(2)

# AN EXPERIMENTAL VERSION OF AN ADVANCED AUTONOMOUS SPACECRAFT COMPUTER

Industry
Library
JUL 2
Industri

checked
Queen
91
C655
G643
1983

# AN EXPERIMENTAL VERSION OF AN ADVANCED AUTONOMOUS SPACECRAFT COMPUTER

Industry Canada
Library Queen
JUIL JUL 20 1998
Industrie Canada
Bibliothèque Queen

by

T. Gomi
I. McMaster

COMMUNICATIONS CANADA
JUIN JUNE 29 1984
LIBRARY – BIBLIOTHÈQUE

24th June 1983

Work done under subcontract to:

Eidetic Systems Corporation,
P.O. Box 13340,
Kanata, Ontario,
K2K 1X5

under DSS Contract #OST82-00056

COMMUNICATIONS CANADA
LIBRARY – BIBLIOTHÈQUE

P
91
C655
G643
1983

DD 4604787
DL 4604821

Government of Canada / Gouvernement du Canada

Department of Communications

DOC CONTRACTOR REPORT DOC-CR-SP -83-034

DEPARTMENT OF COMMUNICATIONS - OTTAWA - CANADA

SPACE PROGRAM

TITLE: An Experimental Version of an Advanced Autonomous Spacecraft Computer

AUTHOR(S): T. Gomi (Applied AI Systems Inc., Kanata, Ont.)
I. McMaster

ISSUED BY CONTRACTOR AS REPORT NO: None

PREPARED BY: Eidetic Systems Corp.
P.O. Box 13340
Kanata, Ontario

DEPARTMENT OF SUPPLY AND SERVICES CONTRACT NO: 15ST.36001-2-0561
SERIAL No. OST82-00056

DOC SCIENTIFIC AUTHORITY: R.A. Millar

CLASSIFICATION: Unclassified

This report presents the views of the author(s). Publication of this report does not constitute DOC approval of the reports findings or conclusions. This report is available outside the department by special arrangement.

DATE: June 1983

C O N T E N T S

# G L O S S A R Y

| | |
|---|---|
| AASC | Advanced Autonomous Spacecraft Computer |
| ACS | Ada Compilation System |
| Ada | DoD defined Ada programming language |
| AI | Artificial Intelligence - a major and significant discipline in Computer Science (AI). |
| APF | Application Function (AASC/iMAX/BAM/APF) - mission-oriented function of a spacecraft. |
| AP | Attached Processor (AASC/PCU/AP) - external i/o processor used in conjunction with iAPX 432-based PCU. |
| APP | Application Process (AASC/iMAX/BAM/AP) |
| ARBY | An Expert System framework for diagnostic applications. |
| BAM | Basic Application Manager (AASC/iMAX/BAM) |
| BPM | Basic Process Manager (AASC/iMAX/BPM) [INTE 82] |
| CMP | Communicating Processes |
| CKB | Compound Knowledge Base (AASC/OAM/CKB) |
| DUCK | Deduction System for the ARBY consultant (AASC/OAM/OCS/DUCK) |
| EOA | Example On-Board Application (AASC/EOA) - station on the AASC breadboard that represents a typical on-board application. |
| EOD | External Object Descriptor - product of the Intel Ada compilation and linkage processes for execution on iAPX 432. |
| ES | Expert System (AI/ES) - a system that mimics experts working in a limited domain. A form of the application of AI technology for practical purposes. |

| | |
|---|---|
| FTM | Fault Tolerance Manager/Management (AASC/FTM) - collective title for functions in the Layers 1 through 4 of the Hierarchy of System Autonomy. Layers of fault-tolerance capabilities arranged in ascending order of abstraction. |
| HLL | High Level Language(s) |
| IF | Interactive Frame (AASC/OAM/OCS/IF) - a unit of exchange between the on-board consultant (OCS) and the user. |
| IFM | Human Interface Component (AASC/OAM/OCS/IFM) - a part of the OCS that deals with the user. |
| IIU | Subsystem I/O Interface Unit (AASC/IIU) - modules that support i/o activities (sensor or effector activities) of a spacecraft. |
| iMAX | Multifunction Applications Executive [INTE 82] - an operating system for iAPX 432 computer. |
| ISO | International Standards Organization, alias International Organization for Standardization. |
| KB | Knowledge Base (AI/KB) - a module in ES or other form of AI system for storing and making available domain specific knowledge. |
| KE | Knowledge Engineering (AI/KE) - a discipline of AI. Deals with practical use of knowledge in achieving design objectives. |
| KR | Knowledge Representation (AI/KR) |
| LAN | Local Area Network |
| MS | Monitor Station (AASC/MS) - a monitor station on the AASC breadboard that represents a subset of ground control functions |
| NISP | Nifty LISP - a LISP macro library developed by Yale University |
| NIU | Network Interface Unit (AASC/NIU) - a module that permits PCU or IIU to access the AASC on-board network. |

NL — Natural Language (AI/NL)
- ordinary spoken, written, or otherwise expressed language.

NLI — Natural Language Interface
- sub-discipline in AI.

NLP — Natural Language Processing
- sub-discipline in AI.

OAM — On-Board Autonomy Manager/Management (AASC/OAM)
- an entity on-board the AASC responsible for maintaining autonomous operation of the spacecraft. Uses both AI and conventional control techniques to govern the spacecraft. Reports to ground control. Also, represents a function of a layer in the Hierarchy of System Autonomy [GOMI 83a].

OCS — On-Board Consultation System (AASC/OAM/OCS)
- an instantiation of ES. Performs consultations on-board spacecraft with users both on-board and off-board (ground).

OSI — Open Systems Interconnection
- a global scheme to interconnect a wide variety of nodes (computer-based and otherwise). An international standard proposed by the ISO, which is being widely accepted.

PCU — Processor Cluster Unit (AASC/PCU)
- a physical description of a node on the AASC that performs algorithmic/heuristic processing. A processing unit with several variations.

TCL — Transport Control Layer

VLAN — Very Local Area Network - network suitable for a very limited geographical area, typically 1~50 m., in length.

N.B. — Throughout the pseudocode and Ada program specifications, the double hyphen (--) is used to indicate comment(s). In the same context, it is common in Computer Science to use an underscore (_) to form an identifier from a collection of names.

ACKNOWLEDGEMENTS

This work was performed under DSS Contract OST82-00056 for the Federal Department of Communications, Communications Research Centre, Shirley Bay, Ottawa, Ontario, Canada.

The authors would like to acknowledge the contribution which the following authorities have made to their understanding of the issues involved in this report: Dr. Nancy Leveson, University of California, Irvine, on software fault-tolerance; Dr. L. Friedman, Jet Propulsion Laboratory, on autonomous systems; Professor R. Reitman, University of British Columbia, on Knowledge Representation; Professor N. Cercone, Simon Fraser University, on Expert Systems; Professor J. Mylopoulos, Dr. John Tsotsos and Mr. T. Shibahara of the University of Toronto, on Knowledge Representation, AI development environments, and Natural Language Processing; Dr. J. Davidson, of Stanford University, on Knowledge Representation and AI languages; and Dr. David Liu, of Stanford University, on reliability. The authors also acknowledge the support of R.A. Millar, of the Communications Research Centre and assistance from M. Inwood with research and preparation of the report.

## ABSTRACT

A detailed system design is presented by the authors under sub-contract to Eidetic Systems Corporation, contractors for the Communications Research Centre, Department of Communications, Ottawa, Canada, under DSS Contract #OST82-00056.

An Advanced Autonomous Spacecraft Computer system has been proposed in previous reports by the same authors. The design of a proof-of-concept system is described in some detail, whose purpose is to provide a test of the novel combination of state-of-the-art design philosophies and software and hardware technology. The system design is structured using a layered, hierarchical methodology. The top layer is On-Board Autonomy Management (OAM), which provides consultation by an Expert System for the next layer, Fault-Tolerance Management (FTM). The FTM maintains reliable functioning of the Application Functions of the spacecraft, in this case a stationkeeping subsystem for a geosynchronous communications satellite. The system executes by distributing its processing over a Local Area Network (LAN) of processor complexes (PCUs) which use the OSI Reference Model protocols for inter-process communication. A Monitor Station (MS) simulates a ground station for satellite monitoring and control.

The OAM uses artificial intelligence techniques, including a rule-based knowledge representation (KR), and reasoning by hypothesis formation, refinement, and selection. The FTM, which can consult the OAM, starts, monitors, and stops the processes involved in executing the mission-oriented application functions (APF) of the spacecraft, and provides guardian ports and owner processes to handle APF faults. The example APF uses a simplified orbital model to simulate stationkeeping using orbit determination and prediction, and manoeuvre planning, to produce thrust control command sequences. Communication over the AASC bus is controlled by Network Interface Units employing Ethernet and OSI protocols up to the Presentation layer. Ada and LISP are used as design and implementation languages for the major part of the system software.

The major hardware components of the proof-of-concept system are a VAX-11/780 for the OAM, iAPX 432 complex for FTM and EOA, and Intel iSBC 8086 for the NIU for the FTM/EOA node. Other hardware choices will be made from a small set of possible choices at implementation time. Development environments, including requisite hardware and software, will

be needed for the OAM, FTM, and NIUs.

The proof-of-concept system is expected to provide confirmation of the effectiveness of the novel combination of elements in the AASC design.

# 1. INTRODUCTION

A spacecraft with the ability to manage its own systems, even in the face of system failures and unforeseen circumstances, was proposed in [GOMI 82a]. In subsequent reports [GOMI 82b,83a] we have brought the concept of an Advanced Autonomous Spacecraft Computer (AASC) further down the path from abstract to concrete. The design philosophy has included:

1. A layered, hierarchical model for complex system functions
2. Knowledge engineering for the highest layers of system management
3. Fault-tolerance, both hardware and software, at all layers below knowledge-based
4. Use of available hardware and software products wherever possible.

These approaches have implied certain other aspects of the design. Layering has implied use of the OSI Reference Model for network communications, and a layered model that integrates intelligence with fault-tolerance. The requirement for knowledge engineering combined with the desire to use available products implies the choice of an Expert System for intelligent autonomy management. To make the fault-tolerance objective explicit, the Fault-Tolerant Computing (FTC) rules [GOMI 82a] were formulated, and these in turn imply a distributed approach to both satellite functions and fault-tolerance management. The last aspect of the philosophy has led to the choice of Ada as a design and implementation language where possible, and to the choice of a fault-tolerant VLSI processor, the iAPX 432.

The actual introduction of Artificial Intelligence (AI) techniques to the design of the AASC is also achieved in the design of the proof-of-concept system. Presently, there are several efforts being made throughout the world in developing a highly autonomous vehicle. Such autonomous vehicles are meant for use underwater [BLID 83],[HECK 80], on ground (rough terrain) [HARM 83],[BULL 83], or in space [VERE 81,83],[WAGN 83],[SOTT 83],[MOGI 83],[ORLA 83]. While their respective modes of operation vary wildly, in particular in the formalism of their lower level sensors and effectors, the methodology employed to design the autonomy me-

chanisms is surprisingly similar: they all depend on AI technology, namely Expert System (ES) technology.

Such was also the notion introduced in the AASC at its conception 15 months ago. We are now at the stage where we can define the linkage between conventional control technology and this drastically different approach to the control problem. This topic is assumed to become dominant in various realtime system research areas as the impact of AI is increasingly felt by system designers and project administrators (see [DeJo 83], for example).

In fact, the AASC has been built as an AI machine from the beginning. To this end, other technologies were meant to be more or less subservient. Networking, with its OSI elegance, will provide an ideal fault-tolerant framework. The FTM will look after the remaining important issue in fault-tolerant computing, namely, the problem of providing software fault-tolerance. Upon this flexible, yet stable bed are built open-ended AI layers. The AASC design differs from other similar autonomous vehicle projects in the clear conception of system reliability as a hierarchy of abstraction or intelligence.

In this report, we specify in detail the structure, hardware, software, and development environment for a proof-of-concept system for the AASC. Hardware and software product choices are made, with some qualifications. Software to be implemented in the ensuing phases of AASC development is described by Ada package specifications, and the development systems for such a project are specified in detail.

Each of these areas is presented as it relates to the OAM (On-Board Autonomy Management), FTM (Fault-Tolerance Management), Networking, a spacecraft application function (Stationkeeping) and the simulated ground station (Monitor Station).

Finally, we present our conclusion that the proof-of-concept system is readily achievable and expandable to a full AASC.

## 2. OBJECTIVE OF THE DESIGN

### 2.1 Testing the AASC Concept

The design document [GOMI 83a] preceding this one set forth the structure and function of an AASC proposed for a hypothetical but realistic spacecraft. The objective of this document is to present a detailed design of a system which, if implemented, would serve as confirmation or negation of the viability of all or some of the concepts embodied in the AASC design.

The important AASC characteristics to be tested by implementing this "proof-of-concept" system are:

1. Use of Artificial Intelligence (AI) techniques, specifically an Expert System, to manage on-board processing functions and provide ground-station control and information

2. Use of a hierarchical, layered model for system structure

3. Use of a Local Area Network (LAN) to support on-board processing functions

4. Use of iAPX 432 or similar fault-tolerant VLSI processor for spacecraft processing functions

5. Use of software fault-tolerance to maintain reliable subsystem control and performance

6. Use of the Open Systems Interconnection (OSI) reference model as a communications design model

7. Use of Ada as a software specification, design, and implementation language.

To judge whether the proof-of-concept test confirms or negates any of these characteristics we must specify criteria against which system performance is to be gauged.

AI Techniques:

If a knowledge base for a chosen on-board fault domain may be built successfully, and if the OAM conducts the successful diagnostic consultation requested by its users, then these techniques will be justified.

Hierarchy:

If inter-layer communication between layers of the Hierarchy of System Autonomy takes place in such a fashion as to support collective achievement and maintenance of the well-being of the spacecraft; and if the proof-of-concept system is implemented successfully and can support the example application function, support is given to the use of hierarchical, layered models for system structure.

LAN:

If the Example On-Board Application function (see Section 3.4) is able to execute successfully within the timing constraints required in a real-life system, including interacting with the Monitor Station (see 3.5), then this characteristic is confirmed.

iAPX 432:

If a fault-tolerant configuration of iAPX 432 processors executes and provides sufficient processing power to support the example application function, then support is given to the use of the 432 for the AASC. We will not attempt in the initial proof-of-concept, to actively test reliability by inducing processor faults.

Software Fault-Tolerance:

If the proposed software structures for Fault-Tolerance Management (FTM) are successful in initiating, monitoring, and stopping the example application function, then support is given to this characteristic of the AASC. If the FTM is further able to handle intentionally-placed (seeded) faults in the application function, much heavier support is given to the concept.

OSI Reference Model:

If the proof-of-concept system is able to support the example application system within real-life timing constraints, then support is given to the OSI reference model as a viable model for on-board communication.

Ada:

If system development deadlines are met while maintaining the quality of the software written, and the

proof-of-concept system is implemented as proposed, then use of Ada as a design and implementation and documentation language will be given support.

## 2.2 Scope of the Proof-of-Concept System

An important part of designing a proof-of-concept system is deciding how far one should go along the road to building a real-life AASC. Clearly, there are a number of limiting factors:

1. Elapsed time to implement and evaluate the system must be reasonable, so as to capture current state-of-the-art technology in one system.

2. The system must be within the current or achievable level of expertise of available implementors.

3. The system must not exceed available financial resources.

4. The system must perform a function or functions sufficiently close in complexity and performance requirements to a real-life AASC to allow judgement of the viability of the AASC design (see Section 2.1).

Taking the above factors into account, the proof-of-concept system will have the characteristics shown in Table 2.1.

Table 2.1: Proof-of-Concept Characteristics.

| Item | Quantity | Description |
|---|---|---|
| Application Function | 1 | Stationkeeping for geosynchronous communications satellite |
| Simulated Ground Station | 1 | Monitor station attached to network |
| OAM | 1 | Consultation using on-board ES techniques |
| PCU | 2 | Support FTM, Stationkeeping Subsystem, and OAM |
| Fault-Tolerance | - | Four-layer FTM resident in same PCU as Stationkeeping function |
| NIU | 3 | Support OSI protocol up to Session layer for Monitor Station and 2 PCUs |
| LAN | 1 | Ethernet bus |

The system will support one Example On-Board Application (EOA) in a PCU, managed by the Fault-Tolerance Manager in the same PCU.

Similarly, there will be only one OAM node, while in a real-life AASC, there could be several OAM stations.

Networking is not multiplied as in the standard AASC design. In a real-life AASC, a ground station is, or is likely to be connected to a global network. The network may have a node which acts as a gateway for spacecraft communications. This gateway communicates via appropriate media with a gateway which is a PCU, or PCU-IIU combination, on board the spacecraft. However, in the proof-of-concept system, the ground station will be simulated by a workstation

attached directly to the LAN.

## 2.3 Extensibility

Implicit in the proposal to implement a proof-of-concept system is the eventual creation of an AASC laboratory prototype, which performs or simulates all the functions of an actual spacecraft computer system. The proof-of-concept system provides, in a very natural way, for a stepwise advance towards that goal. In fact, it is a natural outcome of the very design philosophy of the AASC that allows the gradual progression from the simplest proof-of-concept system to a full-blown prototype.

If we look at the various characteristics described in Section 2.2, we see how each can be expanded, replicated, made more sophisticated, or improved in performance; in other words, how the scope of the system can be increased according to any desired strategy.

Application Function:

To increase the number of functions, one can

- add more PCU-NIU pairs to the network
- design and code more application functions.

To make these functions handle various real physical subsystems, one can

- add NIU-IIU pairs to the network, each supporting a real or simulated physical subsystem.

To increase the realism of the application functions (for instance, use a more realistic orbit-parameter set for Stationkeeping) one can redesign the relevant parts of the application function algorithms, without altering any of the other system architecture.

Ground station:

Two dimensions are available for improvement of the ground station simulation. First, Monitor Station capabilities can be enhanced to allow control over and communication with more application functions. Second, one can progress towards a realistic network link to the ground station by the following steps:

- replace NIU-Monitor Station attached to bus, using instead NIU-PCU-MS configuration

- replace the above with

  NIU-PCU-NIU-link-NIU-MS

- replace the above with

  NIU-PCU-NIU-link-NIU-existing network (not AASC)

  with the MS linked somewhere in the network.

Autonomy:

The sophistication and quantity of knowledge in the OAM can be increased dramatically. Furthermore, additional layers of intelligence can be added above the initial Layer 5 (Intelligence 1).

Fault-Tolerance:

There are at least two options for increasing fault-tolerance capabilities:

- better message format checking in the Presentation layer of OSI

- creation of a sub-layer within Layer 4 of the FTM that can watch processes in remote processors.

PCU:

As stated above, more PCUs can be added at will with no change to the existing software.

NIU:

Same as for PCUs.

# 3. STRUCTURE OF THE SYSTEM

## 3.0 Overview

In the AASC, the key elements of system design are on-board autonomy based on AI technology, and the concept of software fault-tolerance. In addition, the idea of coordinating multiple functional nodes, loosely coupled by standardized inter-process and inter-processor communication links is considered essential as the framework of the system. Such nodes, each of which is built in a fault-tolerant fashion, would jointly achieve the intended autonomy requirements of the system. This use of cooperative distributed processes is also suggested by Lesser [LESS 83]. Nodes are not necessarily physical in their existence, although each of the nodes will have its physicality as attributes.

There are several levels of abstraction (or levels of machine intelligence) within the system, each represented by one or more nodes. The lower levels typically correspond to such tasks as hardware and software fault-tolerance arrangements, storing of data units, and the basic computational activities.

As stated in the objectives above, in the present phase of the AASC development, emphasis is on the interaction between the levels of intelligence represented by each layer, as well as such interactions among elements within a layer. It is interesting to note that there is a great similarity between the structure aimed for in the AASC and that of the Japanese Fifth Generation computer currently under development [FEIG 83a,b].

More basic fault-tolerant computing issues, such as the need for redundancy in implementation as a basis for fault-tolerance, are assumed but, for reasons of economy, are not planned during this phase of development. This means that, in this phase, each level may be implemented without consideration for redundancy. Such techniques, though essential for the building of fault-tolerant systems, are well-understood by a major part of the Fault-Tolerant Computing (FTC) community, and hence can be demonstrated in a more elaborate (and richer) demonstration, which may be developed later. This simplification is manifested in the current design of the proof-of-concept system, for example in the use of a single bus structure as the means of inter-process/processor message exchange. A further economy

in implementation detail is seen in the use of a single physical node to support more than one logical nodes, or more than one level of abstraction. Multitasking is used as one such method of compromise.

It is of the utmost importance that hierarchical architecture as described and specified in the Design Report [GOMI 83a] be clearly established first. This hierarchy will become the framework to realize a rational autonomy control. The current testbed design reflects this thinking. Following the philosophy of E. Dijkstra, C.A.R. Hoare, and Hubert Simon, and as reiterated by reliability researchers like Professor Nancy Leveson of UC Irvine, hierarchical structuring of the problem-solving scheme must be the heart of the design for any complex system. We can point to examples of such notions in the design of modern (post sixties) operating systems, computer graphics, networking, and most importantly, in Artificial Intelligence.

The AASC hierarchy is explained in detail in the above mentioned Design Report, along with meta-rules that govern the layers in the hierarchy. It may be helpful to show the correspondence between the design hierarchy and its implementation in non-hierarchical physical media. The logical structure of a fully implemented AASC would look like Figure 3.1. A subset of logical functions for the proof-of-concept system is shown in the figure, using shaded elements. This is the subset of the AASC hierarchy chosen for the proof-of-concept design. Its physical implementation is shown conceptually in Figure 3.2.

A commercially available, rule-based Expert System framework for system diagnosis is adopted as an instance of the On-Board Autonomy Management (OAM) layer (Layer 5 in the System Autonomy Hierarchy). Similarly, a set of Ada program modules are defined to represent the selected subset of functions of the FTM layers (Layers 1 through 4 in the hierarchy). As an arbitrary example of on-board application, the OAM complex is linked to an application node and used to test the proof-of-concept system. The FTM and suitable application function(s) will be represented in the node.

## 3.1 On-Board Autonomy Management

The On-Board Consultation System (OCS) of the OAM is defined for the proof-of-concept system. For the purpose of this system, the rest of the OAM functions, as described in the Design Report, will exist in a simplified form within

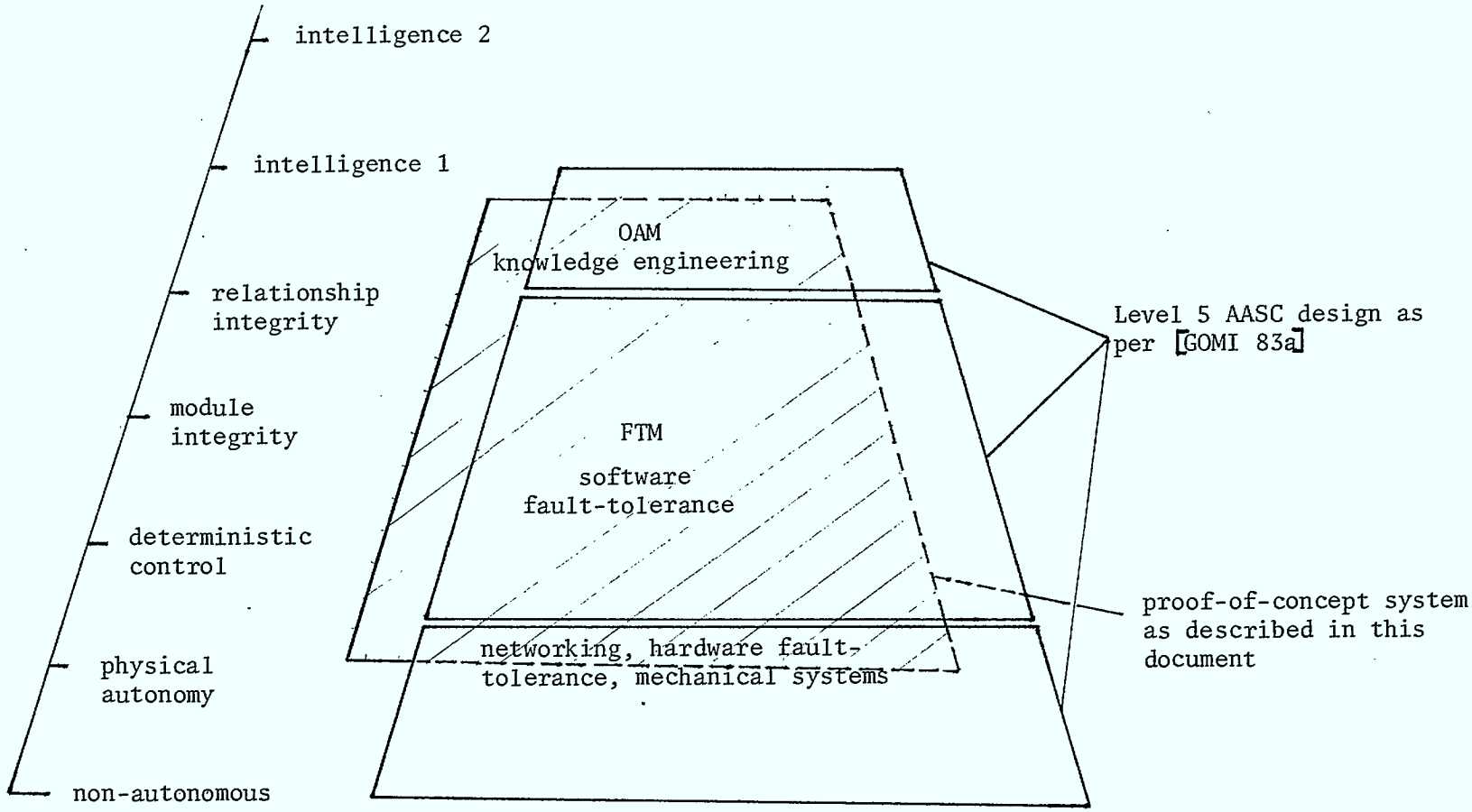
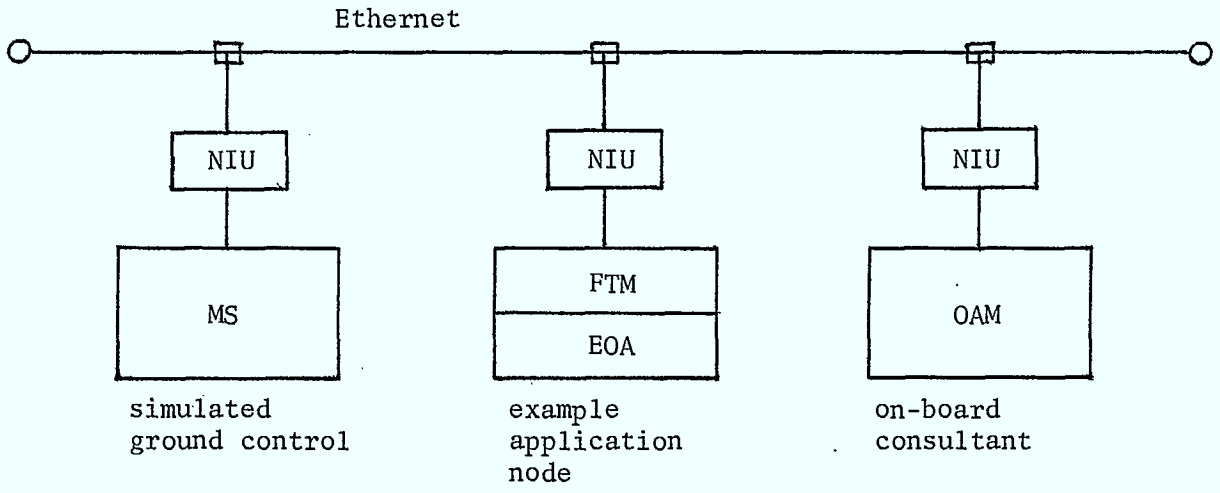



Figure 3.1: Hierarchy of System Autonomy, the AASC Design and the Proof-of-Concept System




NIU: Network Interface Unit
MS: Monitor Station
EOA: Experiment On-Board Application
FTM: Fault-Tolerance Manager
OAM: On-Board Autonomy Manager

Figure 3.2: The AASC Proof-of-Concept System

the interfacing module to the OCS, or within the network protocol. The OAM uses the network protocol to exchange messages with the other portion of the on-board system, other space systems, and also with the ground.

The idea of developing our own Expert System (ES) from scratch for this phase of the AASC has been rejected for the following reasons:

- availability of commercial ES frameworks of a reasonable capability
- lack of experience in constructing a major ES for diagnostic purposes
- excessive cost of full in-house development of a sophisticated AI system

Several available ES frameworks are considered. Some examples are:

- EMYCIN, from Stanford University (is now available only from Teknowledge)
- OPS-5, C. Forgy, Carnegie-Mellon University
- PROSPECTOR, R. Duda, Fairchild AI Lab.
- PSN and its extension, J. Mylopoulos et al., University of Toronto
- INTERNIST-II, E. Pople, University of Pittsburg
- ROSY
- ARBY, D.McDermott, Yale University/Smart Systems Technology Inc.
- EXPERT, Rutgers University.

EMYCIN [VANM 81] was rejected as it already belongs to the older generation of tools; OPS-5 [FORG 81] because information was not obtained in time; PROSPECTOR, [DUDA 81,83] since its characteristics as an ES for exploration did not suit the objectives of the AASC; PSN (and its extension) [MYLO 82],[SHIB 82,83] because it was judged too sophisticated for the current version of the AASC; INTERNIST-II, for the same reason as PSN and also as it is still very much

under development; ROSY, as not enough information was obtained; EXPERT as its development facility requirement is excessive, and also it was judged too general. Therefore, ARBY, of Smart Systems Technology was chosen and the basic system design [GOMI 83a] was adjusted to accommodate the package.

In addition to its general suitability as an introductory diagnostic ES, ARBY has the following list of desirable features which are judged particularly suitable for the current phase of the AASC proof-of-concept system:

(1) ARBY is relatively dynamic in its method of conducting diagnosis. It can respond quickly enough to changing system states and adjust its inference process accordingly. Also, it can describe and retain the changing state of the system as a dynamic history of system performance. This is a major departure from the first generation diagnostic ESs, such as MYCIN.

(2) It has actually been built and tested in the domain of diagnosing electronic systems.

(3) It is judged sufficiently abstract and general purpose to represent and manipulate knowledge in the domain of software fault-tolerance, as well as hardware diagnosis.

(4) Sufficiency of the explanation facilities found in the design Explanation subsystem is considered particularly important for application of AI technology in new space systems. This is because of the greater risk and cost involved in the operation of space systems and, hence, the eagerness on the part of human operators to audit, monitor, and take over, if deemed necessary, the operation.

(5) Inference schemes and algorithms used by the inference engine are judged effective, practical and economical, while achieving a tolerably sophisticated level of reasoning. There are techniques to eliminate redundant hypotheses, as well as to avoid the over-stretching of a single hypothesis. Such techniques are implemented without resorting to an expensive, over-elaborate method. The robustness of the process is anticipated.

(6) Interface to other on-board subsystems is a good compromise between a Natural Language scheme and the rigid inflexible formalism of fixed parameter exchange. The message interface is also very suitable for the AASC on-board and off-board communication format. It is assumed to be fully adaptable to the OSI protocol scheme. Both the module strength and the module decoupling rules are well observed in the design.

(7) The design is open enough to:

- allow future expansion and/or sophistication of selected subcomponents

- permit trimming, reorganization and other modes of optimization to make it an efficient AASC component

- allow later extensions to incorporate more advanced AI features such as a full Natural Language interface and more sophisticated knowledge representation using structural semantic networks ("structural" in the sense discussed by Mylopoulos, DeJong [DeJO 83] and Pople [POPL 83]).

(8) The reasoning scheme is flexible enough to deal with expected spacecraft on-board faults. The inference engine can handle both forward and backward chaining and deals nicely with compounded causes of a fault, as well as multiple faults.

(9) The requirement for a development environment is modest and can be realized on easily accessible facilities (such as the VAX computer).

On the other hand, the following areas of the design of ARBY are identified as cause for some concern:

(1) It is rule-based, and as R. Davis of MIT points out [DAVI 82a,b], using such a rudimentary KR method in diagnostic ES may eventually pose a problem, as such a system is incapable of efficiently describing the internal (topological and relational) structure of the system to be diagnosed - "A rule-based ES, such as MYCIN, does not care if the patient has three legs or two hearts" (Davis). D.

McDermott, the principal designer of ARBY, emphasizes the significance of adopting "shallow" models for conducting diagnosis. Yet we feel that the shallow investigations of relationships between the symptoms and diagnosis may not be enough in more involved cases. The concern still does not affect our choice of ARBY as a good entry system for the AASC.

(2) The KR method used in ARBY is highly domain and application specific. We still cannot foresee the way to generalize inference schemes and eventually extract "reusable" diagnosis knowledge This, we think, is partly because the ES is rule-based. In the KB of the rule-based ES, particularly those written in LISP, knowledge is represented in a relatively ad hoc fashion. We agreed to disregard the generalization issue for our first try.

(3) Again, as a rule-based system, we may experience greater difficulties in debugging the KB. More sophisticated KR are not available in a suitable form for use in the current phase of the project.

### 3.1.1 Operation of the On-Board Consultation System

The OCS (implemented using the ARBY framework) runs under the guidance of the consultation monitor. The monitor manages each consultation session. It is invoked by a consultation request message sent by one of its users (the FTM, the EOA, or the MS). At this point, the user may specify if it wants to continue the consultation carried out previously, or start an unrelated session. If the previous consultation is to be continued, data structures created earlier are made available to the user.

An Interaction Frame (IF) is a means of communication between the user and the OCS and is discussed further in 3.1.3 below). At the beginning, an introductory IF is generated by the OCS to acknowledge the request and initiate a consultation session. It is assumed that, at this time, the Session establishment over the AASC on-board network is completed by the NIUs involved. Such IFs will ask general questions that are typically asked by a domain expert at the beginning of a real-life consultation: "Explain what went wrong." If the OCS encounters, during consultation, a rule which demands a new finding, another IF will be fired to ac-

counted for such a rule.

If the initial information gathering collects some new findings the OCS begins to find, and then refine, hypotheses to account for them. During this process, the OCS will report the progress by sending out messages about the choices it is making. A trivial consultation will terminate relatively quickly. However, in more involved cases, the OCS will eventually pick a test to use to gather additional information. Such a test is also an interaction between the user and the OCS, and hence runs as an IF. The user will be put in "Question Mode" (see 3.1.3 below for "Question Mode").

When the OCS exhausts the hypotheses expansion and comes to the end of a run, it outputs the over-all hypothesis structure that it has inferred. It then puts the user in "Walk Mode" (explained also in 3.1.3 below), so that it can inspect the structure of the consultation session just completed, investigate the reason behind choices of hypothesis, and, in rare cases, change the course of the consultation by altering the data base (the "change" feature is yet to be implemented).

The following pseudocode summarizes the operation of the OCS. However, it must be noted that the representation of the consultation algorithm in a pseudocode is somewhat misleading as it de-emphasizes the heuristic nature of the exchange and processes that take place in a session.

```
task on_board_consultation_session is
  initialize_consultation;
      -- includes a request for an OSI session
      -- establishment to the NIU.

  activate_initial_Interaction_Frame;
      -- "describe what went wrong"


  while findings_to_be_accounted_for loop
    look_for hypotheses to account_for the
    findings;
    ask_for_more_information (findings);
      --  hypotheses are accounted
      --  for in a structured
      --  fashion from the abstract
      --  to the detailed as the loop
```

```
        -- repeats.
      if no more to find then
      findings_to_be_accounted_for := false;
    end loop;


    output overall hypothesis structure;
    put the consultant into "Walk Mode";
        -- consultant explains what
        -- he did altogether for the
        -- user

    while walk_mode loop;
      accept request by the user and
      walk about the result of
      the consultation as instructed;
      If user enters "quit" then walk_mode := false;
    end loop;

end on_board_consultation_session;
```

### 3.1.2 Functions of the OCS

Two major functions are performed by the core of the OCS: the inference conducted by the HYPO module; interaction with the external world (in our case, communication with the FTM, EOA, or the MS) conducted by the IFM module. Both modules are written mainly in Franz LISP, and augmented by the NISP macros. The general purpose deduction retriever DUCK is also needed for their operation.

HYPO tries to find hypotheses which successfully account for a given set of findings. On-board faults described in terms of symptoms and system states are an example of such findings. The findings could be initial observations or any new information, such as test results collected during the course of the consultation. Hypotheses are statements about the nature of the fault or problem which caused or are causing the findings.

Initial findings will make HYPO choose a set of initial hypotheses. In the process of refining them, the OCS attempts to ask the user to make observations, to perform tests, so that HYPO can delete or elaborate hypotheses in the set. Backward-chaining is the main inference methodolo-

gy.

The IFM, or Human Interface Component, is called upon by the inference engine (HYPO) as needed. The net effect of the IFM's function is to add to or modify the assertion database that is built up during a consultation. The Interaction Frame or IF, is a discrete unit of interaction with the user to carry out the investigation.

The IFM asks questions using one of five defined formulae. A very simple IF may ask the user to choose an answer on the state of the system from a set limited number of alternate answers, for example, (FATAL, RAD-ALERT, ORANGE-ALERT, YELLOW-ALERT, STABLE, A-OK). These symbolic expressions are treated as symbols throughout the system and not converted to codes (such as YELLOW-ALERT = -1, for example). This is one difference between AI programming and conventional programming. However, it is pointed out that this level of symbolic manipulation is still a far cry from what can be offered by Natural Language (NL) processing.

In a slightly more sophisticated dialogue the user might be asked "Is the second bank of the main memory complex responding?" If the answer is "Yes", an assertion like (STATUS MEMORY ENABLED) might be put into the assertional database. A yet more complex IF might instruct the user, step by step, to run some interactive tests on on-board equipment (note that "the user" may be a software process in the FTM or the EOA), and report the result to ground using a limited English vocabulary. Such IF might then place several different assertions into the database. There is a way to expand the IF interface and provide a reasonable NL processing facility. This would further increase the flexibility of the system, creating a consultant whose explanations are more articulate, and who can respond more freely to input variations.

A more detailed explanation of how HYPO and IFM work will be found in Section 4.1.

### 3.1.3 User Interface to the OCS

#### 3.1.3.0 Overview

The sole interface of the consultation system is that between it and the user. Normally, the physical implementation of the interface takes the form of a standard alphanu-

meric console attached to the processor. Except for the debugging period, however, the interface in the AASC proof-of-concept system will be between the OAM machine (software/hardware entity) and the Network Interface Unit (NIU). The interface will be used to convey to the OCS requests for consultation and other information demanded by the OCS in the course of the reasoning process; and the result of the consultation, explanations and other reports generated by the OCS for its user.

The NIU that is used with the OAM, like the other NIUs in the AASC, will support the OSI protocol. Hence, the OAM will follow the Session layer protocol to establish a consultation session with the user stations at the beginning of each consultation. This channel will be dissolved at the end of a consultation. It is assumed only one consultation will be supported at a time due to current limitations in the handling capability of the OCS. Therefore, there will be a maximum of only one invocation of Session protocol between the OAM and its users.

There are three types of user within the AASC proof-of-concept system. They are the simulated groundstation or the MS, the simulated application station or the EOA, and the FTM. The OCS will issue requests to these users for <facts> or <findings> as needed to carry out the reasoning process.

#### 3.1.3.1 Interaction Frame (IF)

The unit of exchange between the user and the OCS is called an Interaction Frame, or IF. An IF consists of at least one output and one input message from/to the OCS. It is typically a question issued by the OCS followed by the reply submitted by its user. This unit of exchange is normally a part of a larger framework (a consultation session).

There are several standard frame types that the OAM understands. These standard IFs will be used for the simpler interactions. These include the following:

- an IF that solicits a reply by asking the user to choose it from a known, finite list, e.g.,
  (SUCCESSFUL, NO-CHANGE, OPERATION-FAILURE)

- an IF that requests an answer from the user which can be any subset of a list, i.e., multiple answers are

acceptable

- an IF that demands an integer as an answer,
  e.g., Anticipated length of thrust? -> 1400

- an IF that is used to ask for names of things; it does not place any restrictions on acceptable values.

In addition, the OCS provides the potential user with ways to define its own IF format. The consultation system also permits bundling of IFs that are frequently used in a particular sequence. It is anticipated that several IFs will be developed during the implementation phase of the AASC proof-of-concept system, including several "macro-IFs" that support a relatively lengthy exchange. Examples of lengthy exchange expected on board the AASC are: step-by-step re-loading of portions of on-board software; and periodical report to the ground by the on-board consultant. The NIU guarantees the safe delivery of the elements of an IF (messages) by using its Transport layer protocol. It also supports an exclusive access to the consultant during a consultation session, using its Session layer protocol.

### 3.1.3.2 Question and Walk Modes

A more sophisticated exchange between the OCS and its users takes place when the OCS puts the consultation into Question Mode by asking the user a fact-finding question. Instead of simply providing requested information, the user may then ask the consultant for the meaning of the question, the reason for asking it, its aim, how to collect needed information, and an explanation of related commands. The user can even refuse to answer the question.

When the user asks about either the reasoning behind or the aims of a question, the OCS will put it into Walk Mode. This enables the user to examine either the deductive goals which the answer to the question would help to satisfy; or the overall hypothesis that the OCS is currently entertaining. This feature is obviously useful to ground control (in the present setup, the Monitor Station or MS and its operator).

It is very unlikely that either of the other two users, the FTM and the EOA, would access the Walk Mode facility. Although the algorithms executed within these modes may be very complicated and sophisticated, they nevertheless repre-

sent considerably lower intelligence than the operator at the console of the MS, or the OCS.

In a full-scale AASC application, ground control will typically use this facility to monitor the on-board operation, to execute its auditing duties (see the AASC Design Report [GOMI 83a]), or, in extreme circumstances, to take over control. Even in the case of a take-over, it is likely that ground control would depend on the interfacing functions of the OCS. It is anticipated that only in extreme circumstances would ground control disable on-board autonomy features (the OAM and the FTM) completely, and resort to direct control of lower level on-board functions.

An example of a reply from the OCS to the user in Walk Mode appears below:

```
Goal:  (after (subunit-diag-4 (generator-fault 8))
Rule used:
        1   SUBUNIT-64-RULE-1
How you got here:
        2   (voltage-fluctuation intermittent)
        3   (invoked  (IF 12))
Where are you going:   (after  (subunit-diag-6)
                               (generator-fault 8))
```

The output says that the immediate reason the question is being asked by the OCS (one gets into Walk Mode from Question Mode) is to find out if generator-fault 8 is a result of running subunit-diag-4. The test was run as the SUBUNIT-64-RULE-1 suggested it. The rule had checked intermittent voltage fluctuations, and before that, another IF had been invoked in the current consultation sequence.

The user, in response to such an explanation, may walk further about the deduction structure, change the focus of attention by changing the display, or demand more detailed explanations. Optionally, the user can alter an element of the reasoning sequence. This may or may not be meaningful depending on the element to be changed and the way in which it would be changed.

There are five kinds of display for different purposes. These purposes are:

- editing the concepts in the domain

- investigating successful and failed deductions

- inspecting and editing Interactive Frames

- investigation of the current state of a diagnosis

- miscellaneous minor items

It is anticipated in the near future, the facility to permit changing old answers will be added to ARBY. When this happens, the OCS will become capable of re-enacting its old sessions and of changing its course of action retrospectively.

## 3.2 Fault-Tolerance Management

### 3.2.0 Overview

The Fault-Tolerance Manager (FTM) is composed of four layers. Their relationships are shown in Figure 3.3.

### 3.2.1 Layer 4: Application Function Integrity

Layer 4 is responsible for maintaining reliable execution of each AASC application function (APF). Application functions are such things as Stationkeeping, Thrust Control, Attitude Determination, etc. The functions provided in this layer include those in Table 3.1.

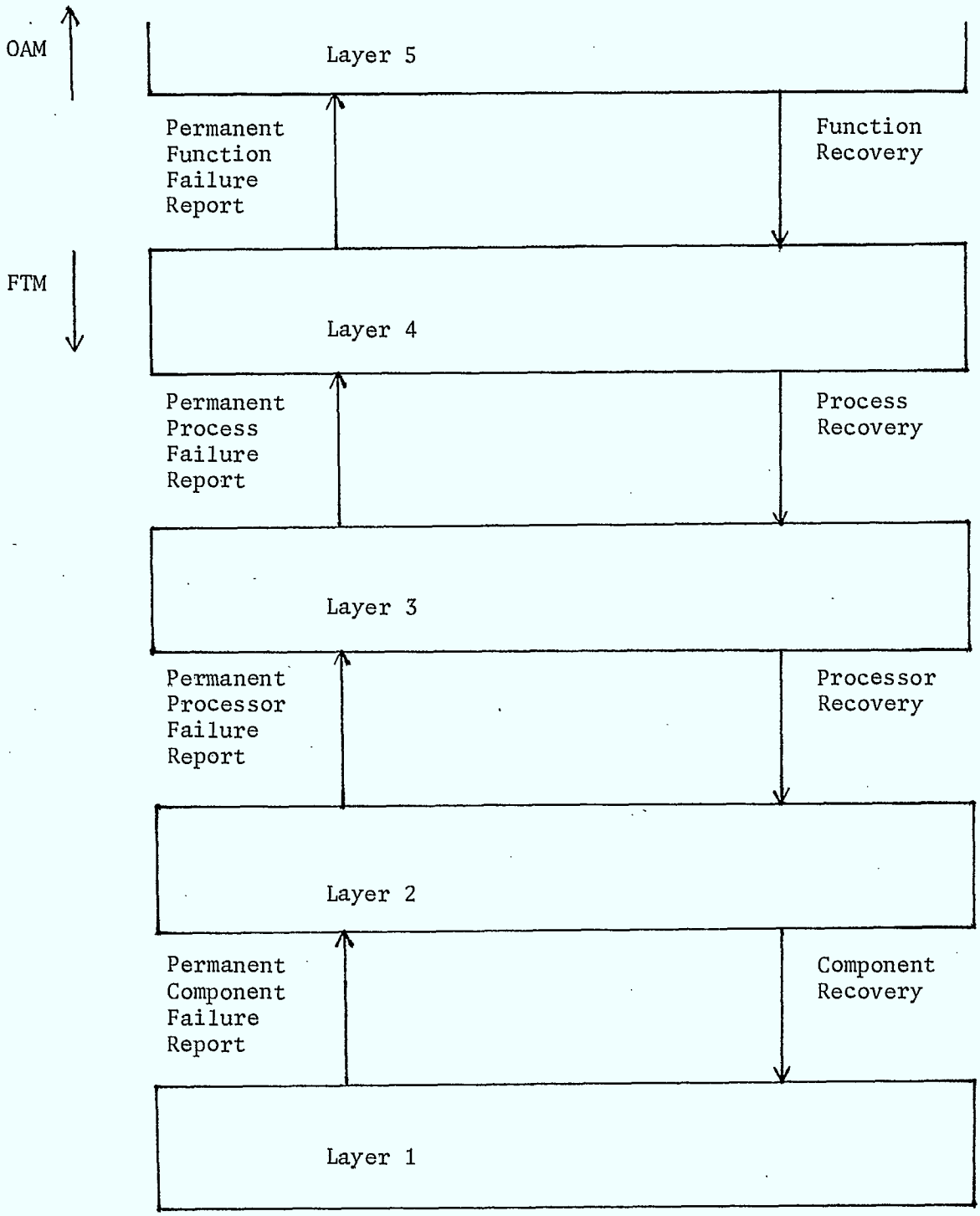



Figure 3.3: Relationships among FTM Layers

Table 3.1: Structure of Layer 4 of FTM

| Function | Performed by |
|---|---|
| initiation, monitoring, and termination of APFs | Basic_Application_Management package (BAM) |
| detection and recovery from transient faults in interprocess communication | OSI_Presentation package and related packages |
| reporting of permanent faults in interprocess communication | OSI_Presentation package and related packages |
| reporting of permanent faults in APF. | Basic_Application_Management package |

### 3.2.2 Layer 3: Process Integrity

Layer 3 is responsible for maintaining reliable execution of each process within each APF. There are two classes of techniques used to accomplish this objective:

1. Software fault-tolerance:

   - assertions
   - exceptions
   - exception handlers
   - N-version programming

2. Guardian/owners

These are not independent classes, but represent different parts of the overall structure of Layer 3. Each of the items in class 1 can only be specified by meta-rules for the design and coding of processes within APFs.

Assertions:

An assertion is a logical expression A, associated with a range of instructions of a procedure P, with the property

that if A = false between execution of the first and last instructions of the range, then P is faulty. More practically, an assertion can be tested at the beginning and/or end of the range, and if it is false, a fault has been detected.

Exception:

An exception is the occurrence, during execution of a procedure, of a state for which there is no valid successor state within the procedure. Stated more simply, it is a condition for which coding has not been provided within the body of the procedure. When an exception occurs, the procedure cannot continue executing.

Exceptions can be:

- system-defined, e.g. integer overflow
- programmer defined.

The latter kind of exception will be raised when an assertion is false. Thus assertions will be placed at key points within a procedure to prevent further execution in the presence of a fault. The choice of these key points is process-dependent, but the following are suggested points:

- start of the procedure
- after an input instruction
- before a procedure reference
- after a procedure reference
- before an output instruction
- at the end of the procedure.

Exception-handler:

For each possible exception defined within a procedure, an exception-handler can be defined. It may perform arbitrary process-specific functions for isolation or analysis of the exceptional condition. For the purposes of the AASC, it must terminate by raising an exception itself. The reason for this is given under Guardian/Owner below.

Each procedure in each process in each application may contain an exception-handler. Its purpose is to handle con-

ditions for which appropriate actions cannot be coded within the procedure. The exception-handler may perform arbitrary functions necessary for isolation or analysis of the exception, but must ultimately send the process within which it is located to a guardian (see below). The exception-handler does this by raising an exception itself. The operating system then sends the containing process to its guardian, where it awaits handling by its owner.

The exception-handler is invoked either by the implicit raising of an exception, such as integer overflow, or by an explicit 'raise' statement for a process-specific exception. An example framework for an exception handler is given below:

```
procedure  xyz (
    a:  type 1;
    b:  type 2);
    wrong_combination:  exception;
      --  This declares the exception.
        .
        .
        .
    begin
        .
        .
        .
        if  a > 0  and b < 0
        then
            raise wrong_combination;
              --  This raises the exception.
        end if;
        .
        .
        .
        exception
              --  This declares the actions
              --  of handling the exceptions.
            when wrong_combination =>
            failure_description :=
                failure_descriptor' ("wrong_
                combination", a, b);
            raise;
              --  This invokes iMAX to send
              --  process to its guardian.
        when  others =>
              --  This handles all other
```

```
            -- exceptions.
            failure_description :=
                failure_descriptor'(null, null, null);
            raise;
    end;
```

The raise statement within each handler causes iMAX to send xyz to its guardian, and thence to its owner, which can examine the state of the xyz, including failure_description, and perform required recovery functions on it.

Guardian/Owner:

When an exception occurs, the question arises: what happens now? The first level of response depends on whether there is an exception-handler within the process. If there is, it is executed. It must terminate by raising an exception itself. Thus, whether an unhandled exception or a handled exception occurs, the operating system sees the exception as the end result.

When a process terminates by means of an exception, the operating system automatically sends the process to its guardian.

A guardian is a special instance of a port [INTE 82]. A port is simply a queue to which objects may be sent and from which objects may be received. A guardian is a special port in that only processes can be sent to a guardian. When any process is started, a guardian is associated with it, and it is to this guardian that the process is sent when an exception is raised within it. In addition, an owner process is defined when the process is started, whose responsibility it is to receive processes from the guardian. The owner can perform isolation, diagnosis, and recovery functions for the process. A generic package for owners is presented in Section 4.2. Figure 3.4 shows the process fault-handling sequence involving exceptions, guardians, and owners.

N-Version Programming:

An obvious technique for tolerating faults is redundancy. Concurrent redundancy implemented in software is called N-version programming. 2-version programming provides fault detection, 3-version provides fault detection, diagnosis and recovery.

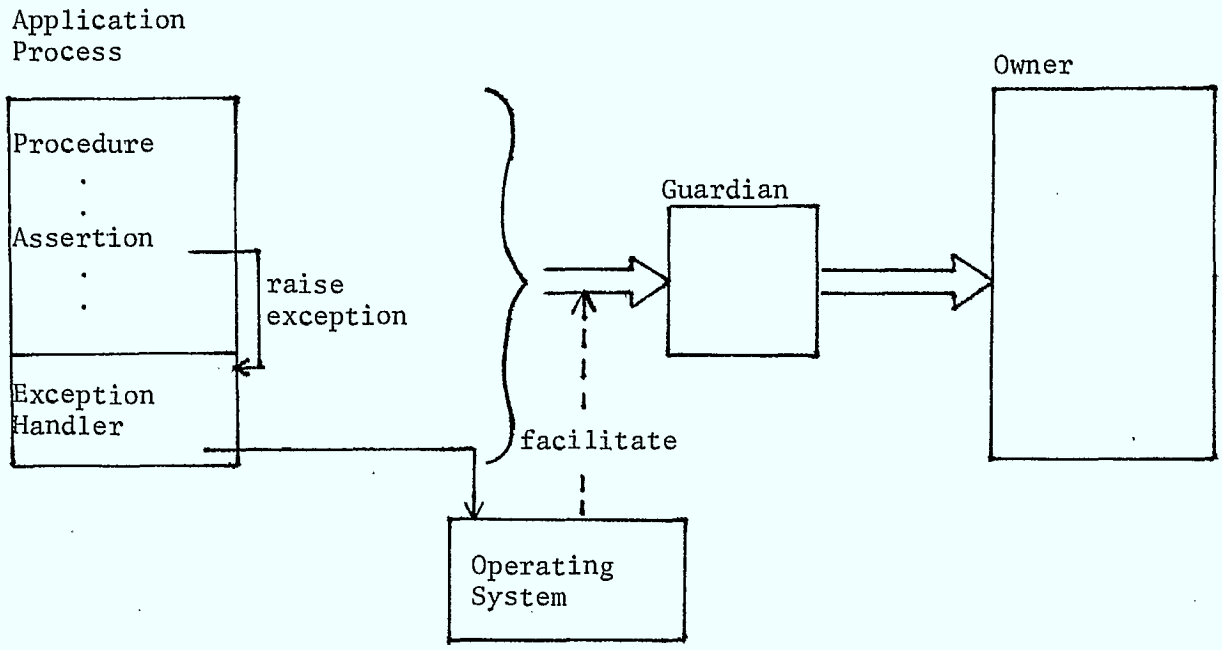



Figure 3.4: Process Fault-Handling

In the AASC, redundancy may be provided at the process level, as shown in Figure 3.5. It has been pointed out [LEVE 82] that rigorous N-version programming would require that the specification, design, coding, and testing of a software module be independently carried out, in order that no systematic error occur in the N versions. Such independence would, it is claimed, raise software costs to an unacceptable level. There are two reasons for mollifying these objections, however.

First, to demand independent specificiations and/or design is to demand much more of the software than is usually demanded of the hardware in equivalent redundant hardware designs. That is for hardware, redundant components are usually simply manufacturers' replications of the same component design. That is, hardware redundancy is at the fabrication level, not at the specification level. If such a level of redundancy is acceptable in hardware, then for software, redundancy at a point intermediate between independent specification and independent coding should be accepted as providing an enhanced level of reliability.

Second, by providing N versions at the process level, redundancy can be applied selectively to critical processes. Hence, cost/benefit can be kept at an arbitrarily reasonable level.

### 3.2.3 Layer 2: Processor Integrity

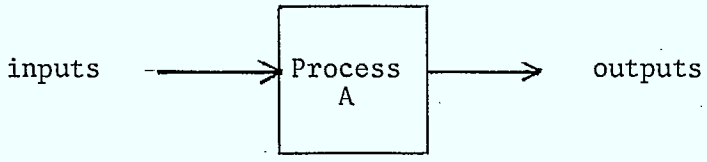
Processors in the AASC are the PCU, NIU, and IP. The PCU achieves fault detection and recovery at the processor level by multiple redundancy of the iAPX 432 GDP (General Data Processor) [PETE 83].

Two levels of redundancy are available in the 432: Master/Checker (double redundancy) and Primary/Shadow or married processors (quadruple redundancy). The latter supplies detection and uninterrupted recovery from processor faults. A detailed description of the mechanisms by which this is achieved appeared in a previous report [GOMI 82a].

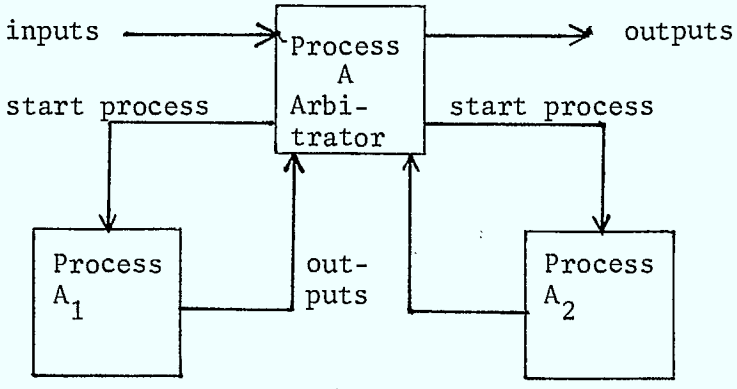
### 3.2.4 Layer 1: Component Integrity

By component, in this case, we mean a component of a processor, not necessarily a physical module. For instance, an adder, a bit plane, a word, a logical comparator. Fault detection and recovery for these components is provided in

1-Version Programming (no redundancy)




2-Version Programming




3-Version Programming

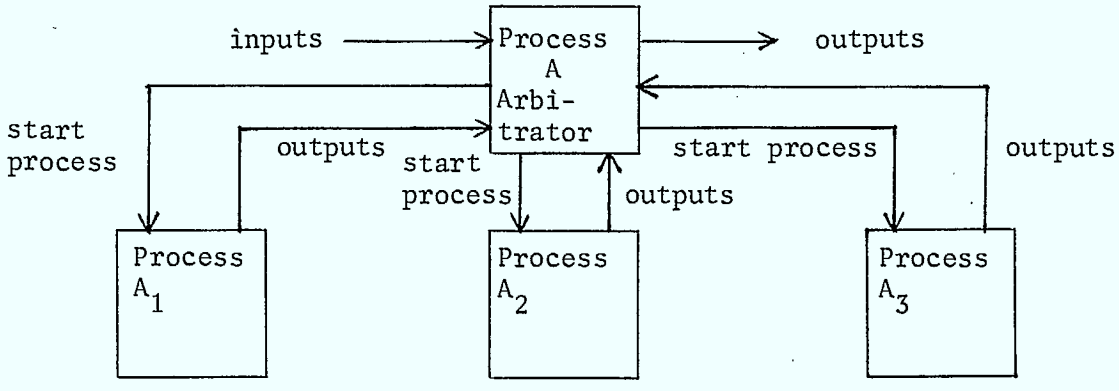



Figure 3.5: Redundancy at the Process Level

the iAPX 432 within the chips, by:

access retry
access re-routing on alternate backplane buses
several bit-correction schemes
bus parity
ECC memory arrays.

## 3.3 Networking

### 3.3.1 Interprocess Communication

Interprocess communication is considered to take place between:

OAM and BAM
  e.g. request to start an APF
OAM and APF
  e.g. request for stationkeeping
APF and APF
  e.g request for current orbit parameters from Stationkeeping to Orbit_Determination subsystem
APF and subsystem
  e.g. Thrust Control program and thruster controllers.

The two communicating processes (CMPs) may be within the same processor complex (PCU) or in different PCUs. The CMP which initiates the communication is called the initiator process, the other is the acceptor process. Neither CMP should have to know the location of the other. A solution to this problem is attempted in the Liberty Net [NASS 82] but, unfortunately, the solution appears to deviate substantially from the OSI Reference Model. Within the OSI Reference Model, the location of two OSI users is unknown above the Network layer. Ideally, then, if two processes CMP1 and CMP2 establish communication, their locations should be known only at the Network layer. If CMP1 and CMP2 are within the same PCU, this would imply the flow of messages between CMP1 and CMP2 as shown in Figure 3.6.

There are two disadvantages of implementing this type of connection for co-located CMPs. First, it implies extra processing steps in going through the PCU-NIU connection and three OSI layers. However, processor speed is sufficient for us to disregard this disadvantage. Second and more important, however, is the fact that it is not guaranteed that

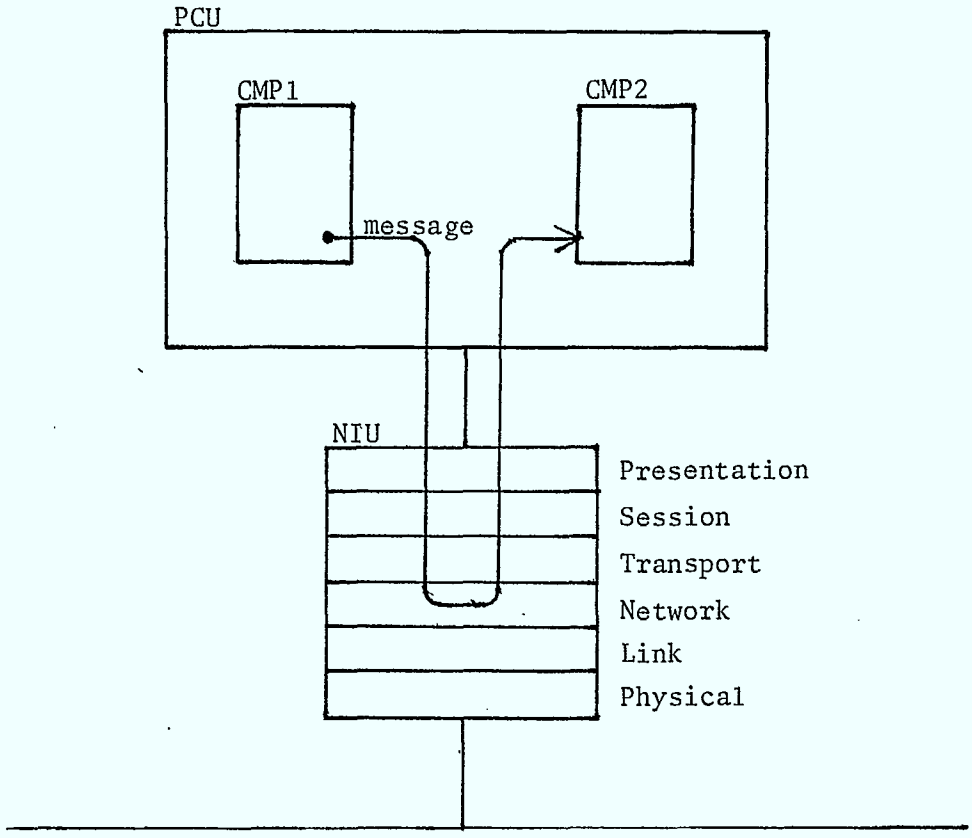



Figure 3.6: Ideal flow of messages between co-located processes

existing available communication packages will provide the kind of connection in Figure 3.6. Available specifications on such products do not indicate whether or not this feature is implemented. For this reason, the ability to determine whether CMP1 and CMP2 are co-located will be contained in an application-layer module within the PCU, through which all inter-process communication will go. Figures 3.7a and 3.7b compare co-located and remote communication paths.

The purpose of the local exchange is to determine whether the destination process is in the same processor or not. If it is, it (the local exchange) uses the local inter-process communication facility to deliver the message [INTE 82]. Otherwise, it transmits the message to the NIU, where the Presentation and lower layers of the OSI implementation take over. A more generalized theory of distributed concurrent processes is presented by Tobiasch [TOBI 82], but the formalism is too experimental to be incorporated here.

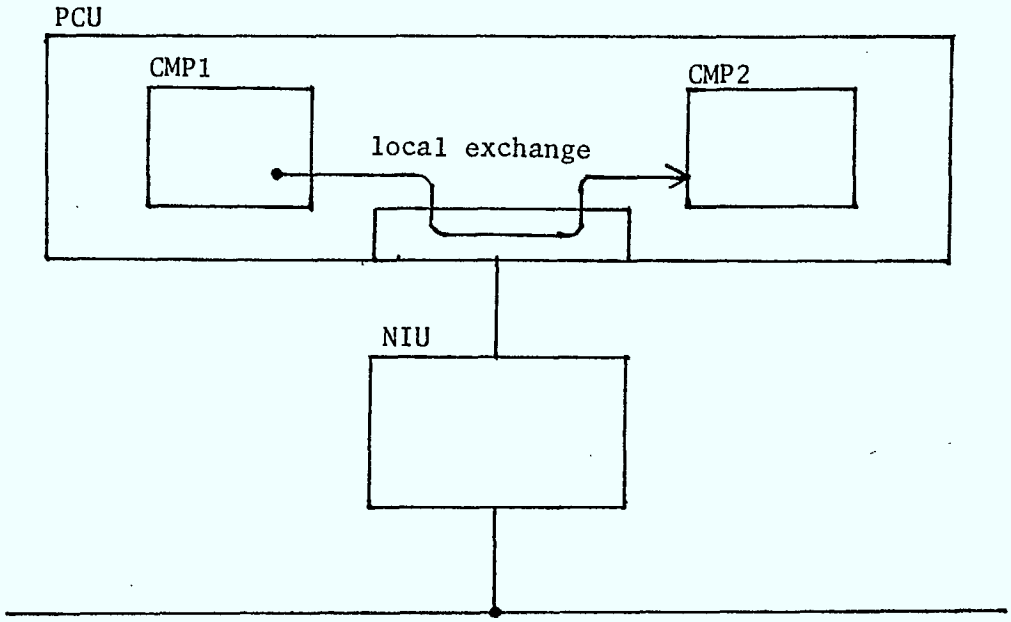
### 3.3.2 OSI Implementation

When two processes communicate across the network bus, they do so through two NIUs, one attached to each of the processors in which the processes run as shown in Figure 3.7a. Each NIU can be considered as providing a service to a communicating process (CMP). In turn, the NIU must send and receive digital signals over the bus. The transformation between the quite abstract function of exchanging messages between CMPs and the exchanging of signals over a bus is divided into six steps within the NIU. This choice is not arbitrary. It is based on the OSI Reference Model [GOMI 82b] [BLAN 81],[SCHI 83],[ISO 82b]. Figure 3.8 shows the layers within the NIU. We describe here the services provided at the Session and Presentation layers.

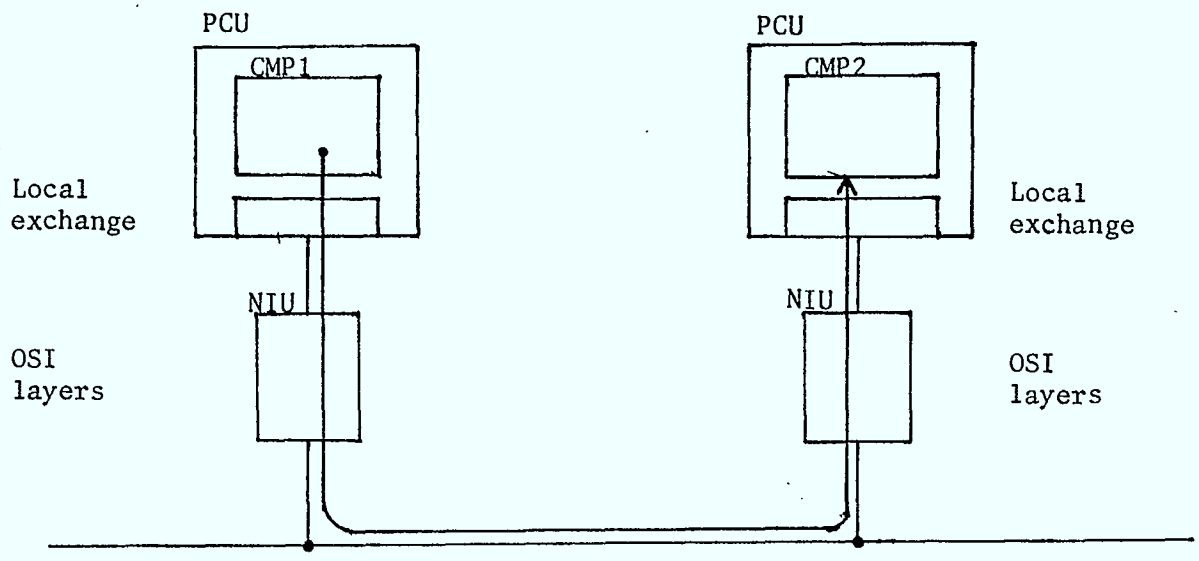
Message exchange is at the application level in the OSI Reference Model. Thus the AASC must provide service at the Presentation layer. The messages may have any content, depending on the application. For the proposed proof-of-concept system, the Presentation layer will provide functions to make inter-process communication reliable:

1. Check message form and content against an APP-specific template

2. Check messages' origins and destinations for correctness, using APP-specific criteria




(a) Co-located processes CMP1 and CMP2




(b) Remote processes CMP1 and CMP2

Figure 3.7: Comparison of communication paths for co-located and remote processes in proposed AASC.

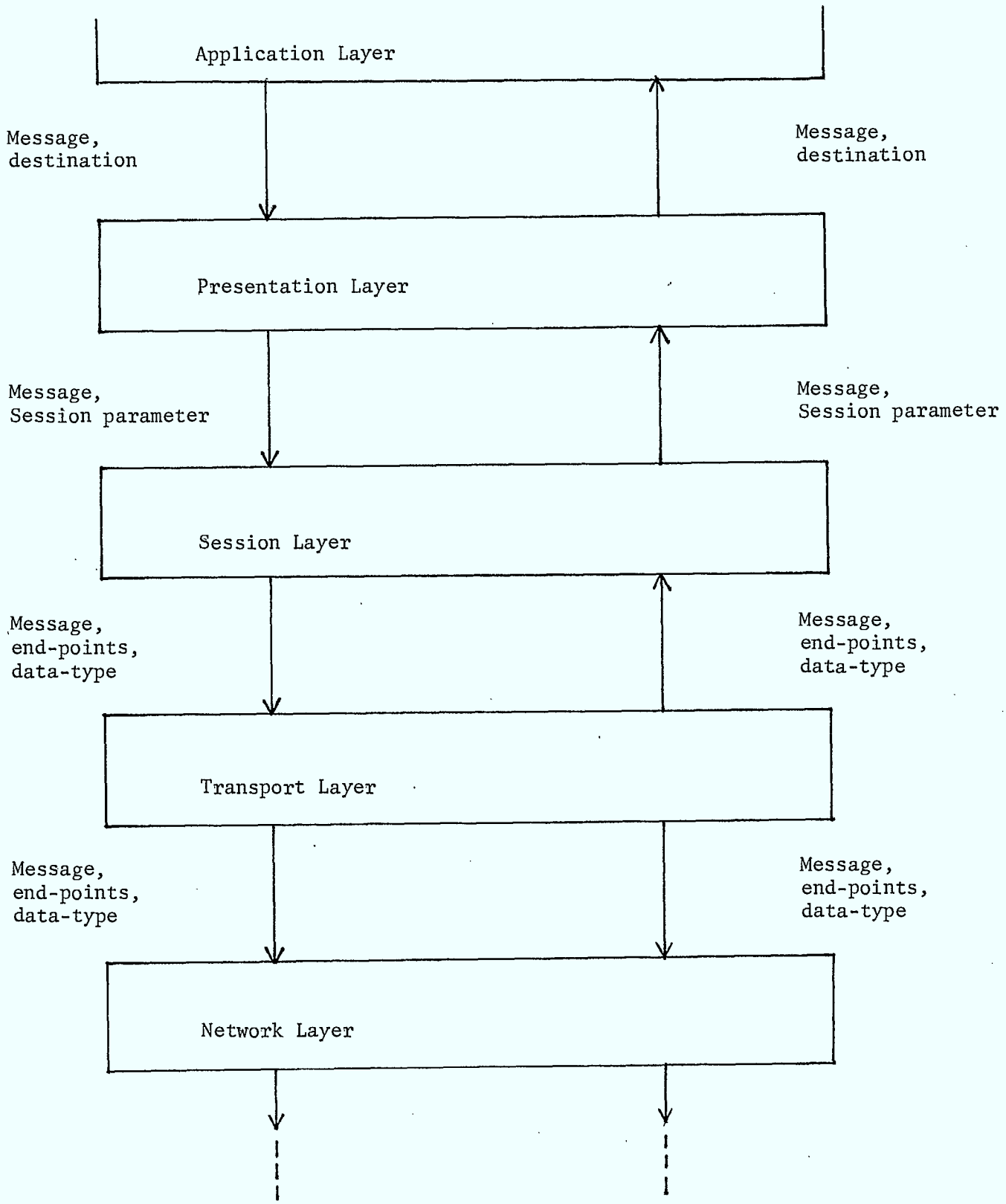



Figure 3.8: Relationships among Interprocess Communication Layers

3. Handle exceptional waiting times and other exceptional conditions

4. Detect patterns of failure of communication.

The Session layer of the OSI model provides the following categories of service to its user [BLAN 81],[ISO 82a]:

1. Session connection establishment/connection

2. Provision of a token or turn to one or both users

3. Expedited data transfer, i.e. immediate delivery regardless of other waiting messages

4. Data transfer.

It may also provide, optionally, the ability to re-synchronize message exchange at some agreed point in the past.

The users of the Session layer in the AASC proof-of-concept system will be the Presentation layer and the local exchange module within the PCU. The local exchange module can be considered an Application layer process.

The following Ada packages specify the detailed services provided by the Session and Presentation layers. (For Ada reference, see [PYLE 81].)

```
-- SESSION PROTOCOL
--   [SCHI 83]

package  OSI_Session  is
  subtype print_name is string (1 .. 12);
  type  connection_spec  is
    record
      initiator_id:  print_name;
      acceptor_id:   print_name;
      connection_id:  print_name;
      data_token_available:  boolean;
      major_token_available:  boolean;
     end record;
  type  direction_spec  is (send, receive);
  type  service_primitive  is (
    request,              --  Invokes, or does not, invoke a
    no_request,           --  service.
    indication,           --  Indicates, or does not, indicate
    no_indication         --  service is invoked.
    response,             --  Completes a service
    no_response,          --  previously indicated, or does not.
    confirmation          --  Completes, for the request
    no_confirmation);     --  indicator, a given service, or
                          --  does not.
type token_transaction is (
    give,
    please);


procedure s_connect (
    result:  out function_result;
    connection:  in out  connection_spec;
    initial_major_token:  in boolean;
    initial_data_token:   in boolean;
    time_limit:  in elapsed_time);


--Function:
--  If the time_limit is zero, the Session layer checks to
--  see if the acceptor process is currently also requesting
--  connection with this process.   If it is, Session
--  establishes the connection and returns the result
--  connection_established.
--  If the specified process is not requesting connection,
```

```
--  the result is process_not_available.
--  If time_limit > 0, Session checks for the specified
--  process' request until either the request exists or
--  the time_limit is exceeded.   If time_limit is
--  exceeded, the result is process_not_available.


procedure  s_release (
    result:  function_result;
    connection:  connection_spec);

--Function:
--  The corresponding process is disconnected.
--  If messages remain to be delivered, the result is
--  messages_waiting.
--  If no messages are awaiting delivery, the result
--  is success.


procedure  s_u_abort (
    connection:  connection_spec;
    message:  any_access);


--Function:
--  Send a process-
--  specific message to the corresponding
--  process, and abort the connection.
--  Other than the specified  message, no
--  undelivered messages will be  delivered.


procedure  s_token_give (
    result:  out  function_result;
    connection:  in out  connection_spec;
    major_token:  in boolean;
    data_token:   in boolean;
    message:  in any_access);


--Function
--  Give the specified tokens to the acceptor
--  process via the specified connection, and
--  deliver the message.   Results are
--  similar to s_data.
```

```
procedure s_token_please (
    result:  out function_result;
    connection:  in  out  connection_spec;
    major_token:  in  boolean;
    data_token:  in  boolean;
    message:  any_access);


--Function:
--  Request that the specified token(s) be given
--  to the initiator, and deliver the message.
--  Results are similar to s_data.


procedure  s_major_synchronization (
    result:  out  function_result;
    connection:  in out  connection_spec;
    serial_number:  out  natural;
    message:  any_access);


--Function:
--  Establish a synchronization point
--  for the purpose of possibly resynchron-
--  izing in the future.   The synchronization
--  point is given a serial number by the
--  Session layer for future reference.
--  If the initiator does not hold the
--  major token, the result is no_major_token.
```

```
procedure  s_resynchronize (
    result:  out function_result;
    connection:  in out connection_spec;
    serial_number:  in natural;
    forced_collision:  in boolean;
    new_major_token:  in  boolean;
    new_data_token:  in  boolean;
    message:  in  any_access);


--Function:
--  Purge all undelivered messages
--  since the synchronization point
--  specified by the serial number.
--  Request establishment of the specified
--  tokens with the initiator as specified.
--  Deliver the message.
--  The acceptor has the option of refusing
--  the re-synchronization by initiating a
--  re-synchronization with the forced
--  collision parameter set.
```

```
procedure  s_data (
    result:  out  function_result;
    connection:  in out connection_spec;
    serial_number:  out natural;
    forced_collision:  out boolean;
    major_token_change:  token_transaction;
    data_token_change:  token_transaction;
    message:  in  any_access);

--Function:
--  Receive via the specified connection
--  any of the parameters, depending on
--  the nature of the corresponding process'
--  activities.
--  If the serial_number is not null, a
--  resynchronization is requested.
--  If forced_collision is true, a
--  previous resynchronization is being
--  rejected.
--  If major_token_change = please,
--  the major token is being requested by
--  the corresponding process.  Similarly
--  for the data token.   If token_change =
--  give  the token is being
--  given to the calling process.
--  The message may be null or non-null.
--  If there is no current message or
--  other communication from the
--  corresponding process, the result is
--  no_communication.

procedure  s_broadcast (
    result:  out function_result;
    initiator_id:  in  print_name;
    broadcast_reason:  in broadcast_reason_value;
    message:  in  any_access);


--Function:
--  Broadcast the message to all connections
--  associated with the initiator.

end  OSI_Session;


-- PRESENTATION  LAYER
```

```
with OSI_Session;
package  OSI_Presentation  is
  type  range_spec  is
    record
      minimum:  natural;
      maximum:  natural;
    end record;
  type  alternation_spec  is
    record
      send_range:  range_spec;
      receive_range:  range_spec;
    end record;
  type message_validation_rules  is
    record
      min_length:  constant  natural := 1;
      max_length:  constant  natural := 256;
      min_value:   string (1 .. max_length;
      max_value:   string (1 .. min_length);
    end record;
  type  message_flow_characteristics  is
    record
      alternation:   alternation_spec;
      min_interval:  natural;
      max_interval:  natural;
      min_response_time:  natural;
      max_response_time:  natural;
    end record;

--Example of message_flow_characteristics:
--  (((1,3)(1,2)),200,400,300,500)
--  which means that the given process
--  must sent between 1 and 3 messages,
--  then receive between 1 and 2 messages.
--  Also, it should not space consecutive
--  messages closer than 200 time units or
--  wider than 400 time units.
--  The process expects response from the
--  correspsonding process no sooner than
--  300 units, and no later than 500
--  units after sending a message.
```

```
--The procedures of the Presentation
--  layer are just those of the Session
--  layer, except for the connect
--  procedure.  In addition, note that
--  extra values of the type function_
--  result have to be defined:
--    message_template_error
--    message_flow_error

procedure  p_connect (
    result:  out  function_result;
    connection:  in out  connection_spec;
    initial_major_token:  in  boolean;
    initial_data_token:  in  boolean;
    time_limit:  in  elapsed_time;
    message_template:  message_validation_rules;
    message_flow:  message_flow_characteristics);

--Function:
--  Same as s_connect, except that
--  message_template is specified to allow
--  checking of message form and message_
--  flow is specified to allow checking of
--  message flow.

end  OSI_Presentation;
```

It can be legitimately asked whether the complexity represented by the OSI Reference Model is appropriate to all kinds of communications, regardless of the sophistication of the nodes in the network. For instance, consider the situation shown in Figure 3.9. One might argue that the simple thruster controller, with very little processing power, does not have the sophistication to follow the protocols required by the OSI Reference Model. Would it not be "simpler" just to connect the controller directly to the PCU with the Thruster Control APF in it?

First, no extra sophistication is required of the thrust controllers. In fact, the purpose of the OSI is to relieve the communicating process of all the burden of handling the mechanism of communication. The implementation of the Presentation layer, the highest layer in the BIU, is such that the application process (in this case the thruster controller) simply sends whatever coded signal it is capable of, and the Presentation layer adds the appropriate information to enable delivery to the Thrust Control APF. Similarly, when the Thrust Control APF sends a control command to the thruster controllers, the Presentation layer sends the bare control signal to the appropriate controller.

This scheme is preferable to an ad hoc connection between the thruster controller and the PCU, for the following reasons:

1. It allows redundancy to be designed into PCUs independent of physical connections.

2. It allows the Thrust Control APF to execute in any PCU on the bus.

3. It allows the mechanisms for communicating with the thruster controllers to be localized in a processor specifically designed for i/o.

4. The modularity of the design allows easy substitution of different components for PCU, NIU, thruster controllers with minimum impact on other components.

5. It allows a uniform design and implementation approach to be used throughout the network, simplifying and speeding up implementation.

## 3.4 Example On-Board Application Model

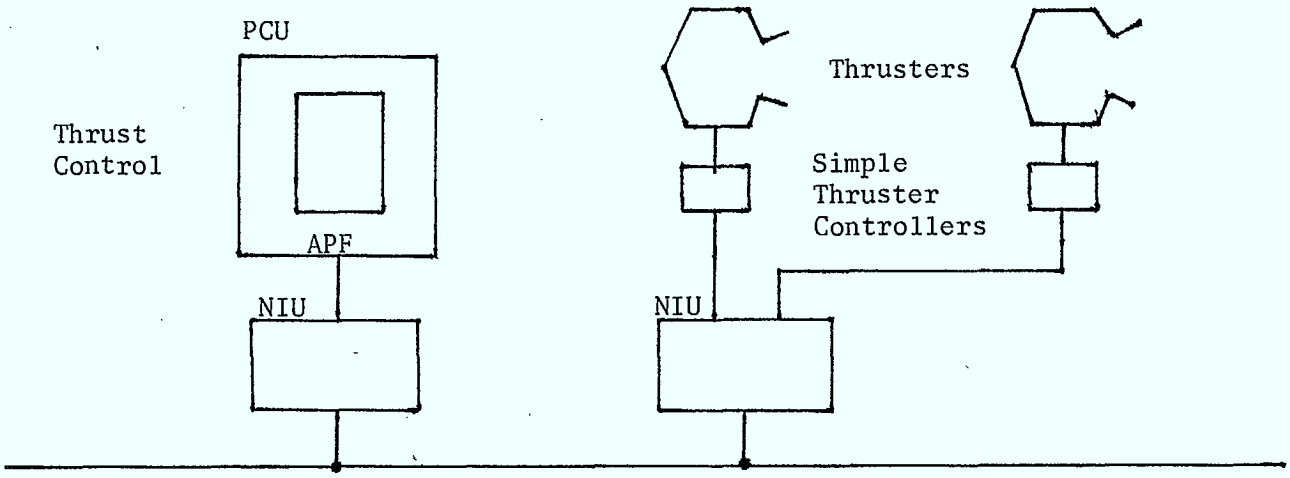



Figure 3.9: Example of a very simple communicating process

The proof-of-concept experiment requires that at least one example of an on-board application function be implemented so as to provide evidence that the proposed AASC can support a realistic subsystem and allow estimation of some quantitative aspects of the proposed system. The Stationkeeping subsystem of a geosynchronous communications satellite provides such an example [EKMA 83]. It is relatively complex, because of the calculations required; it is a critical function of the AASC; and it communicates with at least three other subsystems, namely Orbit Determination, Attitude Determination, and Attitude Control.

The Stationkeeping model assumed here for purposes of software specification is not strictly a realistic one, since it assumes an orbit parameter set which has a singularity for exactly equatorial orbits (ascending node = 0.0). However, different dynamic models may be substituted by altering some data type definitions and some procedure parameters. Further specification and design of the example application processes should be carried out by experienced spacecraft application programmers or by application programmers in conjunction with satellite designers. Existing application programs could be converted to Ada to run in the iAPX 432.

The satellite is assumed to be in a near-equatorial geosynchronous orbit, with its required station at a given longitude with a given deadband. The goal of Stationkeeping is to maintain the satellite within the given deadband with manoeuvres that minimize fuel consumption.

The policies for timing of Stationkeeping functions may be specified in several ways:

1. Ground control decision

2. OAM decision with ground control consultation

3. OAM decision

4. Deterministic algorithm within Stationkeeping function.

For purposes of the proof-of-concept, we will use the last alternative.

For the purposes of rough specification of Stationkeeping functions, we assume the following general sequence:

1. Determine current orbit

2. Predict orbit events

3. Plan manoeuvre

4. Execute manoeuvre

5. Check effects of manoeuvre

Furthermore, we assume that a manoeuvre may require the sequence:

1. Change attitude

2. Change velocity vector

3. Reset attitude

It is important to point out that it is not our intention nor is it within our expertise to specify an actual stationkeeping algorithm. It is sufficient for our purposes to specify a model that approximates the real case and provides similar computational complexity and interprocess relationships.

## 3.5 Monitor Station

The AASC network is connected via a gateway node and link of appropriate medium to a ground station, and thence possibly to a global network. Ground control exists either at the first ground station (no global network) or at some node in the global network [GOMI 83a]. Ground control will conduct dialogues with the OAM, and in exceptional cases issue messages directly to a spacecraft subsystem.

In the proof-of-concept system the configuration will be less ambitious, as shown in Figure 3.10. The analogue of a ground station will be the Monitor Station, which is implemented as a node in the AASC networking. The functions of the Monitor Station are:

1. Accept input from an operator/experimenter

2. Issue requests to OAM

3. Receive and display messages from OAM

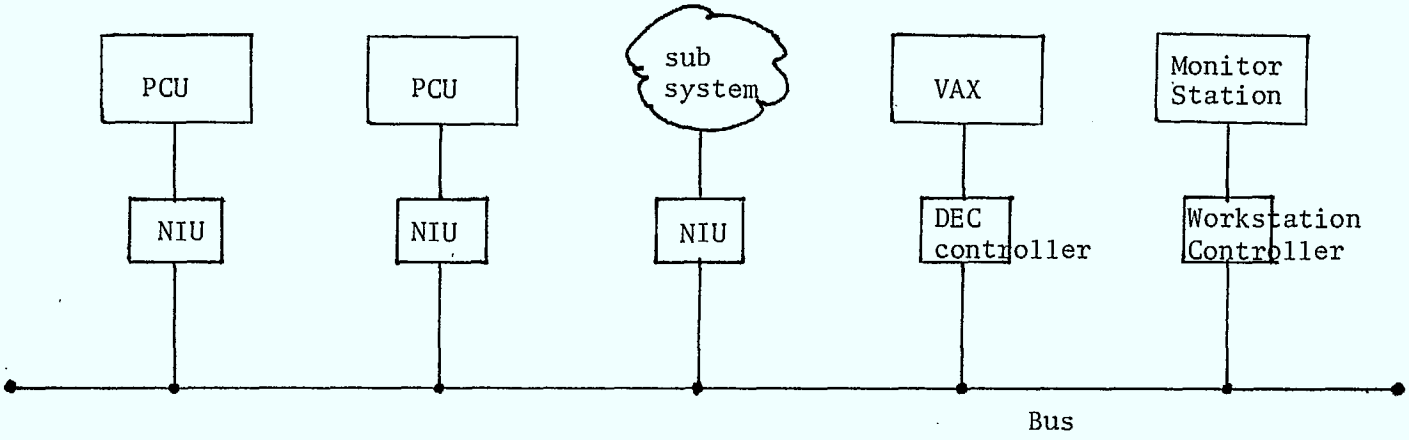



Figure 3.10: Monitor Station Configuration

4. Monitor and display traffic on the bus

5. Monitor and display traffic through each layer of any NIU.

6. Send a message to the FTM.

To accomplish this, the Monitor Station (MS) must be able to:

1. Communicate via the standard AASC communications protocol (see Section 3.3)

2. Intercept any transmission on the LAN and analyze the message contents to determine its contents as they relate to Link, Network, Transport, Session, and Presentation layers in the two communicating NIUs.

Apart from this latter specialized ability, the MS behaves at the system level like any other node on the net.

# 4. SYSTEM SOFTWARE

## 4.1 On-Board Autonomy Management Software

### 4.1.0 Overview

The software of the OAM (for this proof-of-concept design) consists of the On-board Consultation System (OCS) which interfaces the NIU protocol modules, and a few interfacing modules between them. The OCS will be built on the ARBY expert system framework, and has the following major software components:

- UNIX operating system for VAX-11/780, Berkeley Version 4.2 or later
- Franz LISP package that comes as a part of the Berkeley UNIX
- Smart Systems Technology's ARBY expert system for building diagnostic expert systems. It has, as major software subcomponents, the following:
  - HYPO inference engine
  - IFM user interface
  - DUCK deductive retriever
  - NISP LISP macro library

### 4.1.1 Structure of OAM

The OAM consists of the OCS and the OAM Control Subsystem. The functions of the Control Subsystem are not essential to achieving the objectives of the proof-of-concept system, and thus are minimized during this phase of the project. For all practical purposes, the OCS represents the OAM. Therefore, only the functionality of the OCS is described here.

Two major software modules in the OCS play a key role in achieving the consultation. The inference engine, or the HYPO, conducts the reasoning process, while the user interface module, or the IFM, performs the dedicated high level system i/o to the external world. These modules are implemented mostly in Franz LISP, but also use macros defined in the NISP macro library. NISP houses LISP functions and provides easier syntax, data types, improved number handling, and convenient control structures. The LISP syntax, if not semantics, is known to be rather limited in these areas, and

NISP attempts to unload the burden from AI programmers by encapsulating these utility functions in more visible formats.

Access to an assertive database is handled exclusively by a deductive retrieval module called DUCK. The deductive database records the progress of consultation in a relational database formalism. The database is called deductive as it permits deductive access. Figure 4.1 shows the software structure of the OCS.

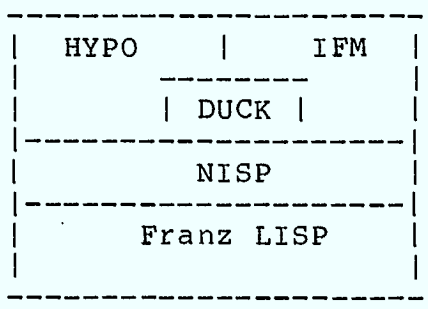



Figure 4.1 OCS Software Structure as supported by ARBY

The reasoning machine (or inference engine), HYPO, is further divided into two distinctive parts: a hypothesis manager, and a hypothesis chooser. The former keeps track of the current complex hypothesis. The latter is responsible for making choices of individual hypotheses.

### 4.1.2 Internal Workings of the Reasoning Machine

The basic operation of the OCS is described in 3.1.1 above. Further details of the internal workings of the consultation system as supported by the ARBY inference mechanism are discussed below.

Sequence:

The reasoning sequence is started when a request for consultation is received by the OCS. Such a request arrives at the OCS via the AASC on-board networking which adopts the OSI protocol formalism. After the necessary initialization, the IFM issues an initial Interactive Frame (IF), demanding a set of findings from the user. It is typically given as a

high level question in order to define the problem. The generalized approach at the beginning helps in picking the right hypothesis from the start. Following this, the DUCK deductive retriever retrieves from the KB hypotheses that account for the initial finding(s). It searches through the KB, backward-chaining through rules using pattern matching techniques. This way, an initial set of hypotheses is created, with at least one hypothesis for every finding. When more than one hypothesis is involved for a given problem, it is said that a complex hypothesis is found.

The system uses the choice rule, or "the survival of the fittest" rule so to speak, to eventually eliminate spurious hypotheses. It amasses evidence either for or against one or another hypothesis by proposing tests to be done to gather more evidence. The problem of complex hypothesis is dealt with by favouring one explanation more than another. The favoured hypothesis is one that accounts for more findings.

Testing:

The choice mechanism (the hypothesis chooser of the HYPO reasoning machine) uses the estimate of profitability to decide which test (request for findings) to run. The profitability of a test is based on the ratio of the change the finding would produce to the cost of producing it.

A test on-board the AASC satellite would be, for example, the running of a diagnostic program on one of its transmitter modules. This involves decommissioning the unit, supplying the power, assigning a communication channel, if the test involved an external station, and supplying other logistic supports such as temperature control around the unit, and orientation management of antenna. Knowledge of the spacecraft design and operation is needed to estimate the costs involved in such operations, and the OCS knowledge base (KB) would contain such information for each of the anticipated tests.

Involvement by the domain expert would be seen in the building of hypotheses as well. In addition to knowledge of faults and their causes, the extent of the benefit anticipated by knowing the possibility of faults is a form of domain knowledge itself. This is a type of knowledge that is related to the way inference is conducted. Study of such "control knowledge" and methods of its application is a hot topic in the current study of ES technology. Also, in the

diagnostic ES, what the expert would do in the event of a fault would be a crucial bit of knowledge. Such knowledge would be stored separately from diagnostic or control knowledge, probably as "rehabilitation knowledge". It would guide the spacecraft in its effort to recover from faults.

In contrast to these symbolically recorded knowledge data in KBs, diagnostic programs that run under the direction of such collective knowledge are a collection of more conventional algorithms written in conventional procedural representation. One of the major differences between the AI programming and conventional programming is that the former permits symbolic (a form suitable for semantic expressions) interaction between units of knowledge embedded in the program, while the latter is limited to numerical and logical manipulation of narrowly defined entities. Hence, while the creation of sophisticated diagnostics program sets for diagnosing transmitter modules may require extensive domain knowledge typically found in the experience of the designer of such modules, these programs are not called AI programs. Being non-AI programs, they do not possess the flexibility that makes symbolic manipulation so attractive and powerful. Only by succeeding in extracting and representing the knowledge of domain experts in a format that permits flexible access and manipulation of that knowledge does on-board intelligence, such as that intended for the AASC, become meaningful.

Estimates of what it costs to perform a test may vary widely from no cost at all (the test has already been conducted, and its result is unlikely to have changed) to very costly, involving running several diagnostic programs, each taking hours, or needing extensive exchanges between the on-board consultant and the external tester (not necessarily human). Simple observations of system status, such as power source voltage checks, will typically have a low cost estimate. Identifying the number of available buffers of a certain type may take more effort, involving messages to be sent to the FTM for request and confirmation, and processing the reply. The OCS may engage in a more elaborate exchange with the FTM. For example, the OCS may direct the FTM to reconfigure an on-board process structure that deals with the control of a vision analysis unit, and then request it to perform a test using a standard series of process test protocols. In such a case, the estimation of the cost would be more involved.

The Reasoning Cycle:

One deduction cycle, as described so far, involves sending out requests for new information to the user, accepting findings, searching KB for hypotheses that best explain the findings, adjusting the score on the hypotheses so far chosen, eliminating spurious hypotheses, elaborating and expanding on the remaining hypotheses, and identifying tests to be conducted to clarify them. The cycle continues until there is no more need to run external tests, or to collect facts. At the end of each cycle, the inference engine (HYPO) checks the profitability of the best proposed test against what it requires to take a decision.

Messages:

In the AASC, such requests are carried over the on-board network (LAN/VLAN) to its users (ground control, the FTM, the EOA, or the MS). Ground control would exchange messages with the OCS via the down-link subsystem or the ground-gateway node, and the up-link or the space-gateway of the ground station. The subsystem structure is shown in Figure 4.2. This is simplified in the proof-of-concept system to merely perform simulated ground control functions or as a monitor station (MS) which is directly linked to the "on-board" network accessed by the OAM, as shown in Figure 4.3.

The exchange between the OCS and the application node takes place on-board. The application nodes are those hardware/software entities that are involved in various on-board sensory/control activities. In executing their own on-board functions, some of them access the OAM via the network. The type and nature of the consultation request depends on the application. See Section 4.4 below for further detail.

The FTM also issues consultation requests, or reports problems to the OAM. This happens when the on-board fault-tolerance manager fails to solve problems by itself. The message format and contents will be determined when the KB of the proof-of-concept system is implemented. This is true for other message formats between the OCS and the EOA and the MS.

The request for a test is sent out to the FTM only after some tests involving the software manager are found to be worth doing. The requests for tests are, as in the case for hardware tests, issued through the IFM module of the OCS, and sent over the on-board network to the FTM. The

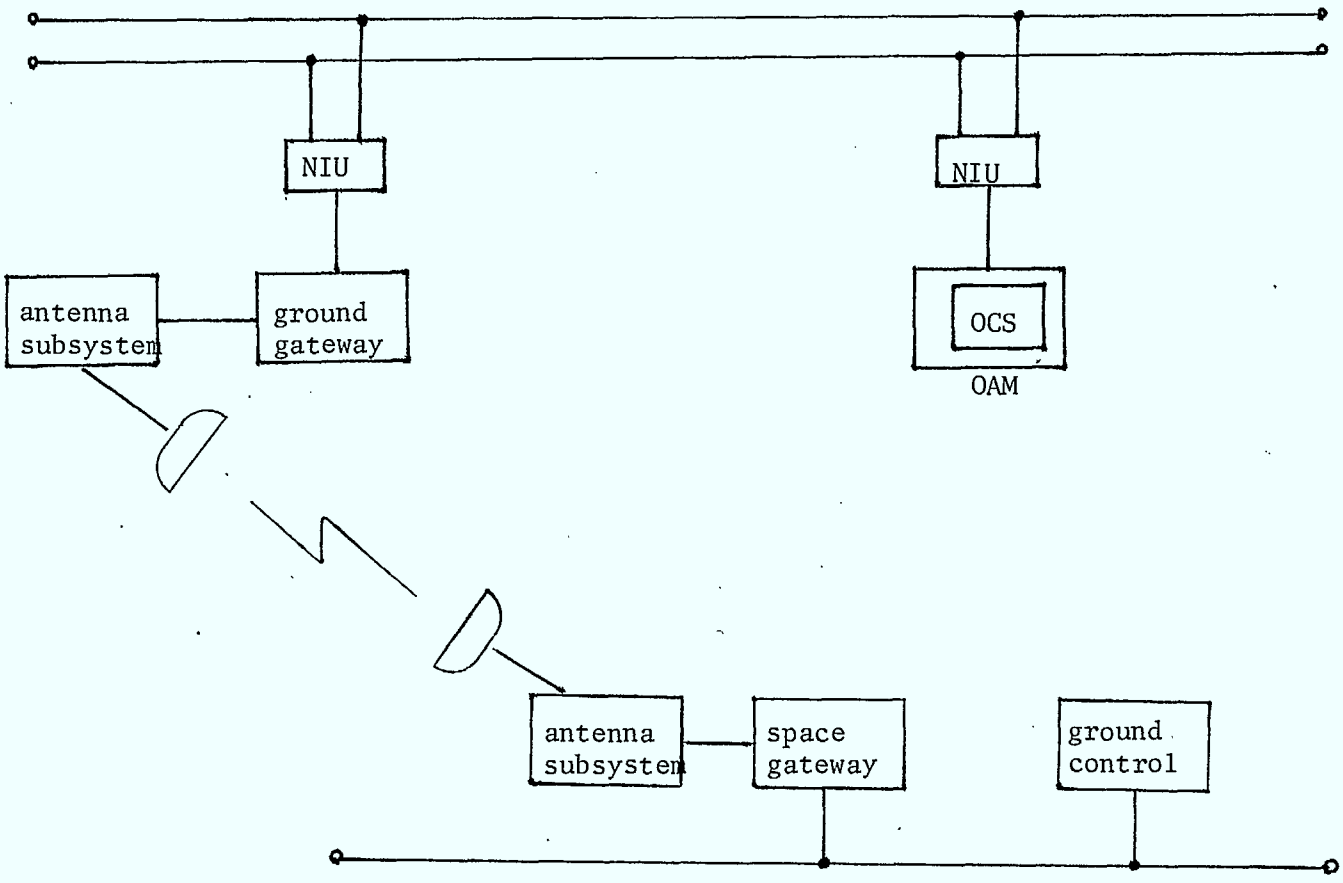



Figure 4.2: Gateway link between ground control and the AASC.

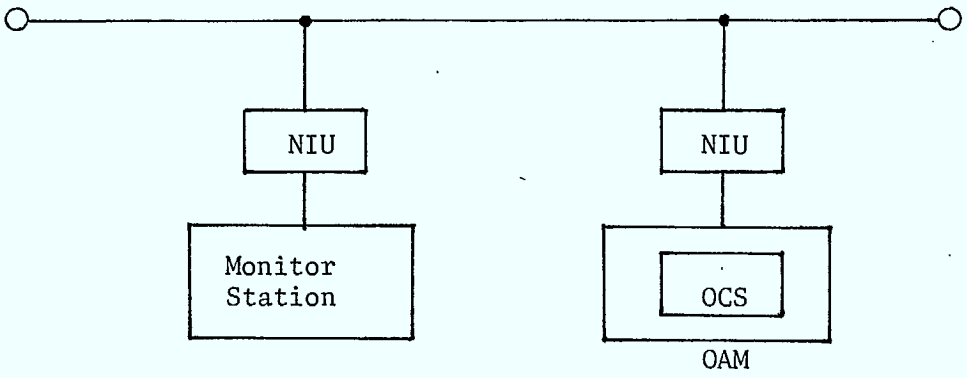



Figure 4.3: Monitor Station as a simplified form of Simulated ground control

Table 4.1: Example of an On-Board Malfunction Analysis

| Findings | Hypotheses | Score by tests | |
|---|---|---|---|
| fi: transmission carrier intermittent | h1: oscillator temperature high | 75 | |
| | h3: connector to antenna subsystem deterioration | 30 | * |
| | h4: transmitter memory controller lost by radiation | 25 | * |
| f2: transmitter control software malfunctioning | h2: on-board loader malfunction | 80 | |
| | h4: memory controller lost by radiation | 65 | |
| | h5: object module in ROM destroyed | 40 | * |
| f3: memory for control software parity error | h4: memory controller lost by radiation | 90 | |
| | h6: RAM memory bank failure | 15 | * |

findings by the FTM on the functioning of on-board software are sent back to the IFM, also via the network. Once received by the IFM, like facts regarding conditions of hardware, the results become new assertions (or findings) in the database. The hypothesis chooser of the HYPO applies the new findings in its attempt to reevaluate the hypotheses on these software conditions.

Control:

The hypothesis chooser uses an algorithm (in the broader sense of the word) to retain "valuable" hypotheses and discard useless ones. For example, let us assume a mismatch is reported by a software voter within the FTM. The OCS, according to a rule in the on-board KB, orders a test from the FTM. Suppose the test only turned up a header syntax error in one of the messages issued by processes involved in the voting. In such a case the OCS, like a human consultant in a similar situation, would attempt to chase this lead as hard as it can, performing those tests which are profitable enough. However, if, for example, there are several findings after the initial test, such as the discovery of a dormant process, memory bank failure, buffer checksum error, etc., the system does not have to chase these findings as hard, and lets the survival of the fittest rule take its own effect.

An Example:

As explained above, a key principle of operation of the HYPO chooser is, "How hard should it try to eliminate all but one hypothesis?" The example shown in Table 4.1 demonstrates this principle.

The table shows the results of testing after a refinement cycle. The example shows multiple hypotheses explaining the same finding, and a single hypothesis explaining multiple findings. Those hypotheses with test results marked by * are the ones to be ruled out due to their "below threshold" score. Hypothesis "h4: lost by radiation", which counts for "f2: software malfunctioning" and "f3: memory parity", does not explain "f1: intermittent transmission" well enough, and hence, is ruled out as the cause for that finding. An important distinction is that it failed to account for the finding, and that it is not necessarily false. Actually, the confidence in "h4: lost by radiation" is increased by the HYPO chooser as it accounts for both "f2: software malfunction" and "f3: memory parity" (the chooser adds bonus points

to those hypotheses that account for multiple hypotheses, each time this happens). The set (h1, h4) explains the findings better than any single or combined hypothesis. Note that "h4: lost by radiation" won out over "h2: loader failure" as the explanation for f2, since this explanation also accounts for the "f3: memory parity" and, hence, obtained a bonus (in this case 50 points).

4.1.3 Refinement of Hypothesis

Once a compound hypothesis is arrived at, as in the above example, the OCS attempts to refine it by obtaining a more specific explanation for each finding. Thus "f4: memory controller lost to radiation" would be traced down to the subcomponent within the memory controller and what type of radiation damage has caused the loss. While such information may not help revive the lost controller, it will greatly assist on-board reconfiguration efforts and provide data for future design improvements. The refinement takes place in the cycle described earlier, and the cycle continues until no further explanation is possible. A set of rules is retrieved from the KB which have the potential for establishing subsidiary hypotheses. Profitability criteria (cost-effectiveness estimates) are calculated on each of them, and hypotheses are ordered according to the results of the calculation.

4.1.4 Treatment of Heuristics

Unlike those in MYCIN or PROSPECTOR, the uncertainty mechanism in ARBY depends on a more practical means for incorporating heuristics into decision judgements - counting and evaluation of the viability of each hypothesis at the end of every refinement cycle. Hypotheses that do not achieve a threshold are considered hypotheses without sufficient evidence, and are discarded. Thus, unlike some other ES, there is no need to combine uncertainty factors of one type or another across conjunction, disjunction, and implication. Carried to extremes, this method may discard an essential hypothesis which did not initially have a sufficient score but might have improved later. However, executed with discretion (mainly by adjusting the threshold appropriately) the method helps greatly to economize in the handling of the necessary heuristics, while maintaining reasonable control over the course of reasoning. The issue of guiding the inference process based on some control algorithm is an on-going research topic in the study of ES (see, for example, JPL's efforts on this issue in the application of ES to

space systems in [GOMI 83c]).

### 4.1.5 Hypothesis Choosing

Hypothesis choosing, and hence pruning, has two main objectives. The first is to prevent stretching a hypothesis in an attempt to explain everything. In the example above, "h4: memory controller lost by radiation" as an explanation for "f1: intermittent transmitter", had that potential. The HYPO chooser, at the end of the current cycle, will eliminate this possibility since that hypothesis explained rather poorly the transmitter problem. However, it is still maintained as a dominant hypothesis for the other two findings. Other low-score hypotheses, namely "h3: antenna connection deterioration" for "f1: intermittent transmitter"; "h5: object module loss" for "f2: transmitter software malfunction"; and "h6: memory bank failure" for "f3: memory parity error"; are similarly discarded.

The second step in the selection process picks up the hypotheses which are the only alternatives left for their respective findings. "h1: oscillator temperature high" for the transmitter malfunction (f1) and "h4: lost by radiation" as the explanation for "f3: transmitter memory parity" are picked up for that reason.

Thirdly, the chooser selects hypotheses for the remaining findings which are not accounted for so far. During this step, if a hypothesis has already been selected by the previous step, it is given a bonus point. Thus "h4: loss by radiation" for f2 beats out "h2: loader failure", as the bonus points of 50, say, are added to h4 because it was selected in the previous step. Otherwise, it would have been lost to "h2: loader failure" as the explanation for f2. Thus, in the above example, we arrived at the selection of the hypotheses set (h1; h4) at the end of the current stage. This "test and refine" approach simulates very well the process of diagnosing complex problems by human experts.

## 4.2 FTM Software

### 4.2.1 Process Management

The mission-oriented functions of the spacecraft are termed Application Functions (APFs). Each is carried out by one or more Application Processes (APPs). The Application Functions are under the direct supervision of the

Basic_Application_Manager (BAM) which is a package of procedures within Layer 4 of the FTM. Its functions are to:

- handle exceptions raised in BPM. These can be classified as
  - errors in the way OAM called BAM
  - transient conditions which can be retried
- detect permanent faults
- detect anomalous conditions or patterns of error
- create a guardian for each APP process to carry out Layer 3 fault-tolerance functions.

To carry out these functions, BAM must know what processes are required to accomplish a given function. Hence there must be enough information passed by OAM to determine function -> process mapping. If this mapping is in the KB, then the information can be formatted and passed directly. If it should only be available at Layer 4, then it must be parameterized by using generic procedures in BAM, or placed in non-volatile storage for constant availability.

Functions in the BAM package will include:

Invoke_function
Stop_function
Restart_function
Suspend_function
Get_function_information

The following Ada package specifications describe the Fault Tolerance Management layer of the AASC.

```
package  FTM_Definitions  is
type  function_result  is (
  success,
  function_invoked,
  no_such_function,
  cannot_create_process,
  connection_established,
  process_not_available,
  messages_waiting,
```

```
    no_major_token,
    no_communication);

end  FTM_Definitions;


-- BASIC  FUNCTION  MANAGEMENT

with  Basic_Process_Management,
Descriptor_Definitions, iMAX_Definitions,
Untyped_Ports, Process_Globals_Definitions;
package  Basic_Function_Management  is

  --Function:
  --  Basic_Application_Management
  --  provides functions to manipulate application
  --  functions that consist of sets of cooperating
  --  processes.   It creates and starts the
  --  necessary processes, can retrieve inform-
  --  ation about them, and can stop and destroy
  --  them.   Furthermore, it attempts to maintain
  --  the reliability of these application functions.

use  Basic_Process_Management;


--   TYPES

type  function_state  is (
  healthy,    --  performing function correctly
  suspended,
  suspect,    --  anomalous conditions are
              --  under investigation
  degraded,   --  the function is degraded
  dead);      --  no aspect of the function is
              --  being carried out.



type  failure_descriptor  is
  record
    exception:  exception;
    --   other fields may have to be added.
  end record:


type  function_result  is (
  success,
```

```
  no_such_function,
  cannot_create_process);


type  process_list_element;


type  process_list_pointer  is
  access process_list_element;


type process_list_element is
  record
    process_field:  process;
    next_process:   process_list_pointer;
    prev_process:   process_list_pointer;
  end record;


type  recovery_descriptor  is
  record
    recovery_result:  function_result;
    --  other fields may be added.
  end record;



type  function_descriptor  is
  record
    name:  print_name;
    component_processes:  process_list_pointer;
    timer:  process;
    state:  function_state;
    error:  failure_descriptor;
  end record;


type  function_report  is
  record
    description:  function_descriptor;
    information:  any_access;
  end record;


type  error_type  is  (
  no_error,
  possibly_transient,
  permanent);
```

```
max_duration:  constant short_ordinal
                := short_ordinal'last;
               --  This gives a way of specifying
               --  the maximum duration for
               --  a particular state.
max_time:  constant short_ordinal
                := short_ordinal'last;
               --  This gives a way of specifying
               --  the furthest possible point
               --  in the future.
```

```
-- FUNCTIONS OF BAM

function Invoke_function (
    f: print_name;
    duration: elapsed_time := max_duration;
    expiry_time: absolute_time := max_time)

  return function_result;

  --Function:
  -- Get function structure from non-volatile
  -- storage.  If duration or expiry time is non-
  -- null, create a timer to limit execution
  -- time of f.  Create one or more
  -- guardians/owners and associate
  -- processes with guardian/owner.
  -- Each owner is capable of handling
  -- faults at Layer 3 for its associated
  -- processes.
  -- Update the function_descriptor in the
  -- function_directory, and start the
  -- processes and timers.


function Suspend_function (
    f: print_name)
  return function_result;

  --Function:
  -- If the function exists and is not dead or
  -- suspended, suspend it
  -- by stopping the component processes and
  -- updating the function directory.


function Stop_function (
    f: print_name)
  return function_result;

  --Function:
  -- The named function is stopped by
  -- stopping the associated processes.


function Restart_function (
```

```
    f:  function_name;
    duration:  elapsed_time;
    expiry_time:  absolute_time)
  return  function_result;

  --Function:
  --  If the function is not dead, restart the
  --  function f by starting the component processes.
  --  If the function has a timer, and duration
  --  and expiry_time are null, the function is
  --  still subject to the timer.
  --  If the function has a timer and duration
  --  or expiry_time are not null, the current
  --  timer is destroyed and a new timer is
  --  created.   If there is no timer, and
  --  duration and expiry_time are null, the
  --  function's processes are allowed to run
  --  indefinitely.
  --  If there is no timer, and duration and
  --  expiry_time are not null, a timer is
  --  created.


function  Get_function_information (
    f:   function_name)
  return  function_report:

  --Function
  --  The function f is queried to obtain a function-
  --  specific description of its state.   This
  --  description is supplied by a process called
  --  xx...x_reporter, where xx...x is the APF print-
  --  name.   This process must be
  --  built into every application function.  Its
  --  purpose is to report its state in application-
  --  specific terms so that Layer 4 of AASC and
  --  OAM can make decisions about the function.
  --  It may be a null process.
```

```
--  OWNER package definition

with  Basic_Application_Manager;
generic
    f:  in  out function_name;
    guardian:  in  out  port;


with  procedure  Isolate_failure (
    failed_process: in process);
    --  This procedure must be custom-designed
    --  for each application.


with procedure  Analyze_failure (
    failed_process: in process);
    remote_threshold:  constant natural
      :=  natural'last;
    --  This procedure must be custom-designed
    --  for each application.

    recent_threshold:  constant natural
      :=  natural'last;


package  Owner  is

--Function:
--  This is a generic definition of an "owner"
--  of one or more processes of an AASC function.
--  The package requires, for an instantiation
--  of an owner of a particular process, the
--  coding of each of the procedures listed
--  above to isolate and analyze the failure
--  of the particular process.
--  Detection of the failure is carried out
--  by the assertions and exception-handler
--  of the specific failed process, or, if
--  no exception-handler exists, by the
--  operating system.  The operating system
--  sends the failed process to its guardian,
--  which was associated with it when the
--  process was created.  The owner receives
--  the failed process from the guardian.
--  The parametric procedures and those
--  defined in the package perform all the
--  functions of Layer 3 of the FTM.   Any
--  exception in these procedures causes
```

```
--  assertion of a permanent failure of the
--  failed process, which is propagated to Layer 4.


procedure  Inform_cooperating_processes (
        failed_process:  in process);

--Function:
--  The procedure broadcasts a message to the
--  Presentation layer to suspend all
--  sessions involving the process.


procedure  Determine_permanence  (
    failed_process:  in process;
    failure_description:  failure_descriptor;
    permanent_failure:  boolean);

--Function:
--  Log the failure event, and determine the
--  frequency of this type of failure for the
--  process.    If the failure frequency is
--  above a threshold, assert a permanent
--  process failure.


procedure  Log_failure_event (
    failed_process:  in process;
    failure_description:  in failure_descriptor);

--Function:
--  Record the failed process name and failure
--  description on non-volatile storage.


procedure  Check_failure_event (
    failed_process:  in process;
    failure_description:  in failure_descriptor;
    permanent_failure:  boolean);

--Function:
--  Look up all instances of the described failure
```

```
--  in the failure event log in the intervals
--     (remote_past, now)
--  and
--     (recent_past, now)
--  If remote_count >  remote_threshold,
--     assert permanent_failure
--  If recent_count > recent_threshold,
--     assert permanent_failure.


procedure  Recover (
    failed_process:  in process;
    failed_processor:  in processor;
    failure_description:  in failure_descriptor);

--Function:
--  If failed_process is not null, and failure_
--  description is not permanent, then recover
--  from transient process fault.
--  If failed_processor is not null, then
--  recover from permanent processor fault.


procedure  Recover_transient_process_fault  (
    failed_process: process);

--Function:
--  Find the most recent checkpoint in the
--  checkpoint data-base.
--  If the checkpoint exists, report the
--  restart to cooperating processes by
--  issuing a request to the Presentation
--  layer to resume any session in existence
--  for this process.
--  Restart the process with checkpoint
--  data if it exists, or in its initial
--  state if not.


procedure  Find_checkpoint (
    failed_process:  in process;
    checkpointed_process:  out process);

--Function:
--  Search the checkpoint data base
```

```
--  for the most recent checkpoint for
--  this process.   If there is no such
--  checkpoint, return null.



procedure  Report_to_cooperating_processes  (
    failed_process:  in process);

--Function:
--  Issue a request to the Presentation
--  layer to resume all existing sessions
--  involving the recovered process.



procedure  Recover_permanent_processor_fault (
    failed_processor:  in processor;
    failure_description:  failure_descriptor);

--Function:
--  Determine failed processor type.   Disable
--  that processor.
--  If the processor is a single point of
--  failure, then assert permanent failure of
--  the process.
--  If an alternate exists, then assign the
--  process to an alternate processor.



procedure  Maintain_process (
    g:  in process_port);

--Function
--  This is the main procedure of the Owner.
--  It receives a process from the guardian,
--  then uses the procedures defined in the
--  package parameter list and within the
--  package to maintain the process.
--  First, isolate the failure.
--  Second, analyze the failure for
--  transience or permanence and any other
--  process-specific characteristics.
--  If the failure is not permanent,
--  attempt recovery.
```

```
--  Report the failure event and, if
--  applicable, the recovery of the
--  process.


procedure  Function_timer  (
    function_description:  in function_descriptor;
    expiry_time:  in absolute_time);

--Function:
--  Periodically checks current time against
--  expiry time.   If current_time >= expiry_time,
--  calls stop_function to stop all processes
--  within the given function.   This process is
--  optionally created when the function is
--  started, and has its iMAX 'periods' parameter set
--  so that it only does one or two time-checks
--  per time-slice, so as not to use up much
--  processor-time.
```

## 4.3 NIU Software

### 4.3.0 Overview

As discussed in detail in 3.3 above, the NIU adopts the OSI protocol structure. The fundamental software requirement is very clear: for whatever software structure that exists on the OAM, the FTM, the Example On-Board Application or the Monitor Station accessibility is established to the OSI model. In search of such software structures, we came across several attempts by the industry and academia. Most of the achievements in terms of transportable software packages are still very exploratory, and hence a considerable amount of work is expected to build the linkage between the above mentioned stations of the proof-of-concept system. In particular, there is no commercially available Session layer protocol module that can be used for the proof-of-concept system: it must be implemented for each of the stations in the AASC. The state of the business in this area (OSI software modules) is very volatile and one must anticipate a considerable amount of effort in writing the "glue" modules to establish the above mentioned linkage between the OSI and the application.

There are a few protocol packages that offer support for Link, Network, and Transport layers. Yet most of them are not necessarily built after the OSI model. A historical reason for this is the large investment made, particularly during the late '70s, to develop and make available to the market, software products that support a hierarchical approach to computer communications. Such products roughly followed the OSI model, which was barely taking shape. A typical example of that type of software is the TCL (Transport Control Layer) protocol offered by XEROX Corporation in the early '80s. Many software houses and manufacturers of business, scientific, and control computer systems adopted it, until in 1982, ECMA adopted the Class 4 of the Transport layer specification by the working group with CCITT/ISO. The move greatly increased the visibility of the international efforts up to that point. By then moves by other standardization organizations, such as the National Bureau of Standards (NBS) and the American National Standard Institute (ANSI) to adopt a protocol structure and detail specification similar to the ISO proposal, put an end to ad hoc protocol development business. Software manufacturers have since pledged adoption of the proposed ISO/OSI protocol. The present problem is caused by this transitional condition. The lack of readily available procotol modules should by no

means be taken as a reason to abandon the OSI approach, because of the logical clarity the model provides, and the increased acceptance of that fact in the industry.

The NIU is responsible for providing network access support up to Presentation layer services to the processes within the PCU. This implies that it must provide implementation of the layers below Presentation in the OSI: Session, Transport, Network, Link, and Physical. Class 4 Transport layer services will be provided by the AASC Transport layer. Class 4 includes error detection, recovery, and correction for all but misdelivered messages.

Session and Presentation layer services are provided by two packages, OSI_Session and OSI_Presentation. For the purposes of proof-of-concept, the Presentation layer provides only one extra service over Session: message format and flow checking. It allows the APP to specify maximum and minimum message lengths, and to set bounds for response time, message exchange patterns, and delivery times.

The Session layer allows an APP to establish a session connection with another APP and exchange data with that APP.

In the following subsections attempts are made to identify the characteristics of each station in regard to accessing the AASC networking, and the currently available options one can pursue to establish such connections.

### 4.3.1 NIU Software for the OAM

The OAM, in its present design, exists in a virtual LISP computer. Since all its users will speak to it by setting up a session, (one consultation session corresponds to a session in the OSI terminology) the OAM must be able to handle needed subfunctions of procotol layer functions up to the Session layer. Assuming that the Ethernet access under Version 4.2 of the Berkeley UNIX permits modules written in Franz LISP to access the UNIX Ethernet driver, one will have to write both Transport and Session layer protocols in LISP. Although this sounds like a tremendous amount of work, experienced LISP users testify that such work can be done with relative ease. For example, D-series machine by XEROX Corp. has all its system software written in LISP, including the operating system and device drivers.

Another option is to adopt Interlan's Network software product for VAX-11 computers which is currently under devel-

opment. The company has made a pledge to provide protocol software which covers all seven layers (with the possible exception of the Application layer for reasons of practicality) of the OSI model. They already have a tentative TCL (which is likely to be a XEROX version) and are working on the Presentation-Session layers. When these materialize, the engineering tasks needed to establish the linkage between the LISP modules (OCS) and the protocol layers will be reduced.

Digital Equipment Corporation is apparently working on an OSI protocol product. When this is completed, it is supposed to provide all seven layers. Running Franz LISP under the UNIS operating system would permit a relatively easy linkage between the two software structures.

4.3.2 NIU Software for the FTM and EOA

Since the FTM and EOA will be implemented using the 432, the issue is to link it to the Ethernet and provide upper layer protocol modules to support 432's effort to access the network.

All i/o, including network access, by the 432 system is done through the links between the 432 complex (PCU) and the Attached Processor (AP). When the network access software and hardware are included, the AP itself becomes the NIU.

In terms of networking, software-wise, the RMX 86 operating system talking to the iNA 950-1 protocol package constitutes a good framework. The interprocessor communication utility package iMMX 800 is used to link the processor that houses the RMX 86, and the iNA loaded on the communication processor.

A more sophisticated implementation involves Intel's new Datacom computer, iSBC 186/51 which is still under development. It would appear that there are engineering issues involved in making it an AP. However, they do not look serious. This would make the interconnection between the 432 and networking far simpler. Our preference is to pursue this route.

The Session layer protocol needs to be implemented. This, in the FTM's case would be written in languages available under RMS 86, such as PASCAL 86, PL/M-86, or "C" for 8086. Whether the Session protocol should be in the processor that houses the AP, or on the Communication processor,

is another level of refinement in specification in describing the NIU software for the FTM, as there a few options in choosing the NIU software.

## 4.4 Example On-Board Application Software

The following packages specify the Stationkeeping function which is used as an Example On-board Application (see Section 3.4). Figure 4.4 shows the relationship of Stationkeeping to other application functions.

```
package Geometry_Definitions  is
    pi:  constant float:= 3.14159265;
  type  angle_in_degrees is new float digits 5;
  type  angle  is new float digits 5;
  revolution_in_degrees:  constant
    angle_in_degrees:=  360.0;
  revolution:  constant angle:= 2*pi;
  subtype circular_angle_in_degrees is
    angle_in_degrees range  0 ..  revolution_in_degrees;
  subtype  circular_angle  is
    angle  range  0 .. revolution;
  subtype  orbit_dimension  is
    float  range  0 .. 35860.0;
  type eccentricity_value  is new float
    range  0 .. 1.0 digits 5;
  type  absolute_time  is new float digits 5;
  subtype  quadrant_angle_in_degrees is
    angle_in_degrees range  0 .. 90.0;
  subtype  quadrant_angle  is angle
    range 0 .. pi/2.0;
  subtype right_semi_circular_angle_in_degrees is
    angle_in_degrees range -90.0 .. 90.0;
  subtype right_semi_circular_angle  is
    angle  range -pi/2.0 .. pi/2.0;
  type  epsilon  is
     float range 0.0 .. 1.0 digits 5;

end Geometry_Definitions;


with Geometry_Definitions;
package  Earth_Position  is
    position_description:  circular_angles_in_degrees;
  subtype deadband_limit_in_degrees is
    float range 0.0 .. 2.0;
end Earth_Position;
```

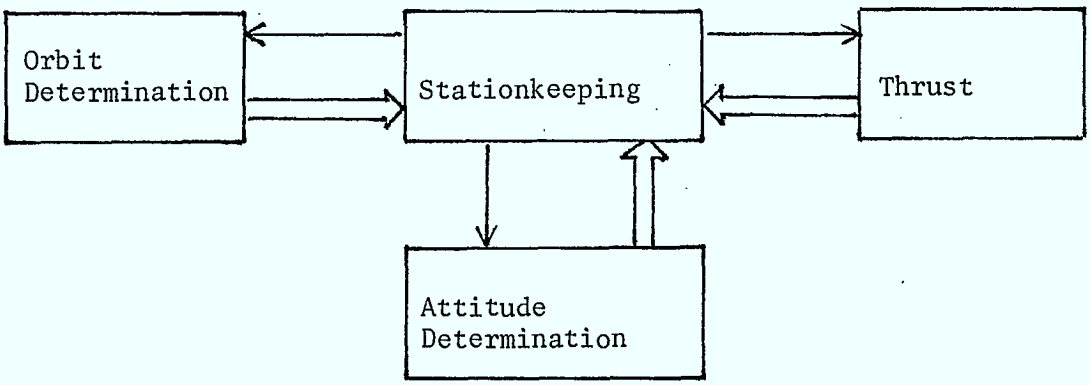



Figure 4.4: Relationship of Stationkeeping to other application functions

```
with Geometry_Definitions;
package  Orbit_Range  is                    --  [EKMA 83][BALL 67]
    low_semimajor_axis:  orbit_dimension;
    high_semimajor_axis:  orbit_dimension;
  type  orientation_spec  is
    record
      cap-omega:  quadrant_angle;      --  Right ascension of ascending
                                       --  node.  Angle measured from
                                       --  vernal equinox to
                                       --  ascending node.

      omega:  circular_angle;          --  Argument of perigee.  Angle
                                       --  measured from ascending node
                                       --  to perigee
                                       --  in plane of orbit in
                                       --  direction of motion.

      i: right_semi_circular_angle     --  Inclination.  The angle
                                       --  between orbit and
                                       --  equatorial planes.
    end record;


  type  orbit_spec is
    record
      a:  orbit_dimension;             --  semi-major axis in km.
      e:  eccentricity_value;
      orientation:  orientation_spec;
    end record;

                                       --  Note:  This is not the only
                                       --  way to specify an orbit.
                                       --  In fact, since there is
                                       --  a singularity in cap_omega
                                       --  if i=0, a different
                                       --  coordinate system is
                                       --  preferable.  Assumptions
                                       --  about lunar, solar,  and earth
                                       --  gravity fields, may be
                                       --  introduced.


  type  position_spec  is
    record
      orbit_description:  orbit_spec;
```

```
      eta:  angle;                          --  Angle from perigee
                                            --  to current position in
                                            --  direction of motion and
                                            --  in plane of orbit.
    end record;
  type  attitude_spec  is
    record
  type  eclipse_spec  is
    record
      entry:  absolute_time;
        --  time of entry into eclipse.
      exit:  absolute_time;
        --  time of exit from eclipse.
    end record;
end Orbit_Range;


with Geometry_Definitions, Orbit_Range;
generic
  type  orbit_observations_spec;

package  Stationkeeping  is
  type  attitude_spec is
    record
      theta:  circular_angle;
      phi:  right_semicircular_angle;
    end record;
  max_thruster_pulse_duration:  constant elapsed_time

  subtype  thruster_pulse_duration_spec  is
    ordinal range 0.0  .. max_thruster_pulse_duration;
  max_number_thruster_pulses:  constant short_ordinal :=  1000;

  subtype  number_thruster_pulses_spec  is
    ordinal range 0.0 .. max_number_thruster_pulses;

  type  thrust_parameters  is
    record
      pulse_delay_angle:  circular_angle;
      pulse_length:  thruster_pulse_duration_spec;
      number_pulses:  number_thruster_pulses_spec;
    end record;



--Assertion and Exceptions
--  The following  show examples  of assertions
--  and the associated exceptions to be raised
```

```
--  to accomplish fault detection.

--  A1:  computed orbit is within pre-specified
--       bounds
--  A2:  check orbit invariant equations

    orbit_invalid : exception;

--  A3:  check manoeuvre attitude and thrust
--       parameters by simulating the
--       manoeuvre, and check resulting
--       attitude and station.

    manoeuvre_invalid : exception;


procedure  Maintain_station  (
    earth_position:  circular_angle_in_degrees;
    deadband:  deadband_limit_in_degrees);

--Function:
--  This is the main procedure and initial process
--  Maintain_station.    It starts executing when
--  the BAM starts the function stationkeeping.
--  Its purpose is to accept a command from the
--  OAM to perform any necessary manoeuvres to
--  maintain the specified station, within the
--  specified deadband.    It must decide
--  whether such manoeuvres can be accomplished
--  and report to the OAM if the given station
--  is not within its pre-defined ability to
--  achieve or maintain.    In this case, the
--  OAM may have to decide to re-acquire
--  station via another function (subsystem).
--  For each major step in determining and
--  carrying out a stationkeeping function, a time-
--  stamped log entry is created and stored in
--  the event_log.


procedure  Get_orbit  (
    orbit:  out orbit_spec);

--Function:
--  Send a request to the Orbit_Determination
--  APF for specification of the current orbit.
--  Record the event on the event_log.
```

```
procedure  Plan_manoeuvre  (
    orbit:  in orbit_spec;
    earth_position:  in earth_position_spec);
    thrust_for_attitude_change:  out_thrust_parameters;
    thrust_for_velocity_change:  out_thrust_parameters;
    thrust_for_final_attitude:  out_thrust_parameters);

--Function
--  Determine epoch of manoeuvres, using orbit
--  prediction methods.   Determine required
--  semi-major axis.   If necessary, iterate
--  above two steps to determine appropriate
--  combination of epoch and semi-major axis.
--  Send a request to the attitude determination
--  subsystem to determine current attitude.
--  Calculate required thrust parameters to
--  attain the initial manoeuvre attitude, to
--  change velocity, and to return to required
--  operational attitude.
--  In order to determine attitude, this
--  procedure can establish communication with
--  the attitude control subsystem, in a real
--  system.   In the proof-of-concept system, a
--  sequential procedure call is used.
--  Record manoeuvre plan on event log.


procedure  Execute_manoeuvre (
    thrust_for_attitude_change:  thrust_parameters;
    thrust_for_velocity_change:  thrust_parameters;
    thrust_for_final_attitude:   thrust_parameters;
    result:  function_result);

--Function:
--  Create command sequences for the thrust
--  subsystem to accomplish the required
--  thruster actions.  Report these sequences
--  to the OAM and await response.  If the
--  response confirms the sequences, send the
--  command sequences to the thrust subsystem
--  sequentially.  Record manoeuvre execution
--  and result on event log.

end  Stationkeeping;
```

```
with Geometry_Definitions, Orbit_Range;
package  Orbit_Determination  is
  type  convergence_criterion_spec  is
    record
      epsilon_a:  epsilon;
      epsilon_e:  epsilon;
      epsilon_cap_omega:  epsilon;
      epsilon_omega:  epsilon;
      epsilon_i:  epsilon;
     end record;



package  Event_Logging  is
  type  event_class_values  is (
    orbit_determination,
    orbit_prediction,
    manoeuvre_plan,
    manoeuvre_execution);

  type  event_spec  (event_class: event_class_values)  is
    record
      time:  absolute_time;
      APF_name:  print_name;
      case  event_class
        when  orbit_determination  =>
          orbit : orbit_spec;
        when  orbit_prediction  =>
          orbit  :  orbit_spec;
          --  other components to be
          --  filled  in later.
        when  manoeuvre_plan  =>
          orbit  :  orbit_spec;
          thrust_for_attitude_change:  thrust_parameters;
          thrust_for_velocity_change:  thrust_parameters;
          thrust_for_final_attitude:  thrust_parameters;
        when  manoeuvre_execution =>
          thrust_for_attitude_change:  thrust_parameters;
          thrust_for_velocity_change:  thrust_parameters;
          thrust_for_final_attitude:  thrust_parameters;
          result:  function_result;
      end case;
    end record;

end Event_Logging;
```

```
with Geometry_Definitions, Orbit_Range;
package  Orbit_Determination  is

procedure  Determine_orbit;

--Function:
--  Receive requests from the OAM or other
--  subsystems to determine current
--  orbit parameters.   To accomplish this, a set of
--  orbit observations are required.   For proof-
--  of-concept, these will be input via the Monitor
--  Station (MS), so a request must be issued to the
--  MS for the observations.   Estimate an initial
--  orbit.   Use an iterative technique to
--  converge on an orbit such that two successive
--  approximations differ by less than the
--  convergence criterion.

end Orbit_Determination;
```

## 4.5 Monitor Station Software

As described in Section 3.5 above, the Monitor Station is a simplified ground control station attached directly to the on-board network for the purpose of monitoring and controlling the proof-of-concept demonstration. The software requirements for the Monitor Station are:

- ability to monitor the on-board network at the Link level
- optionally, the ability to intercept and monitor network traffic at any other level, i.e., Network, Transport, Session, and Presentation
- facility to inject into network traffic at any level of the protocol layers
- capability to tally up message type, count, size, etc., at various levels of the protocol layers for any station on the network
- facility to present the result of the monitoring in a highly visible fashion. Access to some form of graphics display is desirable.
- flexible filing facility to properly store the results of the observations.

The availability of software options which are realistic for the present project environment are the following:

- RSX-11M operating system in conjunction with Interlan's Ethernet driver, TCL protocol module, Netman network monitoring software, and in-house implementation of the Session layer protocol using an appropriate language available under RSX-11M: PASCAL, "C", Ada (one may be available by the time the actual implementation takes place), or MACRO-11.
- RSX-11M operating system in conjunction with DEC's new networking software which includes all seven layers of the OSI and the network monitor program. Interlan's Link layer protocol driver will be used, unless a new RSX-11M device drivers that allow access to network at higher levels are provided.
- iRMX 86 operating system plus iNA 950-1, TCL protocol

module, Ethernet driver (Link and Physical layer driver), and iMMX 800 for iRMX 86. Implementation of Session layer protocol will be necessary.

- iRMX 86 or iRMX 286-based system with appropriate Ethernet controller (Interlan NI3010, Intel iSBC 550 Kit, or a new controller based on 82586/82501 VLSI network access controllers expected in 1984. New version of iNA 950-1 software supporting up to the TCL. Possible Session-Presentation layer protocol modules to be made available in the near future. PASCAL 86, PL/M-86, "C", or other suitable language system to implement the Session layer protocol, if necessary.

- New "Datacom" computer to be made available late 1983 or early 1984 (iSBC 186/51). iRMX 86 operating system plus new TCL and Ethernet driver (offered as packaged software on the Datacom computer) plus Session layer implementation on the Datacom Computer. This solution is preferred over the two before this, as it represents a new generation implementation of the Ethernet access method.

- iRMX 86 operating system plus Interlan's TCL package; Interlan's Ethernet driver for iRMX 86; Netman software; NT10 Non-Intrusive transceiver unit (all the above made available as an Ethernode package).

- 68000-based "baby" machines - super microcomputer workstations, such as Sun Micro Systems, Callan Data Systems, Plexus, Altos, or Pixel. UNIX operating system. Ada, "C", or PASCAL language system. Networking protocol. The Session layer protocol only to be implemented, likely in Ada such as the one by Telesoft.

Software composition using RMX-11M is likely to face problems with filing, as the RSX-11M does not provide a structured filing facility. iRMX 86, on the other hand, has a UNIX-like filing structure but lacks a sufficient graphics support. It is very likely that a thorough search in the market would turn one up.

# 5. SYSTEM HARDWARE

## 5.1 OAM Hardware

The current version of the OAM requires a VAX-11/750 or 780 computer with appropriate real memory and standard disk and tape units, sufficient to run the Berkeley Version 4.2 UNIX, as its hardware foundation. In addition, as the bootstrap device, an 8-inch floppy disk driver (780) or a cassette driver (750) is necessary. To support local monitoring of the operation of the OCS, a standard alphanumeric terminal will be necessary.

## 5.2 Hardware for FTM and EOA

Layers 1 and 2 of the FTM are implemented in hardware, specifically, an iAPX 432 PCU. Layers 3 and 4 are implemented as Ada programs running under iMAX 432 operating system. The Example On-Board Application will run as a process in the 432 PCU, under local control of the FTM.

The PCU will be configured as follows:

The system chosen for the target machine is the Intel iAPX 432. It will comprise a Processor Cluster Unit (PCU) of two General Data Processor boards, one Memory Controller (MC), two Storage Array (SA) boards and a System Bus backplane. There will be one Attached Processor (AP) linked to the PCU by an Interface Processor and an Interface Processor Link. This is a basic target system using the minimum number of boards necessary for a prototype. It would be possible to extend this system up to three GDP boards, in addition to two IPL boards and as many as six SA boards (up to 1.5M bytes of RAM) for use in later stages of development. Details of the components are as follows:

| | |
|---|---|
| Subsystem Processor Cluster Unit | Two General Data Processor boards - iSBC 432/601, which include the system bus and arbitration logic. |
| | Memory Controller board - iSBC 432/604 |
| | Two Storage Array boards - iSBC 432/607 256K bytes each |

| | |
|---|---|
| | Interface Processor Link<br>- iSBC 432/603 |
| | System Bus Backplane<br>- iSBC 432/611 (12 slot) |
| I/O Interface Unit | Interface Processor<br>- iSBC 432/602 |
| | Attached Processor - iSBC 86/12A |
| | Multibus backplane<br>- iSBC 432/615<br>(6 slot) |
| Enclosed chassis | Cardcage - iSBC 432/630 (18 slot) |

5.3 MS Hardware

Based on the software requirements specified in Section 4.5 above, and judging from the availability of certain hardware components in the immediate environment, the following options are identified for the implementation of the Monitor Station hardware:

- PDP-11/45: with maximum main memory and reasonable peripherals; Interlan model NI1010 controller; corresponding Interlan transceiver.

- PDP-11/45: with maximum main memory and reasonable peripherals; DEC's new Ethernet controller, assuming it is different from the Interlan model and becomes available in time.

- iSBC 86/05, 11A, 14, 30 processors with sufficient ROM/RAM memory space to house the respective software; iSBC 550 Ethernet controller; appropriate disk subsystem.

- iSBC 186/51 Datacom Computer with built-in Ethernet controller chips; sufficient memory space; matching Ethernet transceiver (these to be made available as a package); appropriate disk subsystem.

- iSBC 286/14,30 processor with sufficient ROM/RAM memory space; Interlan's Ethernet controller for Multibus with matching transceiver; appropriate disk subsystem.

In all cases, graphic display and standard controller are not included in the specification. However, it is assumed such i/o devices will be chosen at implementation time with appropriate controllers. Also excluded are logistic requirements such as chassis and power supply.

# 6. DEVELOPMENT ENVIRONMENT

## 6.1 OAM Development Environment

### 6.1.0 Overview

The development environment for the OAM consists of the basic VAX-11 hardware with a minimum 2MB main physical memory, and several layers of software. The hierarchy is shown below in descending order of abstraction and the structure is shown in Figure 6.1:

| | |
|---|---|
| IFP and HYPO | - the top ARBY modules, or HLL for AI |
| DUCK | - general purpose deductive retriever |
| NISP | - Nifty LISP, Yale LISP macro set |
| Franz LISP | - a major dialect of LISP |
| UNIX | - Berkeley Version 4.2 or later |
| VAX-11 | - 750 or 780 preferred. |

### 6.1.1. VAX-11

Several hardware options have been investigated so far, but considering all the factors, the VAX-11 seems to be the best compromise, in particular in view of the new acquisition of this machine at Analysis & Simulation Laboratory. Other hardware options investigated include the Altos computer, Sun Micro Workstation, Plexus Systems, Pixel computers, Symbolics 3600, Callan System's Workstation, and LMI's new Lambda computer. The hardware is, however, the least significant element in the environment design, and we can possibly move to other machines without major technical difficulties. The situation may change in the near future and we may choose to change machines in future phases of the project.

### 6.1.2 Berkeley UNIX

Berkeley UNIX Version 4.2 or later is presently the most preferred among AI researchers involved in similar development projects. We have yet to find out whether the Berkeley UNIX can be run under the VMS operating system. However, this seems to be of no practical merit since file format incompatibility would nullify any benefits obtained by avoiding a "two-operating-systems-time-sharing" situation. Most AI utility and library packages are available under the UNIX file format. There are ways to convert UNIX files to VMS format. Nevertheless, it is a layer of processing one wishes to avoid in favour of reliability of the

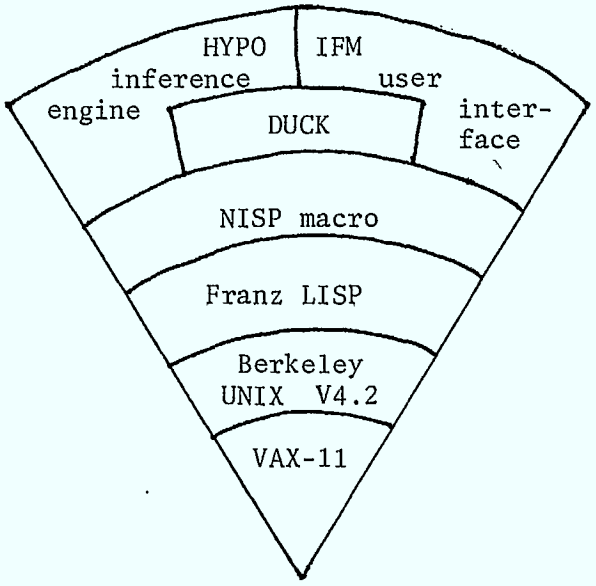



Figure 6.1: The OAM Development Environment

transported software and efficiency of the process. Also important is the fact that UNIX is more than an efficient system program. It is also an educational tool that affects the development habits of even the experienced. At least one AI researcher denounced the UNIS approach and encouraged potential users to stay away from that "best of both worlds" arrangement. We have not enough time to study these issues systematically. However, through our discussions with various AI researchers, we can specify with confidence, the VAX-11 and Berkeley UNIX combination is the most proven formula at the current state of the business. It is desirable to obtain UNIX Version 4.2 or later because of various new features (including formal Ethernet access) and reliability added since Version 4.1.

Standard development facilities under UNIX are typically used for housekeeping such as file management, version and configuration control in the production of AI software, mail handling, file backup, and file transportation (including network access).

### 6.1.3 "Baby Machines"

Another important issue in this respect is the recent progress made in the area of so-called "baby machines" (see [GOMI 83c]). Studying these machines carefully, one can be convinced by the claim of some that within 12 to 18 months we will be seeing the emergence of reliable portable machines that match in throughput and features the VAX-11/750 and sold almost at one tenth of the present price of a standard VAX-11/750 system. There are several companies, both on the manufacturers' and the users' side, who are already looking into that possibility seriously. This includes a few prominent AI corporations in the US. At any rate, as stated earlier, we believe the hardware is the least significant part of the development environment. One should simply be ready to adapt to the changing hardware viability scene.

### 6.1.4 Franz LISP

Franz LISP is one of several major LISP dialects that have been developing over the past 15 years or so. It originated from a group at UC Berkeley, and is now used widely at various AI centres. Again, while there are conversion methods among various LISP dialects, usually it is not worthwhile to attempt a conversion. Franz LISP comes as a part of the Berkeley UNIX package for VAX-11 computers. The UNIX installation procedure automatically makes Franz LISP avail-

able to general UNIX users. Benchmarks on Franz LISP running on a VAX-11/780 against INTERLISP running on a DEC 10 system drew an average 1 : 5 speed ratio both at the basic LISP level and also at the level of AI high level languages which are built using the basic LISPs (Franz LISP is slower). Franz LISP is, nevertheless, preferred at various AI centres because of its elegant integration with the UNIX operating environment and its more advanced features not seen in DEC 10 environment. Again, it is important to be aware of developments around us, as they will affect cost-effectiveness of AI projects that greatly depend on the efficiency of the development environment. It is entirely possible that we may be urged to switch our environment to, say, one based on a new, less expensive LISP machine running GLISP (a new generalized LISP from Stanford AI community).

### 6.1.5 NISP, DUCK, IFM and HYPO

The top three layers of the development environment are the software modules that actually perform key roles in the consultation. These are also the top two modules of the ARBY expert system framework. Their functions are explained in Section 3.1.2 above. It is merely pointed out here that an understanding of their internal functions is important in developing the KB and other domain-specific data structures that are needed for the operation of the OCS.

A set of extensions are provided in the ARBY package (OCS framework) with the aim of assisting the development of the KB. Including these, there are editors defined at each level of the software hierarchy (LISP, NISP, DUCK, IFM and HYPO). Essentially, they have the same syntactic formalism. Similar commands exist at every level. However, the meaning of the commands reflects the level of abstraction at which such commands are defined. There are also several commands which are unique to specific software levels. An example of such commands is the "new-if" command, which is used to define an IF at the IFM level.

The arrangement is quite helpful, as one can maintain the OCS software without actually exiting from the OCS development levels (to, say, the UNIX level). Editors are always at hand at wherever the current development operation is.

### 6.1.6 ARBY Utilities and Debugging Aid

Several utility programs exist in the ARBY framework to assist the development of domain-specific expert systems, such as the OCS. The workspace manager is responsible for storing and retrieving data structures created or edited by the developer/user of the OCS. Such data structures, or objects may be distributed throughout the hierarchy (IFM plus HYPO, DUCK and NISP) and typically include type definitions, IFs, assertions, rules, functions and variables.

Rules are defined using a function defined in the DUCK retriever for this purpose. ACCOUNT-FOR rules and EVIDENCE rules are the two important rule types in defining a KB. For every IF, there must be rules that say when it can be invoked. Also important are rules that say when a given set of arguments is finished.

A set of utility functions will assist the system developer in the following manner:

- a locater utility that locates a rule

- a list program that lists all the rules one can use to conclude a given fact.

These utility functions are accessible only at DUCK level, where most rule definitions take place.

A powerful way to debug emerging data structures in the OCS is the use of Question and Walk Modes, which are described in Section 3.1.3

## 6.2 Fault-Tolerant Management Development Environment

### 6.2.0 Overview

The development environment for the FTM has been chosen to conform to the specific hardware and software fault-tolerant requirements outlined in the AASC Design Report. It is felt that, at this point in time, the needs of both software and hardware are most nearly met by the use of Intel's iAPX 432, which integrates a highly fault-tolerant hardware architecture with the software reliability and security of the Ada programming language and the adaptability of the iMAX operating system. The environment components are depicted in Figure 6.2.

### 6.2.1. Hardware Environment

The hardware components of the environment will consist of a host system, a development workstation and the target AASC computer system. Figure 6.3 shows the functions of the host support.

#### 6.2.1.1 The Host System

This will house the Ada compiler and will support source program preparation, source program compilation and program linking. It will consist of a VAX-11/750 or 780, using the standard VMS operating system, editor and program development facilities.

#### 6.2.1.2 The Development Workstation

The development workstation will allow the updating and loading of linked programs into the target system for debugging and execution. It will also serve as a diagnostic console for the target machine. An Intellec Series III Microcomputer Development System will be used for the debug workstation. It will require an minimum of 192K bytes of RAM, a high-density mass storage subsystem, console interface, and an interface to the target computer system.

#### 6.2.1.3 The Target Computer System

This consists of the PCU described in Section 5.2

#### 6.2.1.4 Communication Links

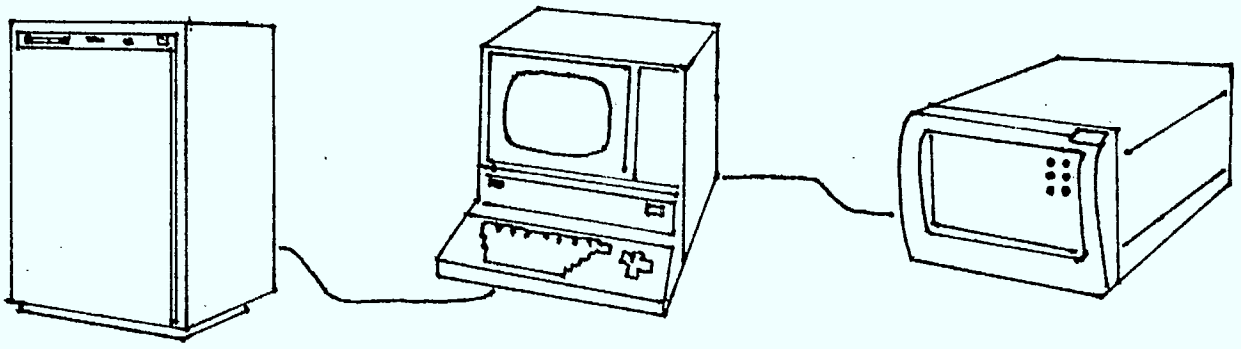
Two serial links are required between the VAX-11 and

Hardware Components:

Host System

Development Workstation

Target System



Software Components:

Ada Compiler
LINK-432
VAX/VMS Editor

Debug-432
UPDATE-432

iMAX 432 Runtime Executive

Figure 6.2: Hardware/Software Development Environment Components

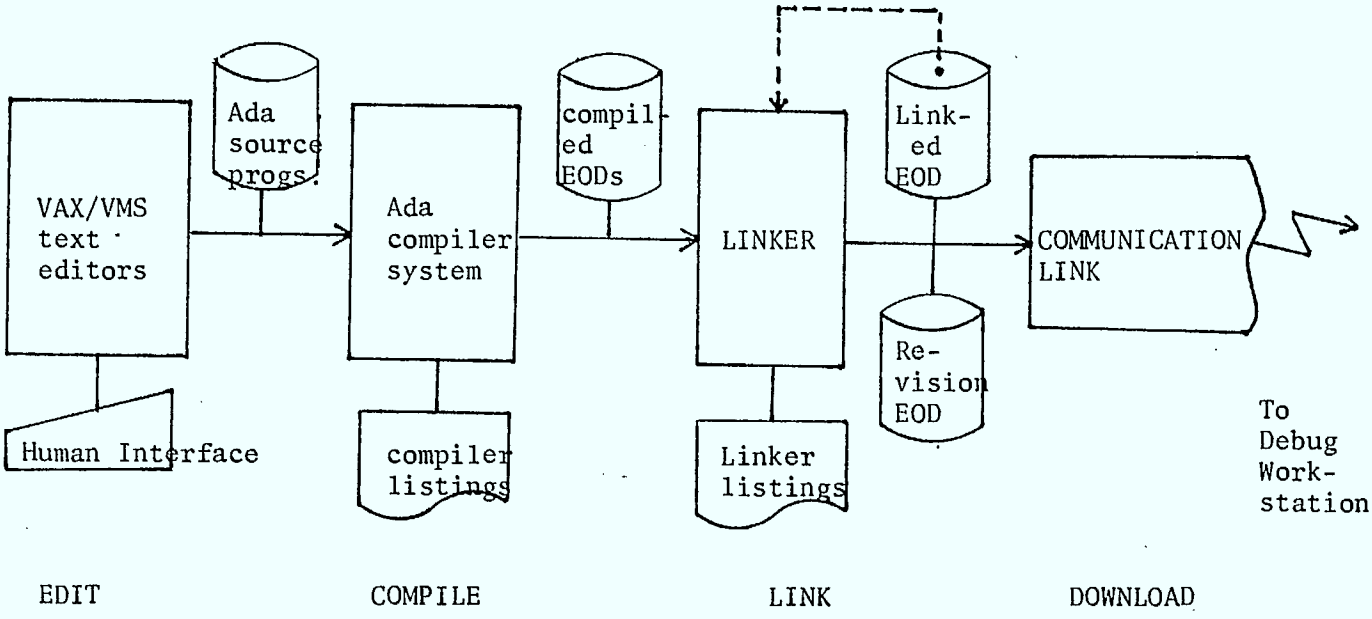



Figure 6.3: Functions of Host Support

Intellec MDS: a synchronous communications link supporting standard 2780/3780 point to point BSC protocol, for which additional VAX communication hardware will be required; and an asynchronous link supporting standard ASCII asynchronous point to point protocol operating via a direct cable connection and standard I/O ports.

6.2.1.5 Interconnect Kit

The Intellec Series III/432 Interconnect Kit will be the link between the Intellec Series III and the System 432/670. It consists of an IP board and an IPL board in addition to those described above, plus Proclink cabling.

6.2.2 Software Environment

This includes a compiler for the Ada programming language, standard VAX/VMS editors and file system support functions, program linker, debugger, and iMAX operating system. The software environment components and development process are shown in Figure 6.4.

6.2.2.1 Ada Compiler System (ACS)

This translates Ada source programs into compiled External Object Descriptors (EODs) which are linker-compatible. It also provides source code listings and error-reporting functions using the REPORT utility. In addition, REPORT generates data structure "templates" which are used in debugging Ada programs on the iAPX 432.

A key feature of Ada is support of separate compilation, i.e. individual design, development and compiling of program units. Units must, however, be compiled in a specific order based on their relationships as specified in environment files for the units. The ACS time-stamps all environment files and compiled EOD modules and warns of inconsistencies.

6.2.2.2 Program Linking

LINK-432 combines compiled EODs into a linked EOD ready for downloading to the debug workstation for execution/debugging. Traditional linker functions are performed in addition to 432-specific functions which include assignment of logical addresses to objects, and building physical 432 access segments and object tables. The linker also acts as a link between the iMAX executive of the user

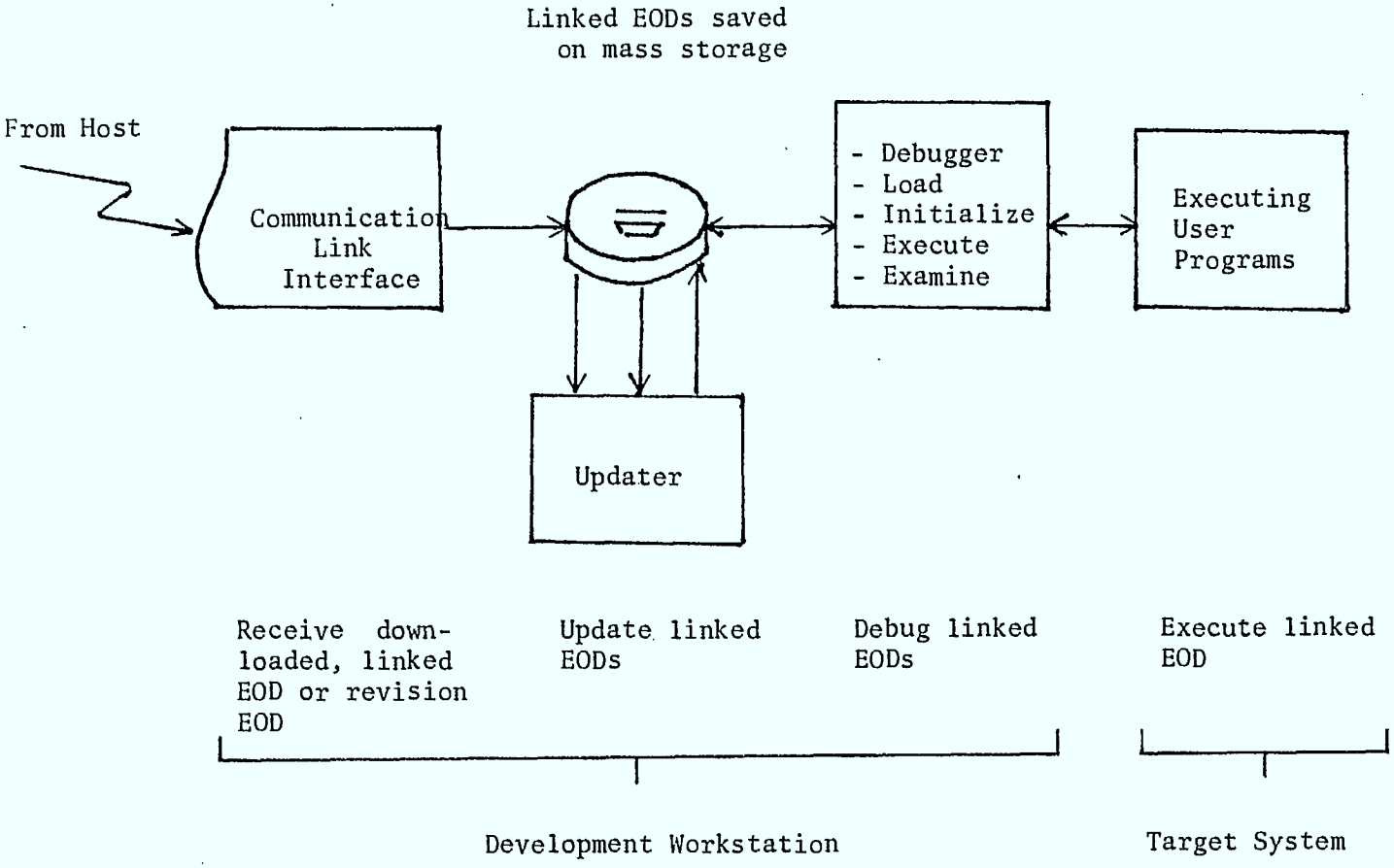



Figure 6.4: Software components and Development Process

application and produces a 432-executable program.

A further function is iterative linking - updating an existing linked EOD by replacing, removing or adding newly coupled EODs to form a new linked EOD, or revision EOD (a shorter version). The linker checks for time-stamp inconsistencies, thus promoting version control.

6.2.2.3 DEBUG-432

DEBUG-432 is contained mainly on the Workstation and supports loading of software into the target system, initiating program execution, setting breakpoints in 432 processes, examining and modifying the contents of variables and data structures, examining the state of failed programs, and debugging multiprocess systems. The use of templates assists in symbolic debugging. These may be generated by REPORT or created individually and stored on the workstation.

6.2.2.4 iMAX 432

The iMAX operating system is a collection of components which executes both on the PCU and the AP. It provides storage, basic process, process communication, input/output services, and initialization services. It may be configured to suit the number of processors in the system. The level of its services may be adapted as required.

## 6.3 NIU Development Environment

Depending on the hardware and software modules used for the particular form of NIU, the following combinations are possible as development facilities for the respective NIU types. Of these, some software products may change their specifications. When these, or other, alterations happen some adjustments will become necessary.

1. for RSX-11M, Interlan controller and driver, TCL protocol, Netman, and in-house Session protocol -

   RSX-11M on a reasonably equipped PDP-11 or LSI-11 model with language facility (PASCAL, FORTRAN, "C") and system utility sufficient to develop Session layer protocol module; test Ethernet with other station; Interlan controller, transceiver and necessary cabling.

2. for Intel's Datacom Computer, new TCL package (non-MIP version) with network management monitor, and in-house Session layer protocol module -

   iRMX 86 based Workstation (Intel's System 86/330 or /380), or Series IV, or NDS-II development system; Intel's network software package; language facility for Session layer development (PASCAL 86, PL/M-86, "C", Ada) and system utility; test Ethernet with another station; transceiver and necessary cabling.

3. for iRMX 86, iSBC 550 controller, iNA 950-1 protocol module with network management monitor, in-house Session layer -

   Intel's Series II, III, IV, NDS-11, or System 86/330, /380 Workstation (the last system is preferred); language facility and system utility for Session layer development, as above; test Ethernet with another station; transceiver and necessary cabling.

4. for iRMX 86, iSBC 550 Kit, new TCL software with network management software -

   Same as above, plus iMMX-800 inter-processor communication package

5. for iRMX 86, Interlan controller, Interlan TCL protocol with Netman software, and in-house Session layer -

   Intel's System 86/330 or /380 Workstation; language and system utility for PASCAL 86, PL/M-86, "C, or Ada; test Ethernet and another station; transceiver and necessary cabling.

# R E F E R E N C E S


BLAN 81 Blanc, R. and J. Heafner, "Summary of Current NBS Protocol Specifications", National Telecommunications Conference Record, IEEE, 1981.

BLID 83 Blidberg, D.R., A.S. Westmeat, and R.W. Coreil, "Expert Systems, A Tool for Autonomous Underwater Vehicles", Proc. of the IEEE "Trends & Applications 83" Conference, May 1983

BULL 83 Bullock, B. et al, "Autonomous Vehicle Control: An Overview of the Hughes Project", Proc. IEEE "Trends and Applications 83" Conference, May 1983

DAVI 82a Davis, R. "Expert Systems - Introduction", Tutorial given at the AAAI-82 Conference, Pittsburg, Pa., August 1982.

DAVI 82b Davis, R. et al. "Diagnosis based on description of structure and function", Proc. of AAAI Conference, 1982.

DeJO 83 DeJong, K. "Integrating AI and Control Theory", Procs. of IEEE "Trends and Applications 83" Conference, May 1983

DUDA 81 Duda, Richard O., and John G. Gaschnig, "Knowledge-Based Expert Systems Come of Age", BYTE, September 1981, Vol.6, No.9, McGraw-Hill.

DUDA 83 Duda, Richard, "Artificial Intelligence and Decision-Making: The PROSPECTOR Experience", paper given at the 1983 NYU Symposium: Artificial Intelligence - Applications for Business.

EKMA 83 Ekman, D.E., "Orbit Control Software for Communications Satellites", Computer, April 1983, Vol.16, No.4.

FEIG 83a Feigenbaum, E. and Pamela McCorduck, "The Fifth Generation - Artificial Intelligence and Japan's Computer Challenge to the World", Addison-Wesley Pub. Co., 1983.

FEIG 83b "Land of the Rising Fifth Generation", High Technology, June 1983

FORG 81 Forgy, C.E. "OPS-5 User"s Manual", Dept. of Computer Science, Carnegie-Mellon University, 1981.

GOMI 82a Gomi, T. and M. Inwood, "FTBBC - The Fault-Tolerant Building Block Computer", Eidetic Systems Corporation, ESC-82-001, 1982.

GOMI 82b Gomi, T., and M. Inwood, "A Fault-Tolerant On-Board Computer System for Spacecraft Applications", Eidetic Systems Corp., ESC-82-002, 1982

GOMI 83a Gomi, T., M. Inwood, and I. McMaster, "The Design of an Advanced Autonomous Spacecraft Computer", Eidetic Systems Corporation, March 15, 1983.

GOMI 83b Gomi, T. "Trip Report - Dr. L. Friedman, Jet Propulsion Laboratory", Department of Communications, Canadian Federal Government, April 1983.

GOMI 83c Gomi, T. "Trip Report - Sun Micro Systems", Department of Communications, Canadian Federal Government, April 1983.

HARM 83 Harmon, S.Y. of Naval Ocean Systems Center, San Diego, "Coordination between Control and Knowledge-Based Systems for Autonomous Vehicle Guidance", Proc. of the IEEE "Trends & Applications 1983" Conference, May 1983.

HECK 80 "Free-Swimming Submersible Testbed (EAVE WEST)" Interim Report prepared for Outer Continental Shelf Oil and Gas Operations, US Geological Survey.

INTE 82 Intel Corporation, iMAX 432 Reference Manual, Intel Corp. Santa Clara, Calif. 95051, 1982

ISO 82a ISO "Draft Basic Connection-Oriented Session Service Definition", ISO TC97/SC16/N1166, ISO, Switzerland, June, 1982.

ISO 82b ISO "Information Processing Systems - Open Systems Interconnection - Connection Oriented Transport Protocol", ISO/DP 8073, ISO, Switzerland, June 1982.

LESS 83 Lesser, Prof. V. "Coordination in Cooperative Distributed Problem Solving Systems", to be pre-

sented at IJCAI-83, Dept. of Computer & Information Sciences, U. of Massachussetts.

LEVE 82 Leveson, N., Talk on Software Fault-Tolerance, Fault-Tolerance Workshop, AIAA/NASA, Forth Worth, November, 1982.

MOGI 83 Mogilensky, Judah, Mitre Corporation, "Manned Space Flight Activity Planning with Knowledge-Based System", paper to be presented at the Computer in Aerospace Conference, October 1983.

MYLO 82 Mylopoulos, J. and T. Shibahara, "Towards a technology for building Knowledge-Based systems: the PSN experience", University of Toronto, 1982

NASS 82 Nassi, I. "The Liberty Net: An Architectural Overview", Proc. of COMPCON 82, IEEE Computer Society, 1982.

ORLA 83 Orlando, Nancy E. "A System for Intelligent Teleoperator Research", paper to be presented at the Computer in Aerospace Conference, October 1983, NASA Langley.

PETE 83 Peterson, C.B. et al. "Two Chips Endow 32-bit Processor with Fault-Tolerant Architecture", Electronics, April 7, 1983.

POPL 83 Pople, Dr. E. "Knowledge-Based Expert Systems: The Buy or Build Decision", paper presented at the 1983 NYU Symposium, Artificial Intelligence - Applications for Progress, May 1983.

PYLE 81 Pyle, I.C. "The Ada Programming Language", Prentice Hall International, 1982.

SCHI 83 Schindler, Sigram, et al, "Open Systems Interconnection - the Teletex-Based Session Protocol: Part 1", Computer Communications, April 1983, Vol.6 No.2.

SHIB 82 Shibahara, T. "CAA: Computer Diagnosis of Cardiac Rhythm Disorders from ECCs", Dept. of Computer Science Report, University of Toronto, 1982.

SHIB 83 Shibahara, T. "CAA: A Knowledge-Based System us-

ing Causal Knowledge to Diagnose Cardiac Rhythm Disorders", paper to be presented at IJCAI-83, Karlsruhe, West Germany, August 1983.

SOTT 83 Sotta, J.P., Matra Space Branch, France, "Generic Approach for Spacecraft Intelligence and Autonomy", paper to be presented at the Computers in Aerospace Conference, October 1983.

TOBI 82 Tobiasch, R. and H. Raffler, "Configurating Software - A Method For Bridging the Gap Between Concurrent Processing and Distributed Processing", Proc. of INFOCOM 82, IEEE Computer Society, 1982.

VANM 81 Van Melle, W., et al., "The Emycin Manual", Report No. STAN-CS-81-885, Stanford University, October 1981.

VERE 81 Vere, S. A., "Planning in Time: Windows and Durations for Activities and Goals", JPL. Report for NASA Contract NAS 7-100. November 1981.

VERE 83 Vere, S. A., Jet Propulsion Laboratory, "Planning Spacecraft Activities with a Domain Independent Planner", paper to be presented at the Computers in Aerospace Conference, October 1983.

WAGN 83 Wagner, Robert E. et al., Ford Aerospace, "Expert System for Spacecraft Command and Control", paper to be presented at the Computers in Aerospace Conference, October 1983.

CACC / CCAC
81678

GOMI, T.
An experimental version of an advanced autonomous spacecraft computer

P
91
C655
G643
1983

DATE DUE
DATE DE RETOUR

SEP 07 1984

LOWE-MARTIN No. 1137