diagram_Window

Location:
geometry_window.h

Description:
This class creates the graphical representation of the mission data. It inherits the class Scroll_Window to enable it to create the window in which the graphics are displayed. This class is instanced in the class Mission_Window.

Inheritance:
Scroll_Window Class

Private Variables:

PHIGS_Structure_ID *
geo_structure pointer to the PHIGS structure which creates the graphical representation of the data

Mission_Window *
mision_window pointer to the window in which the mission data is displayed

float x_scaling the scaling factor along the horizontal direction for the graph

float y_scaling the scaling factor along the vertical direction for the graph

float minimum_x minimum x value for the graph (scaled)

float maximum_x maximum x value for the graph (scaled)

float maximum_y maximum y value for the graph (scaled)

Pint color_index_red color table index number for the color red

Pint color_index_green color table index number for the color green

Pint color_index_white color table index number for the color white

Protected Variables:
None
Public Variables:
None

Private Functions:
void create_axis creates the axis for the graph
int set_proper_zoom sets the proper zooming factor for the graph
void initialize_colors sets color table indices to represent required colors
void display_error_message displays error message in the window
void process_geometry_view processes events detected within the window

Protected Functions:
None

Public Functions:
void create_geometry creates the graphical representation of the data
void destroy_geo_structure frees up the memory taken up by the PHIGS structure which creates the graphical representation

Virtual Functions:
None
FUNCTIONS OF THE PHASE_DIAGRAM_WINDOW CLASS

Location: geometry_window.C

Function: private
void create_axis ()
creates the axis used in the graphical representation

Argument Description:
None

Function: private
void set_proper_zoom ()
zooms the display window by the proper factor to display the graph correctly

Argument Description:
None

Function: private
void set_proper_scale ()
calculates the correct scaling factor which must be used to zoom the window correctly

Argument Description:
None

Function: private
void initialize_colors ()
assigns the correct colors to the color indices to be used by the class

Argument Description:
None
Function: private
void display_error_message()

displays an error message instead of the graph—used whenever the data available is insufficient to create a graph

Argument Description:
None

Function: private
void process_geometry_view(int choice, Ppoint3 *loc_pos, int view_index, Ppick_path *pick_path, Event *event)

used to detect events that occur within the display area of the window

Argument Description:
int choice value entered using the choice logical device
Ppoint3 * loc_pos x, y, and z values of the location selected using the locator logical device
int view_index index of the view in which the event was detected
Ppick_path * pick_path the pick path returned by the logical pick device
Event * event pointer to the event detected

Function: public
void create_geometry()

creates the graphical representation of the mission data

Argument Description:
None

Function: public
void destroy_geo_structure()

deletes the PHIGS structure which creates the graphical representation

Argument Description:
None
THE MISSION_WINDOW CLASS

Class Name:
  Mission_Window

Location:
  mission_window.h

Description:
  This class acts as the central coordinator of all other classes. It creates and controls
display of the mission data. It creates the mission data itself through instancing of
the class Missions and has access to all the functions found within that class
through inheritance. To enable it to create the interface it also inherits the class
Scroll_Window. Whenever appropriate, instancing of the class
Phase_Diagram_Window occurs to display the graphical representation of the
mission.

Inheritance:
  Scroll_Window Class
  Missions Class

Private Variables:
  Phase_Diagram_Window *
    geo_window pointer to the window which displays the graphical
    representation of the mission data
  Pint white_color the color table index for the color white
  Pint red_color the color table index for the color red
  Pint blue_color the color table index for the color blue
  Pint gray_color the color table index for the color gray
  Color_Group *
    menu_color pointer to the set of colors used to display menus
  Color_Group *
    label_color pointer to the set of colors used to display labels
Color_Group *
        input_color  pointer to the set of colors used to display input text
int        menu_face_color  index of the color used in coloring the faces of
        menus
Push_Button *
        autozoom_button  pointer to the push button used for auto zooming
Push_Button *
        mission_parameter_button  pointer to the push button used to toggle the mission
        parameters menu
Push_Button *
        options_button  pointer to the push button used to display options
        menu
Push_Button *
        quit_button  pointer to the push button quit
Push_Button *
        geo_window_button  pointer to the push button used to toggle the window
        that displays the graphical representation of the data
        on and off
Label *
        autozoom_label  pointer to the label used by autozoom_button
Label *
        mission_parameter_label  pointer to the label used by mission_parameter_button
Label *
        options_button_label  pointer to the label used by options_button
Label *
        quit_button_label  pointer to the label used by quit_button
Label *
        geo_window_button_label  pointer to the label used by geo_window_button
PHIGS_Structure_ID *
        mission_structure  pointer to the PHIGS structure which displays the
        mission data
Interface_Manager *
        manager  pointer to the interface manager which governs and
        controls the various windows and menus on the
        screen
Static_Menu *

mission_parameters_address
    pointer to the static menu—mission parameters menu

int exit_mission_parameter_menu
    flag used to indicate exit from the mission parameters menu

int mission_parameter_menu_toggled
    flag used to indicate that the mission parameters menu was toggled

int toggle_window_flag
    flag used to indicate that the window that displays the graphical representation of the data was toggled

float acsynt_title_loc
    the location of the lower-left hand corner of the ACSYNT title bar (in Normalized Projection Coordinates)

float geometry_view_width
    the width of the data display (in World Coordinates)

int initial_data_loading
    flag used to indicate whether data is being loaded for the first time

int this_structure_id
    the id of the PHIGS structure which displays the data (structure displayed by this window)

int mission_parameter_on
    flag indicating that the mission parameters menu is toggled on

int number_precision
    the number of significant digits with which data is displayed

Ptext_align text_alignment
    tells PHIGS how to align the text to be displayed

float horizontal_spacing
    spacing between columns for display of the data

float vertical_spacing
    spacing between rows for display of the data

char * list_of_files
    a list of all files that contain missions

int first_phase_menu
    flag used to indicate initial launching of phase_menu

int first_filename_menu
    flag used to indicate initial launching of file_name_menu

int first_phase_options_menu
int first_options_menu flag used to indicate initial launching of
    phase_options_menu
int first_add_phase_menu
    flag used to indicate initial launching of
    add_phase_menu
int first_parameter_menu
    flag used to indicate initial launching of parameter
    menu
int first_move_parameter_menu
    flag used to indicate initial launching of
    move_parameter_menu
int first_confirm_menu
    flag used to indicate initial launching of
    confirm_menu
int first_message_menu
    flag used to indicate initial launching of
    message_menu
int first_apply_defaults_menu
    flag used to indicate initial launching of
    apply_defaults_menu
int first_row_column_menu
    flag used to indicate initial launching of
    row_column_menu
int first_number_display_menu
    flag used to indicate initial launching of
    number_display_menu
int first_quick_input_menu
    flag used to indicate initial launching of
    quick_input_menu
int first_retr_del_file_menu
    flag used to indicate initial launching of
    retr_del_file_menu
int first_other_variables_menu
    flag used to indicate initial launching of
    other_variables_menu

int first_select_variables_menu
    flag used to indicate initial launching of
    select_variables_menu

Protected Variables:
    None

Public Variables:
    None

Private Functions:
    void set_additional_differences
        specifies differences between the class
        Mission_Window and the class Scroll_Window
    void create_additional_components
        creates extra components found in the
        Mission_Window class and not in the
        Scroll_Window class
    void delete_additional_components
        deletes the extra components created by this class
    void process_additional_event
        processes events generated by the additional
        components created by this class
    void process_geometry_view
        processes events detected within the display area of
        the window created by this class
    char * filename_menu
        creates the menu prompting the user for a filename—
        returns the file name
    void mission_parameter_menu
        creates the menu which displays the mission
        parameters
    void phase_options_menu
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void options_menu</td>
<td>creates the menu which displays the options for editing at the phase level</td>
</tr>
<tr>
<td>void add_phase_menu</td>
<td>creates the menu by which the user can add phases</td>
</tr>
<tr>
<td>void parameter_menu</td>
<td>creates the menu by which parameters can be edited</td>
</tr>
<tr>
<td>void move_parameter_menu</td>
<td>creates the menu by which the user can move parameters</td>
</tr>
<tr>
<td>void confirm_menu</td>
<td>creates the menu which prompts the user for confirmation</td>
</tr>
<tr>
<td>void message_menu</td>
<td>creates the menu used to display messages to the user</td>
</tr>
<tr>
<td>void apply_defaults_menu</td>
<td>creates the menu by which parameters may be reset to their default values</td>
</tr>
<tr>
<td>void row_column_menu</td>
<td>creates the menu by which the user can alter the row and column spacing</td>
</tr>
<tr>
<td>void number_display_menu</td>
<td>creates the menu by which the user can alter the display of the data layout</td>
</tr>
<tr>
<td>void quick_input_menu</td>
<td>creates the menu by which the user can make modification for an entire phase at once</td>
</tr>
<tr>
<td>void retr_del_file_menu</td>
<td>creates the menu by which the user can retrieve or delete files</td>
</tr>
<tr>
<td>void other_variables_menu</td>
<td>creates the menu in which mission parameters are shown</td>
</tr>
<tr>
<td>void select_variables_menu</td>
<td>creates the menu by which the user can select which variables will be displayed by the other_variables_menu</td>
</tr>
<tr>
<td>void create_mission_structure</td>
<td>creates the PHIGS structure which displays the mission data</td>
</tr>
</tbody>
</table>
char * display_properly modifies data for proper screen display
void refresh_window updates window to reflect latest changes
void initialize_colors assigns specified colors to specified color table indices
void scan_for_pick determines if a logical pick input was generated in the display area
void initializer initializes private class variables upon its creation
void toggle_window toggles window to its current and previous size
int get_list_of_files retrieves list of available files which contain missions

Protected Functions:
None

Public Functions:
void phase_menu creates a menu by which parameters can be modified
void turn_ETC_on turns immediate mode on
void turn_ETC_off turns immediate mode off
void open_initial_mission opens a default file when module is first launched
int get_white_color returns the color table index for white
int get_red_color returns the color table index for red
int get_green_color returns the color table index for green
void process_from_mouse processes events generated from the mouse device

Virtual Functions:
None
FUNCTIONS OF THE MISSION_WINDOW CLASS

Location: mission_window.C

Function: private
void set_additional_differences ()
defines the differences between the Mission_Window class and the Scroll_Window class

Argument Description:
None

Function: private
void create_additional_components ()
creates components that are required, but not part of the Scroll_Window class

Argument Description:
None

Function: private
void delete_additional_components ()
deletes the additional components created by this class

Argument Description:
None

Function: private
void process_additional_event (Event *event)
processes events generated by the additional components created by this class

Argument Description:
Event * event pointer to the event generated
Function: private
void process_geometry_view (int choice, Ppoint3 *loc_pos, int
  view_index, Ppick_path *pick_path, Event *event)

  used to detect events that occur within the display area of the window

Argument Description:
int choice          value entered using the choice logical device
Ppoint3 * loc_pos   x, y, and z values of the location selected on the
  screen
int view_index      index of the view in which the event was detected
Ppick_path * pick_path the pick path of the entity selected
Event * event       pointer to the event detected

Function: private
char * filename_menu (char *default_file)

  creates a menu to prompt the user for a file name—returns the name entered

Argument Description:
char * default_file   the default filename which is offered by the menu

Function: private
void mission_parameter_menu ()

  creates the static menu which displays the principle mission parameters

Argument Description:
None
Function: private
void phase_options_menu (int phase_number)
creates a pop-up menu displaying the options available for making modifications at
the phase level

Argument Description:
int phase_number the position of the phase selected within the phase
linked list

Function: private
void options_menu ()
creates a pop-up menu displaying the general options available

Argument Description:
None

Function: private
void add_phase_menu (int active_phase, int entry_code)
creates a pop-up menu by which to add a new phase before or after (depending on
entry_code) the active_phase

Argument Description:
int active_phase the position of the active phase within the phase
linked list—active phase means it was the phase
selected before the menu was launched

int code flag indicating whether to add the code before or after
the active phase (1 = before, 2 = after)
Function: private

void parameter_menu (int phase_number, int parameter_number)
creates a pop-up menu by which parameters can be modified—presents all the
parameters for the phase phase_number, the parameter parameter_number is
highlighted by default.

Argument Description:

- int phase_number: the position of the phase within the phase linked list
to which the parameter selected belongs
- int parameter_number: the position of the parameter selected within the
  parameter linked list

Function: private

void move_parameter_menu (int new_position)
creates a pop-up menu by which the user can move a parameter to a new position.

Argument Description:

- int new_position: the new position of the parameter

Function: private

void confirm_menu (char *prompt)
creates a pop-up menu asking the user for confirmation to the prompt.

Argument Description:

- char * prompt: the prompt to be displayed

Function: private

void message_menu (char *message)
creates a pop-up menu which displays the message message to the user.

Argument Description:

- char * message: the message to be displayed
Function: private

void apply_defaults_menu (int phase_number, int parameter_number, char *title)

creates a pop-up menu by which a parameter can be reset to its default value—the parameter in question is determined by phase_number and parameter_number

Argument Description:

int phase_number the location of the phase within the phase linked list to which the parameter in question belongs
int parameter_number the location of the parameter within the parameter linked list
cchar * title the heading to be displayed by the menu

Function: private

void row_column_menu ()

creates a pop-up menu by which the column and row spacing of the layout can be altered

Argument Description:

None

Function: private

void number_display_menu ()

creates a pop-up menu by which the appearance of the numeric data can be altered

Argument Description:

None

Function: private

void quick_input_menu ()

creates a pop-up menu which allows the user to change all the parameters for an entire phase at once

Argument Description:

None
Function: private
    void retr_del_file_menu ()
    creates a pop-up menu by which mission files can be retrieved or deleted

Argument Description:
    None

Function: private
    void other_variables_menu (int position)
    creates a pop-up menu which displays mission parameters selected as being
    viewable in the select_variables_menu

Argument Description:
    int position              the position within the Mission_Parameters linked
                              list of the first variable to be displayed by the menu

Function: private
    void select_variables_menu (int position)
    creates a pop-up menu by which the user can determine which variables will be
    displayed by the other_variables_menu

Argument Description:
    int position              the position within the Mission_Parameters linked
                              list of the first variable to be displayed by the menu

Function: private
    void create_mission_structure (int row, int column)
    creates the PHIGS structure which displays the mission data—updates the data
    (including graphical representation) to reflect latest changes before posting it to the
    screen

Argument Description:
    int row                  row of data to highlight—used to denote a phase as active
    int column               column of data to highlight—used to denote a parameter as active
Function: private
void refresh_window ()
updates the display window to reflect latest changes

Argument Description:
None

Function: private
void initialize_colors ()
modifies the PHIGS color table—assigns colors to a set of color table indices

Argument Description:
None

Function: private
void scan_for_pick (Pop_Up_Menu *menu, Event *event)
tests whether event contains input from a logical pick device—used whenever the
interface manager sends a message of value zero (0)

Argument Description:
Pop_Up_Menu *
    menu the menu from which the function is called
Event * event the event detected

Function: private
void initializer ()
initializes all private class variables when the data type is first created

Argument Description:
None
Function: private
void toggle_window (float min_x, float max_x, float min_y, float max_y)
toggles the window between its current size and the size specified by the arguments passed

Argument Description:
float min_x minimum x value of the window (Normalized Projection Coordinates)
float max_x maximum x value of the window (NPC)
float min_y minimum y value of the window (NPC)
float max_y maximum y value of the window (NPC)

Function: private
void toggle_window (float min_x, float max_x, float min_y, float max_y)
toggles the window between its current size and the size specified by the arguments passed

Argument Description:
float min_x minimum x value of the window (Normalized Projection Coordinates)
float max_x maximum x value of the window (NPC)
float min_y minimum y value of the window (NPC)
float max_y maximum y value of the window (NPC)

Function: private
int get_list_of_files (char *directory_name)
retrieves a list of files from the directory directory_name that contain mission data—returns 1 if successful; 0 if unsuccessful

Argument Description:
char * directory_name directory path name of the directory in which the files are searched for
Function: public

void phase_menu (int phase_number, int parameter_number)

creates a pop-up menu by which parameters can be modified—presents all the
values for a given parameter; the parameter value for phase phase_number is
highlighted by default

Argument Description:

int phase_number the position of the phase within the phase linked list
to which the parameter selected belongs

int parameter_number the position of the parameter selected within the
parameter linked list

Function: public

void turn_ETC_on ()

turns immediate mode on for certain items of the interface

Argument Description:

None

Function: public

void turn_ETC_off ()

turns immediate mode off for certain items of the interface

Argument Description:

None

Function: public

void open_initial_mission ()

opens a default mission file when the module is first launched—the name of the
default file is hard coded in the function

Argument Description:

None
Function: public
void get_white_color ()
returns the color table index for the color white which was defined by the initialize_color function

Argument Description:
None

Function: public
void get_red_color ()
returns the color table index for the color red which was defined by the initialize_color function

Argument Description:
None

Function: public
void get_green_color ()
returns the color table index for the color green which was defined by the initialize_color function

Argument Description:
None
Function: public

void process_from_mouse (int choice, Ppoint3 *loc_pos, int view_index,
                        Ppick_path *pick_path, Event *event)

    processes events that are generated from the mouse

Argument Description:

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>choice</td>
<td>value entered using the choice logical device</td>
</tr>
<tr>
<td>Ppoint3</td>
<td>loc_pos</td>
<td>x, y, and z values of the location selected on the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>screen</td>
</tr>
<tr>
<td>int</td>
<td>view_index</td>
<td>index of the view in which the event was detected</td>
</tr>
<tr>
<td>Ppick_path</td>
<td>pick_path</td>
<td>the pick path of the entity selected</td>
</tr>
<tr>
<td>Event</td>
<td>event</td>
<td>pointer to the event detected</td>
</tr>
</tbody>
</table>
ADDITIONAL FUNCTIONS

Location: utilities.C

Function: does not belong to a class

int format_type (char *value)

determines if value is purely numeric or if it contains symbols which render it a
"word"—this function is used since PHIGS can only accept string inputs; returns 1
if it is a "word", returns 0 if it is a "number"

Argument Description:
char * value entry whose format is to be tested

Function: does not belong to a class

char * word (char *value)

ensures that value is a single word—leading blank characters are removed; value is
truncated at first blank character which follows a valid character; returns the
modified word

Argument Description:
char * value entry which is to be modified
OTHER CLASSES

The following classes are used by the Mission Profile Input System but were not created by the author. One of the advantages of using an object-oriented language is that it allows for high code reusability. The classes are not discussed in detail. They are listed here for completeness.

For more information on the following classes refer [Steu93]:

- Scroll_Window

For more information on the following classes refer [Woya92 & Woya93]:

- PHIGS_Structure_ID
- Check_Box
- Color_Group
- Event
- Frame
- Interface_Manager
- Label
- Number_Box
- Pop_Up_Menu
- Push_Button
- Static_Menu
- Text_Input
VITA

Francisco Rivera Jr. was born January 11, 1969 in McAllen, Texas. He grew up surrounded by orchards in the small town of Santa Paula, California. He attended the University of California at Santa Barbara where he studied Mechanical Engineering. After graduating in 1991 he decided to head east to Virginia Tech where he did his graduate work in computer-aided design. Upon graduating from Virginia Tech the author will return to work close to home in San Jose, California.