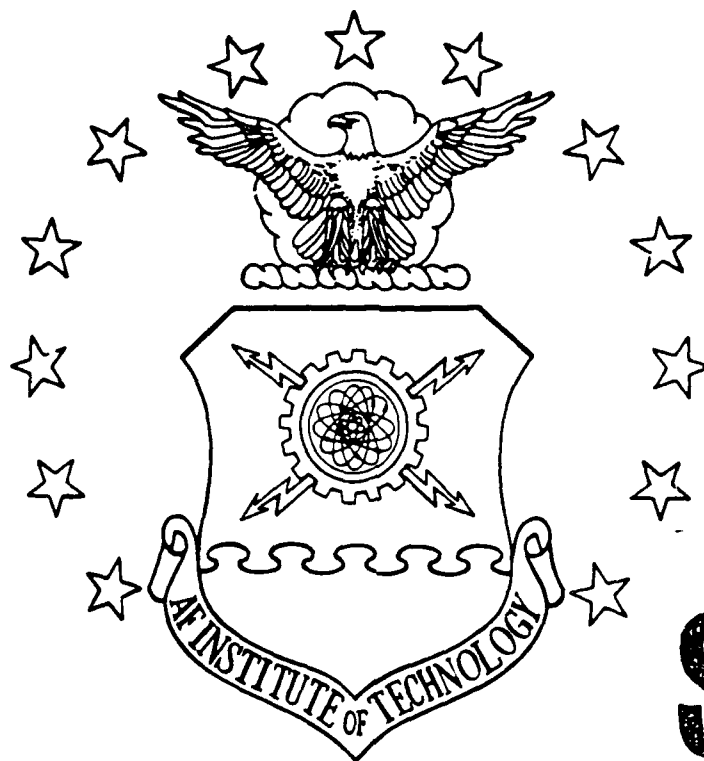


AD-A215 551



DTIC
ELECTE
DEC 15 1989
S B D

USE OF HYPERMEDIA AS A USER INTERFACE
FOR AN ARTIFICIAL INTELLIGENCE-BASED
PROBLEM SOLVER

THESIS

Daniel J. Florian, B.S.

Captain, USAF

AFIT/GCS/ENG/89D-3

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

DISTRIBUTION STATEMENT A

Approved for public release;
Distribution Unlimited

89 12 14 018

AFIT/GCS/ENG/89D-3

①

USE OF HYPERMEDIA AS A USER INTERFACE
FOR AN ARTIFICIAL INTELLIGENCE-BASED
PROBLEM SOLVER

THESIS

Daniel J. Florian, B.S.

Captain, USAF

AFIT/GCS/ENG/89D-3

DTIC
ELECTE
DEC 15 1989
S B D

Approved for public release; distribution unlimited

AFIT/GCS/ENG/89D-3

USE OF HYPERMEDIA AS A USER INTERFACE
FOR AN
ARTIFICIAL INTELLIGENCE-BASED PROBLEM SOLVER

TInesis

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Computer Systems

Daniel J. Florian, B.S.
Captain, USAF

December 1989

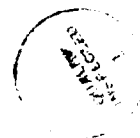
Approved for public release; distribution unlimited

Acknowledgements

I would like to take this *little* bit of time to recognize the people who invested, in their own special way, *so much* of their time in this research project. I am quite sure that without the contributions of these people, the success of this project would have been questionable.

I would like to start by recognizing the most important person in my life: my wife Kathy. It was her constant support that gave me that extra push I sometimes needed to keep going during the rough times. It was her constant love and attention which comforted our children through the many times their "daddy" was working late at school. I will never forget all Kathy did for our family during this time. Also, many thanks and kisses go out to my beautiful children: Joy, Jill, and Beth. Their love and smiling faces make life so much fun and worthwhile.

I want to offer special thanks to these three individuals for contributing their time and their talents: Lt Col Charles Bisbee, Capt Eric Hansen, and Mr Jim Neri. These gentlemen always showed enthusiasm and sincere interest in this project. Never did it appear to me as though they were just going through the motions (although I've been told that I am easily fooled!). Thanks Charlie (can I call you that sir?). Whenever I thought I had too much to do and too little time, I would walk by your office and see someone who really had too much to do and too little time, and this would make me feel better. Your professionalism and perseverance are respected and will be used as a "blueprint" from which I will try to build a copy. Thanks Eric. Your expert knowledge (no pun intended) and friendship will always be remembered and very much appreciated. Thanks Jim. This project is a success because you shared your many years of extremely technical knowledge in such an easily understandable way that even when I was a lieutenant I understood what you were talking about.



Distribution/Availability Codes	
Dist	Avail and/or Special
A-1	

OR

☒
☐
☐

I would like to thank Dr Frank Brown for taking time to read this thesis and also for the opportunity to be a student of his. Thanks also go out to Capt Steve Rasmussen. If it was not for Steve, none of this would have ever happened (i.e., so he is the one to blame!). Lastly, I want to thank the following WRDC/TXI people for their help: Bill Baker for letting me "move into" his shop and for always making me feel like I belong, Mike Wellman for his friendship and for not bumping me off the Macintosh when he needed it, and Wilma Bridges for her cheery disposition and for always offering her help.

I cannot immediately say I would like to do this again, but if I did, I wouldn't mind as much if it involved working with all of you.

Thanks again,

Dan

Table of Contents

Acknowledgements	ii
List of Figures	vi
List of Tables	vii
Abstract	viii
I. Introduction	I-1
1.1 Problem Statement	I-2
1.2 Background	I-2
1.2.1 User Interface	I-2
1.2.2 Hypermedia	I-3
1.2.3 Expert Systems	I-3
1.2.3.1 Expert System User Interface	I-5
1.3 Problem Domain	I-6
1.4 Approach and Methodology	I-6
1.4.1 Knowledge Acquisition Phase	I-7
1.4.2 Hypermedia and Expert System Integration Phase	I-7
1.4.3 Design and Coding Phase	I-8
1.4.4 User Evaluation Phase	I-9
1.5 Hardware and Software	I-10
1.6 Other Support	I-10
1.7 Conclusions	I-11
II. Review of Related Work	II-1
2.1 Hypermedia	II-1
2.1.1 Current Hypermedia Systems	II-3
2.2 AI-Based Problem Solving Strategies	II-4
2.2.1 User Interfaces of Current Expert System Tools	II-7
2.3 Related Military Projects	II-8
2.4 Related Tools	II-9
III. Dual Miniature Inertial Navigation System (DMINS)	III-1
3.1 Background	III-1
3.2 DMINS Testing and Troubleshooting Procedures	III-2
3.2.1 Mode B Testing	III-3
3.2.1.1 Mode B Troubleshooting Procedures	III-4
3.2.2 Mode A Testing	III-5
3.2.2.1 Mode A Troubleshooting Procedures	III-6
3.3 Scope of the Problem Domain	III-6
3.4 Observations	III-7

Table of Contents (cont.)

IV. Prototype Development	IV-1
4.1 Development of Mode B	IV-1
4.1.1 Knowledge Acquisition Phase	IV-1
4.1.2 Hypermedia and Expert System Integration Phase	IV-3
4.1.3 Design and Coding Phase	IV-3
4.1.4 User Evaluation Phase	IV-5
4.1.5 Observations and Conclusions	IV-5
4.2 Development of Mode A	IV-6
4.2.1 Knowledge Acquisition Phase	IV-7
4.2.2 Hypermedia and Expert System Integration Phase	IV-9
4.2.2.1 KMS Provisions for Interprocess Communication	IV-11
4.2.2.2 CLIPS Provisions for Interprocess Communication	IV-13
4.2.3 Design and Coding Phase	IV-13
4.2.4 User Evaluation Phase	IV-15
4.2.5 Observations and Conclusions	IV-15
4.3 Integration of Mode B and Mode A	IV-16
4.4 Constraining KMS's Capabilities	IV-17
4.5 Difficulty During Development	IV-18
 V. Conclusions and Recommendations	 V-1
5.1 Conclusions	V-1
5.2 Recommendations	V-2
5.2.1 DMINS	V-2
5.2.2 Hypermedia System and Expert System Integration	V-3
5.3 Summary	V-4
 Appendix A: DMINS Shop Replaceable Units (SRUs)	 A-1
Appendix B: DMINS Error Messages	B-1
Appendix C: DMINS Calibration Parameters	C-1
Appendix D: First Nine Mode B Tests and Subtests	D-1
Appendix E: CLIPS Source Code	E-1
Appendix F: Frames for Troubleshooting Test B6.1	F-1
Appendix G: Modified CLIPS Main Routine	G-1
Bibliography	BIB-1
Vita	VITA-1

List of Figures

Figure	Page
1.1 Example of a Hypermedia Network	I-4
1.2 GoldWorks II Code to Create a Menu	I-5
1.3 Example of a Rule's User Interface Using Both a Text-based and Hypermedia-based Approach	I-9
2.1 The Representation of Legal Heuristics for Product Liability Written Using ROSIE	II-6
3.1 DMINS ATE Configuration	III-3
4.1 Mode B Test B6.1 Troubleshooting Procedures and Corresponding Hypermedia Frames	IV-4
4.2 Interprocess Communication Scenario	IV-10
4.3 Example of KMS Frame with Two Buttons	IV-11
4.4 Example of a KMS Button's Properties	IV-12
4.5 Same Rule Written in S.1 and CLIPS	IV-15
4.6 Prototype Main Menu Frame	IV-17

List of Tables

Table	Page
4.1 Excerpt From Mode B Troubleshooting Chart	IV-2
4.2 Interprocess Communication Files and Their Function	IV-9
4.3 Four CLIPS Interprocess Communication Rules and Associated Functions	IV-14

Abstract

The use of expert systems as the problem solving strategy in maintenance diagnostic environments has proliferated in the last few years. This is due primarily to the ease with which a diagnostic system can be developed using the expert system approach compared to using other techniques, particularly conventional programming. One important feature which determines the success of such a system is the user interface. Typically, the user interface of an expert system is entirely textual. While developing graphical user interfaces are possible, it requires the programmer to either integrate the expert system with externally written graphics routines, or to use the expert system's own, usually LISP based, programming language. Either method requires experienced programmers to perform many iterations of code development until the user interface is complete. Additionally, in complex problem domains such as maintenance diagnostics, it is often difficult to accurately represent the problem in words alone. Especially for the novice, describing the problem not only in words but also with graphics, facilitates a better understanding of the problem; thus, increasing the probability that the appropriate solution is selected.

This thesis discusses the use of a hypermedia system as the user interface for an expert system. The hypermedia system allowed dynamic creation and editing of the user interface, and collected and transmitted information from the user to the expert system. The expert system, which remains transparent to the user, uses this information to recommend a solution to the problem or to determine more information is needed from the user. Regardless, the expert system communicates these results to the hypermedia system, which then displays them to the user.

Specifically, the prototype developed as a part of this research was designed to help Aerospace Guidance and Metrology Center (AGMC) depot-level technicians troubleshoot the Dual Miniature Inertial Navigation Systems (DMINS) Inertial Measurement Unit

(IMU), which is being used on fast attack submarines. Currently, DMINS technicians use information from automatic test equipment (ATE) to guide their troubleshooting actions. This ATE is driven entirely by test failures resulting from the tested IMU signals being out of the specification limits. In addition to using these signals, the prototype uses IMU signals, not being validated by the current test, to detect problems before a test failure occurs. The capability to find problems prior to a test failing, can significantly decrease the time needed to test an IMU.

A one-day user evaluation of the prototype by an experienced DMINS technician was conducted and documented. The user especially liked the large screen used to display information, the mouse as an input device, the applicability of the prototype as a training aid, and the ease at which the user interface could be modified. The prototype is nearly a complete system, covering the majority of the DMINS troubleshooting knowledge.

1. Introduction

Nearly all problems solvers based on Artificial Intelligence (AI) technology must interact with humans. The interaction, or user interface, consists of queries from the problem solver to the human for more information about the problem, and the presentation of information back to the human. The implementation language (i.e., LISP, Prolog, expert system shells, etc.) of the problem solver is chosen because it has the features needed to solve the problem. Often times, this language is not well suited to implement a worthwhile user interface. In fact, Somers found that a number of expert system shells, including M.1 and Rulemaster, were too limited in their graphics capabilities [Somers, 1988].

In complex problem domains, such as maintenance diagnostics, it becomes extremely difficult to accurately represent the problem in words alone. Particularly with the novice technician, the problem should be described not only with words but also with graphics (i.e., pictures, diagrams, schematics, etc.) [Thomas and Clay, 1988:143]. Having both text-based and graphics-based information better enables the technician to understand the problem; thus, the technician has a higher chance of selecting the appropriate solution.

In the remainder of this chapter, a statement of the problem is given (Section 1.1), as well as background information concerning the user interface, hypermedia, and expert systems (Section 1.2). Section 1.3 describes the problem domain and Section 1.4 outlines the steps involved in developing the prototype. The next two sections, Sections 1.5 and 1.6, detail the resources used and acknowledge the organizations which provided support. Finally, Section 1.7 summarizes this chapter and provides a road map for the remainder of this thesis.

1.1 Problem Statement.

This research project designed and prototyped a maintenance troubleshooting system to help technicians isolate and repair faulty components. It uses a hypermedia system as the user interface, and an expert system shell as the problem solver. The main thrust of this research identifies the interface requirements between the hypermedia system and the expert system shell. Based on these findings, generic interface requirements between hypermedia systems and expert system shells, and design considerations for future hypermedia systems and expert system shells to facilitate these interface requirements, are proposed.

1.2 Background.

In the last few years, hypermedia, the man-machine interface, and expert systems all have been getting much deserved attention. In 1987, the first major conference devoted entirely to hypermedia was held. "Hypertext '87" had participants from five continents and included individuals with graphics, software engineering, philosophy, psychology, medical, and religious backgrounds, to name a few [Smith and Weiss, 1988:817]. Past research in the human factors area is starting to pay off. Features which improve the man-machine interface, such as windows, menus, and pointing devices such as the mouse, are becoming the standard in software design. Because of the continued advancements in computer hardware, Waterman points out that it is now practical for small computers to be designed for and dedicated to particular expert systems [Waterman, 1986: 221]. The following sections provide additional information on the man-machine or user interface, hypermedia, and expert systems.

1.2.1 User Interface. The software of today is becoming increasingly more sophisticated [Akscyn, et al., 1988:835] and often times increasingly more difficult to use. The success of a software program relies on the ability of the human to productively use it

[Gordon, 1989:1]. This becomes an even more complicated problem realizing that there is a continually growing number and diversification of inexperienced users [Saja, 1985:36]. While the use of menus, windows, and pointing devices such as the mouse has improved the quality of the user interface, there is still room for more improvements. Many of these improvements can be found in features of hypermedia systems.

1.2.2 Hypermedia. A hypermedia system organizes information into small units, typically called nodes or frames, and allows these nodes to be interconnected or linked together forming a network. These "links" allow a user to travel from node to node, usually by just clicking a mouse on areas of a frame called buttons. The term "navigate" is used to describe the traversing of a hypermedia network. Nodes can contain both graphics and text and are usually displayed as an entire screen or inside a window [Akscyn, et al., 1988:820]. Ease of use, superior graphics, and multiple user capability are some of hypermedia's desirable features. Figure 1.1 is an example of what a small hypermedia network might look like. The nodes are represented as boxes, and can be thought of as screens, and the links are represented as arrows.

1.2.3 Expert Systems. Expert systems have been around since the mid- to late-70s, and have found their way into such diverse areas as medicine (MYCIN), mathematics (MACSYMA), geology (PROSPECTOR), computer systems (XCON), and military science (KNOBS) [Waterman, 1986:40-48]. Waterman states that an expert system is "a computer program using knowledge to attain high levels of performance in a narrow problem area" [Waterman, 1986:11,239]. The knowledge used by the expert system must be collected from the human experts in the specific problem area and from other existing resources (i.e., user manuals, schematics, databases, etc.). This knowledge must then be coded into the expert system. To facilitate the coding process, an expert system shell, such as CLIPS [Johnson Space Center, 1989], OPS5 [Waterman,1986:362], and Knowledge Engineering Environment (KEE) [IntelliCorp, Inc., 1988], are usually used. Benefits of

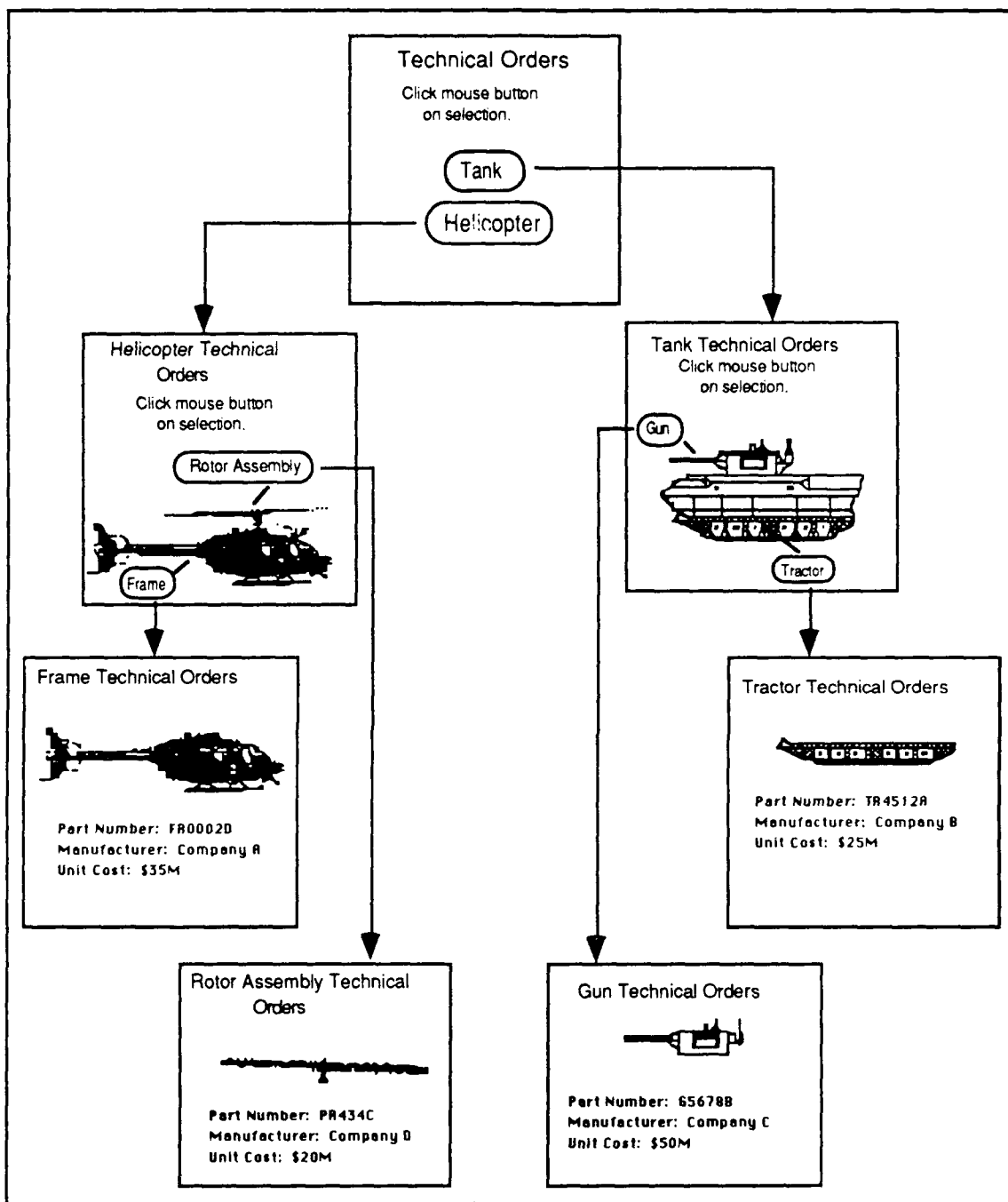


Figure 1.1: Example of a Hypermedia Network

using expert systems include: lower training costs, faster problem resolution for novices, minimization of the impact of employee turnover, and improved effectiveness of the

novices such that their effectiveness approaches that of the experts [AI Squared, Inc., 1989].

1.2.3.1 Expert System User Interface. When expert systems first emerged, their user interface was entirely text-based. Today, the standard interface is still text-based, but many expert systems such as KEE [IntelliCorp, Inc., 1988], GoldWorks II [Gold Hill Computers, Inc., 1989], and S.1 [Teknowledge, Inc., 1987] allow graphics to be incorporated into their user interface. For each of these systems, the screen layouts are programmed using a text editor to input the code. Testing syntax of the code is accomplished and an iterative refinement of the code is used until the graphics object appears in the correct screen location. Figure 1.2 is an example of code needed for GoldWorks II to display a simple menu.

```
(define-instance choose-sev-1 (:is popup-choose-several)
  (instructions ("On a Popup-choose-several-menu,"
    :return "you can make more than one selection."))
  (elements (("Item 1")
    ("Item 2")
    ("Item 3"))))
(x 50)
(y 50 )
```

Figure 1.2: GoldWorks II Code to Create a Menu

The hypermedia systems of today allow on-screen creation, positioning, and editing of graphics, without worrying about screen coordinates, object dimensions, and command syntax. This tremendously simplifies programming the user interface and gives the programmer a "what you see is what you get" (WYSIWYG) approach to screen creation.

1.3 Problem Domain.

The Dual Miniature Inertial Navigation System (DMINS) has been selected as the problem domain for this research. The DMINS is an inertial navigation system used on fast attack submarines, oceanographic survey ships, and aircraft carriers. Depot repair for the DMINS is conducted by the Aerospace Guidance and Metrology Center (AGMC) located at Newark AFB OH. AGMC repairs about 15 DMINS inertial measurement units (IMU) each month; each unit requires a minimum of 70 hours to test. The test technician is responsible for selecting appropriate tests to perform based on output from the DMINS automated test equipment (ATE). Using the results of the testing, the technician will identify none, one, or more than one of the 38 shop replaceable units (SRUs) as being faulty. Because of the complexity of the DMINS, the low number of IMUs tested each month, and the length of the automated tests, a technician needs many years of experience to acquire the in-depth level of knowledge required to perform efficient diagnosis and repair of the DMINS IMU. [Rasmussen, 1988:1369]

An Air Force Institute of Technology (AFIT) thesis written by Capt Skinner also used the DMINS for its problem domain [Skinner, 1988]. Capt Skinner developed a DMINS diagnostics system that blended shallow reasoning techniques, which use empirical knowledge, with model-based reasoning techniques, which use structural and behavioral knowledge. Capt Skinner's system provided assistance to the technician during one phase of the DMINS troubleshooting process. The shallow reasoning part of his research was incorporated in the prototype developed during this research. The prototype developed here encompasses substantially more of the total troubleshooting process.

1.4 Approach and Methodology.

This research consisted of the following phases: Knowledge Acquisition, Hypermedia and Expert System Integration, Design and Coding, and User Evaluation. All

but the User Evaluation phase were concurrently ongoing for the majority of this project. A general discussion of these phases follows, while Chapter IV provides a more detailed discussion:

1.4.1 Knowledge Acquisition Phase. This phase consisted of interviewing DMINS engineers and technicians to collect knowledge of the DMINS functional characteristics, testing environment, and current testing and troubleshooting procedures. Multiple interviews were conducted with two of AGMC's most experienced technicians. Since Capt Skinner concentrated on only one portion of the DMINS troubleshooting process, emphasis was placed on capturing knowledge on the troubleshooting process as a whole (i.e., starting when AGMC received a faulty IMU, and finishing when a faulty SRU is determined). Decision trees representing the technicians' troubleshooting procedures were constructed from this knowledge and later verified by the technicians. Relevant DMINS documentation, such as Technical Orders (TOs) and "in-house" troubleshooting help sheets supplemented the information provided by the technicians.

1.4.2 Hypermedia and Expert System Integration Phase. During this phase, the details of how the hypermedia system and expert system would communicate were decided. Since the hypermedia system will act as the expert system's user interface, it will collect information from the technician about the current problem being diagnosed. This information needs to be communicated to the expert system so that reasoning can take place. The expert system will try to solve the problem by using the knowledge obtained from the user by the hypermedia system. The expert system will either have enough information to identify the faulty component or determine that more information is needed. Regardless, the expert system must communicate these results to the hypermedia system which, in turn, will communicate it to the user.

The hypermedia and the expert systems communicate with each other via files. Each writes information to a file and waits for the other to read that file and respond. The expert system writes the name of the frame to be displayed next to the user, and the hypermedia system writes user responses to queries. Since performance was not a critical issue in developing the prototype, using files to communicate was chosen because it was the simplest way to implement.

1.4.3 Design and Coding Phase. This phase consisted of coding the DMINS troubleshooting knowledge into the expert system and designing a standardized and informative user interface using the hypermedia system. A total of 240 production rules were created: 135 rules were created using Capt Skinner's project, and 105 rules were created to take into account more of the DMINS troubleshooting process and to allow interaction with the hypermedia system.

As a first step, the user interface was implemented using the text-based user interface built into the expert system. Once the logic of the 240 rules was determined to be correct, incorporation into hypermedia was accomplished. A hypermedia frame was created for every rule which, either by seeking or conveying information, interacted with the user. These same rules were then modified to reference the appropriate hypermedia frame. Figure 1.3 displays the user interface of one rule when implemented using text-based and hypermedia-based approaches.

The hypermedia frames were designed so that no input from the keyboard was required. Not allowing keyboard entries eliminated the need for error checking of the user's responses. All user inputs would be made by using a mouse and clicking on buttons. Clicking on a button executes a small program which writes the user response to a file. Because of the way the hypermedia and expert systems communicate with each other, all paths through the network had to be controlled. The user's navigation privileges were

restricted for two reasons: first, to prevent the user from getting lost in the network; and secondly, to prevent the user from clicking on a button which would convey to the expert system either incorrect information or correct information but at the wrong time. The use of graphics, such as schematics, tables, signal diagrams, and equipment panel diagrams, were used on frames to provide the user with additional information.

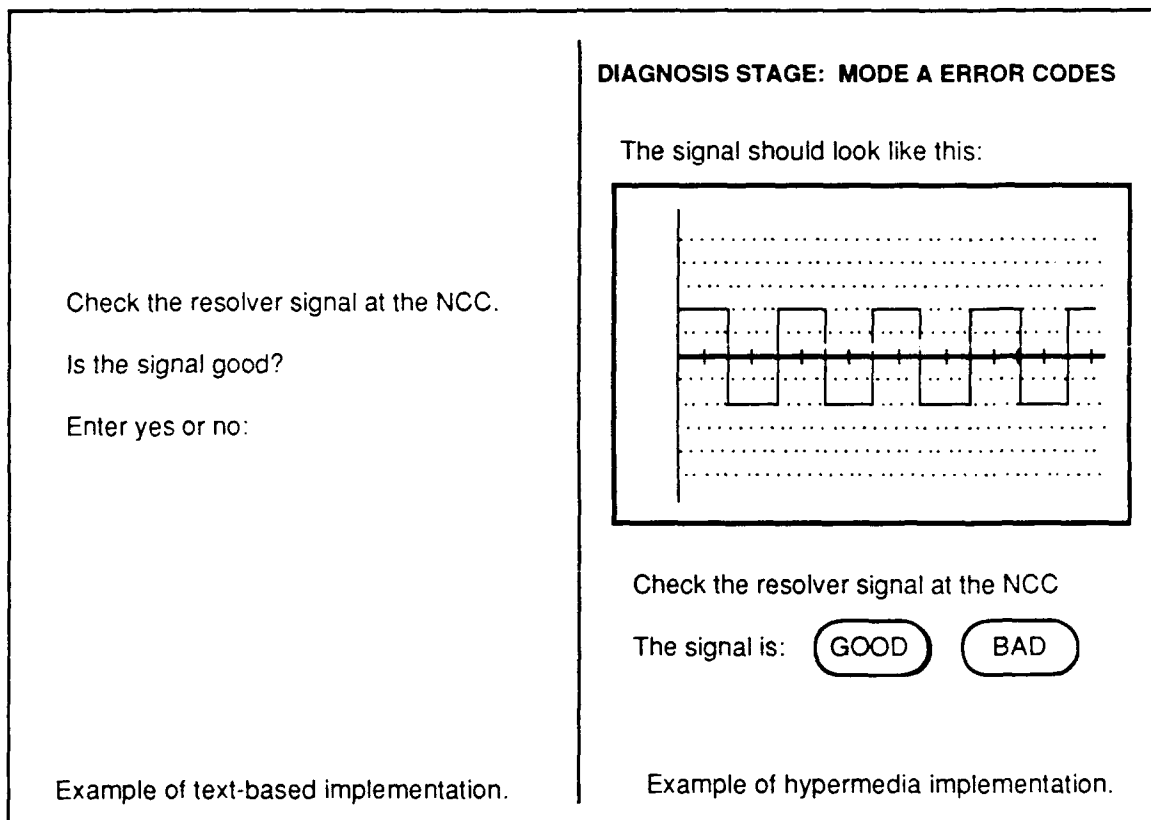


Figure 1.3: Example of a Rule's User Interface Using Both a Text-based and Hypermedia-based Approach

1.4.4 User Evaluation Phase. This phase consisted of a DMINS technician's using the prototype. Numerous test cases were executed, the results of which were compared to known results. The technician provided feedback regarding the design of the user interface, the accuracy of the results, and the prototype's troubleshooting procedures

as a whole. The technician's response to the prototype influenced some of the findings and recommendations documented later in this thesis (See Chapter V).

1.5 Hardware and Software.

The following hardware and software were used to develop the prototype:

Sun 3/60 workstation. This equipment was chosen because of its large screen, speed, and availability. The computer power of this workstation will undoubtedly be available in the personal computers of tomorrow.

Knowledge Management System (KMS). This software application is a hypermedia system developed by Knowledge Systems. KMS was selected because it was available for the chosen hardware suite and had the graphics capability required for the selected problem domain of this project.

C Language Integrated Production System (CLIPS) version 4.2. This expert system was developed by the National Aeronautics and Space Administration (NASA) Artificial Intelligence section. CLIPS was selected because it is available for no charge to offices within the Department of Defense and it provides an easy interface to the "C" programming language.

C language compiler and the UNIX operating system. These tools were used to create the communication link between KMS and CLIPS.

1.6 Other Support.

The sponsor of this thesis, WRDC/TXI, provided the funds to procure the hypermedia system and to support travel to/from NEWARK AFB OH. They also provided use of their computer resources.

Many interviews with DMINS technicians and engineers took place both in person and via telephone. These interviews provided the problem domain knowledge needed to create the prototype.

1.7 Conclusions.

Both expert systems and hypermedia systems have desirable features of their own. Many times, a desirable feature of one will cancel a non-desirable feature of the other. Hypermedia's ease in implementing a user interface, and expert systems' ability to capture knowledge and solve problems, when combined, can increase the probability the problem solver will be a success.

The remaining chapters of this thesis provide a literature review of expert systems and hypermedia (Chapter II); describe the current DMINS testing procedures, testing environment, and ATE configuration (Chapter III); outline the methodology used to develop the system prototype (Chapter IV); and finally, summarize the results of this research project (Chapter V).

II. Review of Related Work

This chapter provides background information concerning hypermedia and Artificial Intelligence (AI) approaches to problem solving in the diagnosis environment. Of the AI techniques available in the diagnosis domain, emphasis will be placed on expert systems since this research chose to implement the prototype using that technique. The importance of the user interface to the success of a problem-solver is also stressed. First, general information and terminology concerning hypermedia will be given. Secondly, AI background information, concentrating on expert systems, is provided. Thirdly, a section outlining several military-related projects is presented. This section includes current projects which use expert systems for diagnosis. In the last section, current tools which combine hypermedia and expert systems techniques are identified.

2.1 Hypermedia.

In a 1945 article, Vannevar Bush described a microfilm based machine called the memex, which was an extensive on-line text and graphics system. The essential feature of the memex was its ability to tie (link) two items together [Bush, 1945]. While Bush introduced the basic ideas of modern hypermedia systems, Theodore Nelson in 1965 actually used the term "hypertext" first. [Nelson, 1965]. The term "hypertext" is used to describe hypermedia systems focusing mainly on textual information [Nielson, 1989]. Halasz gives this definition of hypermedia:

Hypermedia is a style of building systems for information representation and management around a network of multi-media nodes connected together by typed links. [Halasz, 1988:836]

These nodes, sometimes called frames [Akscyn and others, 1988:3], are usually used to represent a single idea or concept [Conklin, 1987:35] and contain a combination of text,

graphics, and buttons. Buttons are textual or graphical regions of the node which, when activated (usually by the clicking of a mouse button), perform one or many functions. For example, clicking a button could cause another node to be displayed, a digitized audio recording to be played, an animation sequence to be shown, and a computer program to be executed [Conklin, 1987:17-18]. Looking back to Figure 1.1, the top node, entitled "Technical Orders", has two buttons each of which is linked to another node. Clicking on the "Tank" button will display the "Tank Technical Orders" node. Clicking on the "Helicopter" button will cause the "Helicopter Technical Orders" node to be displayed next. Conklin calls the collection of all the nodes in a hypermedia network a "hyperdocument" [Conklin, 1987:19]. One of the main differences between a hyperdocument and a conventional paper document is that the hyperdocument is designed to be viewed nonlinearly. Smith and Weiss provide this excellent analogy:

Each node (of a hypermedia network) can be thought of as analogous to a short section of an encyclopedia article or perhaps a graphic image with a brief explanation. The links join these sections to one another to form an article as a whole and the articles to form an encyclopedia. [Smith and Weiss, 1988:817]

The nonlinearity of a hyperdocument allows the reader to choose the sequence in which the information is presented [Charney, 1987:111]. While the reader of a conventional paper document could choose the sequence in which the pages are read, what was read might not make sense or mean what the author intended. A good source of more information on hypermedia is Conklin's survey paper entitled "Hypertext: An Introduction and Survey" [Conklin, 1987]. The following paragraph defines other hypermedia terms.

The term "navigate" is used to describe the traversing from node to node in a hypermedia network. To aid navigation, some hypermedia systems, such as NoteCards, Intermedia, and gIBIS, include a "browser" node which displays a structural or tree diagram of the hypermedia network [Trigg and Irish, 1987:92]. Using the browser node,

the user can select the next node to be displayed or even delete and create nodes [Halasz, 1988:838].

Hypermedia systems have been used to aid software engineering (Neptune [Delisle and Schwartz, 1986]), to create paper and electronic documents (Writing Environment [Smith, 1986]), to develop a tutorial and diagnostics system for the Space Shuttle [O'Reilly and others, 1988], and to demonstrate the concept of a hypertext-based maintenance technical manual [Stone and others, 1982]. Some of the more important hypermedia applications are described in the next section.

2.1.1 Current Hypermedia Systems. This section describes four hypermedia systems: ZOG/Knowledge Management System (KMS), Intermedia, NoteCards, and HyperCard. Many other system are available today for use on personal computers and computer workstations.

ZOG/KMS. ZOG was developed in 1972 at Carnegie-Mellon University. It consisted of a menu-driven interface and allowed sharing of data. In 1980, ZOG was used as a management system aboard the USS Carl Vinson [Akscyn and others, 1988:821]. ZOG, probably the largest and most thoroughly tested hypermedia system, has evolved into the hypermedia application called KMS. ZOG/KMS allows only two nodes to be viewed at a time and does not include a browser node [Conklin, 1987:26]. A "general-purpose, block-structured programming language" is included to "extend the functionality of KMS" [Knowledge Systems, 1989].

Intermedia. This system was developed at Brown University and is the result of two decades of work. It is being developed as an organizational tool to help professors present their lesson material and as an interactive tool for "students to study the lesson material and add their own annotations and reports" [Conklin, 1987:29]. Intermedia consists of a text

editor, graphics editor, timeline editor, and various other editors. Documents and drawings created with these editors can be linked together to form a hypermedia network [Beeman and others, 1987:71].

NoteCards. This application was developed at Xerox Palo Alto Research Center to aid "authors, researchers, designers, and other intellectual laborers" with their ideas. It is fully integrated with a LISP programming environment which allows it to be integrated into other LISP-based programs [Halasc, 1988:839]. Multiple nodes, called "notecards", can be displayed at one time and be of different sizes. NoteCards supports typed nodes which allows nodes to be of a certain type (i.e., text, sketch, graph, etc.) [Trigg and Irish, 1987:91].

HyperCard. This system, developed by Apple Computer, exclusively runs on Apple Macintosh computers. HyperCard features include user-friendly drawing tools, the ability to import graphics files, and its own programming language (HyperTalk). HyperCard does not provide different sized nodes or linking within a field of text. [Apple Computer, Inc., 1987]

2.2 AI-Based Problem Solving Strategies.

The term "artificial intelligence" originated in 1956 at a conference held at Dartmouth College [Schoen and Sykes, 1987:2] and, according to Winston, can be defined as "the study of ideas which enable computers to do the things that make people seem intelligent" [Winston, 1977:1]. But it was not until DENDRAL in the mid-1960s and MYCIN in the mid-1970s that AI applications were considered successful. DENDRAL assists chemists in analyzing spectroscopic data, while MYCIN helps physicians in the diagnosis and treatment of meningitis and bacteremia infections [Harmon and King, 1985:15,134]. Besides medical and chemical diagnosis, AI-based systems have since been used in

geological exploration, management job shop scheduling, computer configuration, job training, and numerous other areas [Blais, et al., 1984:7]. Three popular problem solving strategies used in AI-based diagnostic systems are decision trees, production systems, and model-based reasoning. Each of these will be discussed in the following paragraphs.

Decision trees, sometimes called fault trees, are a simple and efficient way to write down the sequence of tests and conclusions needed to guide a diagnosis [Davis and Hamscher, 1988:305]. They are made up of several sequences of events, each called a path. Each path systematically leads to a conclusion. For example, a path could start with an error message displayed on an ATE terminal and conclude with a recommended repair action. A disadvantage in this strategy is that different paths can lead to the same conclusion. For complex systems, this duplication of data can prove to be too cumbersome to manage [O'Reilly, et al., 1988:470]. Also, a small change in the device could cause a major restructuring of the paths [Davis and Hamscher, 1988:305]. The decision tree strategy is implemented in many maintenance manuals in the Air Force [Davis and others, 1983:7]. Production systems are an alternative approach.

Production or rule-based systems represent knowledge in the form of "condition-action" pairs [Cohen, 1982:157] and are called expert systems. One "condition-action" pair is called a production rule or simply a rule. Figure 2.1 is an example of a rule written in the expert system called ROSIE. Notice the "condition" or "if" part and the "action" or "then" part of the rule.

Waterman defines an expert system as "a computer program that uses expert knowledge to attain high levels of performance in a narrow problem area" [Waterman, 1986:390]. Two popular expert systems are MACSYMA, which is used for symbolic mathematics; and PROSPECTOR, which is used to identify mineral deposits [Hayes-Roth and others, 1983:9-10]. The knowledge used in these expert systems were gathered from an expert in the particular problem domain using knowledge engineering techniques

[Waterman, 1986:152-161]. This type of knowledge is called empirical because it is derived solely from the expert's experience. Much of this knowledge is also considered to be heuristic in nature (i.e., rules of thumb vice algorithmic) because many times the problem is too complex too be solved optimally [Harmon and others, 1988]. The knowledge is then converted into production rules.

If the plaintiff did receive an eye injury
and there was just one eye that was injured
and the treatment for the eye did require surgery
and the recovery from the injury was almost complete
and visual acuity was slightly reduced by the injury
and the condition is fixed,
then increase the injury trauma factor by \$10,000.

Figure 2.1: The Representation of Legal Heuristics for Product Liability written using ROSIE [Hayes-Roth, 1985:922]

An expert system's architecture has three main components. A data store component, also called working memory or data memory, contains facts specifically pertaining to the current problem being diagnosed. Another component is the production rules which represent the domain-knowledge captured during the knowledge engineering sessions. The final component is the inference engine. The inference engine determines which rules to execute based on the facts in working memory. Since rules only execute based on the facts which are known to be true (i.e., found in the working memory), there is no procedural order implied by the rules. Because of the lack of procedural order to the rules, expert systems are said to be data-driven programs. [Brownston and others, 1985:6-7]

For complex systems, the time required to obtain knowledge can be lengthy. As the number of production rules grows, it becomes more difficult to understand the interactions between the rules, to debug them, and to control their behavior [Fikes and Kehler, 1985:912]. Another disadvantage of using this type of system is that it cannot diagnose problems which have not occurred previously, since the experience of the experts is used to make its decisions. Model-based reasoning techniques do not have this problem.

In the last few years, model-based reasoning techniques have become popular, especially in the diagnosis area. Model-based reasoning tries to represent the structure and behavior of a given device. Once the model is correct, discrepancies between the model and the device can be attributed to the problems with the device. These discrepancies are "clues to the location and character of the faults" in the device [Davis and Hamscher, 1988:297-298]. An advantage with this strategy is that it allows you "to begin reasoning about a system as soon as its structure and behavioral description is available" [Davis and Hamscher, 1988:344].

The above-mentioned strategies have been used in problem-solvers which provide assistance in diagnosis, planning, tutoring, and various other environments. Each of these problem-solvers require interaction with a human to successfully solve the problem at hand.

2.2.1 User Interfaces of Current Expert System Tools. Current expert system tools, such as GoldWorks II, KEE, and Nexpert Object, manufactured by Neuron Data Corp., have excellent user interface capabilities built into them. Graphical user interfaces can be implemented using each system's own programming language or by interfacing with externally written code. Most of these languages have a syntax, resembling that of LISP, giving them the capability to add graphical objects like menus, windows, meters, and graphs. GoldWorks II offers two ways to create a user interface:

one for the LISP programmer, for highly sophisticated interfaces, and another for a menu-based interface [Gold Hill Computers, Inc., 1989]. Other expert systems, such as CLIPS and S.1, must interface with externally written code to display graphical objects, otherwise the user interface is entirely textual.

2.3 Related Military Projects.

There are many expert systems being developed in the military today and most seem to have a common implementation feature: interfacing of the expert system with high-level language code. This code is used to write the user interface of the diagnostic system. The following paragraphs describe five of the many military-related diagnostics systems in development.

Central Integrated Test System (CITS) Expert Parameter Set (CEPS). This system applies expert system techniques to the B-1B recorded flight parameters. Rockwell International, Boeing Military Aircraft Company, and Eaton Corporation are all developing portions of the CEPS which includes both on and off-aircraft equipment. The expert system shell called COPERNICUS is being interfaced with "C" language code to implement the user interface. [McArthur, 1989]

Expert Missile Maintenance Aid (EMMA). This system is using expert system techniques to aid the depot level technician in repairing the GBU-15 missile. It is being developed by Rockwell International Autonetics Sensors and Aircraft Systems Division and uses the M.1 expert system shell and "C" language routines to display graphics. It uses data available from the current test equipment to help diagnose faulty components. One problem experienced by Rockwell was dealing with and getting around the standard user text-based interface built into M.1. [Davis and Huebner, 1989:1, 21, 44]

Integrated Maintenance Information System (IMIS). This system is being developed by the Air Force Human Resources Laboratory (AFHRL) located at Wright-Patterson AFB OH. IMIS displays technical order information and diagnostic aids to the maintenance technician using a portable computer which is carried onto the flight line and connected to the aircraft. The user interface has been a very high concern of IMIS and many studies by AFHRL have been done concerning this. Code written in "C" implements both the diagnostic decision algorithm and the graphics routines. Future versions of IMIS will include integrating SMALLTALK-80 with an expert system. [Gunning, 1989]

Intelligent Tutorial/Diagnostic System for the Space Shuttle Main Engine Controller (SSMEC) Lab. This system, developed by Rockwell International Rocketdyne Division, uses hypermedia (HyperCard) to implement both the user interface and the expert system shell. The system included sophisticated graphics, animation, and sound, and required about 50% of the effort necessary to develop an identical system written in Prolog. HyperTalk, the programming language of HyperCard, was used to implement the backward chaining of the expert system. [O'Reilly and others, 1988]

Power Management and Distribution System (PMAD). This system prototype was designed to aid astronauts aboard a space station to diagnose PMAD system faults. The user interface is implemented using "C" routines, a window manager program which is built into the color workstation, and a low-level graphics package. The expert system tool (KEE) remains transparent to the user. The graphics front end and the expert system communicate via a network protocol. [Hester, 1988]

2.4 Related Tools.

This section outlines three tools which combine expert system techniques with hypermedia techniques. People are beginning to see the advantages in merging the benefits

of hypermedia with those of expert systems as tools similar to these are being announced, exaggeratedly, almost daily. One application is a hypermedia system which has added expert system-like features, and two applications are expert systems which have added hypermedia features. The information provided for the first tool was taken from a review found in a computer magazine, while the information concerning the two other tools was taken from company brochures.

HyperX. This tool, published by Millenium Software, incorporates an expert system shell in a hypermedia environment (HyperCard). It allows both forward and backward chaining and provides an excellent explanation facility. All the available graphics tools of HyperCard can be used to create screens in HyperX. Because programs written in HyperX are interpreted, execution is extremely slow. [Rasmus, 1989:259-260]

Intelligent Diagnostic Expert Assistant (IDEA). This system, distributed by AI Squared, Inc., uses model-based reasoning techniques to solve diagnostics problems. IDEA is PC-based and uses hypermedia techniques to implement its help facility. [AI Squared, Inc., 1989]

KnowledgePro. This PC-based system is distributed by Knowledge Garden Inc., and "combines a high-level, object oriented programming language with hypertext and expert systems technology" [Knowledge Gardens, 1989:2].

In the following chapter, the problem domain of this research, DMINS, is described. Details of the troubleshooting procedures are given, as well as background and test equipment information. Then Chapter 4 documents the development of the prototype which is designed to aid the DMINS technician during a troubleshooting session. As is the case with the three tools above, the prototype developed merges both hypermedia and expert system techniques.

III. Dual Miniature Inertial Navigation System (DMINS)

The purpose of this chapter is to familiarize the reader with the Dual Miniature Inertial Navigation System and its current testing procedures, testing environment, and troubleshooting procedures. The information contained within this chapter is the result of knowledge captured during numerous phone conversations with DMINS technicians and engineers, during several visits to Newark AFB, and while reading assorted types of written documentation, such as technical orders and in-house troubleshooting help sheets.

In Section 3.1, DMINS background information is given. Section 3.2 describes the DMINS testing and troubleshooting procedures. Section 3.3 describes the scope of the troubleshooting process implemented by this project. The final section of this chapter, Section 3.4, highlights several observations which arose from interfacing with the DMINS technicians.

3.1 Background.

The DMINS, manufactured by Rockwell International Corporation, is being used on fast attack submarines, oceanographic survey ships, and aircraft carriers; DMINS has been operational since the mid-70s. The DMINS consists of two Inertial Measuring Units (IMUs), which calculate the ship's attitude, heading, velocity, and position; two Blower Assemblies which cool the IMU; two Electric Mounting Bases, which maintain correct alignment between the IMU and the ship's surface; and the Navigation Control Console (NCC), which provides the output from the IMU to the other computers on the ship. The IMU is considered a Line Replaceable Unit (LRU); therefore, no maintenance, besides removal and replacement, is done at sea.

The Aerospace Guidance and Metrology Center (AGMC), located at Newark AFB OH, is the repair depot for all faulty IMUs. At Newark AFB, the technicians use

automated test equipment (ATE) to troubleshoot faults in the IMU. Because the minimum test time is 70 hours per IMU, only approximately 15 IMUs are repaired each month. For this reason, technicians require many years to gain the knowledge and experience required to troubleshoot in an efficient manner.

The ATE is driven by a test controller, an IBM 1800 computer, and a test program, written in assembly language. The test controller is connected to two NCCs, which, in turn, are connected with two IMUs. One of the NCCs is also connected to an IMU Interconnect Console (IMUIC), which is used during Mode B testing (discussed later). The IMUIC provides the capability for performing malfunction detection and isolation, and functional testing of the IMU. The IMUIC allows technicians to verify test point readings at any point in the testing process. The test program controls the communication between the test controller and the NCC, controls the execution order of the tests, and sets up test scenarios. Figure 3.1 diagrams the current ATE configuration.

3.2 DMINS Testing and Troubleshooting Procedures.

An IMU received from the field will undergo two sets of diagnostics tests: Mode A, which checks system performance; and Mode B, which provides automatic fault isolation testing. During both the Mode A and Mode B tests (discussed below), the IMU is mounted on a stationary pier. When an IMU arrives at AGMC, it will undergo Mode B testing first. After successful Mode B testing, Mode A testing of the IMU begins. If the results of both the Mode A and the Mode B tests indicate that the IMU is functioning properly, a Scorsby Test will be run. A Scorsby test is a navigation performance test in which the IMU is mounted on a moveable platform to simulate the field conditions under which the system fails. If an IMU successfully completes the Scorsby test, it is determined to be operating properly and is returned to the field.¹

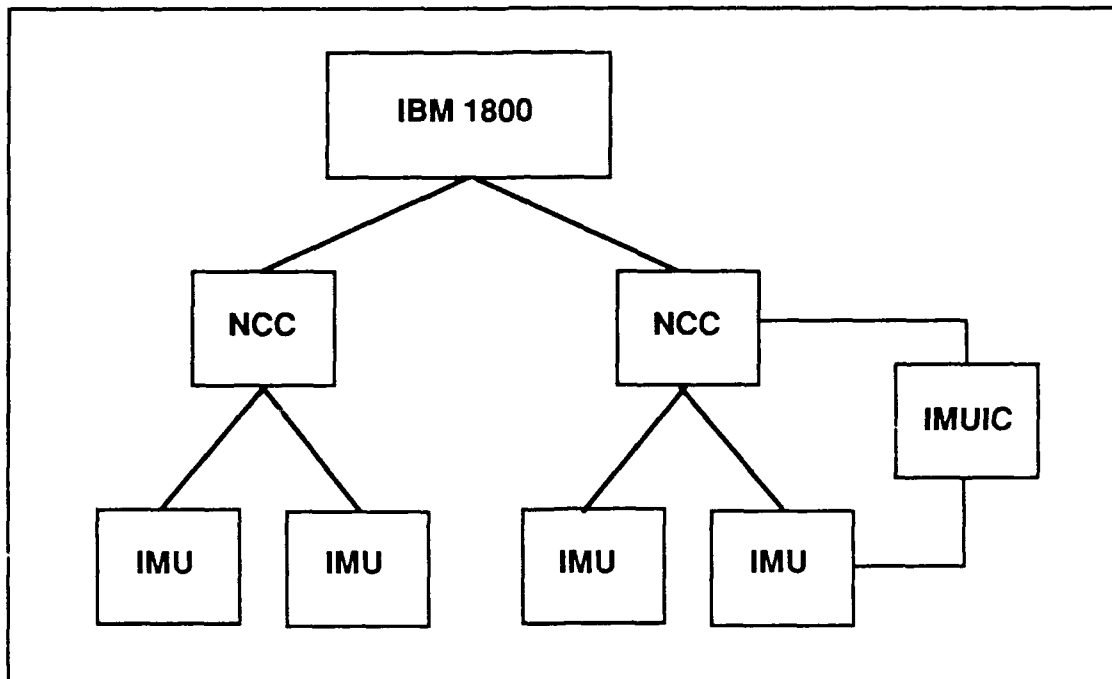


Figure 3.1: DMINS ATE Configuration

Because of the lengthy test times and the relatively small number of IMUs processed by AGMC each month (about 15), a technician's experience is his/her most valuable troubleshooting aid. Using experience, technical manuals, and ATE printouts, a technician determines which of DMINS's 38 Shop Replaceable Units (see Appendix A) should be repaired. The ATE printouts contain three types of information which are used during troubleshooting: signal readings, error messages (see Appendix B), and calibration parameter readings (see Appendix C). Signal readings are used only during Mode B, while calibration parameter readings are used exclusively in Mode A. Error messages occur and are used during both Mode A and Mode B testing.

3.2.1 Mode B Testing. The initial testing of an IMU is performed during Mode B. Mode B consists of 22 tests, each of which can have sub-tests. See Appendix D for a

¹ Sometimes called a retest okay (RETOK).

list and description of the first nine tests. Mode B identifies "hard" or gross failures and tests the functionality of the IMU. Gross failures occur when IMU-generated signals are not within the specification limits. The Mode B tests verify correct operation of IMU functions such as correct direct current power supply, correct alternating current power supplies, correct gyro-hot and gyro-cold alarm conditions, proper built-in test equipment operation, and correct velocity meter acceleration crossover. The 22 tests automatically execute one after the other as long as no failures have occurred. When an IMU successfully completes Mode B testing, Mode A testing will begin. A Mode B run without any of the tests failing will take approximately 4.5 hours.

3.2.1.1 Mode B Troubleshooting Procedures. Output from the ATE consists of a printout for each Mode B test. The printout lists the signal which was tested, the specification limits of that signal, and the status of the signal compared with the specification (i.e., GO or NOGO). Mode B tests are performed with the IMU installed at the test station utilizing the IMUIC. Because the test times of Mode B tests are relatively short compared to Mode A tests, when a Mode B tests fails, the technician will run the same test again. If the test fails again, the technician, using the IMUIC, manually verifies the test point checked during the test to verify that the test equipment is operating properly. If the test point is in fact not within the specification limits, the technician then sorts through various technical manuals in an attempt to identify the failed SRU. Tracking down the origin of the test point signal and the components or modules which could have caused the bad signal is the portion of Mode B troubleshooting in which experience plays an important role. Based on the schematics and the technician's experience, a SRU is determined to be faulty.

During a test, the ATE can also generate error codes. Seventeen of the 62 possible error codes either cause the test to cease execution or require the technician to manually stop the test. The technician will then consult the schematics to determine the SRU which is

causing the error message. Technicians disregard the error codes which do not stop a test from executing, and wait for the entire test to either pass or fail.

3.2.2 Mode A Testing. Mode A testing is accomplished only after an IMU successfully completes Mode B testing. Mode A consists of a sequence of five tests which verify that an IMU's performance meets the specification over long periods of time. Performance failures, like those found during Mode A tests, are typically called "soft" failures. The first three tests calibrate IMU components in preparation for the final two tests which perform navigation tests lasting a minimum of 30 hours. Mode A testing with no failures will take approximately 66 hours. The five Mode A tests are the following:

Shim Calibration (Shim-Cal). An hour-long test which verifies correct performance of the X velocity meter (X-VM) and the Y velocity meter (Y-VM). This test can be run a maximum of three times to allow failed parameters to correct themselves before beginning the next test.

Gyro Calibration (Gyro-Cal). A six hour test which verifies proper XY gyroscope (XY-Gyro) and YZ gyroscope (YZ-Gyro) performance. This test can be run a maximum of three times to allow failed parameters to correct themselves before beginning the next test.

Master Heading. Tests the ability of the IMU to perform a self alignment. This is not a timed test and does not complete running until a technician initiates termination of the test. A typical run time for a successful Master Heading is four hours.

Navigation Alignment (Nav-Align). A 16 hour test that verifies an IMUs ability to navigate.

Navigation Performance (Nav). A 30 hour test of an IMU's navigation performance which can be run a maximum of three times to allow the IMU to try to correct itself. After a successful Nav test, the IMU is delivered back to the field.

3.2.2.1 Mode A Troubleshooting Procedures. Two circumstances initiate troubleshooting an IMU during Mode A testing: calibration parameter readings and error codes. Calibration parameters electromagnetically adjust some of the components of the IMU to improve its performance and keep it operating within the specification. The values of these parameters are checked to see if they are within certain tolerance limits.

The behavior of calibration parameters during and at the completion of each test also provides valuable troubleshooting information. This is true whether the IMU passes or fails the test. Experienced technicians not only care about the value of the parameters, but also about how those parameters vary from one test to the next. Erratic or drastic changes in a parameter from one test to the next, even if the value is within the specification, can be an indicator of a faulty SRU. The calibration parameters of a properly functioning IMU "fine tune" or continually improve themselves from one test to the next.

Troubleshooting error messages during Mode A testing are identical to those during Mode B except that calibration parameter information is also available. Both error message and calibration parameter troubleshooting techniques are contained in the prototype, which is discussed in Chapter 4.

3.3 Scope of the Problem Domain.

DMINS technicians troubleshoot IMUs during both Mode A and Mode B testing, and several sources of information are available to help them. The expert system developed by Capt Skinner as part of his thesis [Skinner, 1988] incorporated only one type of troubleshooting aid available to the DMINS technician: error messages. It also

encompassed only Mode A testing. This thesis effort builds onto Capt Skinner's project by adding calibration parameter troubleshooting to Mode A. Also, a limited Mode B troubleshooting was added. The Mode B portion does not include error message troubleshooting and only includes the first nine Mode B tests. Mode B knowledge was coded entirely in hypermedia while the Mode A calibration parameter knowledge was coded in hypermedia and in the expert system. The intended result is a prototype which mirrors the actual testing and troubleshooting process used by experienced DMINS technicians.

3.4 Observations.

The potential benefits of using an expert system at AGMC became evident early in the knowledge engineering sessions. I talked to two of AGMC's most experienced technicians. These technicians, who now both occupy supervisory positions, kept saying that they wished the operational technicians would do more of the type of troubleshooting "they would do." According to these experienced technicians, most technicians do not use the calibration parameter data as much as they should. If a test passes, even though the calibration parameters provide evidence of a possible problem, these less experienced technicians would continue testing. Adopting the troubleshooting techniques of the more experienced technicians could benefit AGMC tremendously. Faulty SRUs could be found earlier in the testing process, and more IMUs could be processed each month. I have tried to represent "their" way of troubleshooting in the prototype developed during this project. Some of this knowledge is represented in a general form throughout this thesis; a more detailed representation can be found in the hypermedia and expert system code (see Appendix E). The next chapter details the development of the prototype.

IV. Prototype Development

In this chapter, the development of the DMINS prototype is documented and discussed. This prototype is a logical "next step" to the prototype developed during Capt Skinner's research. It encompasses significantly more of the DMINS diagnosis process as both calibration parameter diagnosis and Mode B troubleshooting are added (see Chapter 3), while at the same time, providing a simpler and more informative user interface. This chapter is organized in the same sequence as is the testing of a DMINS IMU (see Chapter 3). Section 4.1 describes the methodology of developing the Mode B portion of the prototype, and Section 4.2 outlines the Mode A part. Both the Mode B and Mode A sections are arranged into subsections corresponding to the methodology outlined in Section 1.4. Section 4.3 describes the integration of the Mode B development with the Mode A development producing the prototype. The next section, Section 4.4, informs the reader of changes made to some of the hypermedia system's basic capabilities. Lastly, Section 4.5 concentrates on problems which occurred due to the software tools used.

4.1 Development of Mode B.

The Mode B part of the prototype was developed using three of the phases described in Section 1.4: Knowledge Acquisition, Design and Coding, and User Evaluation. Because of circumstances discussed in Section 4.1.2 below, the Hypermedia and Expert System Integration Phase was not used during Mode B development. The mechanics involved with each of these phases are discussed presently.

4.1.1 Knowledge Acquisition Phase. Because Capt Skinner's project did not include Mode B testing, the only source of knowledge was the DMINS technicians and engineers. The technicians and engineers I talked with were very interested in

incorporating Mode B because the majority of their troubleshooting is conducting during this phase of testing. Numerous interviews were conducted with Jim Neri, one of the most experienced DMINS technicians, and Capt Steve Rasmussen, an engineer, concerning Mode B testing and troubleshooting procedures. Jim Neri was also involved with Capt Skinner's research and was eager to get started.

The interviews consisted of performing a "walk through" of the first nine Mode B tests and their associated subtests, documenting the specific troubleshooting procedures for each test. When I arrived at the start of the first interview, Jim Neri had already started a chart designed for organizing this troubleshooting information. Mr Neri is in the process of completing this chart to include all 22 Mode B tests. Table 4.1 is an excerpt from that chart which includes test B6.0 information. Each test verifies that one test point signal is within the specification limits. The "Action" column of the chart describes the procedures to follow if the signal is not within those limits. Refer to Section 3.2.1.1 for the Mode B troubleshooting procedures which are common to each test.

Table 4.1: Excerpt From Mode B Troubleshooting Chart

Test	Title	Test Pt	Specification	Action
B6.1	XY Gyro Speed Control Bite	18	3.5 to 6 VDC	R&R AR1, then A1.
B6.2	YZ Gyro Speed Control Bite	19	3.5 to 6 VDC	R&R AR2, then A2.
B6.3	400 Hz Bite	93	2 to 6 VDC	Place PS7 on extender. Observe 17.8-32.2 VRMS. If GO R&R PS7. If NOGO R&R PS8.
B6.4	Azimuth Overrate Bite	94	5 to 6.5 VDC	R&R A7.
B6.5	Servo Disable Bite	21	3.5 to 4.5 VDC	R&R AR7, then A7 or PS7.
B6.6	Free Run Bite	17	5 to 7 VDC	R&R PS7.

4.1.2 Hypermedia and Expert System Integration Phase. As the Mode B chart referenced in the previous section was being created, we realized that the knowledge had a very regular structure. The "Action" column of the chart usually involved only one step: recommending that a component or module be removed and replaced. Very few tests require multiple actions (i.e., checking other test points) before a faulty component or module can be diagnosed. Because the Mode B troubleshooting knowledge is procedural in nature, it requires little reasoning about possible solutions, and because the overall size of the Mode B portion of the prototype was relatively small, a decision was made to implement Mode B troubleshooting procedures using only hypermedia.

Although an expert system could have been used to encode this knowledge, it would have been trivial since each rule would have contained only one condition and only one action (see Section 2.2). Also, each of the conditions would be unique to one rule. It would be like using an expert system to implement each and every path of a decision tree; hence, defeating the nonprocedural execution feature of an expert system. On the other hand, hypermedia can implement decision trees quite easily, using links between frames as paths between decision points. Consequently, the expert system is not used when Mode B troubleshooting is being performed using the prototype.

4.1.3 Design and Coding Phase. The chart mentioned in Section 4.1.1 and the general troubleshooting instructions mentioned in Section 3.2.1.1 were used to create hypermedia frames for tests B1.0 through B9.0. Using the information found in these sections, the troubleshooting procedures for test B.6.1 can be represented as shown in Figure 4.1. Everything contained in a single shaded area is included in one hypermedia frame. In this case, six frames were created for test B6.1. Frames contain queries to the user for more information. Additional information was added to each frame to help the user answer the query. A user responds to a query by clicking a mouse over the button which identifies the correct status of the IMU characteristic (i.e., signal reading, velocity

direction, etc.) in question. Clicking a button would cause the frame which is linked to that button to be displayed next. This new frame would either contain another query or identify the faulty component or module. Appendix F contains the frames a technician would see when troubleshooting Mode B test B6.1 and arriving at the conclusion that the AR1 module should be replaced.

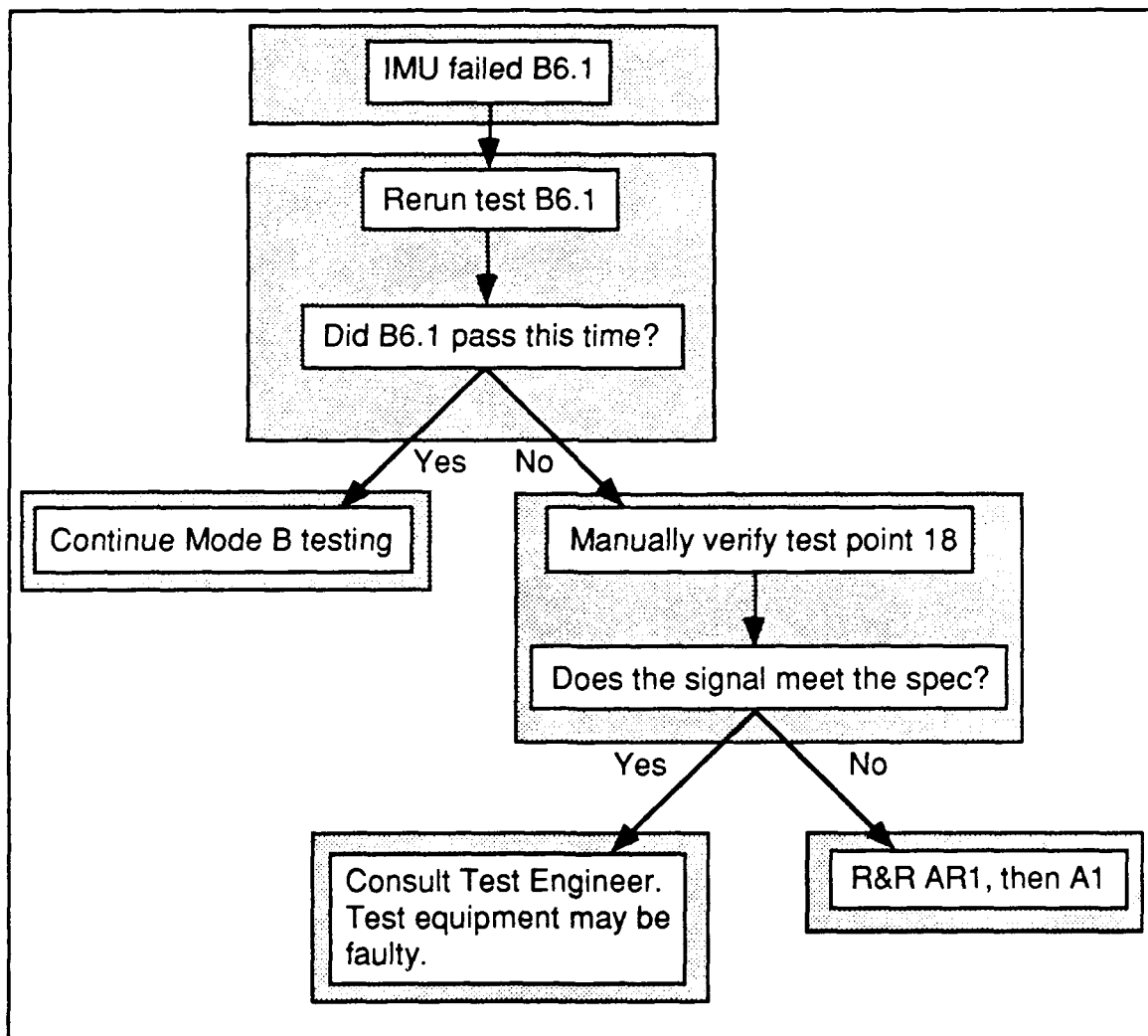


Figure 4.1: Mode B Test B6.1 Troubleshooting Procedures and Corresponding Hypermedia Frames (Shaded Areas)

Figure 4.1 would be identical for each Mode B test except that different test points will be checked and, if the test point signal does not meet the specification, different procedures will be carried out. As stated previously, these procedures usually recommend that a module or component should be replaced (as is the case with B6.1). Not as often, these procedures will have other test points checked before a faulty module is determined (see test B6.3 in Table 4.1).

4.1.4 User Evaluation Phase. A DMINS technician used the prototype during several simulated Mode B troubleshooting sessions. He found the system extremely easy to use and thought it would be an excellent training aid. He particularly liked that each test had a panel diagram, a technical orders page reference, and a specification reading, all of which were displayed on one hypermedia frame. He recommended that the prototype be updated to include the remaining Mode B tests.

4.1.5 Observations and Conclusions. Much of the total troubleshooting time of a DMINS technician is exhausted during Mode B. This time is predominantly spent looking at schematics, tracing signals to determine which module or component might have caused the bad signal reading. The chart developed during this thesis organizes the results of the time spent looking at the schematics and various other documentation into a single document. The chart alone, once completed, will be an invaluable troubleshooting tool and training aid for the DMINS technicians. The Mode B portion of the prototype encoded this chart into a hypermedia system. The advantages of hypermedia over the paper chart include facilitating the ability to contain graphics, the ability to retrieve information quickly, and much improved human factors. While a paper chart usually contains information extraneous to the current problem, such as information concerning other tests, a hypermedia frame can be designed to contain information specific only to the current

situation. Also, a hypermedia system is more easily modified as troubleshooting knowledge changes.

As previously mentioned, a hypermedia-only implementation was selected because Mode B troubleshooting knowledge does not lend itself to an expert system implementation. While this simplified the prototype development immensely, potential problems could surface later. Maintenance of large hypermedia networks can be extremely difficult [Conklin, 1987:39]. A hypermedia network is modified by traversing the network, adding and deleting frames, changing frames, and creating, changing, and deleting links along the way. Because links are modified, it is possible to continually generate unused frames (i.e., frames which are not linked to other frames). Because changes to the Mode B troubleshooting procedures will not occur very often and that the network is relatively small (less than 300 frames), it is doubtful this will happen in the case of the prototype. Another problem with hypermedia is that the logic embedded within the network can not be printed out like program code can be; therefore, the network must be traversed in order to understand its logic.

The Mode B portion of the prototype only aided a technician when a test failed. As mentioned in Section 3.2.1.1, error messages are also used for troubleshooting. Enhancements to this prototype should include error message troubleshooting and the remaining Mode B tests (B10 through B22). Mode A development did include error message troubleshooting and is discussed next.

4.2 Development of Mode A.

The Mode A part of the prototype was developed using all of the phases described in Section 1.4: Knowledge Acquisition, Hypermedia and Expert System Integration, Design and Coding, and User Evaluation. An important feature is that both error message and calibration parameter troubleshooting were incorporated. This feature allows the prototype

to resemble actual Mode A troubleshooting procedures. In contrast to the Mode B knowledge, the calibration parameter knowledge lend itself to an expert system representation. Therefore, development included both the hypermedia system and the expert system. Communication between these two systems was accomplished using shared data files.

The following sections describe the phases of development in detail. Section 4.2.1 outlines the Knowledge Acquisition Phase and the sources of information. Emphasis during this phase was placed on acquiring calibration parameter troubleshooting information. Section 4.2.2 recounts the Hypermedia and Expert System Integration Phase and details the interprocess communication (IPC) features of the prototype. The Design and Coding Phase of the Mode A development is outlined in Section 4.2.3. The next two sections, Sections 4.2.4 and 4.2.5, document the user's evaluation of the prototype and discuss observations and benefits of the Mode A part of the prototype.

4.2.1 Knowledge Acquisition Phase. The purpose of this phase was to collect enough accurate information to begin design and coding. Two main sources of knowledge existed for Mode A troubleshooting procedures: DMINS technicians and Capt Skinner's expert system. Capt Skinner's expert system was used as the main source for gathering knowledge concerning troubleshooting error messages. Capt Skinner's project, written using the S.1 expert system shell, recommended appropriate troubleshooting procedures to diagnose faults in an IMU when one of the 62 error messages occurred. It was learned during one of my knowledge engineering sessions that only 17 of the 62 error messages actually require troubleshooting; the remainder can be ignored because they do not provide enough information to make a decision. This project assumed (i.e., did not verify with the DMINS technicians) that Capt Skinner's production rules concerning those 17 error messages were correct. The majority of time interviewing technicians was spent discussing calibration parameter troubleshooting.

During the interviews with the technicians, Mode A ATE printouts were reviewed to determine how and when the calibration parameters were used, or should have been used by the technician during troubleshooting. In contrast to the Mode B knowledge and the error message knowledge, the calibration parameter knowledge was much more sophisticated (i.e., more than one characteristic of the IMU being tested contributed to reasoning about the problem). Troubleshooting using the calibration parameters is also more individualistic and depends on personal experience and training. When troubleshooting using the calibration parameters, not only is the value of each parameter observed, but also the behavior of each parameter from one test to the next is observed. Since parameter behavior does not cause a test to fail (the parameter's value does this), behavior can even be ignored by the technician. Noticing a parameter's abnormal behavior as early in testing as possible could avoid that same parameter's causing a test failure, many hours later. Also, knowing how close a parameter is to the specification limit is important. For example, if a parameter has an excellent value on one test run, and is extremely close to the specification limit on the next test run, this would indicate a problem with the IMU even though the test passed. The CLIPS code listed in Appendix E documents this type of reasoning in more detail.

Since technicians use both error messages and calibration parameters while troubleshooting during Mode A, information on how and when each is used was also obtained. Because error messages *may* occur at any time during a test and calibration parameter readings *are* available during a test, the technician spends most of the time observing parameter readings and behavior. If one of the 17 error messages which requires troubleshooting occurs (see Appendix B), troubleshooting using the error message begins. Otherwise, troubleshooting is accomplished using the calibration parameter information and test results.

Many notes were taken during my interviews and phone conversations with the technicians. These notes were converted into diagrams represented the troubleshooting procedures of the five Mode A tests. These diagrams were then validated by two DMINS technicians before the design and coding of the prototype began.

4.2.2 Hypermedia and Expert System Integration Phase. This phase consisted of implementing the interprocess communication (IPC) between the hypermedia system and the expert system. The IPC between KMS, the hypermedia system, and CLIPS, the expert system, was accomplished using four files.¹ Two of the files are data files and contain the information that one process wants to relay to the other. The other two files are used as semaphores [Silberschatz and Peterson, 1988:95-101], ensuring that only one process is accessing (reading or writing) a data file at a time. Table 4.2 lists each of the four files and their function.

Table 4.2: Interprocess Communication Files and Their Function

File Name	Function
kmsinfo	Written by KMS, contains user response to queries.
clipsinfo	Written by CLIPS, contains frame name to display next.
ok_kms	Written by CLIPS, a semaphore signifying that the clipsinfo file is ready to be read by KMS.
ok_clips	Written by KMS, a semaphore signifying that the kmsinfo file is ready to be read by CLIPS.

The IPC implementation can be described textually by the following two characteristics and graphically by Figure 4.2:

¹ While there are more efficient forms of IPC, KMS does not support them.

CHARACTERISTIC 1: Only one of the two systems is in control at a time. This system reads in the applicable data file and begins executing. Control ends when information is written to the applicable data file. When CLIPS is in control, it is executing rules based on the information just read and the information already in its working memory. When KMS is in control, it is displaying the frame whose name was just read from the data file, and is waiting for a user response to the query displayed on the frame.

CHARACTERISTIC 2: The system not in control checks the applicable semaphore file periodically to see if the contents of the file indicate that the other system has relinquished control. When this system is not checking the file, a UNIX "sleep 1" command is executed to temporarily suspend this system's execution.

Code internal to each of these systems was written to allow access to the above mentioned files. The following two sections detail what was done to KMS and to CLIPS allowing them to communicate with each other.

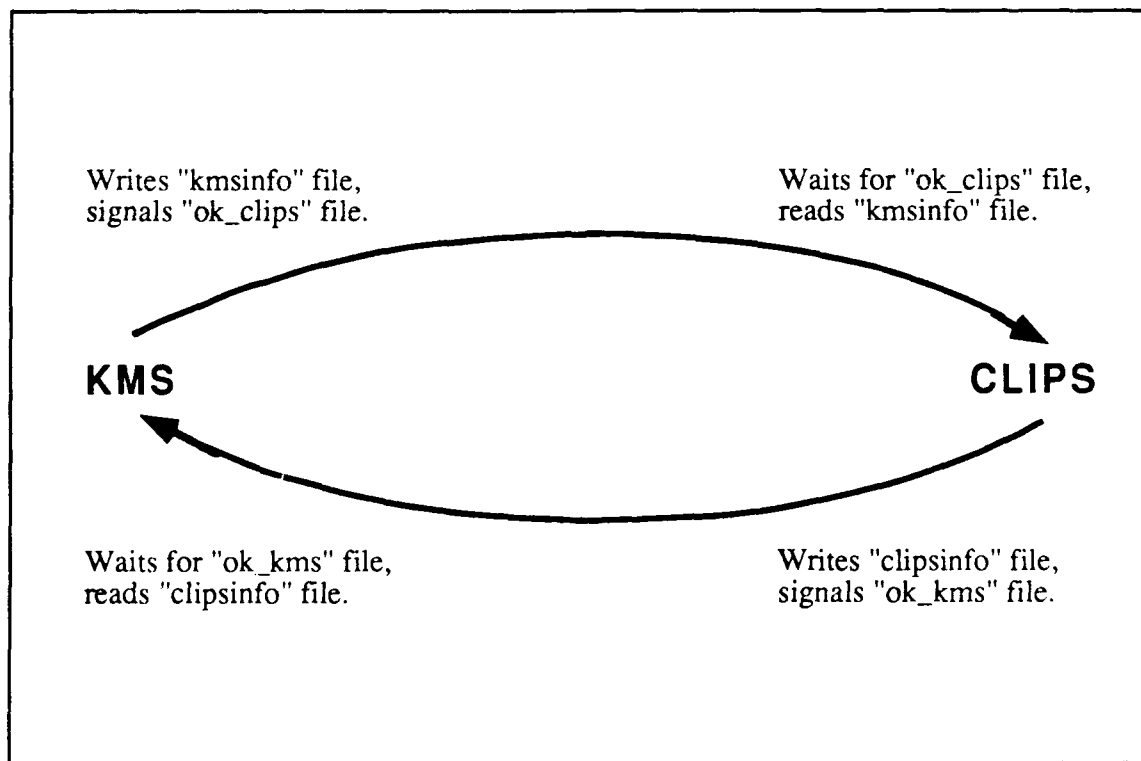
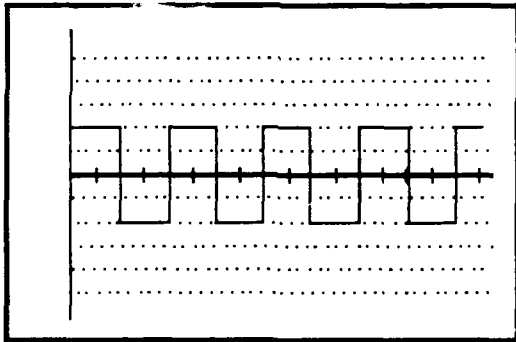


Figure 4.2: Interprocess Communication Scenario

4.2.2.1 KMS Provisions for Interprocess Communication. KMS relays the user's responses to queries to the expert system. These responses are formulated to correspond to conditions embedded in rules in the expert system. When a user clicks on a button to inform the prototype of current characteristics of the IMU, the applicable information is written to the "kmsinfo" file. For instance, in Figure 4.3, the technician would either click on the "GOOD" or "BAD" button to indicate the status of the resolver signal at the NCC.

DIAGNOSIS STAGE: MODE A ERROR CODES

The signal should look like this:



Check the resolver signal at the NCC.

The signal is: ☐ **GOOD** ☐ **BAD**

Figure 4.3: Example of KMS Frame with Two Buttons

In KMS, buttons have properties which can be modified if desired. Figure 4.4 shows the properties of the "BAD" button. Inserting a frame name in the "link" slot would have caused the named frame to be displayed when the button is activated. In the case of

the prototype, the expert system decides which frame to display next; therefore, most "link" slots are empty. The "Action" slots of a button allow the hypermedia programmer to use KMS's built-in programming language to perform various operations when a button is activated. As is the case for this and most of the buttons in the prototype, information is written to the "kmsinfo" file when the button is clicked. The first three "Action" statements accomplish this. The final "Action" statement executes the instructions listed on frame "dflorian8." These instructions perform the IPC *CHARACTERISTIC 2* mentioned in Section 4.2.2. When information from CLIPS is available, the instructions on this frame also read the data from the "clipsinfo" file and display the appropriate frame or frames.

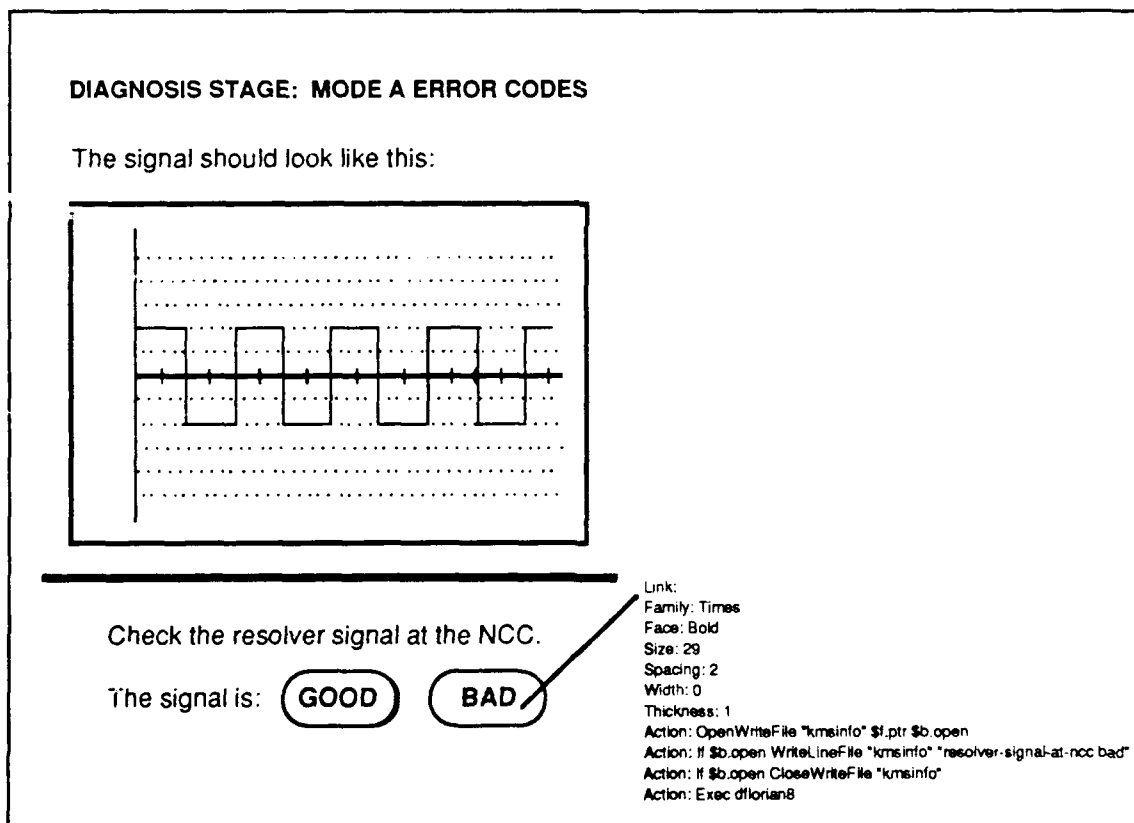


Figure 4.4: Example of a KMS Button's Properties

4.2.2.2 CLIPS Provisions for Interprocess Communication. The expert system chosen for this project, CLIPS, has excellent external interface features. These features allow it to be integrated with external functions and applications, and allow it to be embedded within other programs. Consequently, CLIPS's main procedure file was modified to include "C" language code designed to simplify implementing the IPC. Recompiling CLIPS source code to include the new main procedure file created an executable program specifically tailored for this prototype. The main procedure file was modified to perform the following functions: automatic loading of the rules, reading and checking the contents of the "ok_clips" and "kmsinfo" files, asserting the information contained in the "kmsinfo" file into CLIPS's working memory, and executing the UNIX "sleep 1" command mentioned in Section 4.2.2. Appendix G is a listing of the modified main procedure file. Modifying the main CLIPS procedure eliminated the need to perform all of the above functions in rules. Three additional rules were added to write the information CLIPS wanted to relay to KMS into the "clipsinfo" file. Another rule was added to allow CLIPS to write to the "ok_kms" file to signify that the information contained in the "clipsinfo" file is ready for KMS to process. Table 4.3 lists the four rules and their function, and Appendix E contains the CLIPS code for these rules.

4.2.3 Design and Coding Phase. During this phase, the information collected during the Knowledge Acquisition Phase was used to create production rules (see Appendix E) and hypermedia frames. First, Capt Skinner's expert system was converted from S.1 code into CLIPS code. Figure 4.5 shows the same rule implemented using S.1 code and using CLIPS code. Second, the calibration parameter knowledge and the user interface were encoded using CLIPS. The text-based user interface provided by CLIPS was used until the logic of the CLIPS code was determined to be correct. By using "printout" statements, rules could display textual information to the computer terminal.

Third, and after the logic of the CLIPS code was verified that it accurately represented the troubleshooting knowledge, KMS frames were created for each rule which contained a "printout" statement. These frames were designed to contain not only the information from the "printout" statements, but also panel diagrams, wave diagrams, and various other information to aid the user in answering the query displayed on the frame (see Figure 4.3). Next, the "printout" statements were "commented out" and another statement, which identified the appropriate hypermedia frame to display, was added in its place. This new statement would cause one of the rules listed in Table 4.3 to execute, which in turn would cause the appropriate hypermedia frame to be displayed.

Table 4.3: Four CLIPS Interprocess Communication Rules and Associated Functions

Rule Name	Function
write-one-frame-name	Writes one frame name to "clipsinfo" file. KMS will display this frame in KMS full screen mode.
write-two-frame-names	Writes two frame names to "clipsinfo" file. KMS will display the first frame in half screen mode on the left, and the second frame in half screen mode on the right.
write-one-frame-name-change-value	Writes one frame name, one parameter, and the value of the parameter to "clipsinfo" file. KMS will display the frame in full screen mode and change the parameter value on the screen to the new value.
ok-for-kms-to-read	Writes a one to "ok_kms" file signifying the file "clipsinfo" is ready to be read by KMS.

Finally, integration of the error message rules with the calibration parameter rules was accomplished by adding an additional button to the calibration parameter troubleshooting frames and an additional statement to each of the calibration parameter rules. Activating this button would cause a fact to be asserted into the expert system's working memory

which would not allow the calibration parameter rules to execute. Once all the applicable error message rules fired, and if a faulty component had not been determined, calibration parameter troubleshooting would continue at the point where it was interrupted by the error message.

S.1 Implementation

```
DEFINE RULE rule016
::APPLIED.TO    I:IMU
::PREMISE      check.resolver.signal.at.ncc[I] and
                not resolver.output.at.ncc.good[I]
::CONCLUSION   check.resolver.signal.at.imu[I]
END.DEFINE
```

CLIPS Implementation

```
(defrule rule016
  (check resolver-signal-at-ncc)
  (resolver-signal-output-at-ncc bad)
=>
  (assert (check resolver-signal-at-imu)) )
```

Figure 4.5: Same Rule Written in S.1 and CLIPS

4.2.4 User Evaluation Phase. A DMINS technician used the prototype during several simulated Mode A troubleshooting sessions. Again, the technician was impressed with the system's ease of use. Because this part of the prototype is not driven by failed tests, he recommended that an opening frame be included to discuss the concept of operations of the Mode A part of the prototype. He particularly liked the sample graphs and the ability to perform error message troubleshooting and calibration troubleshooting at the same time.

4.2.5 Observations and Conclusions. Development of the Mode A portion of the prototype included using hypermedia as the user interface to an expert system.

Hypermedia provided a means for quickly developing an efficient, user-friendly, and informative user interface. However, the provisions made to allow the hypermedia system and expert system to share information were cumbersome at best. Because development of the rules and the hypermedia frames were completely separate efforts, too much effort was expended keeping track of which frame corresponded to which rule. For this reason, debugging the expert system was hard to do unless the appropriate frame was also being displayed.

A significant enhancement to Capt Skinner's expert system's capability is the addition of calibration parameter troubleshooting. Most of a technician's time during Mode A is spent observing calibration parameter values and behavior. The way these parameters are used during troubleshooting varies from technician to technician. Because the knowledge used during development came from two of the most experienced technicians, the prototype uses the parameters in much the same way as those technicians. For this reason, the prototype should provide a more efficient and standard way of troubleshooting.

4.3 Integration of Mode B and Mode A. Because the two modes of testing are run in sequence, there is very little integration between them. The Mode B and Mode A parts of the prototype were developed as separate stand-alone packages. Provisions were made to integrate them in the prototype only to allow the user to select which mode of testing to perform. Figure 4.6 is the KMS frame which allows this selection to be made. When the "Mode B Diagnosis" button is activated, the user begins traversing the Mode B frames of the hypermedia network. If the "Mode A Diagnosis" button is activated, the modified CLIPS program starts executing in the background, transparently to the user. From this point on, the information passed between the two systems determine the frames to displayed.

Diagnosis System
for
Inertial Navigation System
AN/WSN-1(V)2
Inertial Measurement Unit

• Mode B Diagnosis	• Mode A Diagnosis
	Tutorial
T.O. Reference	• Exit

Figure 4.6: Prototype Main Menu Frame

4.4 Constraining KMS's Capabilities. Three of KMS's standard features were disabled in the prototype. Without modification, these features would permit navigation to any frame in the network, allow modification of the network, and allow keyboard inputs. Disabling the first two features was required because of reasons discussed below. Not allowing keyboard inputs was accomplished to eliminate writing code to validate user input and to streamline the user interface. All user inputs are accepted by using a mouse to activate buttons on a frame.

KMS allows a user to perform many navigation operations. For instance, a user can perform operations to go back to previously displayed frames, to go to the next frame in the network, to go to a named frame, and to go to the first frame in the network. Controlling the navigation of the user was needed to prevent the user from getting lost in the network, and possibly activating a button which would relay the incorrect information to the expert system, and in turn, relay incorrect information back to the user.

KMS also allows users to modify the network by creating frames and deleting frames which were created by that user. Since the expert system would not recognize these new frames, the new frames in and of themselves would not be a problem. But, as mentioned above, the possibility exists that a user could become lost in the network while busy creating and deleting frames.

4.5 *Difficulty During Development.* One difficulty worth mentioning arose during development. It surfaced during conversion of the text-based user interface into hypermedia (see Section 4.2.2).

During development and prior to implementing the user interface in hypermedia, CLIPS's text-based user interface was used. If information was needed from the user, a "read" statement was included in the rule containing the query. A "read" statement causes CLIPS to suspend execution of all the other rules which are waiting to be executed, until the user responds to the query. When the rules were changed to implement the hypermedia interface, all the "read" statements were taken out of the rules. Hence, when CLIPS would execute a rule which used to contain a "read" statement, it would not suspend CLIPS execution and any activated rules would execute. When all of the rules had executed, CLIPS terminated. This caused an IPC problem because, after the user response was accepted by KMS, KMS would be waiting for information from CLIPS. Since CLIPS had already terminated, KMS would be in a deadlock state. To solve this problem, the rules which wrote information to the "clipsinfo" file (see Table 4.3) were modified to call the "kms_file_ready" function (see Appendix G) which would suspend execution of CLIPS until the information from KMS was asserted into CLIPS's working memory.

The final chapter summarizes this research effort and provides recommendations for further research. Also, conclusions concerning integrating hypermedia technology and expert system technology will be addressed.

V. Conclusions and Recommendations

This chapter identifies conclusions derived from this research and submits recommendations to improve developing this type of system in the future. Of special interest is the importance of the prototype to the Dual Miniature Inertial Navigation Systems (DMINS) organization located at Newark AFB, Ohio, and the enhancements which could be made to the prototype to improve their troubleshooting practices.

5.1 Conclusions.

The prototype developed as a part of this research implemented an expert system's user interface using a hypermedia system. Hypermedia facilitated construction of a user-friendly, informative user interface by allowing the dynamic creation and editing of screens without the need to learn a sophisticated programming language.

While the user interface was considerably easier to create using the hypermedia system compared to using an expert system implementation, problems arose because the user interface and the expert system were separately developed. It became the programmer's responsibility to ensure that relevant information, in the proper form, was made known to both the hypermedia system and the expert system. The programmer had to keep track of which rule and which hypermedia frame were related, and code this into the expert system. Since the expert system contained references to hypermedia frames and not the actual queries to the user, it was hard to recognize the logic embedded in the code. For this reason, maintenance of the expert system could be difficult.

A maintenance environment was used as the problem domain for this system. Specifically, the prototype was designed to help depot-level technicians troubleshoot the DMINS Inertial Measurement Unit (IMU). DMINS testing and troubleshooting knowledge was used to develop the prototype which encompasses a significant portion of the DMINS

troubleshooting procedures and practices. Mode A and Mode B testing, and both error message and calibration parameter troubleshooting, are included. The prototype has the potential to decrease the average test time of an IMU for the following reasons:

- a. Two of the more experienced DMINS technicians were used as primary knowledge sources. The prototype provides standardized troubleshooting procedures based on their vast knowledge and many years of experience.
- b. Because calibration parameter information can be ignored by the technician, the prototype will ensure that this information is used, possibly resulting in the detection of problems earlier in testing.

5.2 Recommendations.

This section identifies several enhancements to the prototype as they would apply to DMINS troubleshooting. Also, ways in which hypermedia and expert system technology could better be integrated are discussed.

5.2.1 DMINS. Several additional features could be incorporated into the prototype to improve the overall performance and capability of the system. With these enhancements, the prototype will become an even more powerful tool which can be used during all phases of testing. The following is a list of features, from the easiest to implement to the hardest, to be considered for future development:

- a. Include Mode B tests 10 through 22, for completeness.
- b. Include error message troubleshooting during Mode B, for completeness.
- c. Obtain multiple technicians to help in development, to prevent the system from being biased towards one way of troubleshooting.

- d. Ensure close involvement of DMINS technicians throughout development, for accuracy and user satisfaction.
- e. Compile the CLIPS code, to produce a run-time program and decrease the response time of the system.
- f. Include an explanation facility, for training purposes.
- g. Add the ability to save the status of a test and to restart the same test at a later time, to mirror actual troubleshooting.
- h. Allow a user to retract responses, to correct inputs and to allow the system to be used for training purposes.
- i. Study previous navigation graphs, which are plotted during the Navigation Performance test, and the relationship between these graphs and the mean time between failure (MTBF) of the associated IMU. For example, a curve that is within the specification limits is shipped even though the IMU looks as if it is drifting. There might be a relationship between this type of graph and an IMU with a small MTBF.
- j. Finally, obtain calibration parameter readings, error messages, and test failure data directly from the ATE, to avoid unnecessary queries to the user.

5.2.2 Hypermedia System and Expert System Integration. To relieve the programmer from having to code the means of communication between the hypermedia system and the expert system, applications having features of both types of systems should be developed. This would eliminate the overhead (i.e., shared files, busy waiting, programmer bookkeeping, etc.) experienced during development of this prototype.

Expert systems need to have the capability to dynamically create screens and allow the screens to be edited as the rules are being created. Not only will this facilitate creating the user interface, it will also allow the user to become more involved with the design and implementation of the system. When creating rules, the expert system could automatically generate a hypermedia frame containing buttons relating to the valid responses to queries. These buttons, when activated, would relay the information in the correct form to the inference engine of the expert system. The automatically-generated hypermedia frame would be available to edit to ensure that an adequate and customizable description of the query is given.

5.3 *Summary.*

This research demonstrated how a hypermedia system could be used to improve the creation and editing of an expert system's user interface. It also demonstrated the need for these two exciting technologies to merge, each borrowing features from the other. The differences between hypermedia systems and expert systems are diminishing as the capabilities of each are improving. Systems incorporating the strengths of both hypermedia systems and expert systems are emerging rapidly and will continue to evolve into more effective development environments.

Appendix A: DMINS Shop Replaceable Units (SRUs)

This appendix identifies the 38 DMINS SRUs and for each identifies the replacement part number.

<u>ID#</u>	<u>SRU Name</u>
3A1	Bandpass Filter and Shift Register
3A2	Bandpass Filter and Shift Register
3A3	Precision Torquing Driver (X)
3A4	Precision Torquing Driver (Y)
3A5	Precision Torquing Driver (Z)
3A7	Platform Electronic Switch
3A8	Shorting Plug
3A9	Precision Current Network
3A10	Stable Platform
3A10A3	Displacement Gyroscope (X-Y)
3A10A4	Displacement Gyroscope (Y-Z)
3A10A7	Velocity Meter (X)
3A10A8	Velocity Meter (Y)
3A10AR1	Resolver Buffer Amplifier
3A10AR5	Gyro Buffer Amplifier (X-Y)
3A10AR6	Gyro Buffer Amplifier (Y-Z)
3PS1	640 Hz Power Supply (X-Y)
3PS2	640 Hz Power Supply (Y-Z)
3PS3	Power Cube
3PS7	400 Hz Power Supply No. 2
3PS8	400 Hz Power Supply No. 1
3PS9	Triangle Generator and Case Rotation Power Supply
3PS10	4.8 KHz Power Supply
3PS11	Frequency Standard
3AR1	D.C. Amplifier (X-Y)
3AR2	D.C. Amplifier (Y-Z)
3AR3	Synchro Signal Buffer Amplifier
3AR4	Gyro Cage Amplifier
3AR5	Thermoelectric Signal Amplifier
3AR6	Gyro Temperature Controller
3AR7	Gimbal Cage Amplifier
3AR8	Platform Signal Amplifier

Appendix A (cont.)

<u>ID#</u>	<u>SRU Name</u>
3AR9	Platform Electronic Control Amplifier (Roll)
3AR10	Platform Electronic Control Amplifier (Pitch)
3AR11	Platform Electronic Control Amplifier (Azimuth)
3AR12	Gimbal Rate Electronic Control Amplifier (Roll)
3AR13	Gimbal Rate Electronic Control Amplifier (Pitch)
3AR14	Gimbal Rate Electronic Control Amplifier (Azimuth)

Appendix B: DMINS Error Messages

This appendix identifies the 62 DMINS error messages and their accompanying message number. Both the error message and the message number are output from the ATE.

Message No.	Error Message	Message No.	Error Message
0101	Automatic Shutdown *	0704	No Input Roll *
0102	IMU O'Load	0801	Plat Stab Abort *
0103	IMU O'Temp *	0901	XVM Precounter Fault
0104	Pwr Interrupt	0902	YVM Precounter Fault
0105	DCC O'Load	0903	Both Precounter Failure
0106	DCC O'Temp	0904	VM Bite Failure
0201	I/C Fault	1001	X Gyro Torque Fault
0202	Comp Tie-In Sw On	1002	Y Gyro Torque Fault
0203	I/C Fault Inhb Enab	1003	Z Gyro Torque Fault
0204	Seq Cnt No Compare	1101	Velocity Unreasonable *
0205	I/C Data Loop Fault	1102	System Not Properly Caged
0206	I/C Fault Cont	1201	Gyro Hot *
0301	In Parity Test Inhb Enab	1202	Gyro Cold
0302	Out Word Par Inhb Enab	1203	Gyro Temp Normal
0303	Input Parity Fault	1301	Mux Decoder DL Fault
0401	Output Word Parity Fault	1302	Cage XY DL Fault
0402	Output Word Parity Cont	1303	Cage YZ DL Fault
0501	IMU Major *	1304	Gyro Start DL Fault
0502	Excess Angle *	1305	Gyro Run DL Fault
0503	Servo Disable *	1306	UYK Good DL Fault
0504	Major Reset Fault *	1307	Input Parity DL Fault
0601	XY Speed Control	1308	Input Parity No DL Fault
0602	YZ Speed Control	1309	Output Word Parity DL Fault
0603	Z Stab *	1401	VT Greater Than 2 Knots *
0604	System In Free Run	1403	VT-VR Greater Than 3 Knots *
0605	Minor Reset Fault *	1404	MINISINS Vel Dif Exceeds Lim
0606	Minor Fault Cont	1405	MINISINS Pos Dif Exceeds Lim
0607	IMU Minor	1501	Parity Test 1 No Go
0701	No Input 3 Axes *	1502	Parity Test 2 No Go
0702	No Input Az *	1503	Parity Test 3 No Go
0703	No Input Pitch *	1504	Put Intercom Test No Go

* Requires immediate troubleshooting action.

Appendix C: DMINS Calibration Parameters

This appendix lists and defines the DMINS calibration parameters. For each parameter, the related component is identified. If a parameters' reading is out of specification, then the applicable component would be suspected as either the cause of a problem, or causing a problem in the future.

<u>Parameter</u>	<u>Description</u>	<u>Component</u>
AX	Velocity Meter Bias	X Velocity Meter
KX	Scale Factor (knots/pulse)	X Velocity Meter
KXY	Cross-Axis Scale Factor (knots/pulse)	X Velocity Meter
AY	Velocity Meter Bias	Y Velocity Meter
KY	Scale Factor (knots/pulse)	Y Velocity Meter
KYX	Cross-Axis Scale Factor (knots/pulse)	Y Velocity Meter
KYZ	Cross-Axis Scale Factor (knots/pulse)	Y Velocity Meter
BX	Gyro Bias	XY Gyro
SX	Scale Factor (degree/hour)	XY Gyro
SXY	Misalignment in the X-Y Plane	XY Gyro
SXZ	Misalignment in the X-Z Plane	XY Gyro
SYX	Misalignment in the Y-X Plane	XY Gyro
SZX	Misalignment in the Z-X Plane	XY Gyro
BY	Gyro Bias	YZ Gyro
BZ	Gyro Bias	YZ Gyro
SY	Scale Factor (degree/hour)	YZ Gyro
SYZ	Misalignment in the Y-Z Plane	YZ Gyro
SZ	Scale Factor (degree/hour)	YZ Gyro
SZY	Misalignment in the Z-Y Plane	YZ Gyro

Appendix D: First Nine Mode B Tests and Subtests

This appendix lists the first nine Mode B tests and each tests' applicable subtests. The first nine tests are incorporated in the prototype developed during this research.

<u>Test</u>	<u>Description</u>
B1.0	IMU Power Up and Thermal Tests
.1	IMU Thermal Switch
.2	IMU Power Shutdown
.3	IMU 64 Hz Clock
.4	AZ Gimbal Motor
.5	IMU Air Flow
B2.0	IMU DC Power Supply Tests
.1	DC Power Supply - +28V
.2	DC Power Supply - +6V
.3	DC Power Supply - -6V
.4	DC Power Supply - +12V
.5	DC Power Supply - -12V
.6	DC Power Supply - +24V
.7	DC Power Supply - -24V
.8	DC Power Supply - +48V
.9	DC Power Supply - +60V
B3.0	IMU AC Power Supply Tests
.1	AC Power Supply - 115V Reference
.2	AC Power Supply - 9.6KHz Triangle Power
.3	AC Power Supply - 6.72KHz Frequency Standard
.4	AC Power Supply - 4.8KHz Angle 0 Frequency Standard
.5	AC Power Supply - 4.8KHz 90 Frequency Standard
.6	AC Power Supply - 4.8KHz
.7	AC Power Supply - 640Hz Angle 0 Frequency Standard
.8	AC Power Supply - 400 Hz Case Rotation
.9	AC Power Supply - 80Hz Angle 0 Frequency Standard
.10	AC Power Supply - 80Hz Angle 0 Case Rotation Triangle
.11	AC Power Supply - 80Hz Angle 120 Case Rotation
.12	AC Power Supply - 80Hz Angle 0 Case Rotation
.13	AC Power Supply - 64Hz Clock

Appendix D (con.)

<u>Test</u>	<u>Description</u>
B4.0	Gyro Temperature Alarm Tests
.1	XY Gyro Hot Indication
.2	XY Gyro Cold Discrete
.3	YZ Gyro Hot
.4	YZ Gyro Cold
B5.0	Thermoelectric Control Tests
.1	Heat Test Limit
.2	Cool Test Limit
B6.0	BITE Status Checks
.1	XY Gyro Speed BITE
.2	YZ Gyro Speed BITE
.3	400Hz BITE
.4	AZ Overrate BITE
.5	Servo Disable
.6	Free Run BITE Fault
.7	AZ Cage
B7.0	BITE Operation Tests
.1	400Hz Servo Disable
.2	4.8Hz Servo Disable
.3	400Hz Bite
.4	AZ Overrate Servo Disable
.5	Free Run
.6	Free Run Reset
B8.0	Cage Discrete Tests
.1	AZ Gimbal Motor
.2	IMU Cage Discrete
B9.0	Resolver Presence
.1	Roll Resolver (2X)
.2	Roll Resolver (36X)
.3	Pitch Resolver (2X)
.4	Pitch Resolver (36X)
.5	Azimuth Resolver (1X)
.6	Azimuth Resolver (36X)

Appendix E: CLIPS Source Code

This appendix contains the expert system source code. All comments are preceded by a semicolon. Commented lines which start with "printout" were used before integrating the expert system with the hypermedia system. Facts asserted which begin with the word "frame" reference a hypermedia frame name. The converted Capt Skinner rules appear in the last 24 pages of this appendix and are the rules which govern the error message troubleshooting. Shim Calibration rules start on Page E-3. Gyro Calibration rules start on Page E-13. Master Heading rules start on Page E-22. Navigation Alignment rules start on Page E-28. Navigation Performance rules start on Page E-31. Rules which assert initial conditions start on Page E-36. Rules which allow communication with the hypermedia system start on Page E-37.

```
;*****
;***
;***      Mode A calibration parameter rules.
;***
;*****

(defrule start-up
  ?if <- (initial-fact)
=>
  (retract ?if)
  (assert (start mode-a)) )

;*** All these parameter rules will only fire if an error code fact has
;*** not been asserted.

;*** This rule finds out which Mode A test to run. It will also
;*** provide a way to exit the program. After each of the five Mode A
;*** tests completes (pass or fail) this rule will fire.

(defrule start-mode-a
  ?sm <- (start mode-a)
  (not (error-code-phase))
=>
  (retract ?sm)
  ;(printout t t "Which test would you like to run?")
  ;(printout t t "Enter shim-cal, ")
  ;(printout t t "    gyro-cal, ")
```

Appendix E: CLIPS Source Code

```
;(printout t t "    master-heading, ")
;(printout t t "    nav-align, ")
;(printout t t "    nav, ")
;(printout t t "    exit: ")
;(assert (begin test =(read)))
(assert (frame dflorian134)) )
```

*** Start Shim Cal test.

```
(defrule start-shim-cal
  ?bt <- (begin test shim-cal)
  (not (error-code-phase))
=>
  (retract ?bt)
  (assert (start shim-cal 1)) )
```

*** Start Gyro Cal test.

```
(defrule start-gyro-cal
  ?bt <- (begin test gyro-cal)
  (not (error-code-phase))
=>
  (retract ?bt)
  (assert (start gyro-cal 1)) )
```

*** Start Master Heading test.

```
(defrule start-master-heading
  ?bt <- (begin test master-heading)
  (not (error-code-phase))
=>
  (retract ?bt)
  (assert (start master-heading)) )
```

*** Start Nav Align test.

```
(defrule start-nav-align
  ?bt <- (begin test nav-align)
  (not (error-code-phase))
```

Appendix E: CLIPS Source Code

```
=>
  (retract ?bt)
  (assert (start nav-align)) )

;*** Start Nav test.

(defrule start-nav
  ?bt <- (begin test nav)
  (not (error-code-phase))
=>
  (retract ?bt)
  (assert (start nav 1)) )

;*** Exit DMINS Expert System

;(defrule exit-dmins
;  ?bt <- (begin test exit)
;  (not (error-code-phase))
;=>
;  (retract ?bt)
;  (printout t t "Goodbye." t t)
;  (assert (frame dflorian135)) )

;*****
;
;***
;
;***      Shim Calibration Rules
;
;***
;*****

;*** At the end of each Shim-Cal, get the status of the AX and AY
;*** parameters. Not much troubleshooting is done until the run is
;*** over. A "too close to spec" parameter is one which it's value is
;*** greater than 0.06 but still within the spec. The only parameters
;*** this test is concerned with is AX and AY.

(defrule get-ax-status
  ?ss <- (start shim-cal ?test-num)
  (not (error-code-phase))
=>
  ;(printout t t "After running Shim-Cal " ?test-num ":" t)
```

Appendix E: CLIPS Source Code

```
;(printout t t "Is AX out-of-spec, too-close-to-spec, or good? ")
;(assert (ax shim-cal ?test-num =(read)))
(assert (frame dflorian136 Test: ?test-num)) )

(deffacts shim-cal-deffact
  (finished shim-cal 0) )

(defrule get-ay-status
  ?ss <- (start shim-cal ?test-num)
  (ax shim-cal ?test-num ?status)
  ?rs <- (finished shim-cal =(- ?test-num 1))
  (not (error-code-phase))
=>
  (retract ?ss ?rs)
  ;(printout t t "After running Shim-Cal " ?test-num ":" t)
  ;(printout t t "Is AY out-of-spec, too-close-to-spec, or good? ")
  ;(assert (ay shim-cal ?test-num =(read)))
  (assert (finished shim-cal ?test-num))
  (assert (frame dflorian365 Test: ?test-num)) )

;*** The technicians look at how the parameters are changing from one
;*** run to the next. From one run to the next, the parameter should
;*** be getting closer to the ideal specification (i.e., the delta
;*** terms should be getting smaller). If this isn't happening, a
;*** problem probably exists. The following rules describe this
;*** behavior. A parameter which is significantly moving away from
;*** spec in consecutive runs has a problem. Other types of changes
;*** are self explanatory. For example, if a parameter was out of
;*** spec for runs 1 and 2 but is significantly getting closer to the
;*** spec, a third run might correct the problem.

;*** If a parameter stays the same (either good, close to, or out of
;*** spec, from run 1 to 2, this rule will fire.

(defrule parameter-change-from-previous-run
  (?parameter1 shim-cal 1 ?p1status)
  (?parameter1 shim-cal 2 ?p1status)
  (?parameter2 shim-cal 2 ?p2status)
```

Appendix E: CLIPS Source Code

```
(test (neq ?p1status too-close-to-spec))
(test (neq ?parameter1 ?parameter2))
(not (error-code-phase))
=>
  ;(printout t t "The " ?parameter " change from the previous run was: ")
  ;(printout t t "   none, ")
  ;(printout t t "   significantly-better (better), ")
  ;(printout t t "   significantly-worse (worse)? ")
  ;(assert (?parameter change 1 to 2 =(read)))
  (assert (frame dflorian137 Parameter: ?parameter1)) )

;*** The following rules describe how a parameter passes a Shim Cal
;*** run. When both parameters pass the same run, a check for gross
;*** is done.

;*** If a parameter is good for two runs and is significantly
;*** moving toward the spec, then it passed the current test.

(defrule good-parameter-got-better
  (?parameter change 1 to 2 better)
  (?parameter shim-cal 1 good)
  (?parameter shim-cal 2 good)
  (not (error-code-phase))
=>
  (assert (?parameter passed shim-cal 2)) )

;*** If a parameter is good and did not significantly change from
;*** previous run, then the parameter passed the current test.

(defrule good-parameter-no-change
  (?parameter change 1 to 2 none)
  (?parameter shim-cal 2 good)
  (not (error-code-phase))
=>
  (assert (?parameter passed shim-cal 2)) )

;*** A parameter which was close to the spec on the previous run is
;*** now looking good; therefore, it passed the test.
```

Appendix E: CLIPS Source Code

```
(defrule too-close-parameter-is-now-good
  (?parameter shim-cal ?test-num too-close-to-spec)
  (?parameter shim-cal =(+ ?test-num 1) good)
  (not (error-code-phase))
=>
  (assert (?parameter passed shim-cal =(+ ?test-num 1))) )

;*** An out of spec parameter in previous run is now good and passes
;*** test.

(defrule out-of-spec-parameter-is-now-good
  (?parameter shim-cal ?test-num out-of-spec)
  (?parameter shim-cal =(+ ?test-num 1) good)
  (not (error-code-phase))
=>
  (assert (?parameter passed shim-cal =(+ ?test-num 1))) )

;*** If the parameter (AX or AY) status is good after the first or third
;*** run, assert that the parameter passed this run. When both pass the
;*** same run, then the IMU will pass Shim Cal.

(defrule parameter-passed-shim-cal-1 ""
  (?parameter shim-cal 1 good)
  (not (error-code-phase))
=>
  (assert (?parameter passed shim-cal 1)) )

(defrule good-parameter-still-good-run-3 ""
  (?parameter shim-cal 2 good)
  (?parameter shim-cal 3 good)
  (not (error-code-phase))
=>
  (assert (?parameter passed shim-cal 3)) )

;*** The following rule signifies that an IMU has passed Shim Cal.
;*** Shim Cal does not check for gross failures which occur if a
;*** parameters upper or lower bound are out of spec. This rule base
;*** checks for gross failures.

;*** If both AX and AY passed, then check for gross failures. A
```

Appendix E: CLIPS Source Code

```
*** parameter has a gross failure if it exceeds its upper or lower  
*** limits.
```

```
(defrule shim-cal-passed  
  ?ax <- (ax passed shim-cal ?test-num)  
  ?ay <- (ay passed shim-cal ?test-num)  
  (not (error-code-phase))  
=>  
  (retract ?ax ?ay)  
  (assert (shim-cal ?test-num passed? yes)) )
```

```
*** These gross failure rules are used in Shim-Cal and in Gyro-Cal.  
*** When a fact is asserted that says either Shim Cal or Gyro Cal  
*** passed, these rules will fire.
```

```
(defrule check-for-gross-failures  
  ?tc <- (?test ?test-num passed? yes)  
  (finished ?test ?test-num)  
  (test (or (eq ?test shim-cal)  
            (eq ?test gyro-cal) ))  
  (not (error-code-phase))  
=>  
  (retract ?tc)  
  ;(printout t t "IMU passed " ?test "." t)  
  ;(printout t t "Check for any gross failures." t)  
  ;(printout t t "Did any parameter exceed the gross limits (yes/no)? ")  
  ;(assert (gross-failure? =(read)))  
  (assert (frame dflorian138 Test: ?test-num)) )
```

```
*** If no gross failures, begin next test.
```

```
(defrule no-gross-failures  
  ?gf <- (gross-failure? no)  
  (not (error-code-phase))  
=>  
  (retract ?gf)  
  (assert (start next-test)) )
```

```
(defrule shim-cal-complete
```

Appendix E: CLIPS Source Code

```
?rs <- (finished shim-cal ?number)
?sn <- (start next-test)
(test (!= ?number 0))
(not (error-code-phase))
=>
(retract ?rs ?sn)
;(printout t t "Shim Cal is complete. Run Gyro Cal." t)
;(assert (start mode-a))
(assert (frame dflorian139)) )

(defrule gyro-cal-complete
  ?rs <- (finished gyro-cal ?)
  ?sn <- (start next-test)
  (not (error-code-phase))
=>
  (retract ?rs ?sn)
  ;(printout t t "Gyro Cal is complete. Run Master Heading." t)
  ;(assert (start mode-a))
  (assert (frame dflorian140)) )

;*** If there was a gross failure, find out what parameter. This
;*** program will assume only one parameter will have a gross
;*** failure. While this does not seem realistic, in reality, AGMC
;*** is not even looking for gross failures at this time. Catching
;*** possible problems early on in the testing cycle can save time
;*** and resources.

(defrule had-gross-failure
  ?gf <- (gross-failure? yes)
  (not (error-code-phase))
=>
  (retract ?gf)
  ;(printout t t "Which parameter had the gross failure? ")
  ;(assert (=(read) gross-failure))
  (assert (frame dflorian141)) )

;*** Depending on the parameter with the gross failure, the
;*** faulty component is determined.

(defrule gross-failure-component
```

Appendix E: CLIPS Source Code

```
?pf <- (gross-failure ?parameter)
(?parameter ?component parameter)
(not (error-code-phase))
=>
  (retract ?pf)
  (assert (faulty ?component)))

;*** The following 3 rules describe how a parameter can fail a Shim
;*** Cal run.

;*** If the status of a parameter is not good during first run, then
;*** that parameter failed the first Shim-Cal. This rule will fail a
;*** parameter for being "too-close-to-spec". While the test
;*** equipment considers this parameter good, another run will
;*** determine if the parameter will move away or toward the
;*** specification. Shim Cal is a very short test compared to the
;*** Mode A tests and it is worth the time now to try and detect a
;*** problem. This type of troubleshooting is only being done by the
;*** most experienced AGMC technicians.

(defrule parameter-failed-shim-cal-1
  (?parameter shim-cal 1 ?status)
  (test (neq ?status good))
  (not (error-code-phase))
=>
  (assert (?parameter failed shim-cal 1)))

;*** If the parameter was out-of-spec in run 1 and run 2 but is
;*** getting closer to a fix, fail the term but make sure another cal
;*** is run to see if it will correct itself.

(defrule parameter-getting-better
  (?parameter change 1 to 2 better)
  (?parameter shim-cal 1 out-of-spec)
  (?parameter shim-cal 2 out-of-spec)
  (not (error-code-phase))
=>
  (assert (?parameter failed shim-cal 2)))
```

Appendix E: CLIPS Source Code

```
*** If a parameter close to the spec, the parameter fails the
*** current test.

(defrule parameter-is-too-close-to-spec
  (?parameter shim-cal ?test-num ?status)
  (?parameter shim-cal =(+ ?test-num 1) too-close-to-spec)
  (not (error-code-phase))
=>
  (assert (?parameter failed shim-cal =(+ ?test-num 1))) )

*** If either AX or AY failed a run, then start another run, unless
*** the run was the third one. AGMC will always run at least 2
*** Shim-Cals.

(defrule run-next-shim-cal
  ?t1 <- (?parameter1 ?status shim-cal ?test-num)
  ?t2 <- (?parameter2 failed shim-cal ?test-num)
  (test (neq ?parameter1 ?parameter2))
  (test (< ?test-num 3))
  (test (or (eq ?status passed)
            (eq ?status failed) ))
  (not (error-code-phase))
=>
  (retract ?t1 ?t2)
  (assert (start shim-cal =(+ ?test-num 1))) )

*** The following rules describe when to stop running Shim Cal and
*** what the suspected faulty component is.

*** If any parameter fails a 3rd Shim-Cal a problem exists in a
*** velocity meter. This will fire if a term is too close to the
*** spec after the 3rd run. This is not how AGMC does it (they would
*** pass the IMU. Since the parameter should be getting better
*** each run, I've chose to fail the IMU. This could mean
*** improved MTBF of IMUs that would have been shipped with
*** marginally accepted specifications.

(defrule parameter-failed-shim-cal-3
  (?parameter1 failed shim-cal 3)
```

Appendix E: CLIPS Source Code

```
(?parameter2 ?status shim-cal 3)
(?parameter1 ?component parameter)
(test (neq ?parameter1 parameter2))
(not (error-code-phase))
=>
(assert (problem with ?component)))

;*** If the parameter stayed either good or bad from one run to the
;*** next, and the change was significantly worse, then there is a
;*** problem one of the velocity meters.

(defrule velocity-meter-problem
  (?parameter1 change 1 to 2 worse)
  (?parameter1 shim-cal 1 ?p1status)
  (?parameter1 shim-cal 2 ?p1status)
  (?parameter2 shim-cal 2 ?p2status)
  (?parameter1 ?component parameter)
  (test (neq ?parameter1 ?parameter2))
  (not (error-code-phase))
=>
  (assert (problem with ?component)))

;*** If a parameter was out of spec for 2 consecutive runs and did not
;*** show any improvement, then a problem exists in a velocity meter.

(defrule out-of-spec-for-2-runs
  (?parameter1 change 1 to 2 none)
  (?parameter1 shim-cal 2 out-of-spec)
  (?parameter1 ?component parameter)
  (?parameter2 shim-cal 2 ?status)
  (test (neq ?parameter1 ?parameter2))
  (not (error-code-phase))
=>
  (assert (problem with ?component)))

;*** If a parameter was good or too close to spec on a previous run
;*** and is now out of spec, a problem exists in a velocity meter.

(defrule parameter-not-then-out-of-spec
  (?parameter1 shim-cal ?test-num ?status)
```

Appendix E: CLIPS Source Code

```
(?parameter1 shim-cal =(+ ?test-num 1) out-of-spec)
(?parameter1 ?component parameter)
(?parameter2 shim-cal =(+ ?test-num 1) ?p2status)
(test (neq ?parameter1 ?parameter2))
(test (or (eq ?status good)
          (eq ?status too-close-to-spec) ))
(not (error-code-phase))
=>
(assert (problem with ?component)) )

;*** A problem was found in only one of the VM parameters. This rule
;*** must fire only if the next rule doesn't.

(defrule problem-with-one-vm
  (declare (salience -200))
  ?pw <- (problem with ?component)
  (not (error-code-phase))
=>
  (retract ?pw)
  (assert (faulty ?component)) )

;*** If a problem with both AX and AY, technician should go see
;*** supervisor. This does not happen very often. It is possible
;*** both velocity meters could be faulty.

(defrule problem-with-ax-and-ay
  (declare (salience -100))
  ?px <- (problem with x-vm)
  ?py <- (problem with y-vm)
  (not (error-code-phase))
=>
  (retract ?px ?py)
  ;(printout t t "Having a problem with both AX and AY is very rare." t)
  ;(printout t t "Both velocity meters could be faulty." t)
  ;(printout t t "See Shop Supervisor." t)
  (assert (frame dflorian142)) )

;*****
;***
```

Appendix E: CLIPS Source Code

```
***      Gyro Calibration Rules
***
*****

*** During Gyro Cal runs, velocity parameters are printed. If these
*** partameters are high, the technician can use the information
*** concerning directiona and angle to determine the faulty
*** component. If the velocities are not high, the technician waits
*** for the run to finish. At this time, the status of some
*** parameters is given. The technician will use the information
*** printed concerning the parameters to do the troubleshooting. By
*** looking at which parameters failed and by how much, a faulty
*** component can be detected. After a Gyro Cal run passes, a check
*** for gross failurs is done using the code found in the Shim Cal
*** section.

*** Find out if any velocities are high at zero degrees. If they are
*** high, then check them again at 90 degrees.

(defrule were-vdifs-high-at-zero?
  ?sg <- (start gyro-cal ?test-num)
  (not (error-code-phase))
=>
  (retract ?sg)
  ;(printout t t "What velocities were high at zero degrees:")
  ;(printout t t "  n-s, ")
  ;(printout t t "  e-w, ")
  ;(printout t t "  both, ")
  ;(printout t t "  none? ")
  ;(assert (high-velocities zero-degrees =(read)))
  (assert (finished gyro-cal ?test-num))
  (assert (frame dflorian143)) )

*** If there were velocities high at zero degrees, find out if any
*** velocities were high at 90 degrees.

(defrule were-vdifs-high-at-90?
  (high-velocities zero-degrees ?vdif)
  (test (neq ?vdif none))
```

Appendix E: CLIPS Source Code

```

(not (error-code-phase))
=>
  ;(printout t t "What velocities were high at 90 degrees:")
  ;(printout t t "  n-s, ")
  ;(printout t t "  e-w, ")
  ;(printout t t "  both, ")
  ;(printout t t "  none? ")
  ;(assert (high-velocities ninety-degrees =(read)))
  (assert (frame dflorian144)) )

;*** If velocities were high in n-s then e-w, then there is a problem
;*** with the x velocity meter.

(defrule vdifs-high-n-s-then-e-w
  ?h1 <- (high-velocities zero-degrees n-s)
  ?h2 <- (high-velocities ninety-degrees e-w)
  (not (error-code-phase))
=>
  (retract ?h1 ?h2)
  (assert (faulty x-vm)) )

;*** If velocities were high in e-w then n-s, then there is a problem
;*** with the y velocity meter.

(defrule vdifs-high-e-w-then-n-s
  ?h1 <- (high-velocities zero-degrees e-w)
  ?h2 <- (high-velocities ninety-degrees n-s)
  (not (error-code-phase))
=>
  (retract ?h1 ?h2)
  (assert (faulty y-vm)) )

;*** If velocities were high in n-s and stayed n-s, then there is a
;*** problem with the yz gyro.

(defrule vdifs-high-n-s-then-n-s
  ?h1 <- (high-velocities zero-degrees ?direction1)
  ?h2 <- (high-velocities ninety-degrees ?direction2)
  (test (or (and (eq ?direction1 n-s)
                  (eq ?direction2 n-s) )

```

Appendix E: CLIPS Source Code

```
(and (eq ?direction1 both)
      (eq ?direction2 n-s) )
(and (eq ?direction1 n-s)
      (eq ?direction2 both) )))
(not (error-code-phase))
=>
(retract ?h1 ?h2)
(assert (faulty yz-gyro)) )

;*** If velocities were high in e-w and stayed e-w, then there is a
;*** problem with the xy gyro.

(defrule vdifs-high-e-w-then-e-w
  ?h1 <- (high-velocities zero-degrees ?direction1)
  ?h2 <- (high-velocities ninety-degrees ?direction2)
  (test (or (and (eq ?direction1 e-w)
                  (eq ?direction2 e-w) )
             (and (eq ?direction1 both)
                  (eq ?direction2 e-w) )
             (and (eq ?direction1 e-w)
                  (eq ?direction2 both) )))
  (not (error-code-phase))
=>
(retract ?h1 ?h2)
(assert (faulty xy-gyro)) )

;*** Having both directions high for both angles does not happen very much
;*** or at all.

(defrule both-vdifs-high
  ?h1 <- (high-velocities zero-degrees both)
  ?h2 <- (high-velocities ninety-degrees both)
  (not (error-code-phase))
=>
(retract ?h1 ?h2)
;(printout t t "This is very rare. Contact your supervisor." t)
(assert (frame dflorian145)) )

;*** If no velocities were high at zero degrees, or if they were high
```

Appendix E: CLIPS Source Code

```
*** at zero but not at 90 degrees, let current test finish and ask
*** operator if the test passed.

(defrule vdifs-ok
  ?hv <- (high-velocities ?degree none)
  (finished gyro-cal ?test-num)
  (not (error-code-phase))
=>
  (retract ?hv)
  ;(printout t t "Velocities are fine." t)
  ;(printout t t "Did Gyro-Cal " ?test-num " pass (yes/no)? ")
  ;(assert (gyro-cal ?test-num passed? =(read)))
  (assert (frame dflorian146 Test: ?test-num)) )

*** If current Gyro-Cal run passes, check for gross failures. A
*** parameter has a gross failure if it exceeds its upper or lower
*** limit. Use the same rules found in Shim-Cal rules section.

*** If first Gyro Cal fails, always run another. Not much
*** troubleshooting is done if the first run fails.

(defrule run-gyro-cal-2
  ?gc <- (gyro-cal 1 passed? no)
  ?rg <- (finished gyro-cal 1)
  (not (error-code-phase))
=>
  (retract ?gc ?rg)
  ;(printout t t "Let Gyro Cal 2 run." t)
  ;(assert (start gyro-cal 2))
  (assert (frame dflorian147)) )

*** If second or third Gyro Cal fails then we need to find out which
*** parameter(s) is/are out of spec. 3 rules.

(defrule gyro-cal-2-or-3-fails
  ?gc <- (gyro-cal ?test-num passed? no)
  (finished gyro-cal ?test-num)
  (test (!= ?test-num 1))
  (not (error-code-phase))
=>
```

Appendix E: CLIPS Source Code

```
(retract ?gc)
(assert (get failed-parameters gyro-cal ?test-num)) )

(defrule get-failed-parameters
  ?gf <- (get failed-parameters gyro-cal ?test-num)
  (finished gyro-cal ?test-num)
  (not (error-code-phase))
=>
  (retract ?gf)
  ;(printout t t "Enter failed parameter or no-more if done. ")
  ;(assert (failed-parameter =(read) gyro-cal ?test-num))
  (assert (frame dflorian148 Test: ?test-num)) )

(defrule more-failed-parameters
  (failed-parameter ?parameter gyro-cal ?test-num)
  (finished gyro-cal ?test-num)
  (test (neq ?parameter no-more))
  (not (error-code-phase))
=>
  (assert (get failed-parameters gyro-cal ?test-num)) )

;*** Technicians look at how each of the parameters failed. The way a
;*** parameter fails determines how the troubleshooting proceeds. If
;*** a parameter just barely fails, you should run another Cal. If a
;*** parameter fails real bad, then if it can be isolated to one
;*** component then Gyro Cal stops. If all the real bad failures
;*** identify more than one component, a third Cal will be run.
;*** If all the parameters are close to spec and this is run 2, then
;*** run a third test.

(defrule how-bad-did-failed-parameter-fail
  (failed-parameter no-more gyro-cal ?test-num)
  (failed-parameter ?parameter gyro-cal ?test-num)
  (finished gyro-cal ?test-num)
  (test (neq ?parameter no-more))
  (not (error-code-phase))
=>
  ;(printout t t "Was " ?parameter " close to spec (yes/no)? ")
  ;(assert (?parameter close-to-spec? =(read)))
```

Appendix E: CLIPS Source Code

```
(assert (frame dflorian149 Parameter: ?parameter)) )

;*** This rule determines if at least one of the parameters that
;*** failed, failed badly.

(defrule no-parameters-way-away-from-spec
  (finished gyro-cal ?test-num)
  (?parameter close-to-spec? no)
  (test (!= ?test-num 1))
  (not (error-code-phase))
=>
  (assert (gyro-cal ?test-num at-least-one-parameter away-from-spec)) )

;*** If no parameters which failed, failed badly, then cannot
;*** conclude with certainty just one component is at fault.

(defrule gyro-cal-3-bad
  (declare (salience -500))
  ?rg <- (finished gyro-cal 3)
  (not (gyro-cal 3 at-least-one-parameter away-from-spec))
  (not (error-code-phase))
=>
  (retract ?rg)
  (assert (gyro-cal 3 inconclusive)) )

;*** This rule takes each parameter and identifies the component it
;*** suggests as being faulty.

(defrule suspected-components
  ?fp <- (failed-parameter ?parameter gyro-cal ?test-num)
  ?pc <- (?parameter close-to-spec? no)
  (?parameter ?component parameter)
  (finished gyro-cal ?test-num)
  (not (error-code-phase))
=>
  (retract ?pc ?fp)
  (assert (?component implicated gyro-cal ?test-num)) )

;*** If more than one parameter failed badly, and they suggested more
```

Appendix E: CLIPS Source Code

```
*** than one component is the problem, run another Cal to try to
*** narrow it down to one component.
```

```
(defrule more-than-one-suspected-component
  (declare (salience -50))
  ?fp <- (failed-parameter no-more gyro-cal ?test-num)
  ?rg <- (finished gyro-cal ?test-num)
  ?c1 <- (?component1 implicated gyro-cal ?test-num)
  ?c2 <- (?component2 implicated gyro-cal ?test-num)
  (test (neq ?component1 ?component2))
  (not (error-code-phase))
=>
  (retract ?fp ?rg ?c1 ?c2)
  (assert (gyro-cal ?test-num inconclusive)))
```

```
*** Based on the results of a failed Gyro Cal 2, the evidence is
*** inconclusive because more than one component is being considered
*** as the primary suspect. This occurs when 2 parameters which
*** are associated with different components, both failed
*** and were not close to the spec. This also occurs if no parameter
*** was close to the spec. Gyro Cal 3 should be run to gather more
*** information. 2 rules.
```

```
(defrule gyro-cal-2-inconclusive
  ?gc <- (gyro-cal 2 inconclusive)
  (not (error-code-phase))
=>
  (retract ?gc)
  ;(printout t t "Evidence was inconclusive. Let Gyro Cal 3 keep running." t)
  ;(assert (start gyro-cal 3))
  (assert (frame dfloria...))
```

```
*** If after Gryo Cal 2 or 3, there is just one parameter out of
*** spec, and it is way out of spec, you have enough evidence to
*** suggest a faulty component. This must fire only if the rule
*** above cannot.
```

```
(defrule conclusive-evidence
  (declare (salience -100))
  ?fp <- (failed-parameter no-more gyro-cal ?test-num)
```

Appendix E: CLIPS Source Code

```
?rg <- (finished gyro-cal ?test-num)
?ci <- (?component implicated gyro-cal ?test-num)
(not (error-code-phase))
=>
(retract ?fp ?rg ?ci)
(assert (faulty ?component)) )

;*** If no failed parameters failed badly, run third Cal.

(defrule run-gyro-cal-3
  (declare (salience -500))
  ?fp <- (failed-parameter no-more gyro-cal ?test-num)
  ?rg <- (finished gyro-cal 2)
  (not (gyro-cal 2 at-least-one-parameter away-from-spec))
  (not (error-code-phase))
=>
  (retract ?fp ?rg)
  ;(printout t t "Let Gyro Cal 3 keep running." t)
  ;(printout t t "The failed parameters could correct themselves in this run." t)
  ;(assert (start gyro-cal 3))
  (assert (frame dflorian151)) )

;*** In Cal 3 if no parameters failed badly, the technicians will
;*** look for repeatability in the component that the parameters
;*** suggest is the problem.

;*** Parameters failing Gyro Cal 3 will be tallied by component.

(defrule parameters-failing-gyro-cal-3
  (failed-parameter ?parameter gyro-cal 3)
  (?parameter ?component parameter)
  (not (error-code-phase))
=>
  (bind ?unique-num (gensym))
  (assert (suspect ?component ?unique-num 1)) )

;*** Keeps a running total of how many parameters are providing
;*** evidence that a component is faulty.
```

Appendix E: CLIPS Source Code

```
(defrule keep-running-total
  ?s1 <- (suspect ?component ?unique-num1 ?weight1)
  ?s2 <- (suspect ?component ?unique-num2 ?weight2)
  (test (neq ?unique-num1 ?unique-num2))
  (not (error-code-phase))
=>
  (retract ?s1 ?s2)
  (bind ?new-weight (+ ?weight1 ?weight2))
  (bind ?new-unique-num (gensym))
  (assert (suspect ?component ?new-unique-num ?new-weight)) )

;*** These facts are needed to tally the times a component is
;*** suspected by the failed parameters.

(deffacts initial-component-count
  (suspect x-vm start 0)
  (suspect y-vm start 0)
  (suspect xy-gyro start 0)
  (suspect yz-gyro start 0) )

;*** Clean up previous facts if not used.

;(defrule clean-up-suspects
; (declare (salience -10000))
; ?sp <- (suspect ?parameter start 0)
; (not (error-code-phase))
;=>
; (retract ?sp) )

;*** If after Cal 3 you couldn't narrow the suspected components down
;*** to one component or if there were no parameters that failed
;*** badly, this rule will fire. Lists the suspected components in a
;*** single fact which is printed to a KMS frame. The deffacts statement
;*** is needed to start the combination of the facts.

(deffacts start-combining-suspects
  (suspect-list) )

(defrule gyro-cal-3-inconclusive
```

Appendix E: CLIPS Source Code

```

(gyro-cal 3 inconclusive)
?sc <- (suspect ?component ?unique-num ?total)
?sl <- (suspect-list $?list)
(test (!= ?total 0))
(not (error-code-phase))
=>
(retract ?sc ?sl)
;(printout t t "Suspect component: " ?component " Evidence: " ?total t)
(assert (suspect-list $?list ?component evidence= ?total)) )

;*** This just writes a list of components to a KMS frame.

(defrule print-suspect-list-to-kms-frame
  (declare (salience -10000))
  ?gc <- (gyro-cal 3 inconclusive)
  ?sl <- (suspect-list $?list)
=>
  (retract ?gc ?sl)
  (assert (frame dflorian373 List: $?list)) )

.*****
.***
.***
.***      Master Heading Rules
.***
.*****

;*** This is not a timed test. The technician is just waiting for
;*** the gyro bias terms to stabilize.

;*** At the beginning and during Master Heading the technician is
;*** asked how long the test has been running and if the bias
;*** parameters are stabilizing. The variable ?status can equal
;*** "start" or "continue."

(defrule are-bias-parameters-stable?
  ?sm <- (?status master-heading)
  (test (neq ?status determine-fault))
  (not (error-code-phase))
=>
  ;(printout t t "Has BXC, BYC, and BZC stabilized (yes/no)? ")

```

Appendix E: CLIPS Source Code

```
;(assert (master-heading bias-parameters-stable? =(read)))
(assert (frame dflorian153)) )

(defrule how-long-has-master-heading-been-running
  ?sm <- (?status master-heading)
  (master-heading bias-parameters-stable? ?hours)
  (test (neq ?status determine-fault))
  (not (error-code-phase))
=>
  (retract ?sm)
  ;(printout t t "How long has Master Heading been running (hours)? ")
  ;(assert (master-heading run-time =(read) hours))
  (assert (frame dflorian366)) )

;*** If the bias parameters are stable and Master Heading has been
;*** running at least 4 hours, then go on to the Nav-Align test.

(defrule master-heading-complete
  ?mh <- (master-heading run-time ?num-hours hours)
  ?bt <- (master-heading bias-parameters-stable? yes)
  (test (> ?num-hours 3.9))
  (not (error-code-phase))
=>
  (retract ?mh ?bt)
  ;(printout t t "Apply bias. TCI 0100. TCI 0101." t)
  ;(printout t t "Master Heading is complete. Start Nav Align." t)
  ;(assert (start mode-a))
  (assert (frame dflorian154)) )

;*** If the bias parameters are stable and the test has not been
;*** running at least 4 hours; or if the bias terms are unstable and
;*** the test has been running less than 6.5 hours, continue the test.

(defrule continue-master-heading
  ?mh <- (master-heading run-time ?num-hours hours)
  ?bt <- (master-heading bias-parameters-stable? ?status)
  (test (or (< ?num-hours 4)
            (and (eq ?status no)
                  (< ?num-hours 6.5) )))
```

Appendix E: CLIPS Source Code

```
(not (error-code-phase))
=>
  (retract ?mh ?bt)
  ;(printout t t "Continue Master Heading." t)
  ;(assert (continue master-heading))
  (assert (frame dflorian155)) )

;*** If bias parameters are not stable and the test has been running
;*** at least 6.5 hours, then a problem exists in either a velocity
;*** meter or a gyro.

(defrule stop-master-heading-test
  ?mh <- (master-heading run-time ?num hours)
  ?bt <- (master-heading bias-parameters-stable? no)
  (test (>= ?num-hours 6.5))
  (not (error-code-phase))
=>
  (retract ?mh ?bt)
  ;(printout t t "Stop Master Heading test." t)
  ;(assert (determine-fault master-heading))
  (assert (frame dflorian156)) )

;*** If the bias parameters do not stabilize either the velocity
;*** parameters have increased suddenly (i.e., VDIF >= .05 for at
;*** least half hour) or there was a gradual change in the bias
;*** parameters.

(defrule determine-fault-in-master-heading
  ?pf <- (determine-fault master-heading)
  (not (error-code-phase))
=>
  (retract ?pf)
  ;(printout t t "Was there a sudden change in velocity parameters (VDIF)")
  ;(printout t t "or a gradual drift in any of the bias parameters (BXC, BYC, BZC)
  (sudden-change or gradual-drift)? ")
  ;(assert (problem-in master-heading is =(read)))
  (assert (frame dflorian157)) )

;*** If a sudden change in the velocity parameters occurred, find out
;*** which velocity parameters. The sudden change rules are also used
```

Appendix E: CLIPS Source Code

*** by the Nav section.

```
(defrule check-velocity-parameters
  ?pi <- (problem-in ?test is sudden-change)
  (test (or (eq ?test master-heading)
            (eq ?test nav) ))
  (not (error-code-phase))
```

=>

```
  (retract ?pi)
  ;(printout t t "Was the change in the n-s, e-w, or other? ")
  ;(assert (sudden-change =(read)))
  (assert (frame dflorian158)) )
```

*** If the sudden change was in the North-South direction, then
*** there is a problem in the x velocity meter.

```
(defrule sudden-change-in-n-s-velocity
  ?sc <- (sudden-change n-s)
  (not (error-code-phase))
```

=>

```
  (retract ?sc)
  (assert (faulty x-vm)) )
```

*** If the sudden change was in the East-West direction, then
*** there is a problem in the y velocity meter.

```
(defrule sudden-change-in-e-w-velocity
  ?sc <- (sudden-change e-w)
  (not (error-code-phase))
```

=>

```
  (retract ?sc)
  (assert (faulty y-vm)) )
```

*** If the sudden change was not isolated in only one direction,
*** rerun Shim-Cal and look for marginal acceptance of parameters.
*** The way Shim-Cal rules are written, terms that are too close to
*** the spec fail that Shim-Cal run.

```
(defrule sudden-change-in-other-velocity
```

Appendix E: CLIPS Source Code

```
?sc <- (sudden-change other)
(not (error-code-phase))
=>
(retract ?sc)
;(printout t t "Rerun Shim-Cal and look for marginal acceptance of parameters." t)
;(assert (start mode a))
(assert (frame dflorian159)) )

;*** If a gradual change in the bias parameters, then a problem exists
;*** in one of the gyros.

(defrule check-bias-parameters
  ?pi <- (problem-in master-heading is gradual-drift)
  (not (error-code-phase))
=>
  (retract ?pi)
  ;(printout t t "Was the drift in bxc, byc, bzc, bxc-and-byc, or other? ")
  ;(assert (gradual-drift =(read)))
  (assert (frame dflorian160)) )

;*** If the gradual drift was in BXC or BYC parameters, then there is
;*** a problem in the x gyro.

(defrule gradual-drift-in-bxc-or-byc
  ?gd <- (gradual-drift ?parameter)
  (test (or (eq ?parameter bxc)
            (eq ?parameter byc)
            (eq ?parameter bxc-and-byc) ))
  (not (error-code-phase))
=>
  (retract ?gd)
  (assert (faulty xy-gyro)) )

;*** If the gradual drift was in the BZC parameter, then there is a
;*** problem in the z gyro.

(defrule gradual-drift-in-bzc
  ?gd <- (gradual-drift bzc)
  (not (error-code-phase))
```

Appendix E: CLIPS Source Code

```
=>
  (retract ?gd)
  (assert (faulty yz-gyro)) )

;*** If the gradual drift is in some other combination of parameters
;*** than the above 2 rules than check the axis headings.

(defrule gradual-drift-in-other
  ?gd <- (gradual-drift other)
  (not (error-code-phase))
=>
  (retract ?gd)
  ;(printout t t "Examine the roll, pitch, and azimuth synchro outputs from the alarm panel."
  t)
  ;(printout t t "Is any one axis off heading more than the others (roll-pitch, pitch-azimuth,
  none)? ")
  ;(assert (axis-off-heading =(read)))
  (assert (frame dflorian161)) )

;*** If the roll-pitch axis is off heading, then there is a problem
;*** with the xy-gyro

(defrule roll-pitch-axis-off
  ?ao <- (axis-off-heading roll-pitch)
  (r (error-code-phase))
=>
  (retract ?ao)
  (assert (faulty xy-gyro)) )

;*** If the pitch-azimuth axis is off heading, then there is a
;*** problem with the yz-gyro

(defrule pitch-azimuth-axis-off
  ?ao <- (axis-off-heading pitch-azimuth)
  (not (error-code-phase))
=>
  (retract ?ao)
  (assert (faulty yz-gyro)) )

;*** If neither axis is off heading more than the other, then rerun
```

Appendix E: CLIPS Source Code

```
*** Gyro-Cal and look for bias shifts and a bad case rotation motor.
```

```
(defrule neither-axis-off-more
  ?ao <- (axis-off-heading none)
  (not (error-code-phase))
=>
  (retract ?ao)
  ;(printout t t "Rerun Gyro-Cal. Look for bias shifts or a bad case rotation motor." t)
  ;(assert (start mode-a))
  (assert (frame dflorian162)) )
```

```
*****
***
***      Navigation Alignment Rules
***
*****
```

```
*** Very little troubleshooting is done during this test.
*** It was very hard to get a definition or an example of when a
*** this test would be stopped before it ran its course. The
*** technician is plotting a RMS point, a latitude point, and a
*** longitude point each hour.
```

```
*** Get the rate of the RMS change and the number of hours that
*** Nav-Align has been running.
```

```
(defrule get-rms-rate-change
  ?sn <- (?status nav-align)
  (nav-align run-time ?num hours)
  (not (error-code-phase))
=>
  (retract ?sn)
  ;(printout t t "Has the rate of RMS change been normal, exceptional, or drastic? ")
  ;(assert (nav-align rms-rate-change =(read)))
  (assert (frame dflorian367)) )
```

```
(defrule how-long-has-nav-align-been-running
  ?sn <- (?status nav-align)
  (not (error-code-phase))
```

Appendix E: CLIPS Source Code

```
=>
;(printout t t "How long has Nav-Align been running? ")
;(assert (nav-align run-time =(read) hours))
(assert (frame dflorian163)) )

;*** Continue running Nav-Align if less than 7 hours have been run,
;*** if less than 12 hours and rms rate of change is not drastic, or
;*** if greater than 12 hours and rms rate of change is normal.

(defrule continue-nav-align
  ?na <- (nav-align run-time ?num-hours hours)
  ?rm <- (nav-align rms-rate-change ?change)
  (test (or (< ?num-hours 7)
            (and (< ?num-hours 12)
                  (neq ?change drastic) )
            (and (>= ?num-hours 12)
                  (< ?num-hours 16)
                  (eq ?change normal) )))
  (not (error-code-phase))
=>
  (retract ?na ?rm)
  ;(printout t t "Continue Nav-Align." t)
  ;(assert (continue nav-align))
  (assert (frame dflorian164)) )

;*** If the rate of rms change is drastic, Nav-Align should be
;*** stopped and Master Heading should be run again. This does not
;*** happen very much. Let Nav-Align run at least 7 hours.

(defrule rerun-master-heading
  ?na <- (nav-align run-time ?num-hours hours)
  ?rm <- (nav-align rms-rate-change drastic)
  (test (>= ?num-hours 7))
  (not (error-code-phase))
=>
  (retract ?na ?rm)
  ;(printout t t "Stop Nav-Align and rerun Master Heading." t)
  ;(assert (start mode-a ,
```

Appendix E: CLIPS Source Code

```
(assert (frame dflorian165)) )

;*** If RMS is looking real good (i.e., lat and long are correcting),
;*** the operator can run an extended Nav-Align. This extends the
;*** length from 16 hours to 30 hours and it can take the place of a
;*** test. This saves 14 hours of testing. It should be initiated
;*** between the 12th and 15th hours. It will not take affect in the
;*** 16th hour.

(defrule run-extended-nav-align
  ?na <- (nav-align run-time ?num-hours hours)
  ?rm <- (nav-align rms-rate-change exceptional)
  (test (and (>= ?num-hours 12)
             (<= ?num-hours 15) ))
  (not (error-code-phase))
=>
  (retract ?na ?rm)
  ;(printout t t "Initiate an Extended Nav-Align (TCI 0732)." t)
  (assert (running nav 1))
  (assert (nav rms good))
  ;(assert (nav run-time ?num-hours hours))
  (assert (frame dflorian166 Hours: ?num-hours)) )

;*** If RMS looked normal but not good enough to run an extended
;*** nav-align before nav-align finished, start Nav test. Nav will
;*** automatically start after Nav-Align finishes unless the
;*** operator intervenes.

(defrule nav-align-complete
  ?na <- (nav-align run-time 16 hours)
  ?rm <- (nav-align rms-rate-change ?change)
  (test (or (eq ?change normal)
            (eq ?change exceptional) ))
  (not (error-code-phase))
=>
  (retract ?na ?rm)
  ;(printout t t "IMU had a good Nav-Align. Let Nav test run." t)
  ;(assert (start mode-a))
  (assert (frame dflorian167)) )
```

Appendix E: CLIPS Source Code

```

*****
;
;***
;
;***      Navigation Performance Rules
;***
;
*****

;*** During the Nav run, the technician will be asked if the RMS is
;*** out of spec or good and how long the test has been running. The
;*** variable ?status will either be equal to "start" or "continue."

(defrule get-rms-status
  ?sn <- (?status nav ?test-num)
  (nav run-time ?hours hours)
  (test (and (neq ?status determine-fault)
              (neq ?status running) ))
  (not (error-code-phase))
=>
  (retract ?sn)
  ;(printout t t "Is the RMS out-of-spec or good? ")
  ;(assert (nav rms =(read)))
  (assert (running nav ?test-num))
  (assert (frame dflorian368)) )

(defrule how-long-has-nav-been-running
  (?status nav ?test-num)
  (test (and (neq ?status determine-fault)
              (neq ?status running) ))
  (not (error-code-phase))
=>
  ;(printout t t "How long has Nav " ?test-num " been running (hours)? ")
  ;(assert (nav run-time =(read) hours))
  (assert (frame dflorian168 Test: ?test-num)) )

;*** Continue current Nav run. RMS is not out of spec.

(defrule continue-current-nav-test
  ?nr <- (nav run-time ?num-hours hours)
  ?rm <- (nav rms good)
  ?ts <- (running nav ?test-num)

```

Appendix E: CLIPS Source Code

```
(test (< ?num-hours 30))
(not (error-code-phase))
=>
(retract ?nr ?rm ?ts)
;(printout t t "Continue running Nav " ?test-num "." t)
;(assert (continue nav ?test-num))
(assert (frame dflorian169 Test: ?test-num)) )

;*** If the RMS is within the spec for 30 hours, then the IMU passed
;*** the Nav test.

(defrule imu-passed-nav
  ?nr <- (nav run-time 30 hours)
  ?rm <- (nav rms good)
  ?ts <- (running nav ?test-num)
  (not (error-code-phase))
=>
  (retract ?nr ?rm ?ts)
  ;(printout t t "Enter TCI 0303 and TCI 0057. IMU passed Nav." t)
  ;(assert (start mode-a))
  (assert (frame dflorian170)) )

;*** Perform a Nav reset if it is the first run and the test has not
;*** been running 25 hours. This automatically begins a 2nd Nav run.
;*** Usually it takes a few hours for a reset to occur after the
;*** operator has requested it.

(defrule perform-nav-reset
  ?nr <- (nav run-time ?num-hours hours)
  ?rm <- (nav rms out-of-spec)
  ?ts <- (running nav 1)
  (test (<= ?num-hours 25))
  (not (error-code-phase))
=>
  (retract ?nr ?rm ?ts)
  ;(printout t t "Perform a Nav reset (TCI 0021)." t)
  ;(assert (start nav 2))
  (assert (frame dflorian171)) )

;*** If after the 25th hour in runs one or two when the RMS goes out
```

Appendix E: CLIPS Source Code

```
*** of spec, let the run complete and start the next run. Usually,  
*** a reset will not have enough time to take effect.
```

```
(defrule let-next-nav-test-start  
  ?nr <- (nav run-time ?num-hours hours)  
  ?rm <- (nav rms out-of-spec)  
  ?ts <- (running nav ?test-num)  
  (test (and (!= ?test-num 3)  
             (> ?num-hours 25) ))  
  (not (error-code-phase))  
=>  
  (retract ?nr ?rm ?ts)  
  ;(printout t t "Continue into next Nav test. Start plotting then." t)  
  ;(assert (start nav =(+ ?test-num 1)))  
  (assert (frame dflorian172 Test: =(+ ?test-num 1))) )
```

```
*** Stop the Nav run and diagnose the fault. If the RMS is out of  
*** spec at any time in the 3rd run or after the 25th hour of the  
*** 2nd run, stop nav and begin to troubleshoot.
```

```
(defrule stop-nav-test  
  ?nr <- (nav run-time ?num-hours hours)  
  ?rm <- (nav rms out-of-spec)  
  ?ts <- (running nav ?test-num)  
  (test (or (eq ?test-num 3)  
            (and (eq ?test-num 2)  
                  (<= ?num-hours 25) )))  
  (not (error-code-phase))  
=>  
  (retract ?nr ?rm ?ts)  
  ;(printout t t "Stop Nav test." t)  
  ;(assert (determine-fault nav))  
  (assert (frame dflorian173)) )
```

```
*** If the RMS is out of spec either the velocity parameters  
*** have increased suddenly (i.e., VDIF >= .05 for at least half  
*** hour) or there was a gradual change in lat or long.
```

```
(defrule determine-fault-in-nav
```

Appendix E: CLIPS Source Code

```
?pf <- (determine-fault nav)
(not (error-code-phase))
=>
(retract ?pf)
;(printout t t "Was there a sudden change in velocity parameters (VDIF)")
;(printout t t "or a gradual drift in lat, long, or both (sudden-change or gradual-drift)? ")
;(assert (problem-in nav is =(read)))
(assert (frame dflorian174)) )

;*** If a sudden change in the velocity parameters occurred, use the
;*** rules found in Master Heading.

;*** If a gradual change in lat or long, then a problem exists in
;*** one of the gyros.

(defrule what-effected-rms
  ?pi <- (problem-in nav is gradual-drift)
  (not (error-code-phase))
=>
  (retract ?pi)
  ;(printout t t "What caused the RMS to go out of spec (lat, long, both)? ")
  ;(assert (gradual-drift =(read)))
  (assert (frame dflorian175)) )

;*** If the gradual drift was in longitude, then there is a
;*** problem in the xy gyro.

(defrule gradual-drift-longitude
  ?gd <- (gradual-drift longitude)
  (not (error-code-phase))
=>
  (retract ?gd)
  (assert (faulty xy-gyro)) )

;*** If the gradual drift was in latitude, then there is a
;*** problem in the yz gyro.

(defrule gradual-drift-latitude
  ?gd <- (gradual-drift latitude)
```

Appendix E: CLIPS Source Code

```
(not (error-code-phase))
=>
(retract ?gd)
(assert (faulty yz-gyro)) )

;*** If the gradual drift is in both lat and long, find out in what
;*** direction.

(defrule gradual-drift-lat-and-long
  ?gd <- (gradual-drift both)
  (not (error-code-phase))
=>
  (retract ?gd)
  ;(printout t t "Did latitude and longitude BOTH drift in the positive, negative, or other? ")
  ;(assert (lat-long-drift =(read)))
  (assert (frame dflorian176)) )

;*** If both latitude and longitude drifted in the positive
;*** direction, then there is a problem with the xy gyro.

(defrule positive-lat-long-drift
  ?ll <- (lat-long-drift positive)
  (not (error-code-phase))
=>
  (retract ?ll)
  (assert (faulty xy-gyro)) )

;*** If latitude and longitude both drifted in the negative
;*** direction, there is a problem with the yz gyro.

(defrule negative-lat-long-drift
  ?ll <- (lat-long-drift negative)
  (not (error-code-phase))
=>
  (retract ?ll)
  (assert (faulty yz-gyro)) )

;*** If latitude and longitude drifted in different directions, use
;*** the Master Heading rules concerning axis off heading.
```

Appendix E: CLIPS Source Code

```

(defrule lat-long-drift-different
  ?ll <- (lat-long-drift other)
  (not (error-code-phase))
=>
  (retract ?ll)
  (assert (gradual-drift other)) )

;*****
;***
;***          Identifies Faulty Component
;***
;*****

(defrule print-faulty-component ""
  ?fc <- (faulty ?component)
=>
  (retract ?fc)
  ;(printout t t "Recommend the " ?component " be replaced." t)
  (assert (frame dflorian177 Component: ?component)) )

;*****
;***
;***          Facts to Relate Parameter to Component
;***
;*****

(deffacts x-vm-parameters
  (kx x-vm parameter)
  (kxd x-vm parameter)
  (ax x-vm parameter)
  (axd x-vm parameter)
  (kxy x-vm parameter)
  (kxyd x-vm parameter) )

(deffacts y-vm-parameters
  (ky y-vm parameter)
  (kyd y-vm parameter)
  (ay y-vm parameter)
  (ayd y-vm parameter)

```

Appendix E: CLIPS Source Code

```
(kylx y-vm parameter)
(kyxd y-vm parameter)
(kyz y-vm parameter)
(kyxd y-vm parameter) )

(deffacts xy-gyro-parameters
  (sx xy-gyro parameter)
  (sxd xy-gyro parameter)
  (syx xy-gyro parameter)
  (syxd xy-gyro parameter)
  (szx xy-gyro parameter)
  (szxd xy-gyro parameter)
  (sxy xy-gyro parameter)
  (sxyd xy-gyro parameter)
  (sxz xy-gyro parameter)
  (sxzd xy-gyro parameter)
  (bx xy-gyro parameter)
  (bx d xy-gyro parameter) )

(deffacts yz-gyro-parameters
  (sz yz-gyro parameter)
  (szd yz-gyro parameter)
  (bz yz-gyro parameter)
  (bzd yz-gyro parameter)
  (sy yz-gyro parameter)
  (syd yz-gyro parameter)
  (by yz-gyro parameter)
  (byd yz-gyro parameter)
  (szy yz-gyro parameter)
  (szyd yz-gyro parameter)
  (syz yz-gyro parameter)
  (syzd yz-gyro parameter) )

;*** Rules to communicate with KMS. The first rule will display
;*** one frame in Large mode. The second rule will display two
;*** frames on the screen. The third rule will display one
;*** frame in Large mode and also change a value of an attribute
;*** on that frame.
```

Appendix E: CLIPS Source Code

```
(defrule write-one-frame-name "writes filename to a file"
  ?fn <- (frame ?filename)
=>
  (retract ?fn)
  (open "clipsinfo" outfile1 "w")
  (printout outfile1 ?filename)
  (close)
  (assert (kms can read file)) )

(defrule write-two-frame-names "writes two filenames to a file"
  ?fn <- (frame ?frame1 ?frame2)
=>
  (retract ?fn)
  (open "clipsinfo" outfile1 "w")
  (printout outfile1 ?frame1 t)
  (printout outfile1 ?frame2)
  (close)
  (assert (kms can read file)) )

(defrule write-one-frame-name-change-value "writes one filename to a file and changes
an item value"
  ?fn <- (frame ?frame1 ?item ?value $?more-values)
=>
  (retract ?fn)
  (open "clipsinfo" outfile1 "w")
  (printout outfile1 ?frame1 t)
  (printout outfile1 ?item t)
  (bind $?stuff (mv-append ?value $?more-values))
  (printout outfile1 $?stuff)
  (close)
  (assert (kms can read file)) )

(defrule ok-for-kms-to-read
  ?kc <- (kms can read file)
=>
  (retract ?kc)
  (open "ok_kms" outfile2 "w")
  (printout outfile2 1)
  (close)
  (kms_file_ready)
```

Appendix E: CLIPS Source Code

```
(get_kms_info) )

;*****
;***
;*** Mode A Error Code rules.
;***
;*****

;*** Error Code Phase Rules written by Florian

;*** This rule obtains the error message from the user.

(defrule get-error-message
  ?se <- (start error-code-phase)
=>
  (retract ?se)
  ; (printout t t "Enter error message: ")
  (assert (frame dflorian16)) )

;*** Talking to Jim Neri, he said only a few error messages
;*** either stop the current test automatically, or would
;*** cause him to manually stop the test. This rule takes this
;*** into account and provides the transition to/from the error
;*** code phase (modified Skinner's code) and parameter phase.

(defrule stop-parameter-phase
  (error-message ?error)
  (test (or (eq ?error velocity-unreasonable)
            (eq ?error vt-greater-than-2-knots)
            (eq ?error imu-major)
            (eq ?error no-input-3-axes)
            (eq ?error no-input-az)
            (eq ?error no-input-pitch)
            (eq ?error no-input-roll)
            (eq ?error automatic-shutdown)
            (eq ?error imu-o-temp)
            (eq ?error excess-angle)
            (eq ?error servo-disable)
            (eq ?error major-reset-fault))
```

Appendix E: CLIPS Source Code

```
(eq ?error minor-reset-fault)
(eq ?error z-stab)
(eq ?error plat-stab-abort)
(eq ?error gyro-hot)
(eq ?error vt-vr-greater-than-3-knots) ))
=>
(assert (error-code-phase)) )

(defrule continue-parameter-phase
  ?em <- (error-message ?error)
  (test (and (neq ?error velocity-unreasonable)
    (neq ?error vt-greater-than-2-knots)
    (neq ?error imu-major)
    (neq ?error no-input-3-axes)
    (neq ?error no-input-az)
    (neq ?error no-input-pitch)
    (neq ?error no-input-roll)
    (neq ?error automatic-shutdown)
    (neq ?error imu-o-temp)
    (neq ?error excess-angle)
    (neq ?error servo-disable)
    (neq ?error major-reset-fault)
    (neq ?error minor-reset-fault)
    (neq ?error z-stab)
    (neq ?error plat-stab-abort)
    (neq ?error gyro-hot)
    (neq ?error vt-vr-greater-than-3-knots) ))
  =>
  (retract ?em)
  ;(printout t t "Disregard this error code.")
  ;(printout t t "Wait for test to end and look at calibration parameters.")
  (assert (frame dflorian385)) )

;*** The following rules are converted Capt Skinner rules. Only
;*** the error messages which either automatically stop the test
;*** in progress, or those which are severe enough for the
;*** technician to manually stop the test are included here.
;*** Capt Skinner's rules for the other error messages are
;*** commented out and included at the end this code.
```

Appendix E: CLIPS Source Code

```
(defrule rule001 "rule001"
  ?ni <- (not-indication-of-fault)
  (error-code-phase)
=>
  (assert (frame dflorian27)))

(defrule rule005 "rule005"
  ?ct <- (contact te)
  (error-code-phase)
=>
  (assert (frame dflorian31)))

(defrule rule007 "rule007"
  ?mt <- (move to mode-b)
  (error-code-phase)
=>
  (assert (frame dflorian33)))

(defrule rule008 "rule008"
  ?ff <- (fault-found ?component continue-testing)
; ?fc <- (faulty-component ?component)
  (error-code-phase)
=>
  (assert (frame dflorian34 Component: ?component)))

(defrule rule009 "rule009"
  ?re <- (rare-error-message)
  (error-code-phase)
=>
  (assert (frame dflorian35)))

;*** Of these error messages, only imu-o-temp, major-reset-fault,
;*** and vt-vr-greater-than-3-knots are used in my project. The
;*** rest of them does not cause the current Mode A test to stop.
;*** They were left in to keep the rule as Capt Skinner wrote it.

(defrule rule013 "rule013"
  ?em <- (error message ?error)
  (test (or (eq ?error imu-o-load)
```

Appendix E: CLIPS Source Code

```
(eq ?error imu-o-temp)
(eq ?error dcc-o-load-o-temp)
(eq ?error dcc-o-temp)
(eq ?error comp-tie-in-sw-on)
(eq ?error i-c-fault-inhb-enab)
(eq ?error i-c-data-loop-fault)
(eq ?error i-c-fault-cont)
(eq ?error in-parity-test-inhb-enab)
(eq ?error out-word-par-inhb-enab)
(eq ?error output-word-parity-cont)
(eq ?error major-reset-fault)
(eq ?error minor-reset-fault)
(eq ?error minor-fault-cont)
(eq ?error both-vm-precounter-failure)
(eq ?error vm-bite-failure)
(eq ?error vt-vr-greater-than-3-knots)
(eq ?error minisins-vel-dif-exceeds-limit)
(eq ?error minisins-pos-dif-exceeds-limit)
(eq ?error parity-test-1-no-go)
(eq ?error parity-test-2-no-go)
(eq ?error parity-test-3-no-go)
(eq ?error put-intercom-test-no-go)
(eq ?error seq-cnt-no-compare) ))
(error-code-phase)
=>
(assert (rare-error-message)) )

;*** This group of rules diagnoses the following four error
;*** messages: no.input.3.axes, no.input.az, no.input.pitch,
;*** and no.input.roll.

(defrule rule014 "rule014"
  ?em <- (error-message ?error)
  (test (or (eq ?error no-input-3-axes)
            (eq ?error no-input-az)
            (eq ?error no-input-pitch)
            (eq ?error no-input-roll) ))
  (error-code-phase)
=>
```

Appendix E: CLIPS Source Code

```
(assert (check resolver-signal-at-ncc)) )

(defrule attribute-resolver-output-at-ncc-good "rules: 15, 16"
  ?cr <- (check resolver-signal-at-ncc)
  (error-code-phase)
=>
  (assert (frame dflorian14 dflorian127)) )

(defrule rule015 "rule015"
  ?cr <- (check resolver-signal-at-ncc)
  ?rs <- (resolver-signal-at-ncc good)
  (error-code-phase)
=>
  (assert (contact te)) )

(defrule rule016
  ?cr <- (check resolver-signal-at-ncc)
  ?rs <- (resolver-signal-at-ncc bad)
  (error-code-phase)
=>
  (assert (check resolver-signal-at-imu)) )

(defrule attribute-imu-resolver-signal-good "rules: 17, 18"
  ?cr <- (check resolver-signal-at-imu)
  (error-code-phase)
=>
  (assert (frame dflorian36 dflorian124)) )

(defrule rule017 "rule017"
  ?cr <- (check resolver-signal-at-imu)
  ?ir <- (imu-resolver-signal good)
  (error-code-phase)
=>
  (assert (check resolver-signal-on-different-station)) )

(defrule rule018 "rule018"
  ?cr <- (check resolver-signal-at-imu)
  ?ir <- (imu-resolver-signal bad)
  (error-code-phase)
=>
```

Appendix E: CLIPS Source Code

```
(assert (check resolver-excitation)) )

(defrule attribute-resolver-signal-on-different-station-good "rules: 19, 20"
  ?cr <- (check resolver-signal-on-different-station)
  (error-code-phase)
=>
  (assert (frame dflorian37 dflorian124)) )

(defrule rule019 "rule019"
  ?cr <- (check resolver-signal-on-different-station)
  ?rs <- (resolver-signal-on-different-station good)
  (error-code-phase)
=>
  (assert (contact te)) )

(defrule rule020 "rule020"
  ?cr <- (check resolver-signal-on-different-station)
  ?rs <- (resolver-signal-on-different-station bad)
  (error-code-phase)
=>
  (assert (check resolver-excitation)) )

(defrule attribute-resolver-excitation-good "rules: 21, 22"
  ?cr <- (check resolver-excitation)
  (error-code-phase)
=>
  (assert (frame dflorian38 dflorian127)) )

(defrule rule021 "rule021"
  ?cr <- (check resolver-excitation)
  ?re <- (resolver-excitation good)
  (error-code-phase)
=>
  (assert (faulty resolver)) )

(defrule rule022 "rule022"
  ?cr <- (check resolver-excitation)
  ?re <- (resolver-excitation bad)
  (error-code-phase)
```

Appendix E: CLIPS Source Code

```
=>
(assert (faulty excitation-module)) )

;*** This group of rules troubleshoots the following two error
;*** messages: velocity.unreasonable and
;*** vt.greater.than.2.knots. In addition, this section uses
;*** the routine Check Gyro Circuit, which starts with rule 043.

(defrule attribute-test "rules: 30-32"
  ?em <- (error-message ?error)
  (test (or (eq ?error velocity-unreasonable)
            (eq ?error vt-greater-than-2-knots) ))
  (error-code-phase)
=>
(assert (frame dflorian45)) )

(defrule rule030 "rule030"
  ?em <- (error-message ?error)
  ?gt <- (gyro-cal test)
  (test (or (eq ?error velocity-unreasonable)
            (eq ?error vt-greater-than-2-knots) ))
  (error-code-phase)
=>
(assert (check position)) )

(defrule rule031 "rule031"
  ?em <- (error-message ?error)
  ?nt <- (navigate test)
  (test (or (eq ?error velocity-unreasonable)
            (eq ?error vt-greater-than-2-knots) ))
  (error-code-phase)
=>
(assert (check velocity-direction)) )

(defrule rule032 "rule032"
  ?em <- (error-message ?error)
  ?ts <- (?test test)
  (test (or (eq ?error velocity-unreasonable)
            (eq ?error vt-greater-than-2-knots) ))
```

Appendix E: CLIPS Source Code

```
(test (or (eq ?test shim-cal)
          (eq ?test nav-align)
          (eq ?test master-heading) ))
(error-code-phase)
=>
(assert (frame dflorian46)) )

(defrule attribute-north-south-or-east-west "rules: 33-38"
  ?cp <- (check ?what)
  (test (or (eq ?what position)
            (eq ?what velocity-direction) ))
  (error-code-phase)
=>
(assert (frame dflorian47 dflorian127)) )

(defrule rule033 "rule033"
  ?cp <- (check position)
  ?ns <- (north-south-or-east-west ?dir)
  (test (or (eq ?dir east-west)
            (eq ?dir north-south) ))
  (error-code-phase)
=>
(assert (wait-for platform-to-slew)) )

(defrule attribute-velocities-changed-directions "rules: 34, 35"
  ?wf <- (wait-for platform-to-slew)
  (error-code-phase)
=>
(assert (frame dflorian48 dflorian124)) )

(defrule rule034 "rule034"
  ?wf <- (wait-for platform-to-slew)
  ?vc <- (velocities-changed-directions? yes)
  (error-code-phase)
=>
(assert (faulty corresponding-velocity-meter)) )

;*** Rules 35-38 were used to transition to deep reasoning. The
;*** following four rules were written by Florian using a
;*** decision tree diagram developed by Skinner and Newark.
```

Appendix E: CLIPS Source Code

```
(defrule velocities-did-not-change-direction
  ?wf <- (wait-for platform-to-slew)
  ?vc <- (velocities-changed-directions? no)
  (error-code-phase)
=>
  (assert (check gyro-circuit)) )

(defrule velocities-other-check-position
  ?cp <- (check position)
  ?ns <- (north-south-or-east-west other)
  (error-code-phase)
=>
  (assert (check gyro-circuit)) )

(defrule check-direction-velocities-not-other
  ?cp <- (check velocity-direction)
  ?ns <- (north-south-or-east-west ?dir)
  (test (or (eq ?dir east-west)
            (eq ?dir north-south) ))
  (error-code-phase)
=>
  (assert (check vm-signal-on-ncc)) )

(defrule check-direction-velocities-other
  ?cp <- (check velocity-direction)
  ?ns <- (north-south-or-east-west other)
  (error-code-phase)
=>
  (assert (check-for level-platform)) )

(defrule attribute-vm-signal-on-ncc-good "rules: 84, 85"
  ?cv <- (check vm-signal-on-ncc)
  (error-code-phase)
=>
  (assert (frame dflorian73 dflorian127)) )

;*** Next 2 rules were originally used by Skinner for different
;*** error messages. I could use them here.
```

Appendix E: CLIPS Source Code

```
(defrule rule084 "rule084"
  ?cv <- (check vm-signal-on-ncc)
  ?vm <- (vm-signal-on-ncc good)
  (error-code-phase)
=>
  (assert (move to mode-b)) )

(defrule rule085 "rule085"
  ?cv <- (check vm-signal-on-ncc)
  ?vm <- (vm-signal-on-ncc bad)
  ;?sv <- (suspect-vm ?suspect)
  (error-code-phase)
=>
  (assert (faulty corresponding-velocity-meter)) )

(defrule attribute-azimuth-equal-zero "rules: 39, 41"
  ?cf <- (check-for level-platform)
  (error-code-phase)
=>
  (assert (frame dflorian49 dflorian124)) )

(defrule platform-is-level
  ?cp <- (check-for level-platform)
  ?ae <- (azimuth-equal-zero? yes)
  (error-code-phase)
=>
  (assert (check gyro-circuit)) )

(defrule rule039 "rule039"
  ?cf <- (check-for level-platform)
  ?ae <- (azimuth-equal-zero? no)
  (error-code-phase)
=>
  (assert (check velocity-meter)) )

(defrule attribute-faulty-velocity-meter "rules: 40, 42"
  ?cv <- (check velocity-meter)
  (error-code-phase)
=>
```

Appendix E: CLIPS Source Code

```
(assert (frame dflorian51 dflorian127)) )

(defrule rule040 "rule040"
  ?cv <- (check velocity-meter)
  ?fv <- (faulty-velocity-meter? yes)
  (error-code-phase)
=>
  (assert (faulty corresponding-velocity-meter)) )

(defrule rule042 "rule042"
  ?cv <- (check velocity-meter)
  ?fv <- (faulty-velocity-meter? no)
  (error-code-phase)
=>
  (assert (check gyro-circuit)) )

;*** CHECK GYRO CIRCUIT ROUTINE
;*** This routine is used for several different error messages
;*** including velocity unreasonable, vt. greater than 2 knots,
;*** imu major, and plat stab abort.

(defrule check-gyro-circuit
  ?cg <- (check gyro-circuit)
=>
  (assert (frame dflorian383 dflorian124)) )

(defrule rule043 "rule043"
  ?cg <- (check gyro-circuit)
  ?sc <- (speed-control-signal good)
  (error-code-phase)
=>
  (assert (check gyro-run-voltage)) )

(defrule attribute-gyro-run-voltage-good "rules: 44, 51"
  ?cg <- (check gyro-run-voltage)
  (error-code-phase)
=>
  (assert (frame dflorian52 dflorian127)) )
```

Appendix E: CLIPS Source Code

```
(defrule rule044 "rule044"
  ?cg <- (check gyro-run-voltage)
  ?gr <- (gyro-run-voltage good)
  (error-code-phase)
=>
  (assert (check pick-off-signals)) )

(defrule attribute-pick-off-signals-good "rules: 45, 46"
  ?cp <- (check pick-off-signals)
  (error-code-phase)
=>
  (assert (frame dflorian53 dflorian124)) )

(defrule rule045 "rule045"
  ?cp <- (check pick-off-signals)
  ?ps <- (pick-off-signals good)
  (error-code-phase)
=>
  (assert (check pick-off-signals-while-gyros-not-running)) )

(defrule rule046 "rule046"
  ?cp <- (check pick-off-signals)
  ?ps <- (pick-off-signals bad)
  (error-code-phase)
=>
  (assert (faulty corresponding-gyro)) )

(defrule attribute-pick-off-signals-while-gyros-not-running-good "rules: 47, 48"
  ?cp <- (check pick-off-signals-while-gyros-not-running)
  (error-code-phase)
=>
  (assert (frame dflorian54 dflorian127)) )

(defrule rule047 "rule047"
  ?cp <- (check pick-off-signals-while-gyros-not-running)
  ?ps <- (pick-off-signals-while-gyros-not-running bad)
  (error-code-phase)
=>
  (assert (faulty corresponding-gyro)) )
```

Appendix E: CLIPS Source Code

```
(defrule rule048 "rule048"
  ?cp <- (check pick-off-signals-while-gyros-not-running)
  ?ps <- (pick-off-signals-while-gyros-not-running good)
  (error-code-phase)
=>
  (assert (monitor speed-control-for-b14-and-up)) )

(defrule attribute-speed-control-for-b14-and-up-good "rules: 49, 50"
  ?ms <- (monitor speed-control-for-b14-and-up)
  (error-code-phase)
=>
  (assert (frame dflorian55 dflorian124)) )

(defrule rule049 "rule049"
  ?ms <- (monitor speed-control-for-b14-and-up)
  ?sc <- (speed-control-for-b14-and-up good)
  (error-code-phase)
=>
  (assert (run next-test)) )

(defrule rule050 "rule050"
  ?ms <- (monitor speed-control-for-b14-and-up)
  ?sc <- (speed-control-for-b14-and-up bad)
  (error-code-phase)
=>
  (assert (faulty corresponding-gyro)) )

(defrule rule051 "rule051"
  ?cg <- (check gyro-run-voltage)
  ?gr <- (gyro-run-voltage bad)
  (error-code-phase)
=>
  (assert (interchange ps1-and-ps2)) )

(defrule attribute-problem-follows-power-supply-card "rules: 52, 53"
  ?ip <- (interchange ps1-and-ps2)
  (error-code-phase)
=>
  (assert (frame dflorian56)) )
```

Appendix E: CLIPS Source Code

```
(defrule rule052 "rule052"
  ?ip <- (interchange ps1-and-ps2)
  ?pf <- (problem-follows-power-supply-card? yes)
  (error-code-phase)
=>
  (assert (faulty corresponding-power-supply-card)) )
```

```
(defrule rule053 "rule053"
  ?ip <- (interchange ps1-and-ps2)
  ?pf <- (problem-follows-power-supply-card? no)
  (error-code-phase)
=>
  (assert (faulty corresponding-gryo)) )
```

```
(defrule rule054 "rule054"
  ?cg <- (check gyro-circuit)
  ?sc <- (speed-control-signal bad)
  (error-code-phase)
=>
  (assert (interchange speed-control-cards)) )
```

```
(defrule problem-follows-speed-control-cards
  ?is <- (interchange speed-control-cards)
=>
  (assert (frame dflorian384)) )
```

```
(defrule rule055 "rule055"
  ?is <- (interchange speed-control-cards)
  ?pf <- (problem-follows-speed-control-card? no)
  (error-code-phase)
=>
  (assert (faulty corresponding-gyro)) )
```

```
(defrule rule056 "rule056"
  ?is <- (interchange speed-control-cards)
  ?pf <- (problem-follows-speed-control-card? yes)
  (error-code-phase)
=>
```

Appendix E: CLIPS Source Code

```
(assert (faulty corresponding-speed-card)) )

(defrule attribute-next-test-good "rules: 57, 58"
  ?rn <- (run next-test)
  (error-code-phase)
=>
  (assert (frame dflorian57)) )

(defrule rule057 "rule057"
  ?rn <- (run next-test)
  ?nt <- (next-test bad)
  (error-code-phase)
=>
  (assert (check speed-stability)) )

(defrule rule058 "rule058"
  ?rn <- (run next-test)
  ?nt <- (next-test good)
  (error-code-phase)
=>
  (assert (run scorsby-test)) )

(defrule attribute-scorsby-good "rules: 59, 60"
  ?rs <- (run scorsby-test)
  (error-code-phase)
=>
  (assert (frame dflorian59)) )

(defrule rule059 "rule059"
  ?rs <- (run scorsby-test)
  ?st <- (scorsby-test bad)
  (error-code-phase)
=>
  (assert (check speed-stability)) )

(defrule rule060 "rule060"
  ?rs <- (run scorsby-test)
  ?st <- (scorsby-test good)
  (error-code-phase)
```

Appendix E: CLIPS Source Code

```
=>
(assert (frame dflorian60)) )

;*** This group of rules corresponds to an imu major error
;*** message. This section also uses the Check Gyro Circuit
;*** Routine which starts with rule 043.

(defrule attribute-previous-imu-major "rules: 61, 62"
  ?em <- (error-message imu-major)
  (error-code-phase)
=>
  (assert (frame dflorian61)) )

(defrule rule061 "rule061"
  ?em <- (error-message imu-major)
  ?pi <- (previous-imu-major? no)
  (error-code-phase)
=>
  (assert (check gyro-circuit)) )

(defrule rule062 "rule062"
  ?em <- (error-message imu-major)
  ?pi <- (previous-imu-major? yes)
  (error-code-phase)
=>
  (assert (check speed-stability)) )

(defrule attribute-yz-speed-control-stable "rules 63, 64"
  ?cs <- (check speed-stability)
  (error-code-phase)
=>
  (assert (frame dflorian131 dflorian127)) )

(defrule rule063 "rule063"
  ?cs <- (check speed-stability)
  ?ys <- (yz-speed-control stable)
  (error-code-phase)
=>
  (assert (faulty displacement-gyroscope-xy)) )
```

Appendix E: CLIPS Source Code

```
(defrule rule064 "rule064"
  ?cs <- (check speed-stability)
  ?ys <- (yz-speed-control unstable)
  (error-code-phase)
=>
  (assert (faulty displacement-gyroscope-yz)) )

;*** This group of rules corresponds to the following error
;*** messages: automatic.shutdown and pwr.interrupt. The
;*** latter error is an error which does not stop the current
;*** Mode A test.

(defrule attribute-operator-initiated-shutdown "rules: 65, 67"
  ?em <- (error-message ?error)
  (test (or (eq ?error automatic-shutdown)
            (eq ?error pwr-interrupt) ))
  (error-code-phase)
=>
  (assert (frame dflorian62)) )

(defrule rule065 "rule065"
  ?em <- (error-message ?error)
  ?oi <- (operator-initiated-shutdown? yes)
  (test (or (eq ?error automatic-shutdown)
            (eq ?error pwr-interrupt) ))
  (error-code-phase)
=>
  (assert (normal-response to-operator-action)) )

(defrule rule066 "rule066"
  ?nr <- (normal-response to-operator-action)
  (error-code-phase)
=>
  (assert (frame dflorian63)) )

(defrule rule067 "rule067"
  ?em <- (error-message ?error)
  ?oi <- (operator-initiated-shutdown? no)
  (test (or (eq ?error automatic-shutdown)
```

Appendix E: CLIPS Source Code

```
(eq ?error pwr-interrupt) ))
(error-code-phase)
=>
(assert (attempt restart)) )

(defrule attribute-restart-possible "rules: 68, 69"
  ?ar <- (attempt restart)
  (error-code-phase)
=>
  (assert (frame dflorian64)) )

(defrule rule068 "rule068"
  ?ar <- (attempt restart)
  ?rp <- (restart-possible? no)
  (error-code-phase)
=>
  (assert (check power-cube)) )

(defrule rule069 "rule069"
  ?ar <- (attempt restart)
  ?rp <- (restart-possible? yes)
  (error-code-phase)
=>
  (assert (frame dflorian65 dflorian127)) )

(defrule attribute-power-cube-good "rules: 70, 71"
  ?cp <- (check power-cube)
  (error-code-phase)
=>
  (assert (frame dflorian66)) )

(defrule rule070 "rule070"
  ?cp <- (check power-cube)
  ?pc <- (power-cube good)
  (error-code-phase)
=>
  (assert (contact te)) )

(defrule rule071 "rule071"
  ?cp <- (check power-cube)
```

Appendix E: CLIPS Source Code

```
?pc <- (power-cube bad)
(error-code-phase)
=>
(assert (faulty power-cube)) )

;*** This rule corresponds to plat stab abort error. It also
;*** uses the Check Gyro Circuit Routine which starts with
;*** rule 043.

(defrule rule077 "rule077"
  ?em <- (error-message plat-stab-abort)
  (error-code-phase)
=>
  (assert (check gyro-circuit)) )

;*** This group of rules corresponds to the gyro hot error
;*** message.

(defrule attribute-cooling-hose-is-connected "rules: 95"
  ?em <- (error-message gyro-hot)
  (error-code-phase)
=>
  (assert (frame dflorian80)) )

(defrule rule095 "rule095"
  ?em <- (error-message gyro-hot)
  ?ch <- (cooling-hose not-connected)
  (error-code-phase)
=>
  (assert (frame dflorian81)) )

(defrule rule096 "rule096"
  ?em <- (error-message gyro-hot)
  ?ch <- (cooling-hose connected)
  (error-code-phase)
=>
  (assert (check meter-m2-position-21)) )

(defrule attribute-position-21-setting-is-normal "rules: 97, 98"
```

Appendix E: CLIPS Source Code

```
?cm <- (check meter-m2-position-21)
(error-code-phase)
=>
(assert (frame dflorian82 dflorian124)) )

(defrule rule097 "rule097"
  ?cm <- (check meter-m2-position-21)
  ?p2 <- (position-21-setting-is-normal? no)
  (error-code-phase)
=>
(assert (move to mode-b)) )

(defrule rule098 "rule098"
  ?cm <- (check meter-m2-position-21)
  ?p2 <- (position-21-setting-is-normal? yes)
  (error-code-phase)
=>
(assert (check meter-m2-position-11)) )

(defrule attribute-position-11-setting-is-normal "rules: 99, 100"
  ?cm <- (check meter-m2-position-11)
  (error-code-phase)
=>
(assert (frame dflorian83 dflorian127)) )

(defrule rule099 "rule099"
  ?cm <- (check meter-m2-position-11)
  ?p1 <- (position-11-setting-is-normal? no)
  (error-code-phase)
=>
(assert (frame dflorian84)) )

(defrule rule100 "rule100"
  ?cm <- (check meter-m2-position-11)
  ?p1 <- (position-11-setting-is-normal? yes)
  (error-code-phase)
=>
(assert (frame dflorian85)) )
```

Appendix E: CLIPS Source Code

*** This group of rules corresponds to z stab error message.

```
(defrule rule108 "rule108"
```

```
  ?em <- (error-message z-stab)
```

```
  (error-code-phase)
```

```
=>
```

```
  (assert (check a-to-d-converter-z-stab)) )
```

```
(defrule attribute-number-of-axes-not-zero "rules: 109, 112"
```

```
  ?ca <- (check a-to-d-converter-z-stab)
```

```
  (error-code-phase)
```

```
=>
```

```
  (assert (frame dflorian89 dflorian127)) )
```

```
(defrule rule109 "rule109"
```

```
  ?ca <- (check a-to-d-converter-z-stab)
```

```
  ?no <- (number-of-axes-not-zero all)
```

```
  (error-code-phase)
```

```
=>
```

```
  (assert (check case-rotation)) )
```

```
(defrule attribute-case-rotation-normal "rules: 110, 111"
```

```
  ?cc <- (check case-rotation)
```

```
  (error-code-phase)
```

```
=>
```

```
  (assert (frame dflorian90 dflorian124)) )
```

```
(defrule rule110 "rule110"
```

```
  ?cc <- (check case-rotation)
```

```
  ?cr <- (case-rotation normal)
```

```
  (error-code-phase)
```

```
=>
```

```
  (assert (move to mode-b)) )
```

```
(defrule rule111 "rule111"
```

```
  ?cc <- (check case-rotation)
```

```
  ?cr <- (case-rotation not-normal)
```

```
  (error-code-phase)
```

```
=>
```

```
  (assert (faulty corresponding-gyro)) )
```

Appendix E: CLIPS Source Code

```
(defrule rule112 "rule112"
  ?ca <- (check a-to-d-converter-z-stab)
  ?no <- (number-of-axes-not-zero one)
  (error-code-phase)
=>
  (assert (interchange gimbal-rate-amps)) )

(defrule attribute-problem-follows-gimbal-rate-amp "rules: 113, 114"
  ?ig <- (interchange gimbal-rate-amps)
  (error-code-phase)
=>
  (assert (frame dflorian91)) )

(defrule rule113 "rule113"
  ?ig <- (interchange gimbal-rate-amps)
  ?pf <- (problem-follows-gimbal-rate-amp? yes)
  (error-code-phase)
=>
  (assert (faulty corresponding-gimbal-rate-amp)) )

(defrule rule114 "rule114"
  ?ig <- (interchange gimbal-rate-amps)
  ?pf <- (problem-follows-gimbal-rate-amp? no)
  (error-code-phase)
=>
  (assert (exchange electronic-control-amps)) )

(defrule exchange-electronic-control-amps
  ?ee <- (exchange electronic-control-amps)
=>
  (assert (frame dflorian382)) )

(defrule rule115 "rule115"
  ?ee <- (exchange electronic-control-amps)
  ?pf <- (problem-follows-amp? yes)
  (error-code-phase)
=>
  (assert (faulty corresponding-electronic-control-amp)) )
```

Appendix E: CLIPS Source Code

```
(defrule rule116 "rule116"
  ?ee <- (exchange electronic-control-amps)
  ?pf <- (problem-follows-amp? no)
  (error-code-phase)
=>
  (assert (move to mode-b)) )

;*** This group of rules corresponds to servo disable error
;*** message.

(defrule rule117 "rule117"
  ?em <- (error-message servo-disable)
  (error-code-phase)
=>
  (assert (check a-to-d-converter-servo-disable)) )

(defrule attribute-all-three-axes-near-zero "rules: 118, 119"
  ?ca <- (check a-to-d-converter-servo-disable)
  (error-code-phase)
=>
  (assert (frame dflorian92 dflorian124)) )

(defrule rule118 "rule118"
  ?ca <- (check a-to-d-converter-servo-disable)
  ?at <- (all-three-axes-near-zero? yes)
  (error-code-phase)
=>
  (assert (check test-point-23)) )

(defrule rule119 "rule119"
  ?ca <- (check a-to-d-converter-servo-disable)
  ?at <- (all-three-axes-near-zero? no)
  (error-code-phase)
=>
  (assert (interchange rate-amps)) )

(defrule attribute-problem-follows-board "rules: 120, 121"
  ?ir <- (interchange rate-amps)
```

Appendix E: CLIPS Source Code

```
(error-code-phase)
=>
(assert (frame dflorian93 dflorian127)) )

(defrule rule120 "rule120"
  ?ir <- (interchange rate-amps)
  ?pf <- (problem-follows-board? no)
  (error-code-phase)
=>
  (assert (move to mode-b)) )

(defrule rule121 "rule121"
  ?ir <- (interchange rate-amps)
  ?pf <- (problem-follows-board? yes)
  (error-code-phase)
=>
  (assert (faulty corresponding-gimbal-rate-electronic-control-amp)))

(defrule attribute-power-supply-4.8khz-good "rules: 122, 123"
  ?ct <- (check test-point-23)
  (error-code-phase)
=>
  (assert (frame dflorian94 dflorian124)) )

(defrule rule122 "rule122"
  ?ct <- (check test-point-23)
  ?ps <- (power-supply-4.8khz good)
  (error-code-phase)
=>
  (assert (check test-point-13)) )

(defrule rule123 "rule123"
  ?ct <- (check test-point-23)
  ?ps <- (power-supply-4.8khz bad)
  (error-code-phase)
=>
  (assert (replace power-supply-4.8khz)) )

(defrule attribute-replacing-power-supply-solved-problem "rules: 124, 125"
  ?rp <- (replace power-supply-4.8khz)
```

Appendix E: CLIPS Source Code

```
(error-code-phase)
=>
(assert (frame dflorian95)) )

(defrule rule124 "rule124"
  ?r4 <- (replace power-supply-4.8khz)
  ?np <- (new-power-supply-solved-problem? yes)
  (error-code-phase)
=>
  ;(assert (faulty-component power-supply-4.8khz))
  (assert (fault-found power-supply-4.8khz continue-testing)) )

(defrule rule125 "rule125"
  ?r4 <- (replace power-supply-4.8khz)
  ?np <- (new-power-supply-solved-problem? no)
  (error-code-phase)
=>
  (assert (move to mode-b)) )

(defrule attribute-power-supply-400hz-good "rules: 126, 127"
  ?ct <- (check test-point-13)
  (error-code-phase)
=>
  (assert (frame dflorian96 dflorian124)) )

(defrule rule126 "rule126"
  ?ct <- (check test-point-13)
  ?ps <- (power-supply-400hz good)
  (error-code-phase)
=>
  (assert (frame dflorian97)) )

(defrule rule127 "rule127"
  ?ct <- (check test-point-13)
  ?ps <- (power-supply-400hz bad)
  (error-code-phase)
=>
  (assert (frame dflorian98)) )
```

Appendix E: CLIPS Source Code

```
*** This group of rules corresponds to excess angle error
*** message.
```

```
(defrule attribute-imu-in-caged-mode "rules: 128, 129"
  ?em <- (error-message excess-angle)
  (error-code-phase)
=>
  (assert (frame dflorian99 dflorian127)) )
```

```
(defrule rule128 "rule128"
  ?em <- (error-message excess-angle)
  ?ii <- (imu-in-caged-mode? no)
  (error-code-phase)
=>
  (assert (not-indication-of-fault)) )
```

```
(defrule rule129 "rule129"
  ?em <- (error-message excess-angle)
  ?ii <- (imu-in-caged-mode? yes)
  (error-code-phase)
=>
  (assert (return-to-test)) )
```

```
(defrule attribute-able-to-restart "rules: 130,131"
  ?rt <- (return-to-test)
  (error-code-phase)
=>
  (assert (frame dflorian100 dflorian124)) )
```

```
(defrule rule130 "rule130"
  ?rt <- (return-to-test)
  ?at <- (able-to-restart? no)
  (error-code-phase)
=>
  (assert (move to mode-b)) )
```

```
(defrule rule131 "rule131"
  ?rt <- (return-to-test)
  ?at <- (able-to-restart? yes)
  (error-code-phase)
```

Appendix E: CLIPS Source Code

```
=>
  (assert (monitor pickoffs)) )

(defrule attribute-angle-occurs-first-on "rules: 132, 133, 135, 136"
  ?mp <- (monitor pickoffs)
  (error-code-phase)
=>
  (assert (frame dflorian101 dflorian127))) )

(defrule rule132 "rule132"
  ?mp <- (monitor pickoffs)
  ?ao <- (angle-occurs-first-on xy-pickoff)
  (error-code-phase)
=>
  (assert (replace ar8)) )

(defrule rule133 "rule133"
  ?mp <- (monitor pickoffs)
  ?ao <- (angle-occurs-first-on yz-pickoff)
  (error-code-phase)
=>
  (assert (replace ar8)) )

(defrule attribute-replacing-ar8-helps "rules: 134-136"
  ?ra <- (replace ar8)
  (error-code-phase)
=>
  (assert (frame dflorian102 dflorian124))) )

(defrule rule134 "rule134"
  ?ra <- (replace ar8)
  ?na <- (new-ar8-helps? yes)
  (error-code-phase)
=>
  ;(assert (faulty-component platform-signal-amp))
  (assert (fault-found platform-signal-amp continue-testing)) )

(defrule rule135 "rule135"
  ?ra <- (replace ar8)
```

Appendix E: CLIPS Source Code

```
?na <- (new-ar8-helps? no)
?ao <- (angle-occurs-first-on xy-pickoff)
(error-code-phase)
=>
(assert (frame dflorian103)) )
```

```
(defrule rule136 "rule136"
  ?ra <- (replace ar8)
  ?na <- (new-ar8-helps? no)
  ?ao <- (angle-occurs-first-on yz-pickoff)
  (error-code-phase)
=>
  (assert (frame dflorian104)) )
```

MODE B DIAGNOSIS

*** SELECT FAILED TEST ***

- B1.0 IMU POWER-UP TESTS
- B2.0 DC POWER TESTS
- B3.0 AC POWER TESTS
- B4.0 GYRO TEMP ALARM TESTS
- B5.0 THERMOELECTRIC CONTROL
- B6.0 BITE STATUS CHECKS
- B7.0 BITE OPERATIONS CHECKS
- B8.0 CAGE DISCRETE TESTS
- B9.0 RESOLVER PRESENCE TESTS
- B10.0 test 10

•EXIT MODE B DIAGNOSIS

NOTE: MODE B TESTS
10 THRU 22 ARE NOT
IMPLEMENTED IN THIS
THESIS EFFORT. DATA WAS
UNAVAILABLE.

FAILED TEST: B6.0 BITE STATUS CHECKS

*** SELECT FAILED SUBTEST ***

·B6.1 XY GYRO SPEED CONTROL

·B6.2 YZ GYRO SPEED CONTROL

·B6.3 400HZ

·B6.4 AZIMUTH OVERRATE

·B6.5 SERVO DISABLE

·B6.6 FREE RUN

·B6.7 AZIMUTH CAGE

·EXIT MODE B DIAGNOSIS

RERUN TEST B6.1: XY GYRO SPEED CONTROL

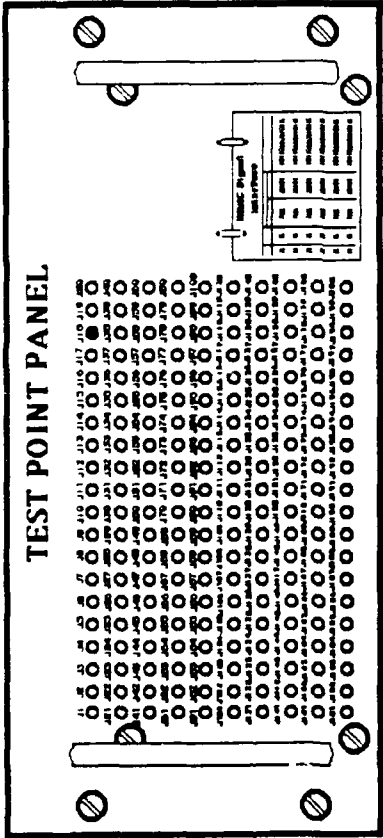
Did test B6.1 pass this time?

·YES

·NO

Troubleshooting Test B6.1

Manually verify test point 18.



IMUIC Signal: J2-F
TO page: 39

The specification is 3.5 to 6 VDC.

Does the signal meet the spec?

☐ YES ☐ NO

Recommend you replace:

D.C. XY GYRO AMPLIFIER (AR1).

NOTE: If AR1 has already been replaced, replace:

BANDPASS FILTER and SHIFT REGISTER (A1).

F-5

***** *CLICK HERE TO CONTINUE* *****

Appendix G: Modified CLIPS Main Routine

This appendix lists the modified CLIPS main routine used in the prototype. CLIPS version 4.20 was used during development.

```

/***** Modified CLIPS main.c routine used in developing the prototype. *****/

#include <stdio.h>
#include <sys/file.h>
#include "clips.h"

#define BUFSIZE 512                                /* max buffer size */

main()                                              /* start of main routine */
{
    init_clips();
    set_watch("all",1);                            /* used when debugging */
    load_rules("dmins.rules");                     /* automatically load rules */
    reset_clips();
    run(-1);                                        /* fire as many rules as possible */
}                                                  /* end of main routine */

int get_kms_info()                                /* routine to read kmsinfo file */
{
    int fd, n;
    char buffer[BUFSIZE];

    fd = open("kmsinfo",O_RDONLY);                 /* open kmsinfo as read only */

    while ((n = read(fd, buffer, BUFSIZE)) > 0)    /* read one line */
        buffer[n] = EOF;                          /* put EOF at end of line read */

    close(fd);                                     /* close kmsinfo */

    system("rm kmsinfo");                          /* delete kmsinfo */

    if (strcmp(buffer,"start mode-a",12) == 0)      /* start another user session */
    {
        reset_clips();
        excise_rule("start-up");                  /* only used once at first start */
        assert(buffer);
    }
}

```

Appendix G (cont.)

```
        run(-1);
    }

    if (strcmp(buffer,"quit",4) != 0)                /* quit CLIPS if session over */
        assert(buffer);

    if (strcmp(buffer,"quit",4) == 0)                /* quit CLIPS if session over */
        clear_clips();

    return;
}

int kms_file_ready()                                /* routine to check ok_clips file */
{
    int is_ready, fd, n;
    char buffer[1];
    char one[1];
    one[0] = '1';
    is_ready = 1;

    while (is_ready != 0)
    {
        system("sleep 1");
        if ((fd = open("ok_clips",O_RDONLY)) != -1) /* wait if file is not ready */
        {                                           /* if file is ready then */
            n = read(fd, buffer, 1);
            is_ready = strcmp(buffer,"1",1);
            close(fd);
        }
        /* read one character */
        /* if = 1 kmsinfo file is ready */
        /* close semaphore file */
    }

    fd = open("ok_clips",O_WRONLY);
    buffer[0] = '0';
    write(fd, buffer, 1);
    close(fd);
    /* if file was ready to read */
    /* set the semaphore file to */
    /* so that it can't be read again */

    return;
}
```

Appendix G (cont.)

```
usrfuncs()                                /* user defined functions */
                                           /* within CLIPS rules */

{
  extern int get_kms_info();
  extern int kms_file_ready();
  define_function("get_kms_info", 'v', get_kms_info, "get_kms_info");
  define_function("kms_file_ready", 'v', kms_file_ready, "kms_file_ready");
}
```

Bibliography

- AI Squared, Inc. Intelligent Diagnostic Expert Assistant (IDEA™) The Field Service Solution. Company brochure, 1989.
- AI Squared, Inc. "Hypertext Help," Company letter, 1989.
- Akscyn, Robert M. and others. "KMS: A Distributed Hypermedia System For Managing Knowledge in Organizations: Communications of the ACM, 31: 820-835 (July 1988).
- Apple Computer, Inc. HyperCard's User Guide. 1987.
- Beeman, William O. and others. Hypertext and Pluralism: From Lineal to Non-lineal Thinking. Hypertext '87, Department of Computer Science, University of North Carolina at Chapel Hill , November 1987, pages 67-88.
- Blais, Curt and others. "Artificial Intelligence to Support Avionics Maintenance Diagnostics: A State-of-the-Art-Assessment," Air Force Human Resources Laboratory. (Jan 1984).
- Brownston, Lee and others. Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming. Reading, MA., Addison-Wesley Publishing Company, Inc., 1985.
- Bush, Vannevar. "As We May Think," The Atlantic Monthly, Volume 176, pages 101-108, July 1945.
- Charney, David. Comprehending non-linear text: The Role of Discourse Cues and Reading Strategies. Hypertext '87, Department of Computer Science, University of North Carolina at Chapel Hill , November 1987, pages 109-120.
- Cohen, Paul R. and Edward A. Feigenbaum. The Handbook of Artificial Intelligence Volume 3: New York: McGraw-Hill Book Company, 1982.
- Conklin, Jeff. "Hypertext: An Introduction and Survey," IEEE Computer, 20: 17-41 (September 1987).

Bibliography (cont.)

- Davis, Kathy and Janet Huebner. Expert Missile Maintenance Aid (EMMA) Phase II Volume 1: Technical Discussion. June 1989.
- Davis, Larry and others. "Expert Computer Systems for Missile Maintenance," Final Report August 1982-August 1983. (AD-A13315).
- Davis, Randall and Walter Hamscher. "Model-based Reasoning: Troubleshooting," Exploring Artificial Intelligence: Survey Talks from the National Conferences on Artificial Intelligence. San Mateo, CA: Morgan Kaufmann Publishers, 1988.
- Delisle, N. and M. Schwartz. "Neptune: A hypertext system for CAD applications," Proceedings of ACM SIGMOD International Conference on Management of Data (Washington DC, May 1986), ACM, New York, 132-143.
- Fikes, Richard and Tom Kehler. The Role of Frame-Based Representation in Reasoning. Communication of the ACM, Volume 29, Number 9, September 1985, pages 904-920.
- Gold Hill Computers, Inc. GoldWorks II Graphics Toolkit User's Guide. February 1989.
- Gordon, Peter J. Human Factors in the Design of User Interfaces and Menus. Unpublished paper. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 1989.
- Gunning, David. Telephone interview. AFHRL/, Wright-Patterson AFB OH, 5 September 1989.
- Halasz, Frank G. "Reflections on NoteCards: Seven Issues For the Next Generation of Hypermedia Systems," Communications of the ACM, 31: 836-852 (July 1988).
- Harmon, Paul and David King. Expert Systems: Artificial Intelligence in Business. New York, NY. John Wiley and Sons, Inc., 1985.
- Harmon, Paul and others. Expert Systems: Tools & Applications. New York, NY. John Wiley and Sons, Inc., 1988.

Bibliography (cont.)

Hayes-Roth, Frederick and others. Building Expert Systems. Reading, MA: Addison-Wesley Publishing Company, 1983.

Hayes-Roth, Frederick. "Rule-Based Systems," Communications of the ACM, Volume 28, Number 9, September 1985, pages 921-932

Hester, Lt. Gina L. A Prototype Fault Diagnosis System for NASA Space Station Power Management and Control. Naval Post Graduate School. September 1988.

IntelliCorp, Inc. KEE User's Guide. May 1988.

Johnson Space Center, Artificial Intelligence Section. CLIPS Reference Manual. April 1988.

Knowledge Gardens Inc. Advertisement for KnowledgePro software. AIExpert, October 1989, page 2.

Knowledge Systems. KMS Action Language Manual. 1989.

McArthur, Capt Tim. Telephone interview. ASD/B1LRE, Wright-Patterson AFB OH, 29 Aug 1989.

Nelson, Thomas H., "The Hypertext," Proceedings International Documentation Federation, 1965.

Nielson, Jakob. "HyperTEXT '87," HyperCard software stack, 1989.

O'Reilly, Daniel and others. "Using Hypermedia to Develop an Intelligent Tutorial/Diagnostic System for the Space Shuttle Main Engine Controller Lab," Marshall Space Flight Center, Fourth Conference on Artificial Intelligence for Space Applications:467-476. (August 1988).

Rasmus, Dan. "HyperX," MacUser, pp. 259-260, January 1989.

Rasmussen, Steven J., "Expert System for Depot Maintenance of the Dual Miniature Inertial Navigation System," 1988 IEEE National Aerospace and Electronics Conference, pp. 1369-1374, May 1988.

Bibliography (cont.)

Saja, Allen D. "The Cognitive Model: An Approach to Designing the Human-Computer Interface," SIGCHI Bulletin, 16:36-39 (January 1986).

Schoen, Seymour and Wendell G. Sykes. Putting Artificial Intelligence to Work: Evaluating & Implementing Business Applications. New York, NY. John Wiley and Sons, Inc., 1987.

Silberschatz, Abraham and James L. Peterson. Operating System Concepts (Alternate Edition). Reading, MA: Addison-Wesley Publishing Company, 1988.

Skinner, Lt James M. A Diagnostic System Blending Deep and Shallow Reasoning. MS thesis, AFIT/GCE/ENG/88D-5. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1988.

Smith, J. H., "WE: A writing environment for professionals," Technical Report 86-025, Department of Computer Science, University of North Carolina at Chapel Hill, August 1986.

Smith, John B. and Stephen F. Weiss. "Hypertext," Communications of the ACM, 31: 820-835 (July 1988).

Somers, Larry. "An Intelligent Knowledge-Based Tutoring System for a Transponder Test Set." Final Report Aug 87-Feb 88. (August 1988).

Stone, David E. and others. "A Hypertext Electronic Job Aid for Maintenance," Final Report, 30 Nov 1982. AD-A133103

Teknowledge, Inc. S.1 Reference Manual. September 1987.

Thomas, Donald L. and Jeffrey D. Clay. "Computer-Based Maintenance Aids For Technicians: Project Final Report," Air Force Human Resources Laboratory, AFHRL-TR-87-44 (August 1988).

Trigg, Randall H. and Peggy M. Irish. "Hypertext Habitats: Experiences of Writers in NoteCards," Hypertext '87, Department of Computer Science, University of North Carolina at Chapel Hill, November 1987.

Bibliography (cont.)

Waterman, Donald A. A Guide to Expert Systems. Reading, MA: Addison-Wesley Publishing Company, 1986.

Winston, Patrick Henry. Artificial Intelligence. Reading, MA. Addison-Wesley Publishing Company, 1977.

Vita

Captain Daniel J. Florian [REDACTED] In 1976,
he moved to St. Louis, Missouri. [REDACTED]

Dan joined the Air Force in November of 1978 and was accepted into the Airman Education and Commissioning Program in September of 1982. He received a Bachelor of Science degree in Computer Science from the University of Missouri at Rolla, Missouri in May 1985 and, after completing Officer Training School, was commissioned as a second lieutenant in the USAF in August 1985. He worked for HQ TAC at Langley AFB, Virginia as an embedded systems project manager until entering the School of Engineering, Air Force Institute of Technology, in May 1988.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED		1b. RESTRICTIVE MARKINGS	
2a. SECURITY CLASSIFICATION AUTHORITY		3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; Distribution unlimited.	
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE		5. MONITORING ORGANIZATION REPORT NUMBER(S)	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GCS/ENG/89D-3		7a. NAME OF MONITORING ORGANIZATION	
6a. NAME OF PERFORMING ORGANIZATION School of Engineering	6b. OFFICE SYMBOL (If applicable) AFIT/ENG	7b. ADDRESS (City, State, and ZIP Code)	
6c. ADDRESS (City, State, and ZIP Code) Air Force Institute of Technology (AU) Wright-Patterson AFB, Ohio 45433-6533		9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
8a. NAME OF FUNDING/SPONSORING ORGANIZATION AI Technology Office	8b. OFFICE SYMBOL (If applicable) WRDC/TXI	10. SOURCE OF FUNDING NUMBERS	
8c. ADDRESS (City, State, and ZIP Code) WRDC/TXI Wright-Patterson AFB, Ohio 45433-6523		PROGRAM ELEMENT NO.	PROJECT NO.
		TASK NO.	WORK UNIT ACCESSION NO.
11. TITLE (Include Security Classification) USE OF HYPERMEDIA FOR AN ARTIFICIAL INTELLIGENCE-BASED PROBLEM SOLVER (UNCLASSIFIED)			
12. PERSONAL AUTHOR(S) Daniel J. Florian, Captain, USAF			
13a. TYPE OF REPORT MS Thesis	13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Year, Month, Day) 891204	15. PAGE COUNT 136
16. SUPPLEMENTARY NOTATION			
17. COSATI CODES		18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)	
FIELD	GROUP	SUB-GROUP	
12	05		
12	09		
		Expert Systems, Hypermedia, Hypertext, User Interface, Maintenance Diagnostics	
19. ABSTRACT (Continue on reverse if necessary and identify by block number) Thesis Chairman: Lt Col Charlie Bisbee (see reverse)			
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS		21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL Charles R. Bisbee, Lt Col, USAF		22b. TELEPHONE (Include Area Code) (513) 255-9265	22c. OFFICE SYMBOL AFIT/ENG

Abstract

The use of expert systems as the problem solving strategy in maintenance diagnostic environments has proliferated in the last few years. This is due primarily to the ease with which a diagnostic system can be developed using the expert system approach compared to using other techniques, particularly conventional programming. One important feature which determines the success of such a system is the user interface. Typically, the user interface of an expert system is entirely textual. While developing graphical user interfaces are possible, it requires the programmer to either integrate the expert system with externally written graphics routines, or to use the expert system's own, usually LISP-based, programming language. Either method requires experienced programmers to perform many iterations of code development until the user interface is complete. Additionally, in complex problem domains such as maintenance diagnostics, it is often difficult to accurately represent the problem in words alone. Especially for the novice, describing the problem not only in words but also with graphics, facilitates a better understanding of the problem; thus, increasing the probability that the appropriate solution is selected.

This thesis discusses the use of a hypermedia system as the user interface for an expert system. The hypermedia system allowed dynamic creation and editing of the user interface, and collected and transmitted information from the user to the expert system. The expert system, which remains transparent to the user, uses this information to recommend a solution to the problem or to determine more information is needed from the user. Regardless, the expert system communicates these results to the hypermedia system, which then displays them to the user.

Specifically, the prototype developed as a part of this research was designed to help Aerospace Guidance and Metrology Center (AGMC) depot-level technicians troubleshoot the Dual Miniature Inertial Navigation Systems (DMINS) Inertial Measurement Unit (IMU), which is being used on fast attack submarines. Currently, DMINS technicians use information from automatic test equipment (ATE) to guide their troubleshooting actions. This ATE is driven entirely by test failures resulting from the tested IMU signals being out of the specification limits. In addition to using these signals, the prototype uses IMU signals, not being validated by the current test, to detect problems before a test failure occurs. The capability to find problems prior to a test failing, can significantly decrease the time needed to test an IMU.

A one-day user evaluation of the prototype by an experienced DMINS technician was conducted and documented. The user especially liked the large screen used to display information, the mouse as an input device, the applicability of the prototype as a training aid, and the ease at which the user interface could be modified. The prototype is nearly a complete system, covering the majority of the DMINS troubleshooting knowledge.