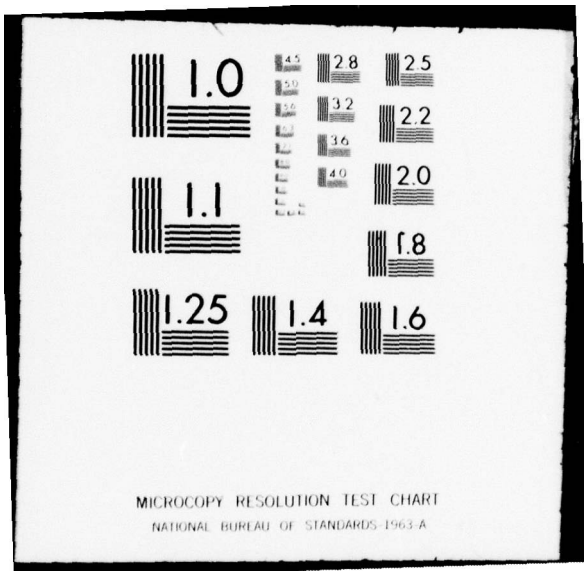
AD-A058 994 SOFTECH INC WALTHAM MASS F/G 9/2
SIMULATION HIGHER ORDER LANGUAGE REQUIREMENTS STUDY.(U)
AUG 78 J B GOODENOUGH, C L BRAUN F33615-77-C-0029
UNCLASSIFIED AFHRL-TR-78-34 NL

1 OF 3
AD A058994




1.0
1.1
1.25
1.4
1.6
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.8
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AD A058994

(18) AFHRL-TR-78-34

LEVEL II

(2)

AIR FORCE HUMAN RESOURCES LABORATORY

DDC FILE COPY

(6) SIMULATION HIGHER ORDER LANGUAGE REQUIREMENTS STUDY

By

(10) John B. Goodenough
Christine L. Braun

SofTech, Inc.
460 Totten Pond Road
Waltham, Massachusetts 02154

ADVANCED SYSTEMS DIVISION
Wright-Patterson Air Force Base, Ohio 45433

(11) August 1978

(9) Final Report for Period 1 February 1977 – 31 January 1978

(12) 208 p.

Approved for public release; distribution unlimited.

(15) F33615-77-C-0029

(16) 6114

(17) 07

DDC
RECEIVED
SEP 25 1978

78 09 21 040

AIR FORCE SYSTEMS COMMAND
BROOKS AIR FORCE BASE, TEXAS 78235

405 932 JOB

NOTICE

When U.S. Government drawings, specifications, or other data are used for any purpose other than a definitely related Government procurement operation, the Government thereby incurs no responsibility nor any obligation whatsoever, and the fact that the Government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data is not to be regarded by implication or otherwise, as in any manner licensing the holder or any other person or corporation, or conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

This final report was submitted by SofTech, Inc., 460 Totten Pond Road, Waltham, Massachusetts 02154, under contract F33615-77-C-0029, project 6114, with Advanced Systems Division, Air Force Human Resources Laboratory (AFSC), Brooks Air Force Base, Texas 78235. Mr. Patrick E. Price was the contract monitor.

This report has been reviewed and cleared for open publication and/or public release by the appropriate Office of Information (OI) in accordance with AFR 190-17 and DoDD 5230.9. There is no objection to unlimited distribution of this report to the public at large, or by DDC to the National Technical Information Service (NTIS).

This technical report has been reviewed and is approved for publication.

GORDON A. ECKSTRAND, Director
Advanced Systems Division

RONALD W. TERRY, Colonel, USAF
Commander

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFHRL-TR-78-34	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* SIMULATION HIGHER ORDER LANGUAGE REQUIREMENTS STUDY		5. TYPE OF REPORT & PERIOD COVERED Final 1 February 1977 – 31 January 1978 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) John B. Goodenough Christine L. Braun		8. CONTRACT OR GRANT NUMBER(s) F33615-77-C-0029
9. PERFORMING ORGANIZATION NAME AND ADDRESS SofTech, Inc. 460 Totten Pond Road Waltham, Massachusetts 02154		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 62205F 61140709
11. CONTROLLING OFFICE NAME AND ADDRESS HQ Air Force Human Resources Laboratory (AFSC) Brooks Air Force Base, Texas 78235		12. REPORT DATE August 1978 13. NUMBER OF PAGES 226
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)* Advanced Systems Division Air Force Human Resources Laboratory Wright-Patterson Air Force Base, Ohio 45433		15. SECURITY CLASS. *(of this report)* Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT *(of this Report)* Approved for public release; distribution unlimited.		
17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)* DoD common language embedded computer systems flight training simulators high order languages JOVIAL PASCAL PL/I programming languages		

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

This report defines high order language requirements for programming flight training simulators. These requirements were determined by analyzing programs written for a variety of simulators. Examples drawn from these programs are used to justify the need for certain HOL capabilities. A detailed specification of simulator HOL requirements is given, following the general structure and organization of the IRONMAN requirements for the DoD Common Language effort. PL/I, FORTRAN, JOVIAL J3B, JOVIAL J73I, and PASCAL are analyzed to see how well each language satisfies the simulator HOL requirements. Although PL/I and JOVIAL J3B were found to be best suited for simulator programming, only FORTRAN was clearly the least suitable language. Since all the languages failed to satisfy some simulator requirements, ~~we specified~~ were specified modifications to each language that would make them

next page

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

78 09 21 010

Item 20 (Continued)

more useful as simulator programming languages. Our analysis of recommended modifications indicated that PL/I was the most easily modified language.

ACCESSION for
NTIS White Section X
DDC Buff Section
UNANNOUNCED
JUSTIFICATION
BY
DISTRIBUTION/AVAILABILITY CODES
Dist. AVAIL. and/or SPECIAL
A

TABLE OF CONTENTS

Section		Page
1	INTRODUCTION	1
2	OVERVIEW	3
3	THE SIMULATOR BENCHMARK	6
4	PROGRAMMING ENVIRONMENT OBSERVATIONS	11
	4.1 Program Development Methods	11
	4.2 Programmer Experience	12
	4.3 Compilation Size and Speed Requirements	13
	4.4 Object Code Size and Speed Requirements	13
	4.5 Program Lifetime and Stability	14
	4.6 Program Reusability	16
	4.7 Program Portability	16
5	PROGRAM ANALYSIS OBSERVATIONS	18
	5.1 Storage Management	19
	5.1.1 Global Data	19
	5.1.2 Local Data	20
	5.1.3 OWN Data	23
	5.1.4 Overlay Programs	23
	5.2 Data Types and Operations	23
	5.2.1 Integer Data Type	23
	5.2.2 REAL Data Type	24
	5.2.3 Status Data Types	27
	5.2.4 Bit Data Type	29
	5.2.5 Boolean Data Type	31
	5.2.6 Character String Data Type	37
	5.2.7 Pointer Data Type	39
	5.2.8 Label Data Type	42
	5.2.9 Procedure Data Type	44
	5.3 Aggregate Data Types	49
	5.3.1 Set Data Type	49
	5.3.2 File Data Type	51

TABLE OF CONTENTS (Continued)

Section			Page
		5.3.3 Array Data Type	54
		5.3.4 Structure Data Type	57
		5.3.5 Union Data Type (or Overlays)	65
	5.4	Control Structures	66
		5.4.1 Conditional Control Structures	66
		5.4.2 Multiprocessing Control	72
	5.5	Program Development Aids	76
		5.5.1 Compile-Time Assignments	76
		5.5.2 Conditional Compilation	77
		5.5.3 Symbol Dictionary	77
		5.5.4 Debugging Support	78
		5.5.5 Onboard Computer Simulation	80
	5.6	I/O	80
	5.7	Machine Dependency	83
6	DETAILED SHOL REQUIREMENTS AND LANGUAGE EVALUATIONS		88
	6.1	General Design Principles	90
	6.2	General Syntax	95
	6.3	Data Types	100
		6.3.1 Numeric Types	100
		6.3.2 Enumeration Types	106
		6.3.3 Boolean Type	108
		6.3.4 Character Type	110
		6.3.5 Bit String Type	113
		6.3.6 Pointer Type	116
		6.3.7 Procedure Types	120
		6.3.8 Array Types	121
		6.3.9 Record Types	126
	6.4	Expressions	129
	6.5	Constants, Variables, and Declarations	132
	6.6	Control Structures	138
	6.7	Functions and Procedures	146

TABLE OF CONTENTS (Continued)

Section			Page
	6.8	Input-Output Facilities	152
	6.9	Parallel Processing	154
	6.10	Specification of Object Representation and Optimization	157
	6.11	Libraries and Separate Compilation	162
	6.12	Language Evaluation Summary	165
7	LANGUAGE MODIFICATION GOALS		167
	7.1	Minimal Cost	167
	7.2	Syntactic Integrity	168
	7.3	Non-Interference with Existing Language Features	168
	7.4	Upward Compatibility	169
8	LANGUAGE EVALUATION AND MODIFICATION SUMMARIES		170
9	IMPLEMENTATION CONSIDERATIONS AND RECOMMENDATIONS		181
10	SUMMARY AND CONCLUSIONS		190
Appendix A – SIMULATOR MODEL			191
	REFERENCES		218
	BIBLIOGRAPHY		219

LIST OF ABBREVIATIONS

AST	Asynchronous Trap
BAM	Binary Angular Measurement (degrees/360)
CRT	Cathode Ray Tube (generically, any display device)
CPU	Central Processing Unit (generically, a computer)
DME	Distance Measuring Equipment
DoD	Department of Defense
DRU	Digital Readout Unit
EOM	Equations of Motion
FIFO	First In First Out
HOL	High Order Language
ILS	Instrument Landing System
IOC	Input/Output Control
I/O	Input/Output
LFI	Linear Function Interpolator
OS	Operating System
SMS	Shuttle Mission Simulator
TTY	Teletype
UPT	Undergraduate Pilot Trainor
VOR	VHF Omnidirectional Range

SIMULATION HIGHER ORDER LANGUAGE
REQUIREMENTS STUDY

Section 1

INTRODUCTION

In most applications, the use of higher order languages (HOLs) for software development is common. However, in a few areas the use of HOLs is rare. In general, such areas are characterized by stringent computer resources. HOLs have recently made substantial inroads into these application areas. In the avionics software area, where size, speed, performance, and reliability of the software for the on-board computer have always been vital, HOLs have been used with success. The B-1 offensive avionics software is written in JOVIAL J3B; the software for the F-16 Fire Control Computer is also written in JOVIAL J3B. The real-time mission software for the Digital Avionics Information System (DAIS) and the Electronically Agile Radar (EAR) is being written in JOVIAL J73I. Portions of some flight training simulators have been written in FORTRAN. The experience with the use of HOLs in avionics projects as well as some recent simulator projects indicates the use of HOLs in real-time flight training simulators is feasible. Given that this is the case, an important question must be answered: which existing HOL, if any, is most suitable as a potential standard for developing real-time training simulator software? What modifications to an existing language must be made to make it suitable? The study described in this report defines simulator HOL (SHOL) requirements and analyzes PL/I, FORTRAN, JOVIAL J3B, JOVIAL J73I, and PASCAL for suitability in meeting these requirements.

In determining HOL requirements for simulator programming, we considered not only simulator needs but also work currently underway within DoD leading to the possible development of a common high order programming language for embedded computer systems applications. Embedded computer systems are defined as systems "integral to a larger military system or weapon, including technical weapons systems, communications, command and control, avionics, simulation, test equipment, training, and systems programming applications" [Fisher, 1976]. Clearly real-time flight training simulators fall within the class of embedded computer systems.

The goal of the common language effort is "the adoption of a very few (possibly only one) common programming languages to be used for the design, development, support, and maintenance of all digital computer software for embedded computer applications in the DoD" [Fisher, 1976]. This means that if the programming of flight simulators has unique characteristics not shared by other embedded computer applications and if these characteristics imply that a SHOL must contain features not needed in other embedded computer applications, then a special purpose simulator programming language will be justified under the common

programming language effort. Our approach to defining the required SHOL characteristics was designed to help decide how specialized simulation programming requirements really are. Given the high level DoD interest in minimizing the number of distinct programming languages, it is reasonable to assume that developing a new or modified language for simulator programming will be possible only if the need can be clearly demonstrated.

Considering this background, the main objectives of this study were:

- to define simulator HOL requirements in a way that can be related to the DoD common language effort;
- to determine which of PL/I, FORTRAN, JOVIAL J3B, JOVIAL J73I, and PASCAL is most suitable for use or modification as a simulator programming language;
- to recommend methods for implementing and enforcing the use of a standard simulator HOL.

The remainder of this report describes our findings and the methods we used in obtaining them.

Section 2

OVERVIEW

In this Section we discuss the general nature of our study, the methods we employed to reach the conclusions reported later, and give an overview of the remainder of the report.

Although the main objective of our study was to define simulator HOL requirements, a subsidiary objective was to develop a general approach for determining HOL requirements in a given application area and to then apply this approach to the simulator area. The general approach we have devised focuses on three sources of language requirements:

- the programming environment, i.e., factors pertaining to the development and maintenance environment of a particular application area, e.g.,
  - long or short program lifetime
  - programmer background
  - potential for program portability
  - compiler efficiency requirements
- functional requirements, i.e., the programs developed for a given application. These programs can be subclassified into two groups:
  - programs that perform application functions; and
  - programs that assist in developing other programs, e.g., data file generation programs, debugging programs, etc.
- language design principles, i.e., accepted and emerging program development methodologies and principles, independent of any particular application area and reflecting current thinking about what the properties of a "good" programming language are, e.g.,
  - linguistic simplicity and uniformity
  - support for structured programming
  - support for modular programming

The first portion of our study was devoted to describing the simulator programming environment and simulator functional requirements. The Link Division of the Singer Company, as subcontractor to SofTech, was our principal source of information on simulator requirements. Link provided simulator programs and documentation for our analysis. They later reviewed our characterization of the environmental and functional requirements to confirm that they accurately reflected simulator needs. Although SofTech was responsible for the language analysis, Link was responsible for ensuring that our language conclusions were based on an accurate understanding of simulator requirements.

Our findings regarding environmental requirements are described in Section 4; functional requirements are described in Section 5. To guide and support our analysis of functional requirements, a benchmark model of a generic flight training simulator was developed. The benchmark model is a key part of our analysis because it represents the entire set of programming tasks relevant to simulator development. Our analysis of language requirements is keyed to this model. Because of the importance of the model, it is discussed separately in Section 3.

Our analysis of the simulator programming environment, simulator program functional characteristics, and language design principles resulted in the specification of simulator HOL requirements given in Section 6. This specification of requirements serves as the definitive basis for evaluating how well existing programming languages could serve in programming simulators. It serves to document the key implications of our study of programming requirements concisely and rigorously.

To facilitate comparison of simulator requirements with the proposed Common Language requirements, the requirements specification in Section 6 has a structure similar to that of the IRONMAN [U.S. DoD, 1977]. The IRONMAN requirements specification is the latest in a series of evolutionary language requirements documents issued as part of the Common Language effort. It is the requirements specification being used to direct preliminary language design efforts completed in Februrary 1978. The IRONMAN specification is the most recent specification of language requirements and thereby most suitably represents the current DoD position on embedded computer application requirements.

Based on our modification of the IRONMAN requirements specification, we selected a set of language features satisfying the requirements using a computerized database of 2267 language features [SofTech, 1977]. This list of features was also used to describe each of the programming languages being evaluated, namely, PL/1, FORTRAN, JOVIAL J3B, JOVIAL J73I, and PASCAL. By analyzing which features satisfied the SHOL requirements specification and were present or absent in each of these languages, we decided how well each language satisfied the SHOL requirements.

Section 6 contains both the SHOL requirements specification and our evaluations of the candidate languages with respect to the requirements. The value of combining the requirements specification and the evaluations in this way is twofold:

- a particular evaluation is most understandable when preceded by a statement of the requirement under consideration.
- a requirement can frequently be understood more readily by reading the discussion of how well the various languages meet that requirement.

An overall summary of how well each language satisfied the requirements is given at the end of Section 6. PL/I and JOVIAL J3B were judged to be the languages best satisifying the requirements without modifications, although only FORTRAN is clearly the least suitable language.

Since all the languages failed to satisfy some of the simulator language requirements, we considered what language modifications would make them significantly more useful as simulator programming languages. To assist in this analysis, we divided the simulator requirements into two classes: those considered essential both to accomplish all necessary simulator programming functions and to meet the more general SHOL design goals of reliability and maintainability, and those considered beneficial in a new language but not of sufficient importance that it is essential to modify a language to satisfy them. In essence, if the non-mandatory requirements can be satisfied with a minor language modification, then the modification should be made, but if the modification is complex or changes the fundamental syntactic and semantic constraints of a language, then its impact as a change outweighs its benefits to simulator programming. For example, changing PASCAL's semicolons to statement terminators instead of separators would be a non-mandatory modification.

Modification issues are discussed further in Section 7, and the modifications selected for each language are presented in Section 8. Based on the extent of the modifications and the usefulness of the modified language, we selected PL/I as the language most suitable for modifications. This decision is discussed at the end of Section 8.

As the final part of our study, we addressed how to support the use of a standard SHOL. Section 9 discusses these issues, which include language design and implementation approaches as well as recommendations for introducing and establishing SHOL usage.

Section 3

THE SIMULATOR BENCHMARK

A significant part of this study involved familiarization with the programming requirements of flight simulators. To assist in this analysis a benchmark simulator problem was developed. This benchmark models a generic flight training simulator, i.e., it does not describe the operation of a particular simulator, but rather incorporates the characteristics typical of simulators in general. The benchmark served two major purposes in the study:

- It provided an overall framework for the entire analysis of simulator functional requirements.
- It provided a frame of reference for presenting results. The material describing language requirements (in Sections 4, 5, and 6) is cross-referenced to components of the benchmark model, allowing the reader to:
  a) Determine the simulator area(s) from which a particular requirement derives.
  b) Determine those requirements which derive from a particular simulator area.

Development of the Benchmark

The basis of the benchmark development was our analysis of simulator programs and design documents. The purpose of this effort was to determine the types of processing required when programming flight training simulators. A major concern was to determine what functions must be performed rather than how they are currently implemented, since the goal of the SHOL is to permit programming the required functions rather than duplicating the programming techniques which are currently used to realize these functions. Thus, the effort had to go beyond a simple investigation of the programming techniques currently employed. For this reason, the benchmark developed is a functional, rather than an operational, representation of a generic flight simulator.

The primary inputs to the programming analysis effort were provided by the Link Division of the Singer Company, serving as a subcontractor to SofTech. At the start of the study, Link presented a two day orientation briefing to SofTech personnel. The briefing included presentations by representatives of each of the major simulation areas as well as discussions of the overall application. The briefing provided a general framework for the subsequent programming analysis and also highlighted issues of particular concern to simulator personnel regarding the use of an HOL.

Subsequently, Link provided extensive documentation to SofTech for study. The materials studied were:

- UPT (Undergraduate Pilot Trainer) - 26 volumes of design documentation plus listings, representing the entire system
- F-14 - documentation and listings from flight and navigation subsystems
- 214A - documentation and listings from the visual (camera/model board) subsystem
- SMS (Shuttle Mission Simulator) - documentation from the visual (digital image generation) subsystem
- other tactics programs - listings (names of systems were not provided to SofTech)

The fact that a single complete system (the UPT) was studied was important to the effort, as it guaranteed that no major functional aspect of simulator programming was overlooked. This would not necessarily be the case if only isolated programs selected as "representative" were studied; the selection of representative programs would have required knowledge that was not available prior to the analysis effort. Certain areas not included in the UPT material were covered by the other simulator documentation. These were:

- camera/model board visual (The UPT visual system was done by a subcontractor to Link, Redifon, and was thus not included in the UPT documentation)
- computer image generation visual
- tactics

Other material was studied which duplicated UPT areas, to help ensure that the analysis concentrated on the functions to be performed, rather than on a single approach to programming those functions.

Other important advantages to the study of actual simulator programs were:

- It was possible to make judgements concerning the degree of efficiency actually required of the object code which the SHOL translator must generate. That is, if a certain algorithm not displaying the maximum efficiency is observed to be adequate, it is possible to conclude that comparable code generated by an HOL compiler will also be adequate.

- It was possible to isolate certain areas of potential inefficiency which occur frequently and which it is thus particularly important that an HOL compiler avoid.

- Areas where use of an HOL would have a particularly beneficial effect on program readability could be observed and highlighted.

As the program analysis proceeded, informal written observations on the programming requirements of the various areas were prepared and submitted to Link for comment. This helped guarantee that no erroneous conclusions were reached. Similarly, Link reviewed the benchmark model as it was developed, as did the Air Force. Based on comments received, the model was reworked. Thus model development was an iterative process, with each iteration reviewed by simulator experts.

Presentation of the Benchmark

The benchmark model, contained in Appendix A, employs SofTech's Structured Analysis and Design Technique (SADT™)[Ross, 1977]. SADT has been found to be a valuable technique for communication between individuals performing analysis in a given problem area and individuals who are experts in that area. The simulator benchmark model assists in communicating the findings of the SHOL investigation to simulator experts.

The model consists of a set of diagrams which form a hierarchical decomposition of a generic flight simulator. The diagrams are made up of boxes representing activities (functions performed), and arrows representing data which is an input or an output of these functions. Each diagram is itself an expansion of an activity box which appears as one of the boxes on a preceding higher level diagram (its parent). The arrows entering and exiting a diagram exactly match the arrows attached to the box on the parent diagram. Figure 3-1 shows a sample SADT diagram and explains the notation used.

The first diagram of the benchmark model, node A-0, contains a single box representing all functions which must be programmed in developing a flight simulator. The decomposition of this diagram appears in the second diagram of the model, node A0, which consists of three activities:

- build simulator

- test simulator

- simulate

The diagram shows that the specification of the aircraft operation, and performance requirements which must be met by the simulator, control the building and testing activities. It also shows that building and testing are an iterative process. The "simulate" box shows the student actions as an input, the simulated aircraft reactions as an output, and the instructor inputs as a control of the activity. An additional output of this activity is the information on student actions which is displayed to the instructor.

Subsequent diagrams further decompose these activities. In particular, diagram A1 decomposes box 1, diagram A2 decomposes box 2, and diagram A3 decomposes box 3. The remaining diagrams further decompose boxes of diagram A3, "simulate," which is the major part of the model, particularly box 3 of A3, "model aircraft functions."

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

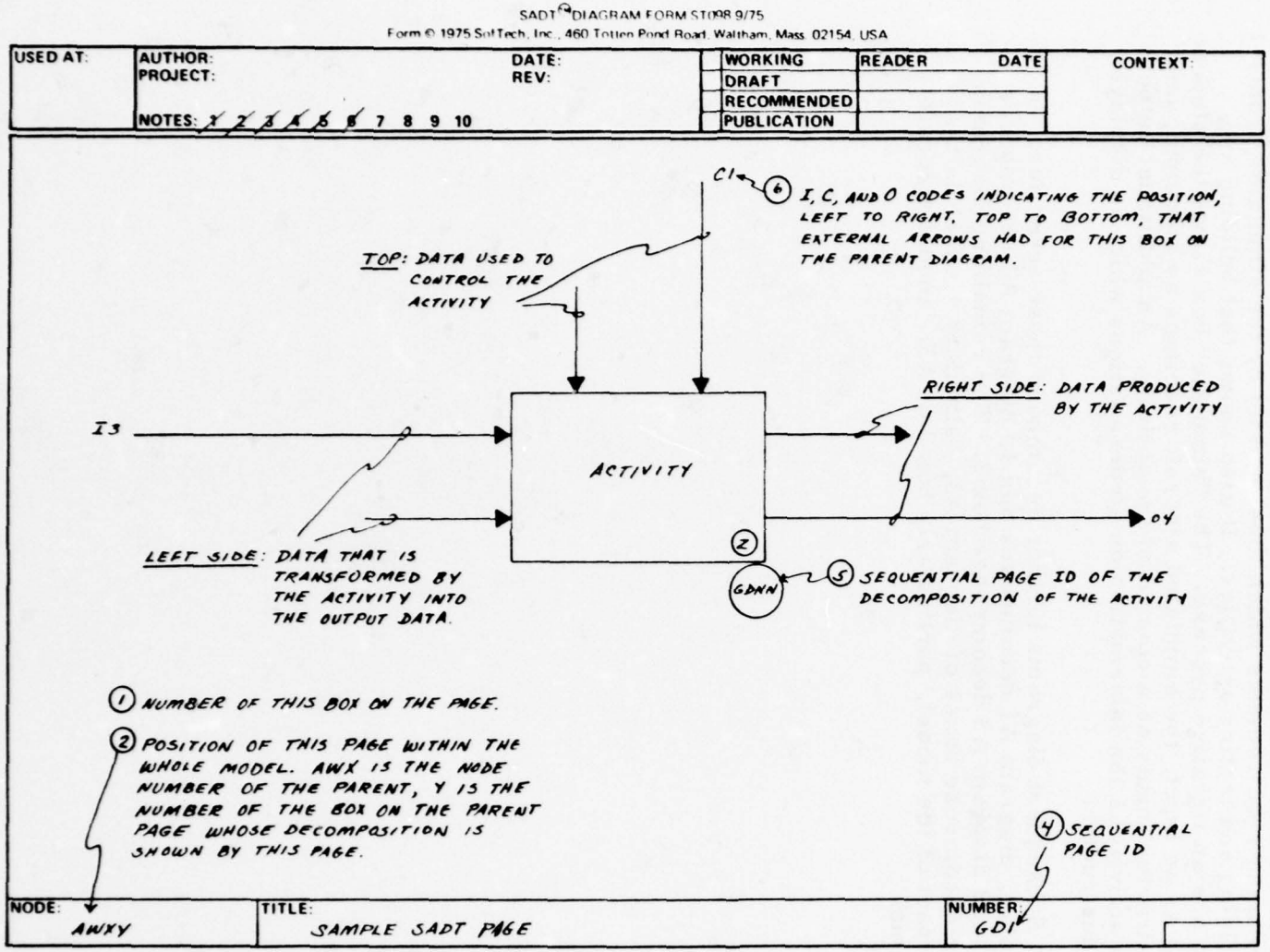


Figure 3-1. Example SADT Diagram

Section 4

PROGRAMMING ENVIRONMENT OBSERVATIONS

Part of our general approach to defining HOL requirements in a given application area is to evaluate the effects of the environment in which programs are developed and maintained. Environmental factors discussed in this Section are:

- program development methods (groups vs. individuals)
- programmer background and experience
- compilation size and speed requirements
- object code size and speed requirements
- program lifetime and stability
- program reusability potential
- program portability potential

Each of these factors has some influence on what language features are most suitable for simulator programming. In subsequent subsections, we discuss the environmental factors for simulator programming and their relation to language features. The information presented here on the simulator programming environment was obtained primarily from discussions with simulator personnel. The findings presented in this and the next Section were used in defining the detailed language requirements specified in Section 6.

4.1 Program Development Methods

Simulators are very large systems programmed by many programmers rather than by a single individual. Coordination between these programmers should be supported by the SHOL. In particular, programmers must be able to interface their programs with those produced by others and must be able to access system data in a consistent manner. The SHOL should diagnose conflicts in these areas. It might also be desirable to control (or at least be able to detect) access to data by a program which should not read and/or alter that data. Section 5.1.1 discusses the methods currently used to support data coordination in simulator development.

The large number of programmers implies a requirement for separate compilation of programs, i.e., individual programmers must be able to separately compile, modify, and test their programs and

then integrate them to form the complete system. Some support for the integration process and for system level testing is also essential.

Use of system data by large numbers of programmers requires libraries of data definitions and subroutine declarations. These facilitate consistent access to global data and allow inconsistent assumptions about forms of data structures and parameters to be detected at compile time. Such inconsistencies can easily arise when groups of programmers are involved.

4.2 Programmer Experience

Simulator programs are produced primarily by individuals trained in an engineering specialty rather than in computer science. This results in a programmer preference for language notation which reflects the engineering notation used in the simulator design descriptions. An example of this is the use of conditional expressions in program documentation, as discussed in Section 5.4.1.1. Inclusion of this feature in a SHOL is primarily justified by this documentation practice.

Another consideration based on programmer experience is a strong preference for fixed point as opposed to floating point. (The reasons for this are discussed in Section 5.2.2.1.) A SHOL should provide programmer control of real number representations. Much programmer concern about the use of floating point comes from fear of loss of control over significance in computations. Concern about the space required by certain real number representations is also apparent. The SHOL should provide access to the various representations available on the target computer.

Most simulators are currently implemented in assembly language, with occasional uses of FORTRAN. Most simulator programmers have not been exposed to other languages. In selecting/designing a SHOL, consideration should be given to the problem of retraining programmers in the language, especially in view of the large number of individuals involved. When a choice is to be made among several language features satisfying a particular functional requirement, programmer background indicates that the choice be made on the grounds of simplicity of use and similarity to commonly-used programming languages.

Another consideration, at least in the Link environment, is the use of a Quality Assurance group to optimize the programs produced by the engineers. This dictates a requirement for program understandability, to ensure that changes made by this group do not alter programs functionally. Language features supporting understandability are discussed in Section 4.5.

4.3 Compilation Size and Speed Requirements

In simulator development, constraints on compiler performance are imposed by the computer being used for compilation. Conventionally, compilation is done on the target computer, i.e., the computer on which the simulation programs will execute. Typically, these computers are of moderate size and speed. Examples of machines used are:

- PDP 11/45
- Honeywell 316, 516
- Interdata 8/32
- SEL 3250
- Harris DC 6024/4

This practice of compiling on the target machine would require that the SHOL be compilable on machines of this size. (Note that disk storage is available with all of the systems.) This restriction would also dictate that the SHOL compiler be implemented in a language supported on the target machine. Clearly, development of compilers in each of the various target machine assembly languages would be costly. An alternative to this would be implementation of the SHOL compiler in the SHOL (bootstrapped). This would require that the SHOL contain the capabilities required for the compiler implementation. This would probably require no features not also needed to produce the various offline simulation support programs, several of which are special-purpose compilers.

Constraints on compiler size and speed might also affect the amount of optimization performed by the compiler. A language offering programmer-controlled optimization allows the programmer to limit the amount of optimization performed by the compiler, thus increasing compiler speed and decreasing core requirements. The programmer will then perform optimization explicitly through appropriate use of the HOL.

The constraints described above could be avoided by use of a single, larger-scale host computer for compilation. A departure of this sort from the current practice involves considerations which cannot be fully addressed here. Among these is the requirement for program modification in the field, discussed in Section 4.6.

4.4 Object Code Size and Speed Requirements

Object code size constraints are imposed by the available core on the target system. A typical simulator might occupy 100K words of core -- 80K for program and 20K for data. An additional constraint stems from the requirement to deliver spare core. (Though core can

be added, it is desirable to keep costs down and to stay within the addressing capacity of the machine.) Simulator programs studied reflect a desire to conserve core (e.g., packing of logicals, use of fixed point for large data tables when floating point would require double words), but not at the expense of operating speed.

Speed of object code execution is of primary importance in simulation. Clearly, the simulator must respond to pilot actions as quickly as the actual aircraft. Simultaneously with this realtime response, other functions such as performance recording must occur. Not only is speed itself important but coordination of the various system components is vital. Small discrepancies between the visual and motion systems, for example, are discernible to the pilot.

Study of the simulator programs has verified the concern for time efficiency over space efficiency. For example, Section 5.2.9.2 discusses the use of inline subroutines to increase speed of execution. The ability to specify inline expansion of subroutines (as opposed to calling the subroutine) allows the programmer to trade space for execution speed. The specification should be part of the subroutine definition, and calls for both types should be written the same way. This facilitates changing the method used (i.e., only one definition, not numerous calls, must be changed) when tuning for the best time-space balance.

Another feature required to provide object code efficiency is programmer control over packing of data. This feature allows the programmer to choose between the space savings possible with packed data (e.g., packed Boolean items) and the speed of accessing provided by unpacked data. A choice of parallel or serial table allocation also provides control over execution speed, since data can be arranged to support efficient accessing.

The need for execution speed has dictated the need for multiprocessing. More than one CPU (typically three or four) are required to obtain the desired performance. Multiprocessing requires that a SHOL support inter-CPU communication and sharing of data. Section 5.4.2 discusses language features directly supporting multiprocessing. In addition, conditional compilation (Section 5.5.2) is useful in adapting programs to the CPU on which they will execute. It allows essentially the same program to operate on different CPUs.

4.5 Program Lifetime and Stability

In general, simulator programs have a long lifetime, since simulators can be used for years before becoming obsolete. This implies that the programs must be understood and modified by programmers who did not produce the original program or make previous modifications. A SHOL should assist in making programs understandable so changes can be made quickly and correctly by programmers

unfamiliar with the program. Thus, program readability should be stressed over programming ease (which might be preferred for programs with a short lifetime).

Simulator programs are fairly stable once they are put into use. Changes are most frequent in the tactics simulation programs, e.g., the programs emulating the onboard computer software. Frequently the simulator user (the customer) makes program modifications in the field. Modification by individuals not involved in program creation and also not primarily involved in simulation engineering disciplines demands program understandability and readability.

Among the kinds of language features that foster understandability are, for example, the status, or enumeration, data type. Section 5.2.3 presents several examples of the use of status data in programming simulator functions. Explicit data declarations are also important. The type of each variable should be stated explicitly (and in a readily-findable location in the program text). In addition, the ability to assign mnemonic names to constants (e.g., the use of the name PI for the constant 3.1415...) enhances understandability. To prevent modification errors, such constants should be a distinct language entity, not just variables initialized to desired values.

Error prevention and error detection features also help to reduce modification errors. Among the features facilitating error prevention and detection are strong typing and range declarations. Strong typing means that implicit type conversions are forbidden (e.g., when assigning a value to a variable). Forbidding implicit type conversions helps to flag errors when programs are modified. Similarly, range declarations, i.e., the specification of the intended value range of a variable, helps to prevent and detect errors. Range information is readily available for simulator data, so a requirement for its specific inclusion in programs should not present a problem.

Many of the language features dictated by general language design principles also contribute to program readability and modifiability. For example, uniformity in language syntax, structured programming constructs, and simplicity of the language all make programs easier to understand. A comment facility which is flexible and convenient to use also encourages production of understandable programs.

The implications of onsite simulator program modification are twofold. If the modifications are made by patching, the SHOL compiler must provide listings of machine code representations of programs and possibly other loading/relocation information. If modification is done by recompilation, it is necessary that the compiler operate on the target machine (see Section 4.3) since the users would not necessarily have access to the host machine used by a cross-compiler.

4.6 Program Reusability

Program reusability (as opposed to program portability, which is discussed in Section 4.7) is concerned with the ability to reuse a program for different purposes for the same target computer family. An example of reusability is the generation of slightly different subroutines adapted to the different simulation capabilities of the cockpits in the Undergraduate Pilot Trainer simulator.

Features supporting reusability are essentially program generation features, i.e., they permit different versions of a program to be easily compiled. Conditional compilation is an essential language feature for facilitating program reusability since reusable routines are often too general purpose for efficient, special purpose use. Conditional compilation is used to remove unneeded generality from such a routine. Use of constant names is another way of adapting a program to different configurations. These constant names can be used to specify configuration-dependent constants for reusable modules.

Neither conditional compilation nor the use of constant names permits onsite patches to programs as a means of adapting to new configurations (e.g., if code has been conditionally dropped out, a patch cannot access it). The use of patches to make the kinds of changes achievable through conditional compilation and use of constant names does not appear to be a significant practice in simulator environments. Configuration changes often require recompilation/assembly because of their complexity.

4.7 Program Portability

Program portability is concerned with the use of HOL source code for different target computers. There is greater need for portability in the simulator applications area than in most. Many programs change very little from one simulator to the next; e.g., the navigation and communications programs; simulation of radio stations does not depend on the particular aircraft involved. A similar situation exists in the tactics area when simulating radar emitters and various types of weapons. Even in simulation areas more dependent on the aerodynamics of the aircraft, portability is possible (e.g., solving six degrees of freedom equations). Visual systems also have much the same processing from one system to the next (e.g., probe and gantry control, altitude limit, visual effects, and cultural lighting controls). Such systems are seldom identical but have considerable processing in common. Since there is significant potential for portability in the simulator area, a SHOL should encourage development of portable programs. Indeed, this is one of the major advantages of using an HOL as opposed to assembly language.

A truly portable program, of course, must not be target machine dependent. This goal is probably unrealizable for simulator systems as a whole, though it may be possible for some modules. However, machine-dependent code can be isolated, thus facilitating the changes required to transport the program.

Machine dependency arises in several ways. The most obvious is in the programming of functions which cannot be implemented in the HOL and must be implemented in assembly language. (Section 5.7 discusses these functions, as required in simulator programming.) Assembly language code should not be intermingled with HOL source code but should be encapsulated to ease its replacement. The HOL should probably require that assembly code be used only in separate assembly language subroutines. In the programs studied, most functions requiring assembly language occur in the monitor area, which might well be an area where not too much portability can be attained due to the nature of the functions required.

Another instance of machine dependency is in programmer-specified data packing. As indicated in Section 4.4, this feature may be necessary to attain required time and space efficiency. It is definitely necessary to describe I/O data, as discussed in Section 5.3.4.1.1. However, it implies that programs cannot be transported to a machine with a different word length (at least, not to a smaller word length -- a larger would simply mean a waste of space).

Machine-dependent packing can be avoided by the use of machine-independent packing attributes when control over packing is needed just to make time-space tradeoffs. Packing attributes allow a programmer a choice of several degrees of packing (e.g., unpacked, medium, dense, tight) without requiring actual specification of bit positions. The packing attributes are then compiled appropriately for the target computer. Data packing specified in this way does not hamper portability. For I/O interface data, programmer specification of actual bit-level packing is necessary. Such specifications, which are machine-dependent, should be encapsulated in some way to support isolation and change. They might, for example, form a separate block or module of the global data-base.

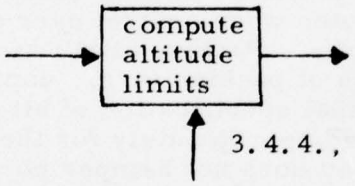
A third source of machine dependency is programmer reliance on internal representations of data, e.g., on the available range and precision of real (fixed or floating) values for a particular target machine. A SHOL can assist portability by providing built-in operations to access implementation information, such as precision, radix, and exponent range of floating point values. A related language feature is the ability to specify machine configuration constants reflecting, for example, machine model, word sizes, etc. These can then be used with conditional compilation to include/exclude machine-dependent code.

Section 5

PROGRAM ANALYSIS OBSERVATIONS

Part of our general approach to defining HOL requirements is to analyze how the functions to be programmed (as opposed to the environment in which the programming takes place) affect the choice of language features. In this Section, we present our observations of simulator programming characteristics, as revealed by our study of simulator programs and discussions with simulator personnel. These observations are part of the basis for the detailed language requirements specification given in Section 6.

The simulator program functions and support program functions are described in Appendix A in the form of an SADT model. This model describes the functional components of a simulator and demonstrates our understanding of the simulator programming tasks. The remainder of this Section contains an analysis of how the functions to be programmed can be supported by various HOL features. Our observations about the relation between simulator program functions and HOL features are cross-referenced to the relevant parts of the SADT model (e.g., a reference to "diagram A33" is to diagram A33 in Appendix A). Similarly, the model diagrams reference this Section of the report. For example,



indicates that language requirements for "compute altitude limits" are discussed in Sections 5.3.4.4 and 5.5.1.1. (Since the first digit of all section references is 5, this digit has been omitted from references in the model.)

The analysis presented here is grouped by language area. The overall organization is:

5.1 Storage Management

5.2 Data Types and Operations

5.3 Aggregate Data Types

5.4 Control Structures

5.5 Program Development Aids

5.6 I/O

5.7 Machine Dependency

5.1 Storage Management

5.1.1 Global Data

The primary storage management facility in the Link simulators is the 'datapool', or system symbol dictionary. System data in this global data base is used and/or updated as required by the simulation programs and by the cockpit and instructor station I/O processing. Diagrams A1, A3, and A33 illustrate datapool use. The datapool is similar to the JOVIAL COMPOOL. When programs reference datapool items, the address and type of the item is retrieved during assembly/compilation by the database system (see Section 5.5).

Datapool items are grouped into blocks. Each item's location is defined by its displacement within the block. This grouping facilitates relocation as well as allowing related items to be grouped together. The data is broken up into 5 groups:

private - cockpit dependent arithmetic

private - cockpit independent

common - cockpit dependent logical

common - cockpit dependent arithmetic

common - cockpit independent

'Common' and 'private' refer to the common and private memories. Cockpit dependent data is data of which there are several copies, one for each cockpit. To provide the user with access to the current cockpit data, the monitor (in the foreground dispatcher - diagram A312) initializes index registers to the base addresses of the data areas for that cockpit. In the UPT simulator, for example:

register I = private - cockpit dependent arithmetic

register K = common - cockpit dependent arithmetic

register V = common - cockpit dependent logical

The V register is a base register used for bit access instructions. Its contents here will actually be the same as the K register, since the common cockpit dependent logical and arithmetic data occupy the same data area. There are actually only two areas, private and common.

The programs which access the cockpit dependent data then can reference, for example:

var,K

and obtain the value of 'var' for the current cockpit. Ordinarily the index register will be dedicated to this purpose. However, the monitor also places the current values in global variables in case the user has to destroy the register values and must then restore them. The programmer need not actually know whether a particular item is in common or private memory. Cockpit dependent variables are referenced by

```
var,R
```

and the assembler (as modified by Link) substitutes either 'I' or 'K' for 'R', based on information from the symbol dictionary.

An HOL implementation might group the data into large tables, one for private data and one for common data, where each table has 4 entries, one for each cockpit. This table could then be indexed by the number of the active cockpit, so that

```
var(CKPT)
```

would refer to the value of 'var' for the current cockpit. Such large tables, however, are rather unwieldy, and their elements would not all be simple items, but would sometimes be tables, arrays, etc., leading to confusing subscripting. A much better representation would be something like the based block of PL/1, i.e., a block based on a pointer variable established for the current cockpit. In any implementation, it would be desirable that the compiled code dedicate an index register to the block. (Typically, most statements in the program will reference data in one of the two blocks, so a good compiler should come close.) Statements explicitly dedicating an index register are also a possibility.

The datapool as used by Link is an important tool for program reusability. The concept is used in each simulator, and many data items are the same in various simulators. Some such global data definition facility seems essential in a simulation HOL, however it be provided. It might be desirable to restrict unauthorized access to global data more directly in this facility. Currently this can be done only indirectly through examination of the cross-reference listings.

5.1.2 Local Data

The simulator programs also make use of a temporary storage area (50 words in the 214A simulator, for example). These are shared by all the programs. Programs using them cannot call one another, but this is not a problem. Generally programs are initiated from the monitor and return to it on completion without calling other programs (diagram A312 illustrates task scheduling). Programs using the temporary storage area cannot be reentrant but reentrancy is required only in a few monitor subroutines.

Use of these temporary locations can make programs unreadable if the temporary storage names are not equated to more meaningful names. As an example of this problem, see Figures 5-1 and 5-2, which illustrate storage usage in display system processing (Box 4 of diagram A35). A simulation HOL with a similar local storage strategy would have to provide some means of assigning meaningful names to these locations in the various programs which use them.

1. SYT02 - CCL WORD 1 ADDRESS
2. SYT03 - CCL WORD COUNTER
3. SYT04 - CAB RANGE
4. SYT05 - CAB WORD 1 ADDRESS
5. SYT06 - CAB WORD 2 ADDRESS
6. SYT07 - CAB WORD 3 ADDRESS
7. SYT08 - FIELD WIDTH
8. SYT09 - NUMBER OF FRACTIONAL DIGITS
9. SYT12 - INTEGER TO BE CONVERTED
10. SYT13 - FRACTION TO BE CONVERTED
11. SYT15 - ANSWER STORAGE
12. SYT17 - CAB WORD 9 ADDRESS
13. SYT24 - RETURN ADDRESS

Figure 5-1. Temporary Storage for Conversion Control Program

1. SYT00 - RETURN Address
2. SYT01 - CCL Address
3. SYT07 - Cockpit Number
4. SYT09 - Address of CAB Word 1
5. SYT21 - Address of CAB Word 2
6. SYT23 - Address of CAB Word 4
7. SYT24 - Address of CAB Word 5
8. SYT25 - Address of CAB Word 6
9. SYT26 - Address of CAB Word 7
10. SYT27 - Address of CAB Word 8
11. SYT10 - Field Width
12. SYT11 - Number of Fractional Digits
13. SYT12 - Sign Flag
14. SYT20 - Decimal Point Flag
15. SYT13 - Integer
16. SYT14 - Fraction
17. SYT17 - Input
18. SYT19 - Answer
19. SYT03 - Private Cockpit Base Address
20. SYT04 - Command Cockpit Base Address

Figure 5-2. Temporary Storage for Parameter Insert Program

5.1.3 OWN Data

OWN data is data local to a subroutine whose value must be retained between invocations of the subroutine. Some uses of OWN data were observed in the programs studied. For example, the 214A program which verifies data on the modelboard contour map (Box 1 of diagram A3353) retains location information between calls. Another example is the Monitor TTY output driver, which must extract a buffer address and character count from the parameter table on the first invocation, then maintain them by incrementing the address and decrementing the count through subsequent invocations. A third example is the timer data (fire and overheat timers, indicated by the two-way output of Box 1 of diagram A33135, and icing timers, Box 2 of the same diagram) used in the Miscellaneous Accessories area. These timers are used to keep track of the time since the indicated problem (e.g., engine overheat) was initiated so warnings, etc. can be turned on after the appropriate interval.

5.1.4 Overlay Programs

Overlay programs are used in several instances in the UPT simulator. For example, system initialization (Box 1 of diagram A31) makes use of programs that are replaced in core after initialization. Overlays are also called in as required to process debugging and display handling functions. Some offline programs (e.g., Math Model Test, Box 2 of diagram A2) are also organized in overlays.

Ordinarily overlay handling is an operating system (OS) function (of course the OS should be implementable in the selected HOL) but in some machines the program must call the OS to bring in overlays; in others, it can be effected automatically. If the program has to request overlays, the language must make this possible. To be usable in real-time applications, overlay handling must be efficiently implemented.

5.2 Data Types and Operations

5.2.1 Integer Data Type

The programs studied show few uses of integers to represent actual simulator numeric data. Some uses were observed in the Miscellaneous Accessories area (diagram A33135) for fire, overheat, and ice timers. Another use was observed in a weapon jettison program (Box 2 of A334). In general, however, most integer variables are used for loop indices, Booleans, enumeration types, array indices, etc. The use to represent Boolean and enumeration values would disappear in a language that supported these data types directly. The remaining uses are not peculiar to simulators, (array indices, loop indices, etc.) but are found in almost any use of a language.

5.2.2 REAL Data Type

REAL data is used throughout the simulator for numeric values. Areas in which mathematical processing is particularly heavy are Aerodynamics (diagram A3312), Visual (diagram A335), and Tactics (diagram A334) modelling and the offline Map Plate Compiler (Box 5 of diagram A35). Most aircraft data (speed, roll, pitch, yaw, altitude, drag, etc.) is REAL data.

5.2.2.1 Fixed Point vs. Floating Point

Both fixed and floating point are used to represent REAL values. Link has indicated that fixed point is always selected unless contract requirements specify floating point. (When FORTRAN is used to program a simulator, however, floating point is used. No attempts at fixed point arithmetic with FORTRAN integers have been noticed.) This preference for fixed point comes partly from a desire to continue using the same data definitions, scaling, etc. used on previous simulators. Of course, once floating point is established, the new definitions could be reused. Other considerations are the programmers' unfamiliarity with floating point and concern about loss of control over significance in computations. A third problem is that floating point is sometimes slower than fixed point (see [Babel, 1974]). Another paper [Goldiez, 1976] compares the relative speed of assembly language and FORTRAN programs; the FORTRAN programs took almost three times as long. The authors blame this discrepancy at least partially on the fact that the FORTRAN versions used floating point while the assembly language versions used fixed point.

The UPT simulator, one of the systems studied, uses primarily floating point. On the UPT computer, the Harris DC 6024/4, floating point is all double word, with 39 bits of mantissa. In the 214A simulator, also studied, all arithmetic data used is fixed point. Much of the data in this simulator is two-word (i.e., 32 bit) fixed point. As two-word arithmetic is not supported by the instruction set, handling such data takes a lot of code. For example, subtracting one such value from another requires five instructions. Addition, if it is necessary to test for overflow, takes nine instructions. A high-level language which supported such a data type would result in much clearer programs. Certainly, support of 16-bit fixed point only is not adequate.

Reasons for the various scale factors used for fixed point data are not always clear, since the range of data items is not always apparent. Presumably scale factors are selected to allow retention of maximum significance during calculations. In all instances where the units represented by fractional values are apparent, the step size is a power of two. Much rescaling occurs during calculations. Some uses of fixed point fractions occur when

integer data is being used, resulting in some loss of efficiency. For example, the code used in the 214A simulator to multiply an integer value of 1, 2, 3, or 4 in R0 by a constant 3 is:

```
MUL   #3B2,R0      constant 3 scaled with 2 integer,
                   13 fractional bits
ASHC  #3,R0        shift register pair left 3 to get
                   result in R0
```

Multiplying by an integer 3 would have given the desired result in R1 in one step. If it had to be in R0, a move instruction, more efficient than a double shift, could then be used. Presumably this sort of thing is due to programmer habit and illustrates that some inefficiency can be tolerated in some fixed point computations.

Several instances of fixed point data occur in the UPT simulator. For example, latitude and longitude are often expressed in BAMs (Binary Angular Measurement), or degrees/360. Sometimes BAMs are represented in double word fixed point, e.g., in the Navigation Environment area (diagram A332, Box 2 outputs). The documentation (UPT Product Specification, Vol. 1, Navigation Environment, p. 31) describes the reason:

> "Since Lat and Long are defined in BAMs (Deg./360) the 23rd bit in the single word is equivalent to 7.82 ft which is not enough resolution to maintain required accuracy. The 48th bit in the double word will be equivalent to 4.7×10^{-7} ft."

The UPT also uses one-word fixed point in certain tables to save space as compared to the two words required by floating point. This occurs, for example, in the LFI (linear function interpolator) routines, which are described in more detail in Section 5.3.4. The LFI subroutines access a table of values in fixed point form. Once the correct value is found, it is converted to floating point before being returned. The reason for having the table in fixed point form is economy of space, since the fixed point values take only one word while two are required for floating point. The precision allowed by the single word is adequate for the values. The breakpoint table, used in the LFI search routines, contains floating point values. The breakpoints require greater precision. In the case of a two or three variable LFI, of course, the value table is much larger than the breakpoint table, so there is more motivation to save space. (In the 214A visual system, which is all fixed point, the value tables are all single precision; the breakpoint tables are sometimes single and sometimes double precision. Altitude breakpoints, for example, are double precision.)

Fixed point values are also used in the display system (diagram A35). These are generally values that appear in large tables or disk files (e.g., saved track data, Box 6 of diagram A35). Fixed point is apparently used to save space over the two words required by floating point.

5.2.2.2 Operations on REAL Data

The standard mathematical operations (+, -, *, /, **) are of course required. Other operations, performed by macros (inline functions) or by subroutines, are described in subsequent subsections.

5.2.2.2.1 Trigonometric Functions

The functions used are SIN, COS, and ARCTAN. The map plate compiler (Box 5 of diagram A35) uses secants and cosecants, but since these are simply inverses of COS and SIN, no separate functions are used. The F-14A and 214A simulators include a single routine to compute both SIN and COS, thus saving time when both are required. The UPT has only the two separate routines.

In the UPT simulator, the routines use fixed point (BAMs) for the angles and floating point for the functional values. Some routines that call them, e.g., Aerodynamics (diagram A3312), maintain angles in floating point and must convert to BAMs to use them. Calls to the SIN and COS routines are preceded by float-to-fixed conversions, and calls to the ARCTAN routine are followed by fixed-to-float conversions.

5.2.2.2.2 LFIs

Linear Function Interpolation is the calculation of a functional value by linear interpolation of its arguments in a predefined table of arguments and associated values. Two routines are used. The 'search' routine looks up the argument in the predefined argument (or breakpoint) list, obtaining the interpolant. The 'value' routine, with this interpolant as a parameter, computes the function value. The data structures used are discussed in Section 5.3.4.

5.2.2.2.3 Limit Functions

A common operation in the programs studied is a limit operation of the form:

MAX (MIN(expression, upper bound), lower bound)

e.g.,

```
MAX(MIN(TEMP02 * 512., 400.), -400.)
```

or

```
MAX(MIN(.95238 * (.525 - HAIFLP), .999), 0.)
```

This is frequently written in the documentation in notation of the form:

$$[.95238*(.525 - HAIFLP)]_{0}^{.999}$$

This function arises from a need to limit a value to an acceptable range, frequently for output to analog hardware. It is particularly prevalent in the Aerodynamics (diagram A3312), Flight Controls (diagram A331, Box 1), Hydraulic System (diagram A33133), and Navigation Radios (diagram A3323) areas.

5.2.3 Status Data Types

There are numerous instances in the simulation programs studied where a status (i.e., enumeration) data type would enhance program readability. Status types would be useful for flags, for table and array indices, and for CASE alternative selectors (see also Section 5.4.1.2).

5.2.3.1 Status Data as Flags

Examples of flags that could be represented by status types occur in the monitor area (diagram A31), in the demo/record/playback area (Box 1 of A35), in the instructor area (Box 2 of A35), and the visual area (diagram A335). In the monitor area, flags are used to synchronize the various processors and to coordinate I/O, the CPU number and cockpit number used by the foreground task dispatcher (diagram A312), etc. In the demo/record/playback area, flags are used to avoid playing a demo which is currently being recorded. In the instructor area, status variables would be useful to describe many of the initial condition settings, e.g., day-dusk-night. In the visual area, subroutines are used with integer parameters indicating X, Y, or Z values are to be processed. The X, Y, Z enumeration type would enhance readability here also.

5.2.3.2 Status Data to Index Tables and Arrays

Throughout the simulators examined, readability would be greatly enhanced by grouping related data items into tables and arrays, even when the resulting structure will have only two or three entries. In such cases, an enumeration or status type is the logical choice to index the structure. Consider, for example, the set of items connected with the DME dial (Navigation Radios, diagram A3323, Box 3):

NDMUNT - DME units wheel

NDMTEN - DME tens wheel

NDMHND - DME hundreds wheel

NDUNTX - DME units output x

NDTENX - DME tens output x

NDHNDX - DME hundreds output x

NDUNTY - DME units output y

NDTENY - DME tens output y

NDHNDY - DME hundreds output y

NDUNT1 - DME units wheel previous pass

NDTEN1 - DME tens wheel previous pass

NDHND1 - DME hundreds wheel previous pass

which could be represented by a 3-entry table (perhaps indexed by an enumeration type UNITS, TENS, HUNDREDS):

```
TABLE DMEDIAL (UNITS:HUNDREDS):
BEGIN
ITEM DMWHEEL    wheel value
ITEM DMOUTX     output x
ITEM DMOUTY     output y
ITEM DMWHEEL1   previous pass wheel value
END
```

Then, for example, the item currently called NDMUNT would be referred to as DMWHEEL(UNITS). Similarly, engine data (diagram A3313, Box 4), wing data (diagram A3312), etc., could be organized in tables indexed by LEFT and RIGHT, or nose position in a table indexed by (UP, DOWN, CENTER).

Many flight data items are actually vectors of X, Y, and Z values. These are all represented by sets of three variables, e.g.,

VXGMI, VYGMI, VZGMI

or

VXPFB, VYPFB, VZPFB

These would be best represented in an HOL using 3-element arrays, which could be indexed with an enumeration type using the identifiers X, Y, and Z, e.g.,

VGMI(X), VGMI(Y), VGMI(Z)

or

VPFB(X), VPFB(Y), VPFB(Z)

Similarly, there are numerous sets of values for roll, pitch, and heading, e.g.,

VCPHI, VCPSI, VCTHTA

which might be represented by 3-element arrays indexed by PHI, PSI, and THETA.

5.2.3.3 Status Data as CASE Alternatives

In the programs studied, there are numerous instances of CASE-like control structures. In some of these the various cases could be most understandably represented by status or enumeration type values. For example, one subroutine in demo/record/playback (Box 1 of diagram A35) has as a parameter an integer indicating the type of processing to be done:

0 = disk read initialization data

1 = disk read demo data

2 = unpack disk data

3 = playback

4 = flyout

Clearly mnemonic values would be more understandable than integers as parameters.

Other examples of CASE alternatives best represented by enumeration types occur in the monitor area (diagram A31). These include the function specifications for the intercomputer communications and background scheduling routines, and the I/O device selection for the input/output control coordinator.

5.2.4 Bit Data Type

Most occurrences of bit data in the simulators studied correspond more logically to the set data type discussed in Section 5.3.1. The major use of bit data is in machine- or device-dependent code.

Instances of bit operations (bit insertion and extraction) could generally be avoided through the use of programmer-defined tables to access the desired bits. These are discussed in Section 5.3.4. Some such instances may be rather forced, however. An example of this occurs in the Math Model Test program (diagram A2) of the UPT simulator in the part of the program which formats instructions for the trace, translating them from binary to mnemonic form. The UPT computer has a very complicated instruction format for this purpose. For example, register specifications are part of

the opcode mnemonic rather than operand fields. Thus there are 42 ADD opcodes, for instance. This translation program involves a main routine and 46 subroutines as well as three tables of opcodes. Processing is roughly as follows:

a. Based on the first six bits of the opcode, one of the three tables is selected, for prefix - 00_8, 77_8, or 'other'.

b. Within the selected table, an entry is chosen by indexing by the first six bits of the opcode for the 'other' table, and by the second six bits for the 00_8 and 77_8 tables.

c. The table entry consists of two words, containing:

opcode mnemonic		
$sub_{no.}$	$sub_{no.}$	$sub_{no.}$

The (possibly temporary) opcode mnemonic is obtained from the first word. The second word contains three 8-bit numbers between 0 and 46, which are subroutine numbers indicating one of the 46 subroutines, or 0.

d. The indicated subroutines are executed in the specified order. An entry of 0 terminates the list.

For example, consider the instruction 00230220_8. Based on the first six bits, the 00_8 table is selected. Based on the next six bits, the 23_8 entry is selected. This entry is:

PRR		
2	1	0

The opcode, 'PRR', is extracted. Subroutine 2 is executed. It extracts the last six bits, 20_8, determines that this indicates 'register A', and replaces the third character of the opcode by 'A', obtaining 'PRA'. Then subroutine 1 is called, looking up the next-to-the-last six bits, 02_8, translating this to 'register J', and obtaining the opcode 'PJA' (Positive of J to A). The various subroutines might also construct the operand field (PJA has no operands). This

implementation is unusual, so such complexity may not be required. The method, however, should be implementable in an HOL, using an array of subroutines or a CASE statement to select the subroutine calls. With a simpler machine, this sort of processing can be done nicely using a programmer-specified table to define (or overlay, really) the instruction word. The table will have variant record types, i.e., different definitions of the word, based on the various instruction formats. The necessary fields can be accessed directly without explicit bit extraction. The DC 6024/4 machine, however, might have too many instruction formats for this to be practical.

Another area where bit manipulation is necessary is in some types of number conversions. These usually involve 'logical or' and shift operations. It is possible to do this without shifts, technically. One UPT FORTRAN program for formatting display screen images (Box 4 of diagram A35) does conversion using addition instead of logical or's, and multiplication and division by powers of two instead of shifts. This does not enhance readability, however, and could be very inefficient if the multiplications and divisions do not compile as shifts.

Bit accessing is occasionally used in 'coding tricks' to gain efficiency. For example, in the 214A Visual System (diagram A335), the code used to implement the test

"If cockpit = 2 or 3, return to caller"

is:

```
BITB #2,QCKPT     test bit 2 of cockpit number
BEQ  V$560A       if off, execute program
JMP  V$560Z       otherwise, jump to last statement
```

A compiler could probably not be expected to generate such a test, even if it was known that the range of QCKPT was 1-4. However, only two instructions are saved over a more explicit translation of the IF statement.

5.2.5 Boolean Data Type

Certain simulator areas involve large quantities of Boolean, or logical, data. This is particularly true in the Navigation and Communications area (diagram A332). Much of the data corresponds to hardware input and output values. In fact, the datapool allows specification of 10 different types of Boolean items:

- VB - logical bit
- DI - discrete digital input
- LO - discrete lamp driver level output

DO - discrete logical level output

WI - hardware bounded digital word input

WL - hardware bounded lamp driver level word output

WD - hardware bounded logic level word output

PI - software bounded digital word input

PL - software bounded lamp driver level word output

PD - software bounded logic level word output

A major issue with Boolean data is the degree of packing used, with a tradeoff between space used and speed of access. One objection to FORTRAN expressed at Link was the necessity of allocating full words to logical values. The programs studied allocate logicals to bits, bytes, and words, depending on the type of efficiency required.

The 214A Visual System (diagram A335) uses many Booleans for flags and indicators. These are all represented by bytes, the minimum addressable unit on the PDP-11. Two exceptions were noted where bit data was used for flags. In one instance, three bits in one byte were used as flags indicating that the roll, pitch, and heading of the probe are in sync with the aircraft. There is no particular use made of the fact that the flags are stored this way - it is just done to save two bytes. Since the machine has bit set and test instructions, there is no additional cost in processing. In the other case, the four top bits of a word are set to indicate which of four cultural lighting boxes are to be turned on. The code which turns on the boxes is a loop which shifts this word left and tests if it becomes negative (sign bit set) to determine whether to turn on each box. An array of Booleans, which could be accessed by the loop index already in use, would support this concept equally well. Packing would be required if only one word was to be used.

Many packed Booleans are also used. In the Hydraulic Systems area (diagram A33133), for example, 14 flags dealing with landing gear and landing gear door positions are packed in a single word. Communications Booleans are also often packed. These are hardware inputs, and the packing is thus determined by the hardware. If the packing could be specified appropriately, organization of this data into tables would improve clarity. For example, consider the actual input layout described in Figure 5-3. For logical purposes, this data should be treated as a table indexed by operator and cockpit (except NO1PTT and NO2PTT, which correspond to operator only). This could be represented by a PL/I structure (see also Section 5.3.4) such as:

```
DECLARE 1 NRADIO (1:2),        indexed by operator
        2 NOPTT,               operator push to talk
        2 NCKPT (1:4),         indexed by cockpit
```

NO2W0I	NO1W1I	NO2W3I	NO1W2I	
IOS #8	IOS #8	IOS #8	IOS #8	
				high-order
		NO22UR	NO12UR	BIT 15
		NO21UX	NO11UX	14
		NO21UR	NO11UE	13
		NO24IS	NO14IS	12
		NO24OS	NO14OS	11
		NO23IS	NO13IS	10
		NO23OS	NO13OS	9
		NO22IS	NO12IS	8
		NO22OS	NO12OS	7
		NO21IS	NO11IS	6
		NO21OS	NO11OS	5
NO24UX	NO14UX	NO24XV	NO14XV	4
NO24UR	NO14UR	NO23XV	NO13XV	3
NO23UX	NO13UX	NO22XV	NO12XV	2
NO23UR	NO13UR	NO21XV	NO11XV	1
NO22UX	NO12UX	NO2PTT	NO1PTT	BIT 0
				low-order
WORD 0	WORD 1	WORD 3	WORD 2	

Figure 5-3. Communications Input Word Layout

```
3 NOXV,     VHF xmit selected
3 NOOS,     override selected
3 NOIS,     instructor selected
3 NOUR,     UHF receive selected
3 NOUX;     UHF xmit selected
```

The only problem with this is that there is no apparent way to describe how the bits are actually packed. (It is possible that the data could be repacked appropriately by the executive, or that the packing coming from the hardware could be altered.) The program currently must perform complex bit manipulation to access data in the order desired, anyway. For example, one sequence of code, with the objective of obtaining NO14UX, NO13UX, NO12UX, NO11UX packed into the right-most four bits of a single word, proceeds as follows:

a. load word 2

b. mask to zero all bits except bit 14 (N011UX)

c. shift left 3 (into bit 17)

d. save result

e. load word 1

f. rotate right 5 (to get the 5 bits into the high-order bits)

g. mask to zero all bits except the top 5

h. add the word previously saved (bits 23, 21, 19, 17 now contain desired values)

i. clear the accumulator extension (extends on high-order end)

j. do 4 times:

   j1. shift left double 1 bit (shift desired bit into extension)

   j2. shift left (single) 1 bit (drop unwanted bit)

k. extension now contains desired result

The amount of processing involved makes reformatting in the executive seem like a reasonable alternative.

Various documentation techniques are used to describe Boolean equations. Some programs use English, as in this example from the 214A Visual System (diagram A335), determining if the aircraft is above the clouds:

```
ABOVE CLOUDS IF
    OFF MODEL AREA OR
    OUT OF TOLERANCE OR
    LIGHTS NOT READY OR
    COMMANDED POSITION CHANGE TOO BIG OR
    PROBE NOT IN SYNC OR
    A/C ALT > CEIL + FIELD ALT + CLOUD THICKNESS
```

Some use FORTRAN or pseudo - FORTRAN, as in this example from Flight Controls (diagram A331):

```
FELTRD = TMPL01 .AND. .NOT. (UMLTRE.LT.UMLTCE) .OR.
    .NOT. FELTRD .AND. (FTRIME .LT. 25) .AND.
    UMLTRE .OR. UMLTCE .AND. (FELTRD .OR.
    TMPL01 .AND. .NOT. FELTRU)
```

A decision table representation (see Section 5.4.4.1 for examples) would be more readable but might require some simplification by the programmer. A less unwieldy notation than that of FORTRAN would also be helpful. Use of longer and more descriptive names might also improve readability, though it would make the expression even longer.

Another documentation technique, from the Communications area (diagram A332, Box 1), using data from Figure 5-3, is:

$$O1VXN = O1PTT \cdot O11XV \cdot \overline{O11OS} \cdot \overline{O11IS}$$

= O1PTT	2	2	2
= O1PTT	3	3	3
= O1PTT	4	4	4

In this assignment, the cockpit number (1-4) is used to determine which of the four lines applies. The implementation of this expression is interesting, and represents a degree of efficiency which might be difficult to obtain in an HOL. For example,

```
OVXN(1) = NOT (OPTT(1) AND OXV(1,CKPT) AND NOT
          OOS(1,CKPT) AND NOT OIS(1,CKPT))
```

would be a reasonable HOL expression of the equation using the data structure described previously. (See Figure 5-3 for allocation of the bits; these are the same variables without the initial 'N's.)

The actual implementation is:

a. Set O1VXN = true.

b. Using a mask obtained from a table of four masks indexed by cockpit number, mask input word 2 to clear all bits except O1PTT, O1nXV, O1nOS, and O1nIS. (n is the cockpit number)

c. Compare result to another value, also obtained from a table of four indexed by cockpit, which has bits set in the O1PTT and O1nXV positions only.

d. If equal, set O1VXN = false.

This sequence takes advantage of the fact that only one possible set of values,

O1PTT, O1nXV, $\overline{\text{O1nOS}}$, $\overline{\text{O1nIS}}$

will result in a value of false for the equation. It uses only seven machine instructions. However, a programmer who has done the necessary analysis to discover that this is possible could instead write:

```
OVXN(1) = (NOW2I(1) AND MASK(CKPT)) NE MVAL(CKPT)
```

masking and comparing to the desired masked value. Unfortunately, the intent of the operation is no longer clear.

Another example (from the same program) of efficiency gained by explicit knowledge of the Boolean packing is the implementation of the three successive equations:

```
NO1SPR = .NOT. (NOSPON .AND. NSPRO1 .AND. .NOT.
         NO1PTT .AND. .NOT. NO2PTT)
NO2SPR = .NOT. (NOSPON .AND. NSPRO2 .AND. .NOT.
         NO1PTT .AND. .NOT. NO2PTT)
NMONSR = .NOT. (NOSPON .AND. .NOT. NSPRO1 .AND.
          .NOT. NSPRO2 .AND. .NOT. NO1PTT .AND.
          .NOT. NO2PTT)
```

The variables that the equations have in common, NOSPON, NO1PTT, and NO2PTT are tested first. If NOSPON is false or NO1PTT or NO2PTT are true, all three expressions are true. To obtain the same degree of efficiency, an HOL implementation would probably have to make this test explicitly, e.g.,

```
IF NO1PTT OR NO2PTT OR NOT NOSPON THEN
     BEGIN
     NO1SPR = 1;
     NO2SPR = 1;
     NMONSR = 1;
     END
ELSE
     BEGIN
     NO1SPR = NOT NSPRO1;
     NO2SPR = NOT NSPR02;
     NMONSR = NOT (NOT NSPRO1 AND NOT NSPR02);
     END
```

The IF-THEN-ELSE expression of the problem seems more understandable as well as more efficient. That, and the fact that it is actually implemented this way, indicate that the equations were probably derived this way and then converted to the "single assignment statement for each variable" form for documentation purposes. Therefore, it doesn't seem that expressing the equation in the more efficient form in an HOL would be a problem for the programmers.

5.2.6 Character String Data Type

Character processing is not significantly used in the main aircraft modelling portion of the simulator (i.e., that processing illustrated by diagram A33). However, it is required for display I/O in the Training or Instructor area (see diagram A35), for numerous offline support programs (which create data files used during simulation) and for debugging support (e.g., Math Model Test, Box 2 of diagram A2).

5.2.6.1 Offline Character Processing Requirements

The offline programs process card image input and some produce printed output. Programs which create offline data files include:

radio station file creation (Box 4 of diagram A332)

malfunction compiler (Box 3 of diagram A35)

initial condition file creation (Box 2 of diagram A35)

screen image file creation (Box 4 of diagram A35)

map plate compiler (Box 5 of diagram A35)

tactics scenario file creation (diagram A334)

radar emitter data file creation (Box 3 of diagram A334)

These offline programs have the text processing requirements characteristic of compilers. Operations such as the PL/I INDEX and SUBSTR functions seem desirable. However, no uses of varying length character strings were noticed.

As an example of text processing requirements, the radio station file creation program (Box 4 of diagram A35) requires interpretation of input commands, testing of individual characters in strings, and insertion of individual characters into strings. This processing is implemented in FORTRAN using octal equivalents of the characters. Direct language support of character handling would result in much more readable code.

In the I/O performed by the offline programs, conversions between numeric forms and ASCII character strings are required. Some of these conversions could perhaps be supported by the I/O features of an HOL, particularly in output conversions. Input conversions are more difficult. To process a statement like

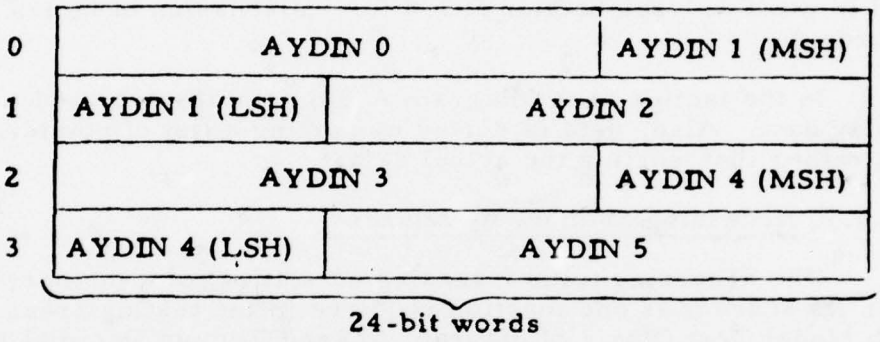
INPT var, value

from the Math Model Test program (Box 2 of diagram A2), the variable would have to be read and looked up in the datapool. Then the value would have to be read based on the variable type. (The input format might also have to be more fixed.) In an HOL, this could probably best be done by simply reading the card as a character string and then invoking an explicit conversion of the appropriate substring. (Another problem with trying to read or write with the various formats is that fixed point formats involve scaling information as well as type.)

5.2.6.2 Display Character Processing

Display character data is constructed and output by the Instructor System programs (diagram A35). Typically a different character set than that of the CPU will be used by the display. In any case, some of the display characters will be display commands or orders, and these would not be representable as characters of the computer. An HOL should provide some means of defining bytes representing these characters by specifying the bit equivalent and then using these in character manipulation expressions.

Display I/O ordinarily involves a block transfer, i.e., a movement of blocks of characters by a hardware operation to the display memory. The display memory may have a different word length than the CPU. In the UPT system, which uses an Aydin display, for example, the Aydin memory has 16-bit words, while the computer has 24-bit words. On input from the Aydin, each 16-bit word is stored right-justified in a 24-bit word. On output, however, the 16-bit words are packed into the 24-bit words as follows:



Construction of Aydin data involves conversion from the Harris ASCII code to the display code, as well as insertion of 16-bit display commands. Packing of the Aydin words into Harris words also requires some bit handling, i.e., shift and logical 'or'. Some programs construct the entire Aydin image with 16 bits/word and then call a routine to pack it. Interpreting Aydin input as character strings requires extracting the rightmost 2 bytes of each word and packing them into a string.

5.2.7 Pointer Data Type

In the programs studied, which are primarily in assembly language, uses of address data must be examined to determine possible requirements for pointers in an HOL. Most address data is simply used to point to a table element, and thus it corresponds to array or table indices (or subscripts) in an HOL. Some addresses are used as subroutine parameters to indicate the address of an input table. In an HOL this could be accomplished through call-by-reference parameters rather than with explicit use of addresses. Some uses of addresses for list processing functions correspond more logically to HOL pointers. Another use, which is not precisely a use of pointers, is the accessing of a memory location given its address.

5.2.7.1 List Processing Pointer Usage

The UPT monitor (box 1 of diagram A3) uses queues in which the elements are linked by pointers. There are two types of queues used: I/O request queues and an error request queue. The error request queue is circular, while the I/O queues are FIFO queues. Both queue types are constructed by request handling routines. These routines receive an input parameter that is the address of (pointer to) a parameter table for the request. They then insert this table into the appropriate chain. (Figure 5-4 illustrates an I/O request parameter table.) Queue elements are removed (unlinked) after being processed by the appropriate request handler. Note that the I/O request chain uses variant record types; the record type is determined by the TYPE field in word 0. (see Section 5.3.4 for a discussion of tables and records.)

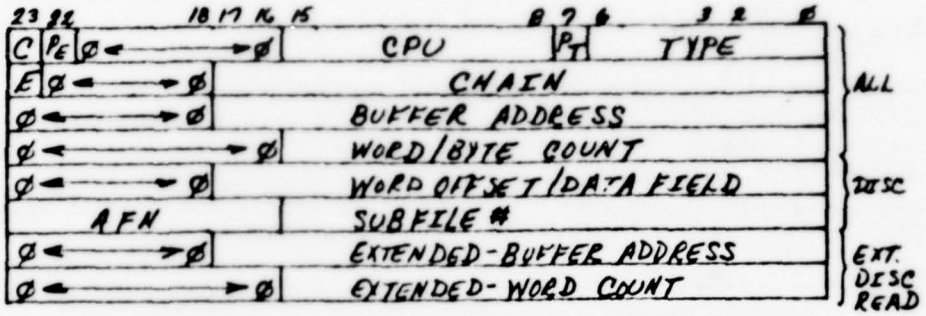
In the tactics area (diagram A334) pointers are used to chain display data. Also, data is sorted by sorting a list of pointers to the data rather than sorting the actual data.

5.2.7.2 Accessing Memory by Address

The accessing (either reading or setting) of a memory location given its address is one function required in the testing areas, i.e., Math Model Test (Box 2 of diagram A2) and Remote Decimal Readout (Box 3 of diagram A2), and in the programs associated with the instructor display/setting of datapool values (Box 4 of diagram A35). For example, the Remote Decimal Readout program provides access to core locations based on octal address. Similarly, the Math Model Test program provides a symbolic debugging capability, i.e., setting of datapool variables or printout of their values.

In a system written using an HOL, some such debugging system would still be useful. It might be available as a general tool provided with the HOL, i.e., a general-purpose symbolic debugger based on the global data base facility, or it might be implemented by the user. In either case, it should be possible to program the debugger in the HOL. Implementation of a debugger requires the same use of addresses as required by the display programs. Specifically, a symbolic datapool name specified by the programmer is looked up in the datapool and its address is obtained. The address must then be used to access the value of the symbol in order to display or alter it.

This function is probably not required in general-purpose HOLs, and it can lead to security problems. However, it is required in order to implement these simulator support programs. Some HOLs support it, but not very directly. For example, a PL/I pointer



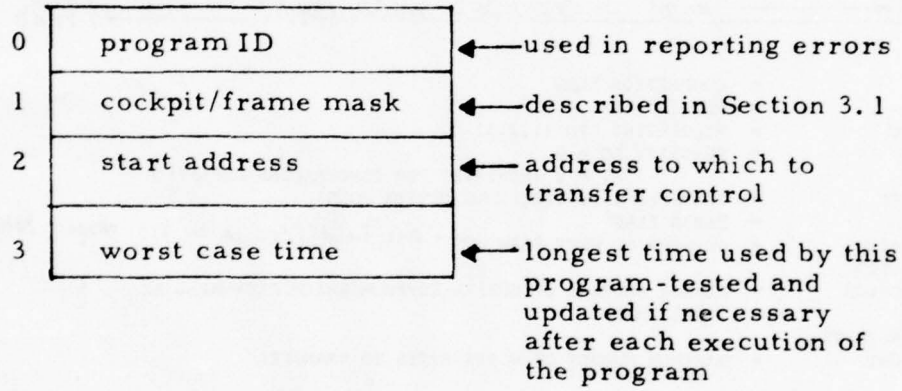
C	=	COMPLETION FLAG
P	=	PENDING FLAG
CPU	=	REQUESTING CPU (1,2,3)
P_T	=	PRIORITY LO = 0 HI = 1 (RESERVED FOR DEMO/RECORD/REPLAY)
TYPE	=	TRANSFER TYPE (INCLUDES DEVICE CODE)
E	=	ERROR FLAG
CHAIN	=	POINTER TO NEXT PARM ← link field, filled in by I/O request handler
BUFFER ADDRESS	=	MEMORY ADDRESS OF BUFFER TO/FROM WHICH DATA WILL BE TRANSFERRED
WORD/BYTE COUNT	=	MAXIMUM NUMBER OF WORDS/BYTES TO TRANSFER
		AYDIN ONLY
DATA FIELD	=	ADDRESS OR REGISTER ASSOCIATED WITH TYPE FOR AYDIN COMMAND
		DISC ONLY
WORD OFFSET	=	OFFSET INTO FILE (MUST BE SECTOR BOUNDARY FOR WRITES.) OFFSETS INTO SUBFILES ARE NOT ALLOWED.
SUBFILE #	=	NUMBER OF SUBFILE TO READ FROM INDEXED FILE.
AFN	=	ASSIGNED FILE NUMBER
		EXTENDED INDEXED READ ONLY
BUFFER ADDRESS	=	MEMORY ADDRESS OF BUFFER TO WHICH DATA WILL BE TRANSFERRED ON THE EXTENDED INDEXED READ
WORD COUNT	=	MAXIMUM NUMBER OF WORDS TO TRANSFER ON THE EXTENDED INDEXED READ

Figure 5-4. I/O Parameter List Structure

variable could be overlaid with an integer containing the correct address. Alternatively, in PL/I and in some JOVIAL implementations, an array could be overlaid to location zero, and accessed using the address as a subscript. Neither of these methods is particularly desirable, however. Some means of providing symbolic debugging, etc. (perhaps using another technique) should be available in a simulation HOL.

5.2.8 Label Data Type

In the simulators studied, only one instance was observed which might best be implemented with a label data type, specifically with an array of labels, though other alternatives are possible. This is task dispatching using the foreground task table (diagram A312), which would most likely not use the same table representation in an HOL. The current task table entries are of the form:



In an HOL implementation, program start addresses stored in a table would not be convenient for executing the programs in sequence. On the other hand, a sequence of calls cannot be used because of all the checking that must be repeated, e.g.:

```
IF PROGRAM1 IN THIS COCKPIT AND FRAME THEN
    BEGIN
    CALL PROGRAM1
    IF PROGRAM1_TIME >PROGRAM1_WORST-CASE-
        TIME THEN PROGRAM1_WORST_CASE_
        TIME = PROGRAM1_TIME
    END
IF PROGRAM2...
etc.
```

Alternatives would be an array of subroutines or an array of labels, e.g.,:

```
FOR I = 1 TO N
     BEGIN
     IF PROGRAM(I) IN THIS COCKPIT AND FRAME THEN
          BEGIN
          CALL PROGRAM(I) or GOTO LABEL(I)
          IF PROGRAM_TIME(I) > PROGRAM_WORST_CASE_
                  TIME(I) THEN PROGRAM_WORST_CASE_
                  TIME(I) = PROGRAM_TIME(I)
          END
     END
```

or a CASE statement (see Section 5.4), e.g.:

```
FOR I = 1 TO N
     BEGIN
     IF PROGRAM(I) IN THIS COCKPIT AND FRAME THEN
          BEGIN
          DO CASE I
                  CALL PROGRAM1
                  CALL PROGRAM2
                  .
                  CALL PROGRAMN
                  END
          IF PROGRAM_TIME(I) etc.
          END
     END
```

Something like a JOVIAL SWITCH or FORTRAN computed GOTO could also be used. In any of these methods, the three task table elements other than program address could still be in a table organization.

5.2.9 Procedure Data Type

There are three main classes of procedures in the simulators studied. These are:

a. The main simulation programs invoked through the foreground task table (as illustrated in diagram A312). These programs do not call one another; each returns to the foreground dispatcher on completion.

b. System subroutines, which are invoked by the various simulation programs to perform general-purpose service functions.

c. Subroutines internal to the main simulator programs.

Various subroutine calling and parameter passing techniques are used, as discussed in Section 5.2.9.3. Particularly in the subroutines internal to the main programs, there is a lack of consistency in methods used.

5.2.9.1 System Subroutines

The system subroutines provided in the UPT system are characteristic of those used in all simulators studied. These are:

LFI argument search

single variable LFI

two variable LFI

three variable LFI

sine function

cosine function (calls sine routine)

arctangent function

random number generator

I/O request handler

The F-14A and 214A simulators, unlike the UPT, include a single routine which computes both sine and cosine, as discussed in Section 5.2.2.2.1.

In addition to these routines, which are regarded as part of the monitor, the UPT system employs two general routines which access the surface radio facility data files (see Box 4 of diagram A332). These area:

DDP — (Digital Data Preselect) These subroutines are given a radio type and frequency from the aircraft dial. They search the preselect file to determine whether a station has been tuned. If one has, the data for the station is read into core from the real time radio data file. There are 4 DDP subroutines, which correspond to TACAN, VOR, DME, and ILS radio types.

RECEIVER — This subroutine uses the data that has been read into core and the aircraft data to compute ranges and bearings from the aircraft to the radio facility.

Various system macros are also provided (see Section 5.2.9.2). Functions provided by these include an LFI linear argument search, absolute value, and various limit functions (see Section 5.2.2.2.3).

5.2.9.2 Inline Subroutines

Both general purpose functions and functions specific to the various simulation programs are frequently implemented by macros, i.e., inline subroutines, rather than by actual subroutines. Some conditional assembly is employed in the macros. For example, in one macro, omission of the first parameter indicates that the parameter is already in the correct accumulator and need not be loaded.

Actually, all UPT system subroutines are accessed through macros. In the case of these subroutines, the macros set up the parameters, call the executive routine, and store results. For example, the routine LFI2 (double variable LFI) is called by a macro of the form:

```
LFI2   F200T, F200F, R, FMA01I, R, FCL01I, R
```

which expands to:

```
TME   FCL01I, R  |
TMA   FMA01I, R  }      input parameters
TOJ   F200T      |
BSL*  ZLFI2             call
TXM   F200F, R          store result
```

The use of inline functions indicates a choice of speed over space efficiency. Some fairly large subroutines are expanded a number of times. For example, in the UPT aerodynamics system (diagram A3312):

LIMFL - floating point limit - 8 instructions - expanded 24 times

LIM1I - limit to ± 1.0 - 9 instructions - expanded 9 times

LIM2N - limit to ± 2.0 - 11 instructions - expanded 8 times

LFIL - LFI linear search - 17 instructions - expanded 8 times

An HOL should allow the programmer a similar control over time-space tradeoffs. Ideally, the specification of inline expansion should be part of the subroutine definition; calls for both types should be written in the same way. This facilitates changing the method used (i.e., only one definition, not numerous calls, must be changed) when tuning for the best time-space balance.

In some cases within individual simulation programs studied, internal subroutines are used for operations which could be done much more efficiently inline, or with macros. Perhaps this reflects a desire to save space, but in at least one example, from the 214A Visual System (diagram A335), the extra code required to set up the subroutine call is such that this does not happen either. The use of the subroutine saves only one word of storage, and 34 more words are executed than would be required in an inline implementation.

5.2.9.3 Subroutine Calling Conventions

5.2.9.3.1 Main Simulation Program Calling

As discussed in Section 5.2.8, these programs are called from the foreground dispatcher (diagram A312) with an indirect transfer to the task table address. No parameters are passed; all communication is through the datapool.

5.2.9.3.2 System Subroutine Calling

In the UPT system, a single convention is used for all system subroutine calls, as illustrated in the example in the previous section. Subroutines are called indirectly through pointers, perhaps to facilitate relocation. (This is not true in the 214A simulator.)

Almost all parameter passing is in registers. In cases where a parameter is an array or table (e.g., LFI routines, I/O request handler) the parameter address is passed (call-by-reference in an HOL). In the LFI routines, there is a need for array parameters of varying sizes. In this implementation, the length is in a word preceding the array. A table parameter of varying length is also used by the display conversion routine (Box 4 of diagram A35). This table is a Conversion Control List - a list of values to be converted to Aydin form.

None of the UPT system subroutines save and restore registers. The UPT machine does not have many registers and most are taken up with parameters. The routines do leave the dedicated cockpit index registers (see Section 5.1.1) intact.

In the 214A system, calling of system subroutines is less consistent. The stack convention of the PDP11 is rarely used. Generally, input parameters and returned values are passed in registers, e.g., from the 214A Visual System (diagram A335):

```
MOV   TEMP+2, R0
MOV   VNRPTC,R1        inputs
MOV   VNRALT,R2
JSR   PC,Z$LFI2        call (note call is direct, unlike UPT)
MOV   R2.VBDELN        output
```

One executive subroutine is called with parameters passed in the temporary storage area (described in Section 5.1.2) and with one parameter a global variable. For example, also from the 214A Visual System:

```
MOV   #1,VALTTB        set datapool item to table #
MOV   #VFC1T,TEMP      value table pointer (focus)
MOV   #VTL1T,TEMP+2    other value table pointer (tilt)
MOV   #VNRMP1,R3       breakpoint table pointer (in register)
JSR   Z$VNOR           normalize for LFI
MOV   R0,VNRPTC        store result
```

Apparently the routine Z$VNOR looks for parameters in the appropriate temporary and datapool locations.

5.2.9.3.3 Internal Subroutine Calling

There is considerable variation in the methods used to invoke internal subroutines. Techniques appear to be selected based on individual programmer preference. Most such routines are called directly. Various parameter passing methods are used. Sometimes parameters are passed in registers and sometimes in local storage. Sometimes the value is passed and sometimes its address (even in cases when it is not a table or array). Occasionally the address where the output is to be stored is passed to the routine, while in other cases the routine returns the output in a register.

Most internal subroutines do not save and restore registers. A few of the subroutines within the monitor (background dispatcher, I/O request handler) must be reentrant and these do save and restore registers, using a local stack.

Some internal routines have multiple entry points. For example, a display program (Box 4 of A35) has two entry points (indicating start/stop refresh). This could be implemented differently, e.g., with a flag as a parameter. An HOL multiple entry point capability seems unnecessary and undesirable.

The flow of control between monitor routines is handled in a variety of ways. Some routines have multiple entry points while others use instruction modification to alter control flow. For example, from the UPT monitor:

- One group of initialization routines (Box 1 of diagram A31) are to be called only on the initial start and not on restarts, when other initialization is repeated. The calling instructions are changed to no-ops after the routines complete.

- A few interrupt handlers must (conditionally, not always) return to the foreground dispatcher rather than to the application program which was interrupted. They do this by replacing the return address, saved in the first location of the interrupt handler at the call, by the desired return address in the dispatcher.

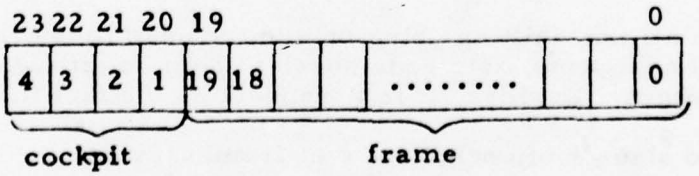
- The Aydin interrupt handler (as one example) is entered at the top as a result of an interrupt and at an internal entry point for the first Aydin request. When this internal entry point is used, the calling routine inserts the desired return address in the first location of the interrupt handler, and then makes a direct transfer to the entry point.

The first and third cases here could certainly be handled other ways, either using flags or a different grouping of functions. The second case is a little more complicated because a subroutine is used. (Actually an interrupt handler is used, but interrupts here cause execution of one of a table of subroutine calls.) The subroutine must be able to return either to the routine which invoked it or to the routine which invoked that routine (the dispatcher). Other interrupt handlers do not return at all to the routine which was interrupted (e.g., for an interrupt caused by the key used to stop TTY output). This is related to the problem of describing interrupt handling in an HOL (see Section 5.7).

5.3 Aggregate Data Types

5.3.1 Set Data Type

As mentioned in Section 5.2.4, some uses of bit string data in the simulators studied correspond more logically to an HOL set type. An example is the use of bit strings in the monitor for frame and cockpit masks (see diagram A312). The frame mask is a string of 20 bits, one of which is on to indicate which of the 20 frames is active. The cockpit mask is a string of four bits, one representing each cockpit. A bit in the cockpit mask is on if the cockpit is in use during the particular simulation run. The cockpit mask is constructed at initialization time, based on operator input in response to the question 'COCKPIT ENABLES?', and remains constant throughout the run. The frame mask is updated each frame by shifting. When the 1 bit is shifted out of bit 19, it is put in bit 0. These masks are used in determining when to execute a task from the foreground task table. Each task entry includes a 24-bit (1-word) cockpit/frame mask of the form:



The cockpit mask includes a 1 bit for each cockpit to which the task applies. Some programs must be called for each cockpit, while others are non-cockpit dependent and need be called only once per frame. The visual programs are called for only two of the cockpits. The frame mask here indicates during which frames the task is to be executed. The simulation programs will execute at 20, 10, 5, 2, or 1 frame/cycle. Some training programs do not require equal spacing and use rates other than these. The sequencing through the foreground task table for a given frame is essentially:

```
FOR COCKPIT = 1 TO 4 "SEQUENCE THROUGH COCKPITS"
     BEGIN
     IF COCKPIT ENABLED
          BEGIN
FOR TASK = 1 TO N "SEQUENCE THROUGH TASKS"
     BEGIN
     IF TASK THIS COCKPIT AND FRAME "MASK"
          CALL TASK
     END
END
END
```

This is actually somewhat more complicated, because cockpits 3 and 4 are 1/2 cycle out of phase with cockpits 1 and 2, so that their frame 10 is at the same time as cockpits 1 and 2 frame 0. This is required to prevent buffer use conflicts during record/playback (Box 1 of diagram A35). Thus, in the above loop, the frame mask (the one indicating which frame is currently active) is shifted 10 bit positions after cockpit 2 processing.

With all this shifting going on and the need to check for a shift out of bit 19 each time, this code doesn't seem so efficient that an HOL need replicate it. Besides, it is dependent on the fact that

$$\text{word size} \geq \#\text{ of cockpits} + \#\text{ of frames/cycle}$$

An HOL implementation could use a set data type to represent the cockpit and frame indicators in each task table entry and to represent the set of enabled cockpits. The active cockpit and frame would be loop indices, which would be tested for inclusion in the sets of cockpit and frame indicators for each task.

In the UPT training system, a demo request mask is used in conjunction with the frame masks. (See demo/record/playback, Box 1 of diagram A35.) The demo request mask contains 15 bits, one of which is set to indicate the demo requested by the instructor. A corresponding 15-bit value indicates which of the demos are available in the file. A set data type could also be used for this.

Bit strings are also used as parameters to the background dispatcher and to the intercomputer communications routine. The parameter to each routine is a string of bits in which each bit position represents one of the functions requested. The bit is set to 1 for each requested function. The sign bit is always 0, and the bits immediately to its left represent the functions in order of decreasing priority (see Figure 5-5). The routine must determine the leftmost 1 bit which is set. This is implemented by a floating point normalize instruction, which shifts the rightmost 23 bits left until bit positions 0 and 1 differ (i.e., bit 22 = 1) and returns the shift count. The parameter word could certainly be implemented as a set in an HOL, and priority of functions could be represented if this set is a power set of an ordered enumeration type. It isn't clear, though, how the operation "find the highest element" could be represented so that it could be expected to compile to a floating point normalize, or even a "find the leftmost 1 bit" instruction.

Another use of bit string data occurs in malfunction simulation (Box 3 of diagram A35). Expressions from the malfunction data set are evaluated, and when one is found to be true, the indicated malfunction is turned on by setting one of 96 bits in a packed 4-word array of bits. These bits might be specified as a set data type.

5.3.2 File Data Type

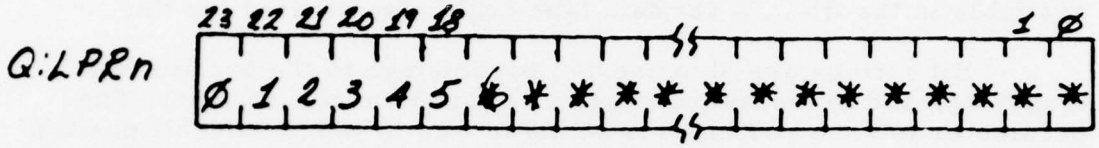
This section describes the disk files used in the simulators studied. Other I/O issues, as well as the actual disk I/O processing, are discussed in Section 5.6.

The simulator monitor provides disk I/O capability, supporting both direct and indexed files. Figure 5-6 illustrates the indexed file structure. The functions provided are direct read and write, indexed read (writes must be made by treating the file as a direct access file), and extended index read (in which two contiguous subfiles of an indexed file may be read into different parts of memory in one request).

Disk files created and used during simulation include:

demo recording file (Box 1 of diagram A35)

track history file (Box 6 of diagram A35)



INTER-COMPUTER (LO-PIG) REQUEST REGISTER FORMAT

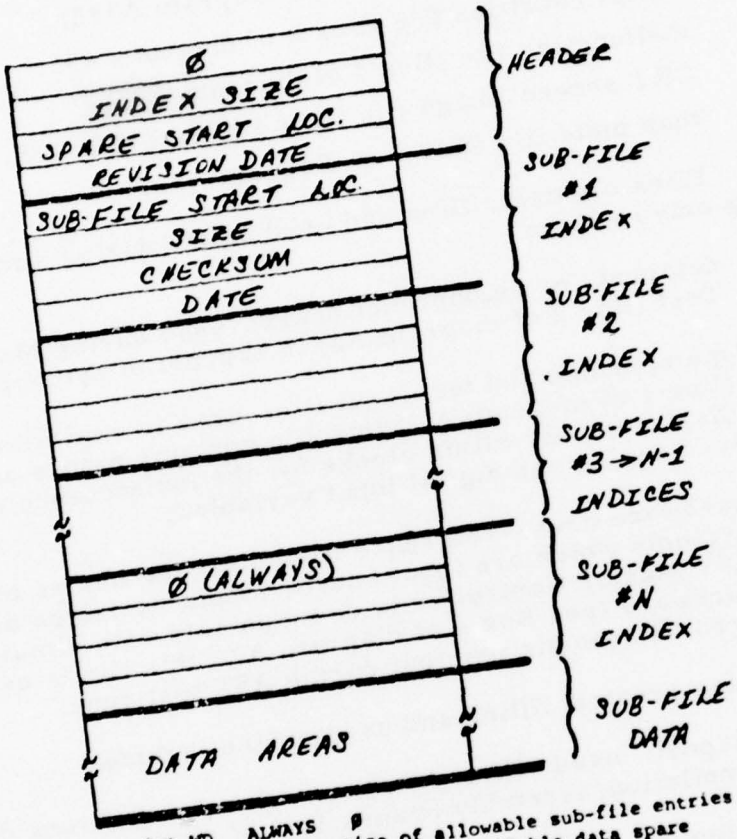
n: N FOR NEXT CPU
P FOR PREVIOUS CPU

INTER-COMPUTER COMMUNICATIONS ASSIGNED REQUEST LEVELS

0	MUST BE UNASSIGNED (ALWAYS = 0)
1	HALT CPU
2	ARM/TRIGGER DISC INTERRUPT[1]
3	DISC XFER REQUEST DATA TRANSMITTAL
4	TTY/CRT OUTPUT INITIATE[2]
5	RELEASE MONITOR WAIT STATE
6	EOM WAIT RELEASE[3]
*	UNUSED LEVELS (INVALID)

1. CPU2 ONLY (CPU1,3 INVALID)
2. CPU2,3 ONLY (CPU 1 INVALID)
3. CPU1 ONLY (CPU2,3 INVALID)

Figure 5-5. Intercomputer Communications Parameter Word



HEADER:

1st WD	ALWAYS Ø
2nd WD	Maximum size of allowable sub-file entries
3rd WD	Location of first available data spare
4th WD	Latest date of file revision

INDEX: Sub-File Indicies are order dependent (i.e., 1st Index = Sub-File #1, 2nd Index = Sub-File #2, etc.)

1st WD	Start location of sub-file data (Ø = NO ENTRY)
2nd WD	Sub-file size
3rd WD	Sub-file checksum
4th WD	Latest date of sub-file revision

DATA: Sub-file Data areas are non-order dependent thus allowing sub-file additions in any order without having to restructure file.

Figure 5-6. Indexed File Structure

Files created offline and used online during simulation include:

radio station file (Box 4 of diagram A332)

initial condition file (Box 2 of diagram A35)

malfunction file (Box 3 of diagram A35)

CRT screen image file (Box 4 of diagram A35)

map plate file (Box 5 of diagram A35)

Files created offline and used online during simulator testing only include:

datapool, or symbol dictionary: used during Math Model Test (Box 2 of diagram A2) in support of symbolic debugging;

maintenance and test input file: used in executive I/O test (Box 1 of diagram A2); this file contains names and displacements within blocks for the maintenance and test load analog and digital input variables;

hardware cross-reference file: lists the names of the symbols which are input and/or output variables used by AST Master Controller to communicate with simulator hardware (see Box 4 of diagram A3); this file is used for error diagnostic printout during AST test runs;

Files created offline and used offline include:

datapool: used offline to obtain symbol definitions during compilation/assembly (Boxes 1, 3 of diagram A1)

system cross-reference file: maintained throughout system development (diagram A1)

various data files (airfield data, etc.) used by the map plate compiler (Box 5 of diagram A35)

5.3.3 Array Data Type

Throughout the simulators studied, readability could be much enhanced by grouping various related items into data aggregates. In general, most of these groupings would correspond to tables or structures, as described in Section 5.3.4. One exception is the vector and matrix data which is heavily used in the Aerodynamics (diagram A3312), Tactics (diagram A334), and Visual (diagram A335) areas. These vectors represent flight data (e.g., the accelerations and velocities computed in the equations of motion program -- Box 5 of diagram A3312), and contain three elements corresponding to X, Y, and Z coordinates.

A simulation HOL should provide some support for vector and matrix operations. One program, the 214A Visual Position and Velocity program (Box 2 of diagram A3351), performs matrix and vector operations almost exclusively. It could be rewritten in about 20 lines in a language supporting these operations (instead of its current 409 lines of assembly language). Such operations would have to allow different scalings of the operands as well as operations in which one operand is single-word and the other is double-word (operands are of different precisions). Operations used are:

addition and subtraction of vectors

multiplication of a vector by a scalar

multiplication of a vector by a matrix

cross product of vectors

dot product of vectors

In one example from the visual programs, a vector is multiplied by a rotation matrix of the form:

$$\begin{bmatrix} \text{COS } \Psi & -\text{SIN } \Psi & 0 \\ -\text{SIN } \Psi & -\text{COS } \Psi & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

The code used takes advantage of the 0's and -1 in the matrix and does not perform the full multiplication. In a high level language, the programmer could obtain this result by writing the multiplication out explicitly on an element-by-element basis. Possibly, though, the multiplication provided might include checks for zero elements.

Efficient compilation of matrix operations is important. Some possible difficulties which might be encountered were observed in the way the 214A visual programs are currently implemented. For example, the programs frequently fail to use loops when performing the same operation on the three elements of a vector. This can result in a considerable increase in program size with little saving in speed. In one instance, a 21-word operation is repeated three times, rather than using a loop. If a loop setup requires 5 words, a loop implementation would require 26 words rather than the 63 used by the actual implementation. On the other hand, there is one instance where a loop is used to accomplish the equivalent of three MOV instructions. The code used is:

```
MOV   #3,R3        "counter"
CLR   R4           "two different offsets required"
CLR   R5
```

```
30$: MOV   VXPFB1(R4), VXPFBT(R5)        "move"
     ADD   #2, R5         "update indices"
     ADD   #4, R4
     SOB   R3, 30$        "subtract one and branch"
```

A better implementation is simply:

```
MOV   VXPFB1, VXPFBT
MOV   VYPFB1, VYPFBT
MOV   VZPFB1, VZPFBT
```

An HOL implementation of operations on vectors should be sophisticated enough to use loops when more efficient, as in the first example, and repeated code when this is preferable, as in the second.

Another possibility of matrix or array use occurs in the computation of DC bus load in the Electrical System (Box 2 of diagram A33132). Here the bus load is initialized to 15 and then various device flags are tested; appropriate values are added to the bus load for each device which is on, i.e.

```
LLDBUS = 15
IF (GPUPRP) LLDBUS = LLDBUS + 3
IF (GPUBST) LLDBUS = LLDBUS + 17
.
.
```

There are quite a number of tests, leading one to look for a simpler representation. Possibilities are:

a. The flags could be considered a 1 x n matrix of 1's and 0's, and the loads an n x 1 matrix of values. Matrix multiplication will then give the sum of selected loads, e.g.,

```
LLDBUS = 15 + [FLAGS] [LOADS]
```

b. If the flags are an array of n Booleans, and the loads an array of n values, a loop can be used, e.g.,

```
LLDBUS = 15;
FOR I = 1 TO N;
    BEGIN
IF FLAG(I) THEN LLDBUS = LLDBUS + LOAD(I);
END
```

If desired, an enumeration type describing the conditions represented by the flags (GPUPRP, GPUBST, etc.) could be used to index the arrays. A table could be used to combine the two arrays, e.g.,

```
TABLE DCLOAD(1:N)
    BEGIN
    ITEM FLAG Boolean
    ITEM LOAD Integer
    END
```

The second method would probably result in more efficient generated code, and its intent is clearer.

5.3.4 Structure Data Type

As mentioned previously, much simulator data could logically be organized into structures (or tables) which group related items together. Several examples have already been given, for example in Sections 2.3.2 and 2.5. Most structures would replace many data items which now all have individual names, with indexable structures, thus using fewer names. The following sections discuss structure organizations used, structure operations required, and examples of major simulator structures.

5.3.4.1 Structure Organization

5.3.4.1.1 Programmer-Specified Allocation

Allocation of components within a structure may be accomplished automatically by a compiler or may be specified explicitly by the programmer. Programmer specified packing seems necessary in cases where the table describes data for an I/O interface. Examples of this are the master disk directory, illustrated in Figure 5-7, the construction of I/O command words by the I/O routines, and the interpretation of device status words. (Perhaps a status word could be represented as a set (see Section 5.3.1) rather than as a programmer-specified table, since in general each bit represents a discrete condition.) One example of I/O data which might be difficult to handle in this way, however, occurs in the 214A visual system gantry feedback processing (Box 1 of diagram A3352), which has as an input the table:

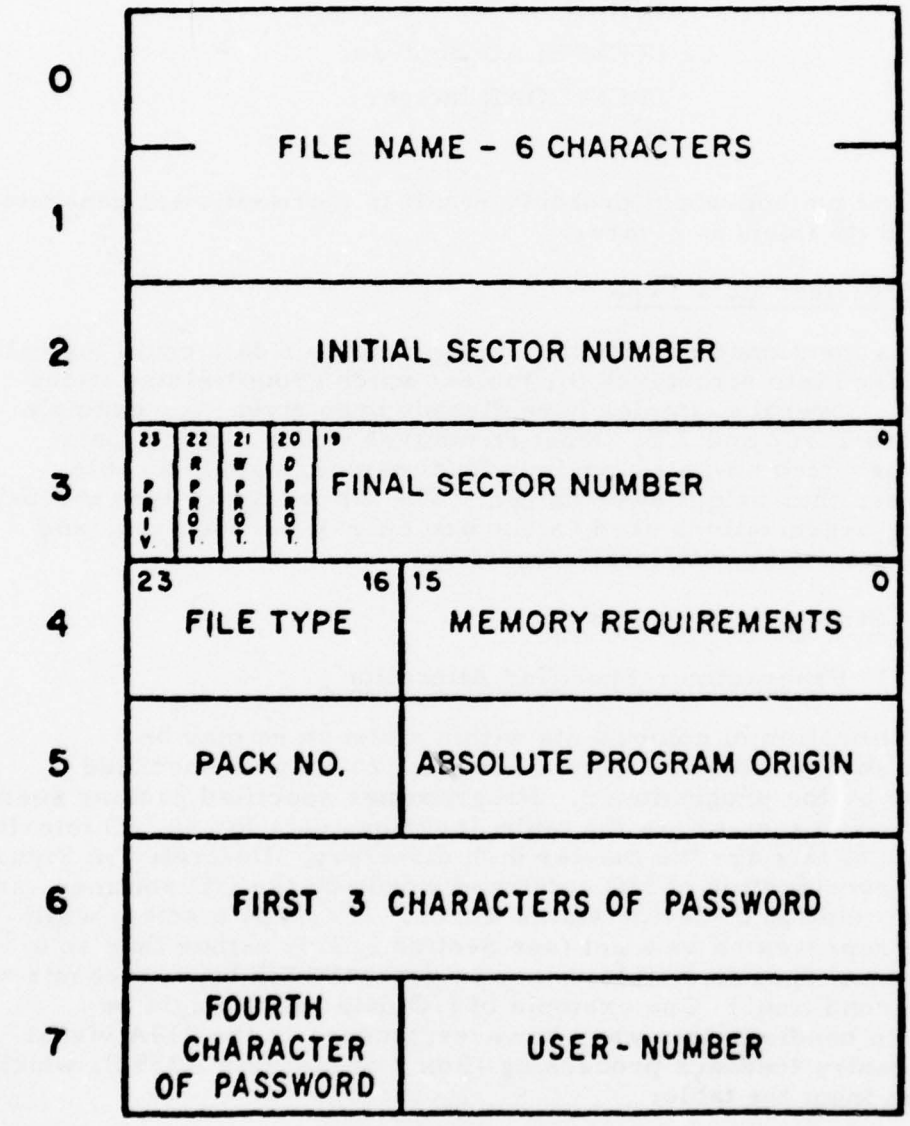
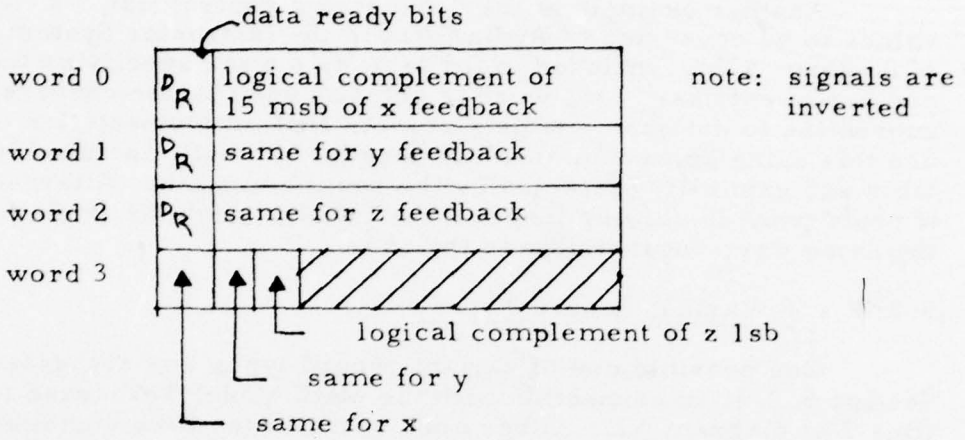


Figure 5-7. Master Disk Directory Entry



It would be difficult for a high-level language to support direct extraction of the 16-bit feedback values. (However, perhaps the data organization could be changed as discussed in Section 5.2.5.)

Another use of programmer-specified tables is described in connection with the Math Model Test (Box 2 of diagram A2) trace feature (Section 5.2.4).

5.3.4.1.2 Serial vs. Parallel Organization

In tables constructed to match hardware inputs or outputs, as discussed above, the choice of serial or parallel organization should probably be left to the programmer. For example, the UPT system's AST Linkage requires parallel organization. Most tables which would be constructed from what are currently separate data items could equally well be represented logically by either organization and both are used in current implementations. The major reason to select one method over the other is to facilitate data accessing as it is done in the particular table. Since simulators have significant efficiency requirements, this option should probably be made available to programmers.

5.3.4.1.3 Variable Length Tables

Little use of tables with dynamically varying length was observed in the programs studied. An example of such tables, however, occurs with the foreground task table (Box 3 of diagram A312) illustrated in Section 5.2.8. This table is preceded in core by its number of entries. This number is reset to zero if the operator, at initialization, specified a "maintenance and test" load (Box 1 of diagram A2). It could potentially be changed dynamically to any value and then be used by the foreground dispatcher. However, it should not be because the task table is preset and not changed during execution. There is no reason this special case couldn't simply be implemented with a flag indicating "maintenance and test."

Another example is the "conversion control list," a list of values to be converted to Aydin form in the Instructor System (Box 4 of diagram A35). Included in the table is a word specifying the number of entries. This word is actually used by the conversion subroutine to determine table size. An HOL implementation could use this same approach, in which length is actually included in the table and explicitly extracted by the routine using it. Alternatively, it could provide varying length tables, which might be implemented the same way, but invisibly to the user.

5.3.4.1.4 Variant Record Types

One possible use of variant record types was discussed in Section 5.2.4, in connection with the Math Model Test trace formatting (Box 2 of diagram A2). Other uses occur in the files containing surface radio station data (Box 4 of diagram A332) and radar emitter data (Box 1 of diagram A334). These files contain different record types for each radio type or emitter type.

5.3.4.1.5 Non-Distinct Component Names

There are many instances in the programs studied where several tables have the same organization and kinds of components. Some convenient notation for this would be useful, e.g., from the 214A visual system visibility effects program (Box 4 of diagram A3354):

```
TABLE UICDATA 7; "old instructor inputs"
    BEGIN
    ITEM CCLG _____; "cloud ceiling"
    ITEM CCTH _____; "cloud thickness"
    ITEM CDDN _____; "day/dusk/night indicator"
    ITEM CMIN _____; "minimum lighting"
    ITEM CRND _____; "random lighting"
    ITEM CSTG _____; "stagefield lighting"
    ITEM CVIS _____; "visibility"
    END
TABLE VICDATA LIKE UICDATA; "new instructor inputs"
```

Of course, some means of distinguishing between components of the two tables is then required. There are instances where one such table is assigned to a corresponding one. A convenient notation for this, e.g.,

```
ENTRY(UICDATA,0) = ENTRY(VICDATA,0)
```

or simply

```
UICDATA = VICDATA
```

would be useful.

5.3.4.2 Operations on Structures

5.3.4.2.1 Table Assignment

As mentioned above, assignment of one table to another of corresponding layout is sometimes required. This occurs, for example, in assignment of current values of variables to previous-pass values, as in the example above from the visibility effects program (Box 4 of diagram A3354). In another example, from the Navigation Environment area (Box 2 of diagram A332), the sequence:

```
NUE = FUE
NVE = FVE
NSPSHD = FSPSI
NCPSHD = FCPSI
NSINNP = FSTHET
NCOSNP = FCTHET
NSINNR = FSPHI
NCOSNR = FCPHI
NGALT = FHGED
NTAS = FVPKTS
NROT = FRA
NSLEW = SNLSLW
NRESET = UILRF
```

represents assignment of a set of "nav freeze" values to a corresponding set of navigation variables, and could be written as one table assignment (with greater clarity and less chance of error). Performing operations on entire corresponding tables in one statement, e.g.,

```
TABLE1 = TABLE2 + TABLE3
```

indicating addition of all components, would also be useful.

5.3.4.2.2 Substructure Selection

Many simulator operations which might use table data for clarity would benefit from the ability to perform operations on substructures. For example, in the 214A visual system gantry feedback program (Box 1 of diagram A3352) much of the data could be organized into structures with 3 entries indexable by X, Y, and Z.

For example,

```
TABLE VGNTRY (X:Z) 5; "gantry data"
    BEGIN
    ITEM VVGMF single fixed point; "velocity"
    ITEM VVGMI double fixed point; "position"
    ITEM VVPFB double fixed point; "position feedback"
    END
```

could describe much more clearly a set of data which currently uses 15 different identifiers in the datapool (two names are used for the two halves of the double fixed point values). With such data structures, it would be necessary that the X, Y, and Z values of a given component could be treated as a vector in whatever vector operations are provided, e.g., in the above example, the assignment

$$\overline{VVPFB} = \overline{VVGMI} * 2.5$$

should be possible.

In another example, from Electrical Systems (diagram A33132), comparable operations are frequently performed with 'left' and 'right' values. Grouping the values into a table or array with two entries indexed by "LEFT" and "RIGHT" could allow a single operation to be used. For example, the code used to set left and right generator relay indicators is:

```
LRYGNL = ((EARPML.GT. 40.).OR. LRYGNL.AND.
          (EARPML.GT. 38.)).AND. LSGLON.AND..NOT.
          (EBRYGL.OR. UMLLGF)
LRYGNR = ((EARPMR.GT. 40.).OR. LRYGNR.AND.
          (EARPMR.GT. 38.)).AND. LSGRON.AND..NOT.
          (EBRYGR.OR. UMLRGF)
```

Combining these in a single operation, e.g.,

```
LRYGN = ((EARPM.GT. 40.).OR. LRYGN.AND.
        (EARPM.GT. 38.)).AND. LSGON.AND..NOT.
        (EBRYG.OR. UMLGF)
```

could improve understandability and decrease the possibility of typographical error.

Another instance is the assignment in the Communications area (Box 3 of diagram A332) documented by:

$$NO11CL = (MPX + NO11IS + NO21IS) \cdot ISPRI$$

2	=	2	2
3	=	3	3
4	=	4	4

This operation, which uses data illustrated in Figure 5-3, uses four assignments, one for each cockpit. If the data is in a doubly-indexed structure (by operator and cockpit) as proposed in Section 5.2.5, this assignment might be written:

$$NOCL(1,*) = (MPX + NOIS(1,*) + NOIS(2,*)) \cdot ISPRI$$

5.3.4.3 LFI Structures

A type of data structure required throughout a simulator is that used by Linear Function Interpolation (see Section 5.2.2.2.2). The data tables used to represent LFIs consist of tables of breakpoints and tables of values which correspond to the breakpoints. The programs studied employ both single and double variable LFIs. Three-variable routines are mentioned but not used. The breakpoint table(s) and the value table are both defined in the program as lists of constants. However, they appear in separate parts of the program (separately compiled modules) and are used separately.

Typically, several different LFI functions might have a variable in common, and these variables might have breakpoint lists in common. For example, the UPT aerodynamics LFIs (see diagram A3312):

$F100(\alpha, M)$

$F805(\alpha, \delta_{FW})$

$F807(\alpha)$

all have α as an independent variable. In F805 and F807 the breakpoint list for α is the same, while F100 has a different α breakpoint list.

When the value of an LFI variable is first determined (e.g., α above), an "LFI search" routine is called to search for its position in any associated breakpoint lists. The resulting value, the interpolant, is used in later processing as a parameter to the "LFI value" routine. Thus one routine uses the breakpoint list and one uses the value list. For example, in the UPT aerodynamics system, most breakpoint lookup occurs in the Equations of Motion module (Box 5 of diagram A3312), while value computation using these breakpoints occurs throughout the system.

A double variable LFI is allocated as two breakpoint lists, X and Y, and one value list, F(X, Y). The value list is allocated as a FORTRAN two-dimensional array would be, with X increasing faster. In the UPT simulator, the breakpoints are two-word floating point values but the values are one-word fixed point. The value lookup subroutine, however, converts the fixed point value to floating point before returning it. The reason for this is economy of space, and one-word floating point is not available on the UPT computer. The precision allowed by the single word is adequate for the values. In the 214A simulator, which uses only fixed point, the value tables are all single precision; the breakpoint tables are sometimes single and sometimes double precision. The altitude breakpoints, for example, are double precision.

A more readable presentation of LFIs would dictate that the breakpoint and value lists be specified together (and thus presumably be allocated together). There appears to be no logical reason why this could not be done as long as the definition of the structure was made available to both routines. A problem in supporting LFI representation in an HOL is that each LFI does not have a unique breakpoint list, but rather several LFIs share lists. (There are, of course, unique value lists for each.) It would be wasteful to repeat the breakpoint lists, and repeating the lookup process would be intolerably inefficient. Perhaps the best approach would be to define the lists separately in a global data base and simply attempt a more readable layout which makes associations clearer, e.g., value lists sharing a common breakpoint list could be grouped together under the breakpoint list definition; this would not work for double variable LFIs, however.

One simulation HOL study [Goldiez, 1976] gives statistics on relative speed of assembly language and FORTRAN LFI routines. The FORTRAN programs took almost three times as long. This is clearly unacceptable. The author's comments suggest that the FORTRAN code generated was very inefficient because it recomputed array subscripts excessively.

5.3.4.4 Modelboard Contour Map

The most complex data structure observed in the programs studied is the 214A Visual System modelboard contour map (Box 1 of diagram A3353). This table gives a maximum elevation indicator (a 3-bit value) for every 4-inch square on the modelboard. The actual elevation corresponding to the 3-bit value is found in an 8-element array, to which the 3-bit value is an index. Because many 4-inch squares will have the same elevation value, each does not have a distinct 3-bit value associated with it. The squares are grouped into larger blocks of 5 by 6 squares (20 x 24 inches). The modelboard contains 468 such blocks. Only those 20 x 24 blocks which are distinct

have an associated bit map. A 468-byte vector maps each block into the associated bit map. The program allows up to 256 distinct blocks. The bit map consists of 6 words, each containing 5 3-bit values, thereby covering the 30 4-inch squares in the block. The best data structure arrangement for accessing this information would be something like:

```
ARRAY BLKPTR (0:38, 0:11) BYTE; "indices into BITMAP
                                 for each block"

TABLE BITMAP (0:255) 6;
    BEGIN
    ARRAY BITARY(0:4, 0:5) bit 3 packed;
    END
```

The double indexing (for X and Y coordinates) is desirable to support calculation of the correct table value - otherwise the program would have to compute a single value from the X and Y values. The bit map could be represented without too much loss of clarity as a three-dimensional array having dimensions (0:255, 0:4, 0:5). It is necessary that the bit values be packed. The particular size chosen for the blocks, leading to the 5 x 6 grouping of bits, is clearly based on the word size of the machine (i.e., 16/3 = 5). Other numbers could certainly be used, but the decision of what size of block will be optimal must be based on some knowledge of the particular model-board -- it is clearly not random.

5.3.5 Union Data Type (or Overlays)

In general, any necessary overlay capability required in simulators can be logically provided through the use of structures with variant record types, discussed in Section 5.3.4.1.4. There are instances where overlays might be utilized to obtain a capability not explicitly provided by the language, for example accessing memory locations by address, as described in Section 5.2.7.2. These cases should really be handled in a manner which makes the intent more understandable.

5.4 Control Structures

There are many instances in the simulators studied in which program understandability could be greatly improved through the use of modern HOL control structures. In some cases, the program documentation reflects an awareness by the programmer that such control structures are needed, but frequently even the documentation does not take advantage of the enhanced readability that would be provided. The following subsections present examples of potential uses of various control structures in the simulators studied.

5.4.1 Conditional Control Structures

5.4.1.1 IF_THEN_ELSE Control Structures

Simulation programs contain complex conditional expressions controlling the assignment of values to variables. Any simulation language must provide a readable way of writing such assignments. The examples in this section illustrate the range of conditional assignment control that must be supported by a suitable HOL.

The programs studied contain numerous examples of complex conditional assignments to variables. These are frequently expressed in the documentation by multiplying a logical variable or expression by the various operands, e.g., from Flight Controls (Box 1 of diagram A331):

$$X = (15.08 * FPE * \overline{WPLAY}) + (15.08 * FERPB * WPLAY) - 4.0$$

Expressed in FORTRAN notation, this might become:

```
X = 15.08 * FPE - 4.0
IF (WPLAY) X = 15.08 * FERPB - 4.0
```

A better representation, which more closely resembles the documentation, might be the ALGOL-like:

```
X = 15.08 * (IF WPLAY THEN FERPB ELSE FPE) - 4.0
```

This form might also compile more efficiently.

Another conditional expression example is (also from Flight Controls):

```
      IF (.NOT. (FELTRU .OR. FELTRD)) GO TO 04
      TEMP00 = 2.25 * QTM
      IF (FELTRU) TEMP00 = -TEMP00
      FTRIME = AMIN1(AMAX1(FTRIME * TEMP00, -8.0), 25.0)
   04 .....
```

where FELTRU and FELTRD (which are flags indicating "nose up or down") may both be false, but cannot both be true. This might be expressed using IF–THEN–ELSE, as

```
IF (FELTRU OR FELTRD) THEN
    BEGIN
    IF FELTRD THEN TEMP00 = 2.25 * QTM
              ELSE TEMP00 = -2.25 * QTM
    FTRIME = AMIN1(AMAX1(FTRIME * TEMP00, -8.0), 25.0)
    END
```

or, with a conditional expression, as

```
IF (FELTRU OR FELTRD) THEN FTRIME =
    AMIN1(AMAX1(FTRIME * QTM * (IF FELTRD THEN
    2.25 ELSE -2.25), -8.0), 25.0)
```

The 214A Visual System (diagram A335) has many expressions similar to:

```
VR = .1744 * UP - .1744 * DOWN
```

or, alternatively:

```
VR = .1744 (UP - DOWN)
```

In these cases, UP and DOWN are single bits in the test box input. Both may be off or exactly one may be on.

Many of the conditional assignments are documented in pseudo-FORTRAN, which leads to inconsistent and error-prone statements. For example, from the Communications area (Box 1 of diagram A332):

```
VHAUL = ((VHVCL/2) + .499)) IF LDOPP + (0.0) IF
    (LDOPP + RAINT + VTUNE)
```

Here the test of RAINT and VTUNE serves no purpose, but the flowchart indicates that the intent is:

```
VHAUL = IF (NOT LDOPP OR RAINT OR VTUNE) THEN 0.0
        ELSE VHVCL/2 + .499
```

Some very complex conditional assignments occur in the Navigation Radios area (diagram A3323). Some assignments are so complex that both the documentation and the pseudo-FORTRAN are frequently incomprehensible and often clearly incorrect. An example is the description of the setting of variable VGRE:

VGRE = (NAVR-VGRI) IF [(VGPWR .OR. .NOT. ERCOT) .AND.
.NOT. FSTER]

VGRE = (-VGRI) IF [(VGPWR .OR. .NOT. ERCOT) .AND.
FSTER

VGRE = (85-VGRI) IF [(ERCOT .AND. (VGPWR .OR. (ERTIM ≤
260))) .OR. .NOT. VGPWR .AND. (ERTIM ≤ 260)]

VGRE = ($C\lambda_{AC}$*K*$C\psi_{HDG}$) IF [(ERCOT .AND. (VGPWR .OR.
(ERTIM > 260))) .OR. .NOT. VGPWR .AND.
(ERTIM > 260)]

In this example, the conditions specified for the four different assignments are not mutually exclusive. For example, the set of conditions

VGPWR, ERCOT, $\overline{\text{FSTER}}$, (ERTIM > 260)

satisfies the first, third, and fourth. Examination of the flowcharts suggests that the intent is simply:

```
IF VGPWR OR NOT ERCOT
    THEN IF FSTER THEN VGRE = -VGRI;
                  ELSE VGRE = NAVR - VGRI;
    ELSE IF ERTIM ≤ 260 THEN VGRE = 85 - VGRI;
                        ELSE VGRE = CλAC*K*CψHDG;
```

One problem that also occurs in the documentation of these conditional assignments is that the same logical expression appears in numerous equations, instead of being tested once preceding them, i.e.:

A = B IF COND C IF NOT COND

D = E IF COND F IF NOT COND

G = H IF COND I IF NOT COND

instead of:

```
IF COND THEN
     BEGIN
     A = B;
     D = E;
     G = H;
     END
ELSE
     BEGIN
     A = C;
     D = F;
     G = I;
     END
```

If the programmers were actually using FORTRAN rather than assembly language, we assume they would not implement this the way it is documented. It is not only less clear, but it is in all likelihood much less efficient.

Another instance in which the lack of IF–THEN–ELSE control structures has a severe negative impact on readability occurs in the following sequence, used to set item LNBUS (from Electrical Systems - diagram A33132):

```
      IF (.NOT. LBUSDC) GO TO 24
      IF (LPWEXT .OR. UQLBSE) GO TO 17
      IF (LBAT .AND. LSWBAT) GO TO 19
      LNBUS = 0.1
      GO TO 25
19    IF (UMLBTY) GO TO 20
      LNBUS = 0.9
      GO TO 22
20    LNBUS = 0.8
22    IF (LRYGNL .OR. LRYGNR) LNBUS = LNBUS + 0.1
      GO TO 25
17    LNBUS = 1.0
      GO TO 25
```

```
24   LNBUS = 0.0
25   IF (EBRYGL .AND. EBRYGR) LNBUS = 0.5 * LNBUS
```

Evidence that this is error-prone may be found in the fact that the logic does not match that in the flowchart for the operation. A more readable representation is:

```
IF NOT LBUSDC THEN LNBUS = 0.0;
     ELSE IF LPWEXT OR UQLBSE THEN LNBUS = 1.0;
       ELSE IF NOT (LBAT AND LSWBAT) THEN LNBUS = 0.1;
          ELSE BEGIN
             IF UMBLBTY THEN LNBUS = 0.8;
                        ELSE LNBUS = 0.9
             IF LRYGNL OR LRYGNR THEN LNBUS
                = LNBUS +0.1;
             END
IF EBRYGL AND EBRYGR THEN LNBUS = 0.5 * LNBUS;
```

Alternatively, a single assignment of a logical expression to LNBUS could be used, e.g.,:

```
LNBUS = (IF EBRYGL AND EBRYGR THEN 0.5 ELSE 1.0)*
        (IF NOT LBUSDC THEN 0.0
           ELSE IF LPWEXT OR UQLBSE THEN 1.0
             ELSE IF NOT (LBAT AND LSWBAT) THEN 0.1
                ELSE ((IF UMLBTY THEN 0.8 ELSE 0.9) +
                (IF LRYGNL OR LRYGNR THEN 0.1
                ELSE 0.0)))
```

The previous example seems more readable, though this one makes it clearer that assignment to LNBUS is the intent.

Another approach to the description of conditional assignments is the decision table. For example, a decision table describing the preceding assignment is:

LBUSDC	0	1	1	1	1	1	1	1	1	1	1
LPWEXT	-	1	-	0	0	0	0	0	0	0	0
UQLBSE	-	-	1	0	0	0	0	0	0	0	0
LBAT	-	-	-	0	-	1	1	1	1	1	1
LSWBAT	-	-	-	-	0	1	1	1	1	1	1
UMLBTY	-	-	-	-	-	0	0	0	1	1	1
LRYGNL	-	-	-	-	-	0	1	-	0	1	-
LRYGNR	-	-	-	-	-	0	-	1	0	-	1
LNBUS	0.0	1.0	1.0	0.1	0.1	0.9	1.0	1.0	0.8	0.9	0.9

5.4.1.2 CASE Control Structures

Conditional processing corresponding to the CASE construct occurs primarily in the simulator support programs (monitor, debugging, etc.). Instances of this include:

- foreground task table processing, described in Section 5.2.8 (diagram A312)
- selection of function based on input parameter by inter-computer communications or by background dispatcher (see Section 5.3.1)
- transfer based on interrupt number in monitor interrupt handlers
- I/O device routine selection by IOC coordinator (currently only one device is connected to this, but the program allows for more) (see Section 5.6)
- I/O conversion processing based on symbol type in Math Model Test (Box 2 of diagram A2)
- selection of correct function based on function select knob in processing Remote Decimal Readout Unit (DRU) inputs

An interesting type of CASE-like conditional assignment occurs in the ground control display program (Box 6 of diagram A35) when selecting messages for the instructor. For example, a message describing how close the pilot is to the desired glideslope is selected as follows:

$-0.14^{\circ} \leq SGTANG \leq 0.14^{\circ}$	"ON GLIDE PATH"
$0.14^{\circ} < SGTANG < 0.42^{\circ}$	"SLIGHTLY ABOVE GLIDE PATH"
$-0.42^{\circ} < SGTANG < -0.14^{\circ}$	"SLIGHTLY BELOW GLIDE PATH"
$SGTANG \geq 0.42^{\circ}$	"WELL ABOVE GLIDE PATH"
$SGTANG \leq -0.42^{\circ}$	"WELL BELOW GLIDE PATH"

An HOL representation for this might be a CASE statement with ranges (rather than single values) for alternative selection.

A similar assignment occurs in the Aerodynamics area (diagram A3312), here expressed in the "multiplying by Booleans" notation:

```
X = ((.1155556 + .000154074 * (X-750.)) * (RC .LT. 750.)
    + (.1969444 + .0001085184 * (X-1500.)) * (RC .GE. 750.
      .AND. RC .LT. 1500.)
    + (.275556 + .0000786112 * (X-2500.)) * (RC .GE. 1500.
      .AND. RC .LT. 2500.)
    + (.370833 + .000635184 * (X-4000.)) * (RC .GE. 2500.
      .AND. RC .LT. 4000.)
    + (.4775 + .00005333335 * (X-6000.)) * (RC .GE. 4000.
      .AND. RC .LT. 6000.)
    + (.4775) * (RC .GE. 6000.) * (-1.0)*(FRCIND .LT. 0)
```

5.4.2 Multiprocessing Control

Multiprocessing is required in the simulators studied since all use more than one CPU. This section describes the overall flow of control in the UPT simulator in order to illustrate multiprocessing requirements.

The UPT system uses three CPUs, each of which has private memory. There is also common memory accessed by all three. The application programs are distributed among the three CPUs so as to provide the necessary speed of execution. A single application program (e.g., flight) uses only the one CPU to which it is assigned. Application programs operate in parallel on the different CPUs, but do not interact directly with one another. All interaction and

sequencing is controlled by the monitor. Necessary synchronization between the CPUs is provided by the monitor through the Equations of Motion (EOM) Syncing Function, described later in this section.

Some monitor routines exist in identical form in all three CPUs, while others exist in only one. Duplication of a routine allows more efficient processing (by eliminating a need for inter-CPU communication) and allows the routine to access private memory data. A single routine, on the other hand, allows a saving of core. In some cases, a single function (e.g., disk I/O) is performed partly by a single routine in a master CPU and partly by duplicated routines in the other two 'slave' CPUs. For example, this organization is used when only one CPU can communicate with a particular peripheral.

Monitor execution begins with system initialization, in which the individual CPUs periodically halt themselves and wait for restart by another CPU. Upon completion of initialization, the basic execution cycle is initiated by a Real Time Clock interrupt in CPU1. (Countdown was initiated by the initialization process.) This interrupt causes execution of the Master Timing Routine, which in turn interrupts CPUs 2 and 3, passing control to the Slave Timing Routines in these CPUs. (The different interrupt levels control the selection of the routine to be executed, through a vector of subroutine call instructions.)

All three timing routines initiate the foreground dispatchers (see Section 5.2.8) by interrupting their respective CPUs. The dispatcher calls each of the required simulation programs from its task table. Each program returns to the dispatcher when it completes. The dispatcher then calls the next required program. After the last simulation program completes, the spare time subroutine is called to compute the spare time for the cycle. Then the foreground dispatcher is exited and the CPU returns to a wait state until the next Real Time Clock interrupt occurs to restart the cycle. (Diagram A312 illustrates this sequence.)

Other processes, such as I/O and background processing, are initiated by interrupts (either hardware or software triggered) which occur asynchronously with the basic cycle. For example, I/O to the simulator hardware occurs twice per frame on countdown of the Interval Timer, while TTY output, if active, is triggered by the 120-Hz clock interrupt.

Communication between CPUs is performed via interrupts or via common memory. Data may be communicated through the common memory. Control flow is handled by interrupts into one CPU triggered by another CPU. For each CPU interface (6 in all), there is a word in common memory in which bits are set indicating, in priority order, the functions requested (see Figure 5-5). Thus one

CPU requests a function of another by setting a bit in the appropriate word and triggering the interrupt. The functions which may be requested are:

- halt - requested if a fatal error or power fail occurs in another CPU
- arm/trigger disk interrupt - used by the slave disk handler to indicate to the master that a slave disk transfer is complete
- disk transfer request - sent by the slave disk handler to the master when a disk request has been made in the slave; sent by the master to the slave when the master has completed setup for the transfer
- TTY/CRT output initiate - sent by IOC coordinator in CPU1 to a slave TTY/CRT driver to grant TTY/CRT output privilege
- release monitor wait state - sent to release the receiving CPU from a wait state; used only during initialization, where CPUs are coordinated via wait/release
- EOM wait freeze - used to release EOM sync wait, described below

Three of these functions are in support of the I/O structure of the simulator while three support other control coordination between CPUs. As 'halt' is used only for exceptions and 'release' only during initialization, only the 'EOM wait freeze' function is used during regular execution.

The EOM syncing function is required to keep the simulation programs running on the three CPUs properly synchronized. In particular, the relationship of the Equations of Motion (EOM) program to the flight programs must be kept constant, since EOM inputs are generated in flight. Similarly, the motion primary cues program must execute after the EOM program since it uses output from EOM. Figure 5-8 illustrates the desired sequencing. Note that the flight programs are in CPU3, while the EOM and motion primary cues (MOT_n) are in CPU1. The correct order of the EOM and MOT programs is assured by their position in the CPU1 task table. The EOM Sync programs are used to delay the EOM programs until the flight programs finish.

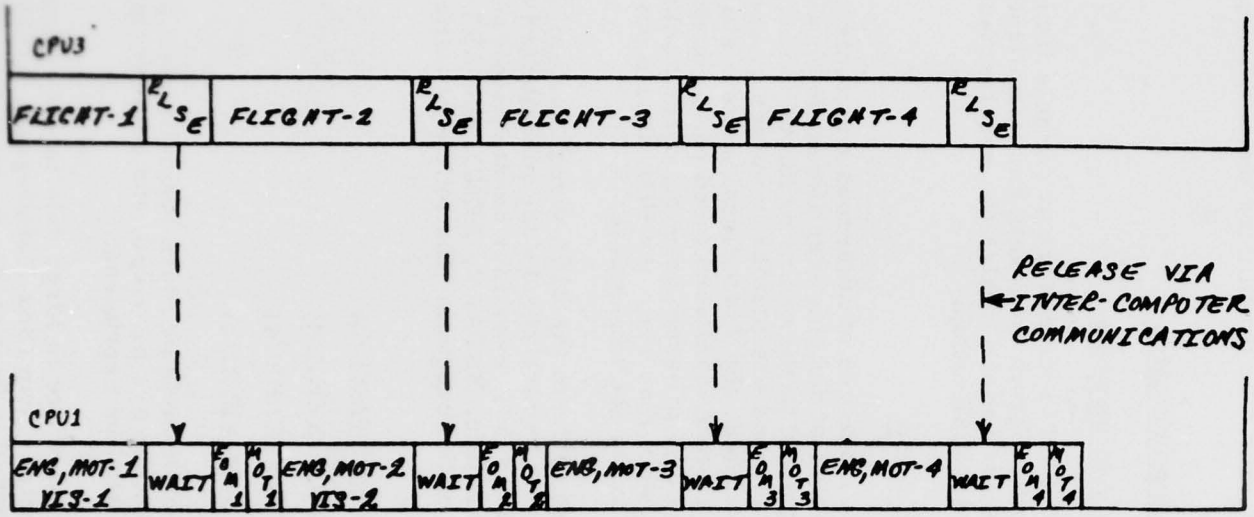


Figure 5-8. EOM Syncing Relationship of Flight Parameters to EOM, Motion Programs

The CPU3 sync release program immediately follows the flight programs in the CPU3 task table. It invokes the inter-CPU function listed above which transfers control to the CPU1 sync release program. This program computes spare time expended while waiting in CPU1, and then terminates the CPU1 monitor wait state.

5.5 Program Development Aids

5.5.1 Compile-Time Assignments

Values are assigned to identifiers at compile time for two purposes -- creation of program constants and initialization of program variables. This section illustrates the requirements for program constants and initialization.

5.5.1.1 Constant Definition

The simulators make use of numerous constants. An HOL should allow some method of defining identifiers which will have constant values, as opposed to the use of ordinary variables for this purpose. The constants used are primarily numeric, both fixed and floating point. For example, the 214A visual system altitude limit program (Box 1 of diagram A3353) uses fixed point constants with values of 1/5 and 1/6. The program uses octal constants and must describe in comments what they are. An HOL should permit an understandable definition of such constants.

In another example from the 214A visual system, the offline data verification program, which checks the modelboard contour map (Box 1 of diagram A3353), uses four constants preset to modelboard dimension information, XSTART, XEND, YSTART, YEND. On the first execution of the program, the following initialization occurs:

```
XLOW = 4*CEIL(XSTART/4)
XHIGH = 4*FLOOR(XEND/4)
YLOW = 4*CEIL(YSTART/4)
YHIGH = 4*FLOOR(YEND/4)
```

This is an operation which could be more logically done at compile time. If compile time expressions are provided, the relationship between the two sets of values can still be expressed.

Most constants used are in large data tables. Examples of this are the modelboard contour map (Box 1 of diagram A3353) and the LFI tables, both described in Section 4. A simulation HOL should provide a convenient and readable method for establishing such tables of constants.

5.5.1.2 Variable Initialization

Use of compile-time initialization of variables occurs only in the offline programs. All initialization of realtime variables is done dynamically. None of the instances noted involve setting of large tables of data.

5.5.2 Conditional Compilation

Conditional assembly (compilation) is used in the simulator support programs to attain the reusability of one program on several CPUs. In some cases, conditional assembly is used to provide variations between the versions. For example, the device codes accepted by the I/O request handler depend on the CPU. The data presetting in the "system description modules" also employs conditional assembly based on CPU. Conditional assembly is also used in the remote digital readout (DRU) program, to adapt the program to its CPU. (There is a copy in each CPU.) Its main use is in the code which tests the CPU select knob to see if that CPU has been selected, i.e., CPU2 compares the knob value to '2', etc.

5.5.3 Symbol Dictionary

As discussed in Section 5.1.1, the simulators use a global data base facility, the symbol dictionary or "datapool." Offline programs are provided to support the use of this dictionary. The basic capabilities of the data base system include:

- creation, update, printout, etc. of the symbol dictionary (a disk file)
- retrieval of symbols defined in the symbol dictionary during assembly
- creation, update, printout, etc. of a system cross-reference file

Various error detection capabilities are included in these programs. For example, a list of symbols not referenced in any module may be printed.

The offline programs used to create simulator data files (e.g., the malfunction compiler; see Section 5.3.2 for a complete list) are also sensitive to the symbol dictionary, allowing use of program symbols in their inputs. For example, in the malfunction compiler (Box 3 of diagram A35), an input expression might be:

LEF = ALT(19500/20200)*AIRSP(200/210)+T(50/)

indicating:

"turn on malfunction LEF when 19500 $\leq$ ALT $\leq$ 20200 and

200 $\leq$ AIRSP $\leq$ 210 or T $\geq$ 50"

The compiler translates this to a binary representation, looking up and inserting the location and type information for the datapool symbols LEF, ALT, AIRSP, and T.

5.5.4 Debugging Support

The major debugging aids provided to support simulator debugging are the remote decimal readout unit (DRU) program and the Math Model Test program.

The remote decimal readout unit (DRU) is a peripheral device which allows control of the CPUs from remote locations in the simulator complex. The functions it provides are:

- reading or setting of any core location in octal, scaled fixed point, floating point, or BAMs (Binary Angular Measurement)
- setting of a trap address at which a specified register's contents will be printed (on first execution of trap address only)
- display of a selected bit (only) of a selected location
- halting of all other tasks (i.e., except the remote decimal readout task), and restart

Figure 5-9 illustrates the DRU control panel.

The DRU program runs as a task in the foreground task table (Box 3 of diagram A312). The program tests the various switch settings, etc. and responds accordingly. No I/O is performed in the program. The DRU I/O is done in the twice/frame update performed by the AST Master Controller (see Section 5.6). This program, when implementing the Halt function, makes itself the only task.

The Math Model Test program (Box 2 of diagram A2) is an offline program used for testing and debugging simulation programs. It executes a card stream of input commands, which request such functions as:

- loading a program
- setting of datapool variables or printout of their values (variables are referenced by name, values are specified or printed according to their type as indicated in the datapool)

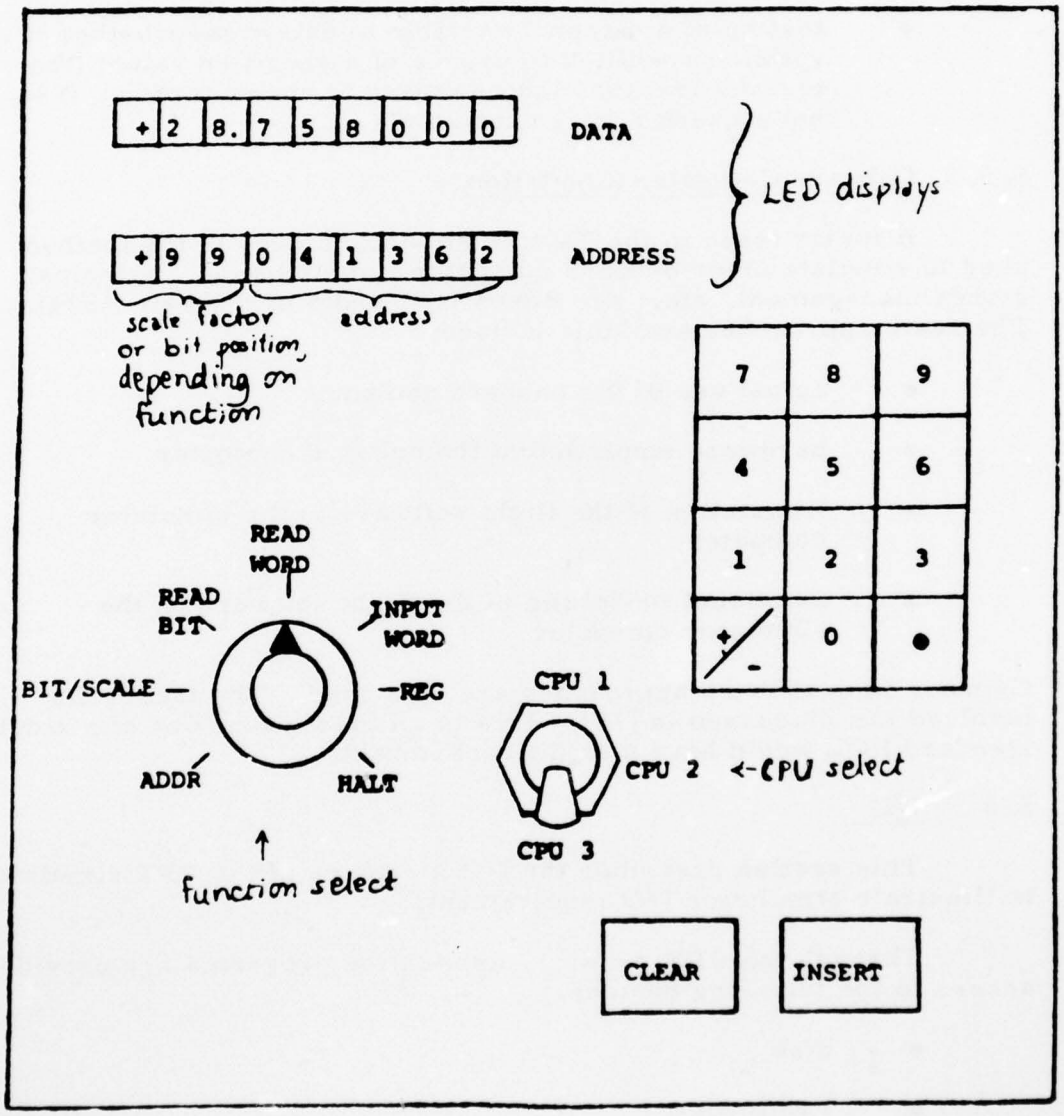


Figure 5-9. Remote Digital Readout Panel

- setting or printout of memory locations specified in octal
- tracing execution
- timing execution
- testing of a datapool variable to determine whether it is within a specified tolerance of a specified value; (the test occurs when the command is encountered -- it is not a check during execution).

5.5.5 Onboard Computer Simulation

A major issue in the Tactics simulation area is the method used to simulate other onboard computer systems (e.g., avionics, stores management, etc.; see Boxes 2, 3, and 4 of diagram A334). The basic approaches available include:

- actual use of the onboard computer
- hardware emulation of the onboard computer
- translation of the flight software to the simulator computer
- functional modelling of the flight software on the simulator computer

Combinations of these approaches are also used. The trade-offs involved are discussed in [18]. This is an area where use of a single standard HOL would have a significant impact.

5.6 I/O

This section describes the I/O structure of the UPT simulator to illustrate simulation I/O requirements.

Through the UPT monitor, application programs are provided access to the following devices:

- disk
- Aydin CRT
- TTY/CRT

All requests are made through the I/O request handler, using a parameter table as illustrated in Figure 5-4. This request handler passes control to the individual device request handler based on the TYPE field. Each request handler maintains a request queue, into which it links the request. (Actually, the disk handler has two chains, high and low priority.) Other processing depends on device type and will be described briefly below. Each CPU has an I/O request handler, a disk request handler, and a TTY request handler. Only CPU2 has an Aydin request handler, and Aydin requests are invalid in other CPUs.

The Aydin handler first tests the status of the Aydin device. Assuming there are no error conditions (in general, errors are handled by setting an indicator in the I/O parameter table), the request is tested to see if it is a status check, in which case the routine returns with the completion bit set in the parameter table. The status is in a dedicated memory location from which it may be obtained by the caller. Otherwise the request is added to the Aydin request chain. If it is the only request, it is processed immediately (by entering in the middle of the interrupt handler). Otherwise, the request handler returns and the chained request will be processed after an Aydin completion interrupt, when its turn comes. When a request is processed, the appropriate I/O command is constructed and initiated. If it is a block transfer, control is then relinquished and it will interrupt when complete. Then the completion bit is set in the associated parameter table and the next request on the chain is processed. If the request to be processed is a read of the Aydin register, the command is made, and the status repeatedly checked to wait for completion since this function does not interrupt when complete. Then the completion bit is set in the parameter table and the next request is processed.

The disk I/O process, if in the master disk CPU (#2), proceeds similarly to the Aydin I/O process. The processing required, however, is more complex. Both direct and indexed files are supported. Figure 5-6 illustrates the indexed file structure. The functions provided are direct read and write, indexed read (writes must be made by treating the file as a direct access file), and extended index read (in which two contiguous subfiles of an indexed file may be read into different parts of memory in one request). Because of the complexity of this process, only CPU 2 contains a driver to handle disk request chaining, error handling, interrupt servicing, building of command words, and indexed file/subfile searches. (CPU 2 was selected because it does the most disk I/O.) CPUs 1 and 3 contain slave disk drivers which pass the parameter table to CPU 2, and then pick up from CPU 2 the command words to initiate the actual block transfer. This allows a saving of core and processor time in CPUs 1 and 3. Doing the actual transfer in the individual CPU allows the buffer used to be in private memory. (Note that the parameter table must be in common memory.) Figure 5-10 illustrates the disk I/O control flow.

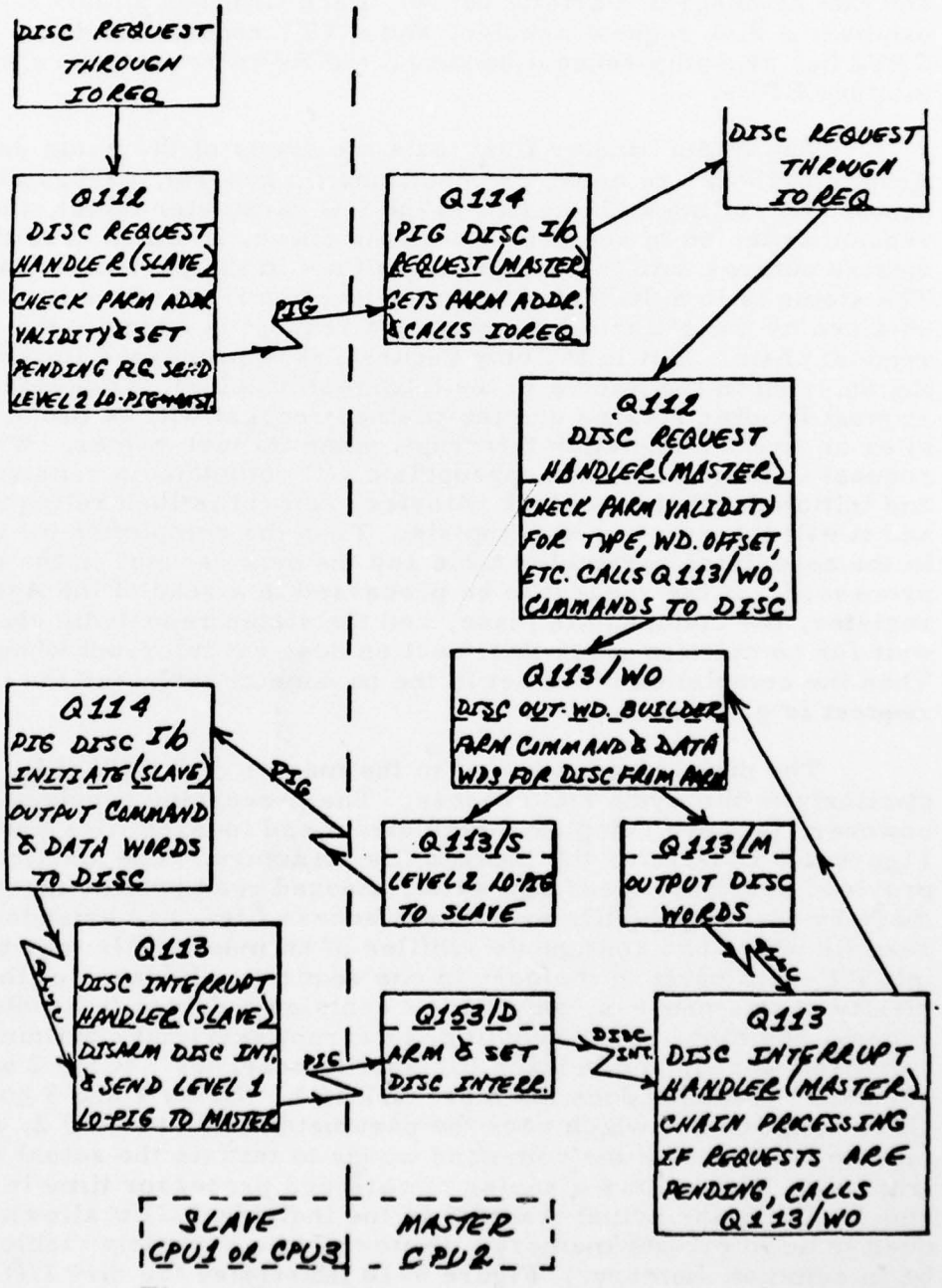


Figure 5-10. Disk I/O Overview

The TTY/CRT I/O is performed through a device called the IOC coordinator. This device controls I/O to any device which uses single word/byte transfers exclusively. Although in this configuration only the TTY/CRT is attached to it, this would also include a card reader/punch or line printer. Only teletype output is a callable function. It runs as a background task, which is scheduled for the first character by the TTY output handler and thereafter whenever an output interrupt occurs. The input routine is scheduled when an input interrupt occurs. TTY input is used primarily by the debug routine, which ties in directly to the input routine.

The IOC coordinator itself resides only in CPU 1. Its execution is triggered by the 120-Hz clock interrupt. (An output request to the teletype does not result in immediate initiation of the output. The request is simply added to the chain, to be processed at the clock interrupt.) The actual input and output drivers and interrupt handlers exist in all three CPUs. Figure 5-11 illustrates the TTY I/O control flow.

Another type of I/O also occurs in the monitor. This is the input and output between the simulator hardware and the data base. This transfer is done through a special device called the AST Master Controller. This device can perform analog to/from digital conversions. Figure 5-12 illustrates this system. This I/O is not requested by programs. It is performed twice per frame on countdown of the interval timer. At 20 msec into the frame, special updates (visual and remote decimal readout unit) are made. At 45 msec, normal updates (everything) are made. A datapool module contains a collection of pointers which define a chain of data to be transferred, and transfer of the entire chain is made with a single invocation. Each update operation, or transfer, is preceded by the transfer of a test data chain. The test data transfer interrupts when complete, allowing verification of a successful test. The actual data update does not interrupt on completion. All this occurs in CPU 1.

5.7 Machine Dependency

Certain types of processing performed in the simulator monitors (as in most executives) require low-level, machine-dependent code. Examples of these functions are:

- setting of the system clocks (Real Time Clock, 120-Hz Clock, Interval Timer)
- enabling, disabling, intercepting, and triggering of interrupts
- memory protection

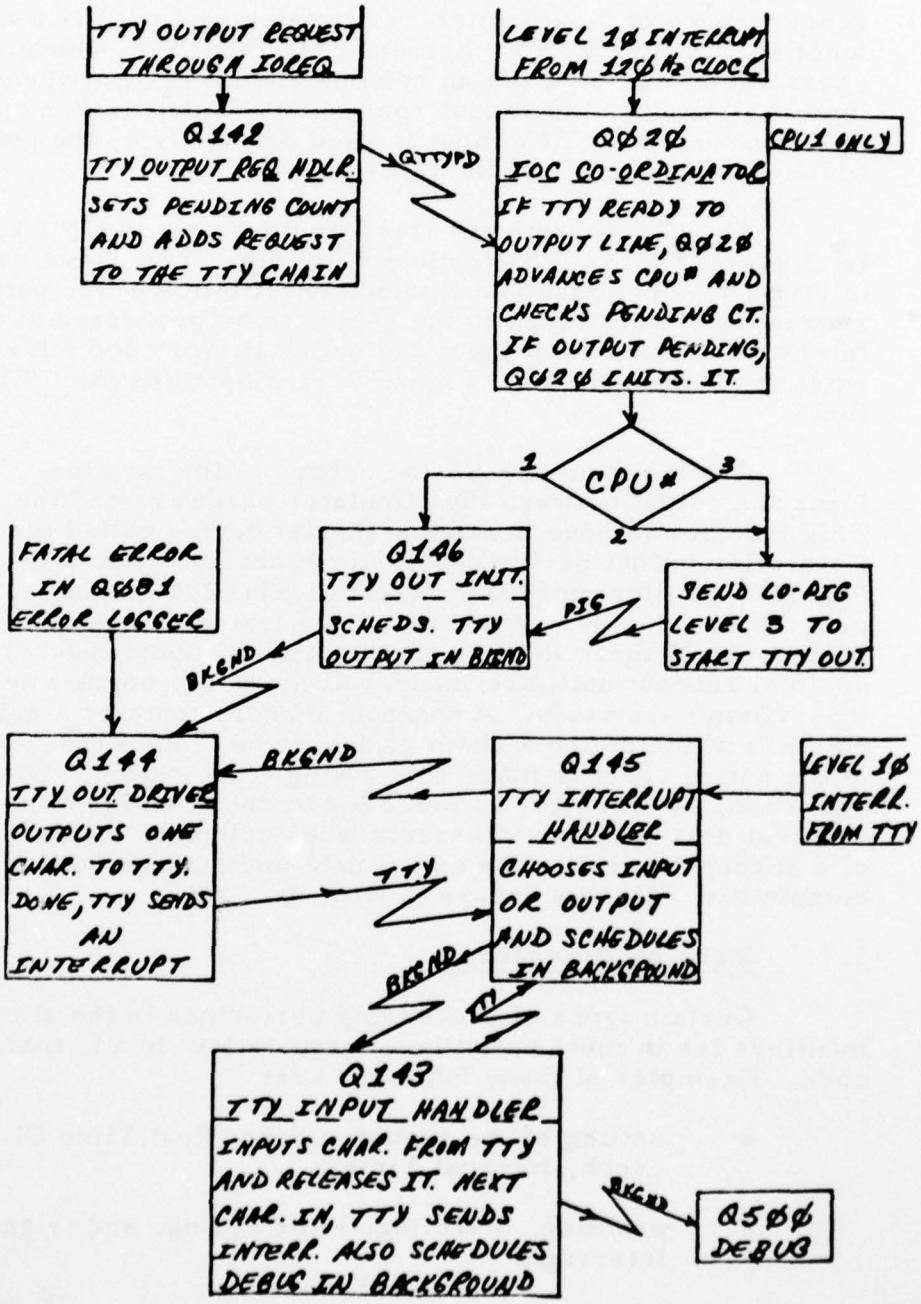


Figure 5-11. TTY I/O Overview

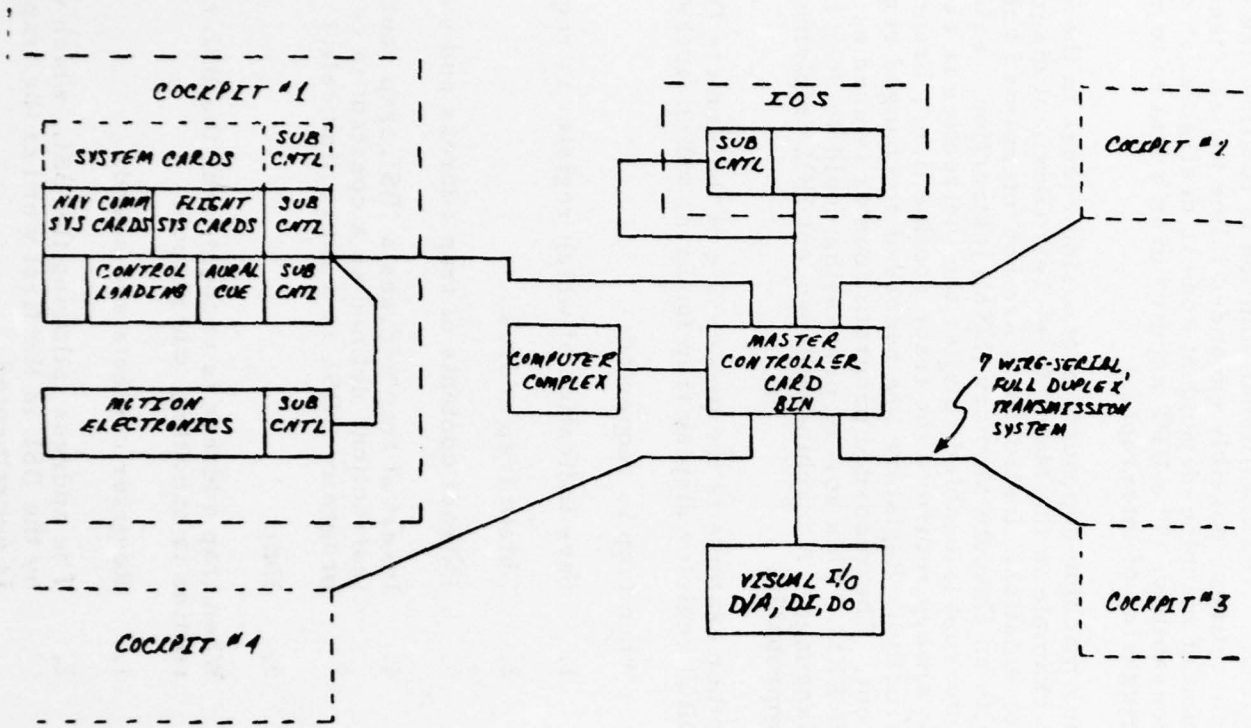


Figure 5-12. AST Master Controller I/O Overview

- specification of a location for the hardware address trap
- I/O (at the lowest level)

It isn't clear how functions such as these can be provided in a machine-independent HOL. The traditional solution is to allow use of assembly language subroutines, possibly expanded inline for efficiency reasons. Encapsulation of machine-dependent code is desirable, in order to facilitate reusability. (The UPT monitor uses a macro to perform software triggering of interrupts.)

Other machine-dependent processing occurs in the debugging areas. An example is the Math Model Test (Box 2 of diagram A2) trace implementation. Instructions are not interpreted but are executed with an Execute Memory (EXM) instruction. Prior to the EXM, an interrupt is enabled so after the instruction is executed, control will always return to the trace routine (i.e., branches will not be taken directly). Registers are recorded so changed registers may be printed out. The recorded program counter is used as the address for the next EXM. This sort of processing could be done in encapsulated assembly language. A machine-independent HOL implementation of it seems improbable.

Another example is the processing of the Remote Decimal Readout (DRU) register display trap function, which works as follows:

a. When trap is requested:

   1. Save indication of which register is requested.
   2. Obtain trap address.
   3. Extract contents of trap address and save.
   4. Insert at trap address a 'BSL trap routine' instruction, defined as a constant in current program. (BSL is a subroutine call.)
   5. Exit.

b. When trap address is encountered, the BSL to the trap routine is executed, whereupon:

   1. Register contents are saved.
   2. The address following the BSL, which was stored by the BSL in the first word of the trap routine, is decremented.

3. The saved previous contents of the trap location are restored by an indirect store through the first word.

4. The desired register is displayed as it was stored on entry to this routine.

5. A standard subroutine return, indirect branch through the first word, now returns to the interrupted routine to execute starting with the trapped instruction.

This routine is not reentrant since registers are not saved in a stack and single memory locations are used for the various indicators. The only implication of this is that instructions within this routine may not themselves be trapped. HOL subroutine linkage mechanisms would not provide this sort of linkage, i.e., decrementing the return address and using the saved return address as a pointer.

Section 6

DETAILED SHOL REQUIREMENTS AND LANGUAGE EVALUATIONS

This Section presents the high-order language (HOL) requirements which should be met by a language for programming flight training simulators and evaluates the candidate languages with respect to these requirements. The requirements were derived by analyzing the functional and environmental factors relevant to simulator programming.

This Section describes the specific simulation language requirements determined by our detailed study of the application area as described in Sections 4 and 5. It is intended to serve as a definitive basis for evaluating how well existing programming languages could serve in programming simulators. The benefits to be derived from this presentation of language requirements are these:

a. The key implications of our detailed study of programming requirements are presented concisely and rigorously.

b. *Use of the document as an evaluation guide* guarantees that no simulation programming requirements will be overlooked.

c. The document addresses specific SHOL requirements as well as general principles which must be considered throughout language design or evaluation.

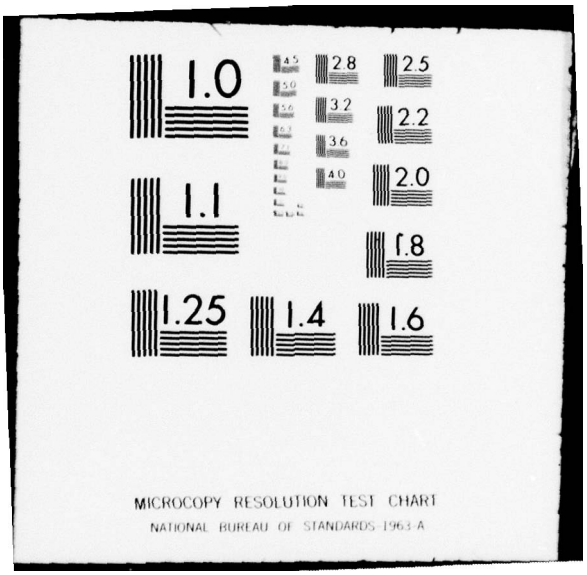
There is one area of requirements that is not addressed in this document -- exception handling (i.e., specification of the program action to be taken when a routine encounters some condition it is not prepared to deal with, e.g., overflow, time-out, or inconsistencies in some data base). The current state of the art with respect to exception handling language features is quite undeveloped; not much of significance can be said with confidence about what minimal exception handling requirements should be. Moreover, to accurately assess these requirements in the simulator area would require a more detailed study of coding and design practices than was possible in this study. Consequently, we have chosen to leave requirements specifications open in this area.

This Section is organized with an outline similar to that of the IRONMAN [DoD, 1977]. However, the purpose this document serves is somewhat different than the purpose the IRONMAN serves. In particular, this document addresses just HOL requirements for design, implementing, and maintaining flight simulator software as opposed to requirements of all embedded computer systems. Consequently we have deleted some IRONMAN requirements that are inapplicable to simulator programming, added others

AD-A058 994 SOFTECH INC WALTHAM MASS F/G 9/2
SIMULATION HIGHER ORDER LANGUAGE REQUIREMENTS STUDY.(U)
AUG 78 J B GOODENOUGH, C L BRAUN F33615-77-C-0029
UNCLASSIFIED AFHRL-TR-78-34 NL

2 OF 3
AD
A058994




1.0
1.1
1.25
1.4
1.6
2.8
2.5
3.2
2.2
3.6
2.0
4.0
1.8
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

that are consistent with the IRONMAN but more specific, and finally, changed some IRONMAN requirements because they were inappropriate for the simulator programming environment and application. Another source of differences between this document and the IRONMAN is that this document is intended to describe requirements rather than to guide a language design effort. Consequently, unlike the IRONMAN, language capabilities not specifically required or forbidden may be provided in a language, although it is not expected that such capabilities will make the language more suitable for programming flight simulators. If this document were to be used to guide a language design effort, some requirements would be specified in greater detail and some would be rephrased to ensure a uniform language. Other modifications would depend on whether the design was to proceed by modifying a particular language or was to be created without such constraints.

Some requirements specified here are considered essential to support simulator programming -- i.e., a language would have to have all these features to be usable in programming all simulator functions. These requirements are marked with an asterisk (*). Other requirements are considered desirable but not essential. These are features recommended for inclusion in any language specifically designed for this application. In recommending modifications to the candidate languages to attain a usable SHOL, only the marked requirements were judged to be necessary. Other features, however, were weighed in determining the overall suitablity of the particular language.

Each language requirements section begins with the statement of a goal that describes the overall objectives to be met by the SHOL in that language area. Following the goal are several supporting concepts that aid in the attainment of the stated goal. Finally, following each stated concept are one or more specific language requirements that realize that concept. Following each set of requirements for a particular concept, PL/I, JOVIAL J3B and J73I, PASCAL, and FORTRAN are discussed with respect to those requirements. At the end of each section, languages are ranked according to how well they meet the goal of that section.

A precise and consistent use of terms has been attempted in stating requirements. Potentially ambiguous terms have been defined in the text. Care has been taken to distinguish between requirements, given as text, and comments about the requirements, given as bracketed notes.

The following terms have been used to indicate where and to what degree individual requirements apply:

must is required	indicates a required capability to be provided by a language or its translator.

not required	indicates a language capability that, if present in an existing language, need not be used. A language having such a capability is usable for simulator programming, but is less desirable than a language not having the capability.
not desired must not	indicates a language capability that must be absent from a language (generally because its presence, even if not used, would degrade object code efficiency or the ability of the translator to detect program errors).
need only	indicates a minimal required capability. A language providing a more extensive capability is acceptable even though the additional capability is not needed in the simulator application.
should	indicates a desired goal but one for which there is no objective test.
shall attempt	indicates a desired goal but one that may not be achievable given the current state-of-the-art, or may be in conflict with other more important requirements.
must require	indicates a requirement placed on the user by the language and its translators (language is subject).
must permit	indicates a requirement placed on the language to provide an option to the user (language is subject).
may	indicates a requirement placed on the language to provide an option to the user (user is subject).

6.1 General Design Principles

Goal

By analyzing the functional and environmental requirement of the simulator application area, the general principles to be observed in SHOL design were determined. These principles are to be followed in meeting each of the specific requirements detailed in subsequent sections.

Supporting Concepts

1A. Application Suitability.

The SHOL must support the programming of simulator software.

Requirements

1A-1. The language must provide the functional capabilities necessary for the production of flight training simulation online and support programs.

1A-2. A language containing only features required by the application is considered more desirable than a language containing additional features. [The intent is to permit a subset of an existing language to be used if it satisfies the requirements and if use of the subset can be administratively controlled.]

Language Evaluations

Subsequent sections discuss the degree to which each of the languages meets simulator programming requirements. The only one of the languages which provides a significant number of unneeded features is PL/I. Some of these (e.g., PICTURE attributes) have no interaction with features which would be used by the simulator programmer, while others may require that the programmer be aware of them in order to ensure correct use (e.g., specifiable lower array bound). Excess capability, of course, increases translator size and implementation cost, and may impact translation speed even if unused.

1B. Correctness.

The language must aid in the development of properly-working programs.

Requirements

1B-1. The language should avoid error-prone features [i.e., features that are difficult to use correctly] and maximize automatic detection of programming errors.

1B-2. Translators must produce explanatory diagnostic and warning messages, but must not attempt to correct programming errors. [Such corrections are seldom appropriate and encourage undisciplined programming habits.]

1B-3. There must be no language restrictions that are not enforceable by translators.

Language Evaluations

Each of the languages contains some error-prone constructs, which are noted more explicitly in subsequent sections. The excess generality of PL/I makes it more difficult than the other candidates to learn to use correctly. On the other hand, PL/I provides good facilities for error detection at translation and execution time, as does PASCAL.

1C. Maintainability

As discussed in Section 4.5, the long lifetime of simulator programs makes ease of maintenance a major design goal.

Requirements

1C-1. The language should emphasize readability over writability, [i.e., it should emphasize the clarity, understandability, and modifiability of programs over programming ease, since programs are usually maintained by programmers who were not involved in their development].

1C-2. Explicit specification of programmer intent should be possible and be encouraged [e.g., declarations of the range of values a variable can have; see 3A-5].

1C-3. Defaults should be provided only for instances where the default is stated in the language definition, is always meaningful, reflects common usage, and can be explicitly overridden.

Language Evaluations

FORTRAN is the least readable, and hence least maintainable, of the candidate languages. Specific deficiencies (i.e., lack of features supporting readability) are noted in the following sections. Examples of FORTRAN deficiencies include numeric statement labels (which PASCAL has also), implicit declarations, and limited identifier length. The JOVIAL variants (J3B and J73I) are fairly readable, though their data declaration statements have a somewhat unreadable format. PL/I and PASCAL are probably the most readable of the candidates, overall.

1D. Efficiency.

As discussed in Section 4.4, the SHOL must support development of efficient object programs.

Requirements

1D-1. Where possible, features should be chosen to have a simple and efficient implementation in many object machines, to avoid execution costs for available generality where it is not needed, to maximize the number of safe optimizations available to translators, and to ensure that unused and constant portions of programs will not add to execution costs.

1D-2. Unduly complex optimization by translators should not be required to obtain efficient object code.

1D-3. Programmers should be able to control time/space tradeoffs through appropriate use of language features [e.g., packing directives; see Section 6.10].

Language Evaluations

The excess generality of PL/I can affect object program efficiency even if the unneeded features are not used, but in most cases the impact of such excess generality is only on translation efficiency. None of the languages make explicit to the user which features are most costly.

Section 6.10 discusses the control provided over time/space tradeoffs by the various languages. In general, the JOVIAL variants are best in this respect, while FORTRAN provides the least capability.

1E. Simplicity.

Simplicity is desired in the SHOL in order to enhance readability and to make the language readily learnable by simulator programmers (who are primarily engineers whose experience with programming languages is not extensive, generally including only assembly language and FORTRAN experience).

Requirements

1E-1. The language should use familiar notations where such use does not conflict with other goals.

1E-2. It should have a consistent semantic structure that minimizes the number of underlying concepts.

1E-3. It should be as small as possible, consistent with the needs of the application. [See also 1A-2.]

1E-4. It should have few special cases and should be composed from features that are individually simple in their semantics.

1E-5. The language should have uniform syntactic conventions and should not provide several notations for the same concept.

Language Evaluations

The language which best meets these requirements is probably FORTRAN, particularly because simulator programmers are already familiar with it. Also it is a relatively "small" language -- i.e., it does not contain a large number of constructs or redundant features. PASCAL is also a concise language, but it deviates from familiar usages more than any of the other candidates. PL/I, because of its emphasis on generality, contains a large number of constructs and permits many variations in notation.

1F. Implementability.

Design of the SHOL must take into account the implementability of the language. As discussed in Section 4.3, simulation program translators have traditionally been required to operate on machines of modest capacity.

Requirements

1F-1. The semantics of each feature should be sufficiently well specified and understandable that it will be possible to predict its interaction with other features.

1F-2. To the extent that it does not interfere with other requirements, the language shall facilitate the production of translators that are easy to implement and are efficient during translation. [See also 1D-3.]

Language Evaluations

As all of the candidate languages have been implemented and used, their semantics are well understood though not all are well documented. Because PL/I has so many constructs, there are many interactions between features, making the language more difficult to specify and implement than simpler ones. Its implicit conversion philosophy compounds this problem.

Language Evaluation Summary

Because of the general and diverse nature of these requirements, any attempt to rank languages with respect to them would be inappropriate. Since these general requirements form the basis for the specific requirements in subsequent sections, in effect the remainder of the evaluation document serves this purpose.

6.2 General Syntax

Goal

SHOL syntax must be selected in keeping with the goals of simplicity and maintainability, and with general language design principles. Syntactic conventions must encourage the production of readable programs and must where possible eliminate opportunities for programmer error.

Supporting Concepts

2A. Character Set.

To allow program portability, all SHOL translators must employ the same source character set, and the character set should be widely supported.

Requirements

2A-1. Every construct of the language must have a representation that uses only the 64-character subset of ASCII:

```
 !"#$%&'()*+,-./
0123456789:;<=>?
@ABCDEFGHIJKLMNO
PQRSTUVWXYZ[\]^_
```

Language Evaluations

All of the candidate languages except PASCAL and PL/I are compatible with 64-character ASCII. PASCAL uses the character '↑', and PL/I uses '¬' and '|'. PL/I, however, is compatible with EBCDIC, as are FORTRAN and J3B, but not PASCAL or J73I.

2B. Grammar

The SHOL grammar must contribute to program readability and ease of learning the language and must make common syntax errors easy to detect and diagnose by translators.

2B-1. The language must have a free form syntax [i.e., the semantics of constructs should not depend on their position within a line].

2B-2. Multiple occurrences of a language defined symbol appearing in the same context must not have essentially different meanings. [For example, the assignment operator should be different from the relational equality operator; division of integers yielding an integer result should not be represented with the same symbol that yields a real result.]

2B-3. The language must not permit unmatched brackets of any kind [e.g., BEGIN and END must be paired one for one].

2B-4. All key word forms that contain declarations or statements must be bracketed -- that is, must have a closing as well as an opening key word. [This requirement and the previous one help avoid programmer errors due to confusion over the lexical extent of the various program constructs.]

2B-5. There must be no control definition facility [i.e., no means of defining new control structures].

2B-6. The structure (i.e., syntax) of expressions must not depend on the types of their operands. [This is motivated by a desire for language uniformity.]

2B-7. The precedence levels (i.e., binding strengths) of all infix operators must be specified in the language definition and must not be alterable by the user.

2B-8. The precedence levels should be consistent with standard practice.

Language Evaluations

Of the candidate languages, only FORTRAN does not have a fully free form syntax.

None of the languages are consistent in always using different symbols for different meanings. For example, all but PASCAL use the same symbol for integer and real division. Most also use compound statements in a variety of ways in control structures, rather than employing distinctive syntax.

PL/I, in permitting multiple closure of blocks, permits unmatched brackets. Most of the languages also use compound statements to delimit the lexical extent of keywords, rather than providing individual closing keywords for each.

None of the languages allow the definition of new control structures. None allow the user to alter precedence levels of operators.

All languages employ a standard set of operator precedence levels except for PASCAL, which has only four levels and is not consistent with standard practice.

2C. Mnemonic Identifiers.

The language must allow the programmer to select meaningful and informative identifier names. This is particularly important given the large number of identifiers required in a simulator system and the use of these identifiers by groups of programmers.

Requirements

2C-1. Mnemonically significant identifiers [more than eight characters long] must be permitted.

2C-2. There must be a break character for use within identifiers. [e.g., the underscore character, as in RATE_OF_CLIMB].

Language Evaluations

All languages except FORTRAN permit identifiers of more than eight characters. All except PASCAL allow a break character in identifiers.

2D. Static Typing.

In keeping with general language design principles and in support of program maintainability and efficiency, the types of values must be determinable from the source program.

Requirements

2D-1. The value type of each variable, array or record component, expression, parameter, and function result must be determinable at translation time. [A value type specifies the set of values associated with a program element. "Type" is used in this document to mean value type.]

2D-2. There must be no implicit conversions between value types -- explicit conversion operations shall be provided.

2D-3. A reference to an identifier that is not declared in the most local scope must refer to a program element that is lexically global, rather than to one that is global through the dynamic calling structure. [This is the normal block structure scoping rule.]

Language Evaluations

Most value types are determinable at translation time in all of the languages. The only exceptions involve the attributes of formal procedure parameters. Only J73I requires a specification of parameter attributes of such procedure parameters, and only J73I and PASCAL require specification of their result attributes. (J3B does not allow procedure parameters.)

Implicit conversion is prevalent in PL/I, and, to a lesser extent, in J73I. There is little in FORTRAN, J3B, and PASCAL. In general, it is restricted in these languages to conversions between integer and real types. PASCAL allows implicit conversion only of integers in assignment to reals. Most of the languages restricting implicit conversion, however, do not provide all of the explicit conversion operations which might be necessary. For example, only PL/I provides explicit conversion from numeric types to character strings.

As required all the languages bind free names statically rather than dynamically.

2E. Comments

The SHOL must provide a comment facility that is easy to use, so commenting is encouraged. The comment facility should allow comments to be used with maximum readability.

Requirements

2E-1. The language must allow comments to be embedded within program text [e.g., a comment bracketed by special left and right bracket symbols, and preceded and followed by program text on the same line, or a comment terminated by end of line].

2E-2. Bracket symbols must be short -- no more than two characters.

Language Evaluations

Of the candidate languages, only FORTRAN does not provide the required flexibility in comment placement. All languages provide short comment bracketing symbols.

Language Evaluation Summary

The candidate languages are ordered, according to the degree to which they meet SHOL general syntax requirements, as follows:

J3B- Most of the requirements are met.

Deficiencies are:

- use of same symbol for different meanings
- lack of closing keywords
- lack of explicit conversion operations

J73I- Most requirements are met. Implicit conversions can occur.

Deficiencies are:

- use of same symbol for different meanings
- lack of closing keywords
- implicit conversions
- lack of explicit conversion operations

PASCAL- Is not ASCII compatible. Operator precedence levels are non-standard.

Deficiencies are:

- use of same symbol for different meanings
- lack of closing keywords
- not ASCII compatible
- non-standard precedence levels
- no break character in identifier
- lack of explicit conversion operations

FORTRAN- Comment facility and statement formatting are inflexible. Mnemonically significant identifiers are not permitted.

Deficiencies are:

- not free format
- use of same symbol for different meanings
- identifier length too restricted
- comment placement inflexible

PL/I- Is not ASCII compatible. Multiple closure and implicit conversions are permitted.

Deficiencies are:

- not ASCII compatible
- use of same symbol for different meanings
- multiple closure
- implicit conversions

6.3 Data Types

Goal

The SHOL must provide the value types required to represent simulator data, must support efficient processing with them, and must allow them to be used in a readable and understandable manner. Notation for and support of the various types should be consistent between types and whenever possible should correspond to common practice.

6.3.1 Numeric Types

Goal

The language must support integer and real (both fixed and float) numeric types. Uses of numeric data in simulator programming are discussed in Sections 5.2.1 and 5.2.2. The SHOL must allow the programmer to use the various real number representations available on the target computer. Requirements for this are discussed in Section 5.2.2.1 and in Section 4.2.

Supporting Concepts

3A. Numeric Type Definitions.

Integer, fixed, and floating point types, with programmer control over precision, must be provided. Representations should correspond to standard usage.

Requirements

*3A-1. The language must provide types for integer, fixed point, and floating point numbers. [See Sections 5.2.1 and 5.2.2.]

3A-2. The minimum accuracy of each floating point variable [e.g., number of decimal digits] and the minimum accuracy of each fixed point variable [e.g., maximum value of the least significant bit] must be specified in programs.

3A-3. Such accuracy specifications must be interpreted as the minimum accuracy to be supported by an implementation -- it is sufficient for implemented fixed point accuracies to be limited to powers of two.

*3A-4. Various sizes of real number representations are required. [Section 5.2.2.1 discusses the use of both single and double word representations to allow accuracy vs. space tradeoffs.]

3A-5. Declarations of the range of numeric variables must be optional (see also Section 6.10), and need only be specified with constant values. [Range specifications make programs more understandable and can improve object code optimization, but there is insufficient experience with their use to make range declarations mandatory. Range declarations may also be used to specify (implicitly) the minimum number of bits occupied by fixed point values.]

Language Evaluations

Of the candidate languages, only J3B and PL/I provide both fixed and floating point real number types. Only J73I and PL/I allow specification of floating point accuracy. PL/I allows specification in decimal digits, while J73I requires specification in bits. In neither case is the interpretation of the accuracy implementation dependent. Only PL/I allows specification of fixed point accuracy. Again, accuracy may be specified in decimal digits, and interpretation is not implementation dependent.

Programmers may select various sizes of real number representations in all of the languages except PASCAL and J3B (for fixed point). PL/I and J73I allow explicit specification of the size. None of the languages allow explicit declaration of read variable ranges.

3B. Numeric Literals.

The SHOL must allow the programmer to specify numeric literals (i.e., numbers of the form 1, 5.6, etc.) in a readable and consistent manner.

Requirements

*3B-1. Numeric literals are required. [See Section 5.2.]

3B-2. Embedded spaces must be permitted in real literals. [These would enhance readability in the long literals occasionally used in simulator programming, as in the example in Section 5.4.1.2.]

3B-3. Numeric literals must have the same value in programs as in data. [i.e., literal values input during program execution shall have the same value as if they had been processed by the translator.]

Language Evaluations

All of the candidate languages provide numeric literals for both integer and real types. Only FORTRAN permits embedded spaces. None of the languages appear to require that program and data literals convert equivalently.

3C. Numeric Operations.

The language must provide a uniform set of the basic arithmetic and comparison operations for numeric types. Trigonometric operations must also be supported. Section 5.2.2.2 discusses simulator requirements for numeric operations.

Requirements

3C-1. There must be operations [i.e., functions] for conversion between numeric value types and for conversion from other types (e.g., character, bit string) to numeric types. [See Section 4.5.]

*3C-2. There must be operations for addition, subtraction, multiplication, division with real [fixed point and floating point] result, and negation for all numeric value types. [See Section 5.2.]

3C-3. There must be operations for integer and fixed point division with integer result and remainder. [A particular requirement for this occurs in the camera/modelboard visual systems, for locating modelboard positions, as discussed in Section 5.3.4.4.]

3C-4. There must be operations for specifying the accuracy of fixed and floating point addition, subtraction, multiplication, and division results.

3C-5. Default scaling rules for fixed point operations need not produce results more accurate than the accuracy (i.e., scale) of the least accurate operand. [e.g., 1.1+20.01 may yield 21.1.]

3C-6. Absolute value and max/min functions (allowing more than two arguments) must be provided for all numeric value types. [See Section 5.2.2.2.3.]

3C-7. For real value types, square root and trigonometric functions are required. [Trigonometric functions are used in analog I/O handling and in various display-related programs such as the map plate compiler; see Section 5.2.2.2.1.]

*3C-8. There must be equality [i.e., equal and unequal] and ordering operations [i.e., less than, greater than, less or equal, and greater or equal] between elements of each numeric value type; [see Section 5.4.1.1].

3C-9. There must be a means of explicitly testing whether a numeric value is within a given range [e.g., the chained comparison; see Section 5.4.1.2.]

3C-10. Numeric values must be considered equal if and only if they represent exactly the same abstract value. [i.e., accuracy specifications must not be taken into account in testing for equality; otherwise A=B and B=C does not imply A=C.]

Language Evaluations

All of the languages allow explicit conversion from real to integer types, and all except PASCAL from integer to real. Of the languages providing more than one real representation, all but J3B provide representational conversions.

Only PL/I provides all desired conversions from other types, but J73I and J3B provide some support.

All of the languages provide the basic arithmetic operations required, but only PL/I provides accuracy-defining specifications for their results. Of the numeric functions required, only FORTRAN and PL/I provide max/min functions, and only FORTRAN, PL/I and PASCAL provide square root and trigonometric functions. (PL/I provides more trigonometric capability.) All languages have an absolute value function.

All of the languages have numeric relational operations required, but none have a capability for range testing, such as chained comparisons. In all of the languages, numeric comparisons are exact.

Language Evaluation Summary

The candidate languages are ordered, according to the degree to which they meet SHOL numeric data type requirements, as follows:

PL/I- Most major requirements are met.

Deficiencies are:

- accuracy specifications are implementation dependent
- no range specifications
- no embedded spaces in literals
- program and data literals not required to convert equivalently
- no chained comparisons

J3B- Control over numeric accuracy is inadequate. Not all desired arithmetic operations are provided.

Deficiencies are:

- no conversions from character or bit to real
- no accuracy specifications
- no range specifications
- no embedded spaces in literals

- program and data literals not required to convert equivalently
- no real representational conversions
- no accuracy-defining specifications for results of computations
- no max/min, square root, or trigonometric functions
- no chained comparisons

FORTRAN- Fixed point real numbers are not provided. Most other requirements are met.

Deficiencies are:

- no conversions from other types to numeric types
- no fixed point reals
- no accuracy specifications
- no range specifications
- program and data literals not required to convert equivalently
- no accuracy-defining specifications for results of computations
- no chained comparisons

J37I- Fixed point real numbers are not provided. Not all desired arithmetic operations are provided.

Deficiencies are:

- no fixed point reals
- accuracy specifications implementation independent
- no range specifications
- no embedded spaces in literals

- program and data literals not required to convert equivalently
- no accuracy-defining specifications for results of computations
- max/min, square root, or trigonometric functions
- no chained comparisons

PASCAL- Fixed point real numbers are not provided. Many required accuracy controls, conversions, and numeric operations are not provided.

Deficiencies are:

- no conversions from other types to numeric types
- no fixed point reals
- no accuracy specifications
- no control over size of real number representations
- no range specifications
- no embedded spaces in literals
- program and data literals not required to convert equivalently
- no integer to real conversions
- no accuracy-defining specifications for results of computations
- no max/min functions
- no chained comparisons

6.3.2 Enumeration Types

Goal

The SHOL must provide a status, or enumeration, data type. As discussed in Section 5.2.3, use of enumeration types to represent flags, case alternatives, and array indices would greatly enhance program understandability.

Supporting Concepts

3D. Enumeration Type Definitions.

Enumeration types are required for program readability.

Requirements

*3D-1. There must be value types that are definable in programs by ordered enumeration of their elements [e.g., type angle = (phi, psi, theta)].

Language Evaluations

Of the candidate languages, only J73I and PASCAL provide enumeration types. The J73I form is rudimentary -- essentially a sequence of integers with names. There are no enumeration variables in J73I.

3E. Enumeration Literals.

Enumeration values should be expressible in a readable and natural manner.

Requirements

3E-1. The elements of an enumeration type may be identifiers.

3E-2. Enumeration value names of different enumeration types must be permitted to be identical.

Language Evaluations

In J73I, enumeration literals are lexically distinct from identifiers. J73I allows duplicate enumeration names in different lists, while PASCAL does not.

3F. Enumeration Operations.

Operations provided for enumeration types must allow their use as flags, case alternatives, and array indices.

Requirements

*3F-1. There must be at least equality and inequality operations between elements of enumeration types. [This is to ensure uniformity; equality should be an operation defined for all types.]

*3F-2. There must be successor and predecessor operations on each enumeration type. [i.e., operations producing the next and preceding elements of an enumeration type's value set; these operations are inherent in the notion of an enumeration type.]

Language Evaluations

Both J73I and PASCAL support equality and inequality operations on enumeration types. Only PASCAL provides successor and predecessor operations.

Language Evaluation Summary

The candidate languages are ordered, according to the degree to which they meet SHOL enumeration data type requirements, as follows:

PASCAL- The basic requirements are met.

Deficiencies are:

- duplicate names may not be used

J731- The capability provided is rudimentary.

Deficiencies are:

- no enumeration variables
- literals are lexically distinct from identifiers
- no successor and predecessor operations

J3B, FORTRAN, PL/I- Enumeration types are not provided.

6.3.3 Boolean Type

Goal

The SHOL must provide a Boolean data type. As noted in Section 5.2.5, Boolean data is heavily used in simulators, particularly in the Navigation and Communications area.

Supporting Concepts

3G. Boolean Type Definitions.

The Boolean data type facility in the SHOL should contribute to program readability and allow programs to be structured more clearly.

Requirements

*3G-1. A Boolean data type is required [see Section 5.2.5].

3G-2. Boolean expressions must be evaluated in short-circuit mode [e.g., A OR B must not cause the evaluation of B if A is true].

Language Evaluations

Only FORTRAN and PASCAL support an actual Boolean data type. The other languages use bit strings of length one. Only FORTRAN requires short-circuit evaluation of Boolean expressions. (This is not specified in PASCAL.]

3H. Boolean Literals.

A useful Boolean facility requires literals as well as variables.

Requirements

*3H-1. Boolean literals (TRUE and FALSE) are required.

Language Evaluations

Both FORTRAN and PASCAL have TRUE and FALSE literals. J3B provides built-in constant names TRUE and FALSE for the bit strings '1' and '0'

3I. Boolean Operations.

The standard Boolean operations must be provided in a uniform manner.

Requirements

*3I-1. There must be operations for conjunction, inclusive disjunction, and negation (i.e., AND, OR, and NOT) of Boolean value types. [These are the most frequently used Boolean operations; see Section 5.2.5.]

*3I-2. There must be equality and inequality [i.e., exclusive or] operations for Boolean types. [The operations are required for uniformity.]

Language Evaluations

Both FORTRAN and PASCAL provide the desired operations.

Language Evaluation Summary

The candidate languages are ordered, according to the degree to which they meet SHOL Boolean data type requirements, as follows:

FORTRAN,
PASCAL- All desired capabilities are provided.

J3B, There is no Boolean data type (J3B does provide
J73I, PL/I- TRUE and FALSE bit string literals).

6.3.4 Character Type

Goal

The SHOL must provide a character data type and associated character operations to support instructor display programs and offline simulation support programs. Simulator character handling requirements are discussed in Section 5.2.6.

Supporting Concepts

3J. Character Type Definitions.

The character data type should be supplied in a manner which contributes to program efficiency, and should not allow excess generality not required by the application. The feature should be natural and easy to use.

Requirements

*3J-1. A fixed length character string data type is required [as opposed to representing strings as arrays of characters; a separate data type is needed to promote program understandability].

3J-2. Explicit specification of string length is required, and the length must be specified with a constant value. [Only use of fixed length character strings was observed.]

*3J-3. It must be possible for the programmer to define new character sets. [Some simulator peripherals require character sets other than the built-in ASCII.]

Language Evaluations

All of the candidate languages except FORTRAN provide a character data type. In PASCAL, however, character strings are represented as arrays of single characters.

Neither PL/I nor J73I require explicit specification of string length, and PL/I permits strings of non-constant length.

None of the candidate languages support the explicit definition of new character sets.

3K. Character Literals.

A character literal facility allowing representation of character constants for the various simulator peripherals is necessary.

Requirements

*3K-1. Fixed length character string literals are required.

*3K-2. The character code used for a literal must be programmer-definable. [This is necessary to support the concept of definable new character sets.]

*3K-3. It should be possible to include unprintable characters in string literals. [Formatting codes in strings for visual displays are an example of the kind of unprintable characters needed here.]

Language Evaluations

All of the candidate languages (including FORTRAN) provide character string literals. None allow the programmer to define the character code used. Only PL/I and J731 supports the inclusion of unprintable characters.

3L. Character Operations.

Character operations provided in the SHOL must support the types of character processing performed in simulators, which are primarily display formatting and offline data file compilation.

Requirements

*3L-1. There must be operations for substring extraction and assignment (the substring length must not be restricted to a constant value), access to string length, string replication by a constant factor, and location of a given substring within a string (i.e., INDEX) [see Section 5.2.6.1].

*3L-2. Equality and inequality must be defined on character types.

3L-3. Ordering operations must be defined on the built-in character set.

*3L-4. There must be operations for conversion from other types (e.g., numeric, Boolean) to character type. [These operations are required for completeness; it should be possible to obtain a printable representation of a built-in type.]

Language Evaluations

Substring extraction and assignment is provided by J3B, J73I, and PL/I. Only J731 and PL/I allow access to string length, and only PL/I supports string replication and substring location. Only PL/I provides conversions from other types to character.

All of the languages (except FORTRAN) support equality and inequality operations on character types, and all but J3B provide ordering operations.

Language Evaluation Summary

The candidate languages are ordered, according to the degree to which they meet SHOL character data type requirements, as follows:

PL/I- Most desired functions are provided. Excess capability is supported.

Deficiencies are:

- explicit length specification not required
- non-constant length specification allowed
- definition of new character sets not supported

J73I- Fewer of the desired functions are provided.

Deficiencies are:

- explicit length specification not required
- definition of new character sets not supported
- no string replication or substring location
- no conversion from other types to character

J3B- Fewer desired functions are provided.

Deficiencies are:

- definition of new character sets not supported
- no inclusion of unprintable characters in literals
- no access to string length, string replication, or substring location
- no ordering operations
- no conversion from other types to character

PASCAL- Strings are supported as arrays. Few of the desired functions are provided.

Deficiencies are:

- strings represented as arrays
- definition of new character sets not supported
- no substring extraction or assignment, access to string length, string replication, or substring location
- no conversion from other types to character

FORTRAN- There is little support for character data.

Deficiencies are:

- no character data type
- no inclusion of unprintable characters in literals
- no string operations or relations
- no conversion from other types to character

6.3.5 Bit String Type

Goal

The SHOL must provide a bit string data type and associated operations. These are required in simulator programming for manipulating

I/O values and for representing vectors of Booleans such as the frame and cockpit masks and the malfunction indicator vector. Sections 5.2.4 and 5.3.1 discusses uses of bit string data.

Supporting Concepts

3M. Bit String Type Definitions.

The bit string data type should be natural and easy to use and should allow efficient implementation.

Requirements

*3M-1. A bit string data type is required [see Sections 5.2.4 and 5.3.1].

3M-2. Explicit specification of bit string length is required and must be specified with a constant value. [No use of varying length bit strings was observed.]

Language Evaluations

Bit strings are provided in J3B, J73I, and PL/I. Neither PL/I nor J73I require explicit length specification. PL/I allows strings of non-constant length. (PASCAL provides a capability similar to bit strings with the set data type.)

3N. Bit/String Literals.

There must be a natural notation for specifying bit string literals.

Requirements

*3N-1. Fixed length bit string literals are required.

3N-2. Literals must be specifiable in bases 2, 8 and 16. [Examples of literals in all these bases have been observed.]

Language Evaluations

All three languages which support a bit string data type provide bit string literals. PL/I allows specification only in base 2, J3B allows only base 16, while J73I allows all three.

3O. Bit String Operations.

The operations provided for bit strings must support the construction and decomposition of strings representing I/O values, as well as the masking and shifting of strings representing indicator vectors.

Requirements

*3O-1. There must be operations for bit-by-bit conjunction, inclusive disjunction, exclusive disjunction, and negation [AND, OR, XOR, and NOT] defined for bit strings. [These are the normal bit string operations.]

*3O-2. There must be operations for substring extraction and assignment (the substring length must not be restricted to a constant value), access to string length, string replication by a constant factor, and location of a given substring within a string (i.e., INDEX). [A use of bit string extraction is discussed in Section 5.2.4.]

*3O-3. Equality and inequality must be defined on bit string types.

3O-4. There must be operations for left and right shifting of bit strings.

3O-5. There must be operations for conversion from other types (e.g., numeric) to bit string type. [Accessing the representation of a value is necessary in I/O and other conversions.]

Language Evaluations

All three languages which support bit string provide AND, OR, and NOT operations. All provide the desired conversion operations. PL/I does not provide XOR, while J3B and J73I do. All three provide substring extraction and assignment, J73I and PL/I provide access to string length, and only PL/I supports string replication and substring location. Shift operations are provided only in J3B and J73I.

Only J73I and PL/I support equality and inequality operations.

Language Evaluation Summary

The candidate languages are ordered, according to the degree to which they meet SHOL bit string data type requirements, as follows:

J73I- Most requirements are met.

Deficiencies are:

- explicit length specification not required
- no string replication or substring location

J3B- Fewer capabilities are provided.

Deficiencies are:

- no base 2 or 8 literals
- no access to string length, string replication, or substring location
- no equality/inequality operations

PL/I- Several desired capabilities are not provided.

Excess capabilities are supported.

Deficiencies are:

- explicit length specifiction not required
- non-constant length strings permitted
- no base 8 or 16 literals
- no XOR or shift operations

PASCAL- Some similar capabilities are provided by the set data type.

FORTRAN- Bit strings are not supported.

6.3.6 Pointer Type

Goal

A pointer data type (see Section 5.2.7) is needed to support certain types of processing performed in simulator executives (e.g.,

I/O request queues) and for use with dynamically allocated storage, which might be required in a digital image generation visual system. The facility provided should meet these requirements without providing excess generality at the expense of efficiency.

Supporting Concepts

3P. Pointer Type Definitions.

The pointer data type must be provided in a manner consistent with the static typing of the SHOL, thereby supporting program maintainability and reducing opportunity for error.

Requirements

3P-1. A pointer data type is required.

3P-2. Explicit specification of the type pointed to must be required for each pointer definition.

3P-3. Explicit dereferencing of the pointed-to value shall be required. [Dereferencing is the operation of accessing the object pointed to by a pointer value, e.g., P→A in PL/I, P↑.A in PASCAL, or A(P) in J3B. Requiring explicit dereferencing contributes to program understandability.]

*3P-4. It must be possible to define objects which are dynamically allocated. [See 3R-1 below.]

Language Evaluations

A pointer data type is provided by all of the candidate languages except FORTRAN. PASCAL pointer definitions do not specify the type pointed to, and in J3B such specification is permitted but not required. The only languages which require explicit dereferencing are J3B and PASCAL, though it is available in J73I and PL/I also.

Only PL/I and PASCAL support dynamic allocation of data objects.

(Note that J73I pointers are declared and represented as integers. There is not an actual pointer type.)

3Q. Pointer Literals.

The concept of a null pointer must be representable in a distinctive and readable manner. Pointer constants specifying addresses of data objects are also required by the uses of pointers in simulator executives [see 3R-3].

Requirements

3Q-1. There shall be a NULL pointer literal. [This is the literal normally provided for pointer types.]

Language Evaluations

Of the languages supporting pointers, only J73I does not provide a NULL pointer literal.

3R. Pointer Operations.

Pointer operations supporting simulator requirements, as discussed in Section 5.2.7, must be provided in a manner compatible with the goal of object code efficiency.

Requirements

*3R-1. There must be operations for the allocation and deallocation of dynamic storage. [Explicit deallocation is required as opposed to garbage collection, because of the negative impact of garbage collection on efficiency.]

3R-2. Identity and non-identity relations must be defined on pointer types. [These are the equality operations for the pointer type.]

*3R-3. There must be an operation for converting from integer to pointer values. A program using this operation must be flagged by the translator, so use of this capability can be administratively controlled. [This capability is needed to provide some SHOL support software, as described in Section 5.2.7. Since its use can impair program understandability, its use should be highlighted. Such highlighting would not be possible if the capability were provided by an assembly language subroutine (see 10C).]

Language Evaluations

Only PASCAL and PL/I provide operations for allocation and deallocation of dynamic storage.

All of the languages which have pointers except J73I provide identity/non-identity operations. (J73I simply uses integers as pointers, so relational operators are available.)

None of the languages support explicit conversion from integer to pointer. (In J73I, since pointers are integers, the desired capability is, in a sense, provided.)

Language Evaluation Summary

The candidate languages are ordered, according to the degree to which they meet SHOL pointer data type requirements, as follows:

PL/I- Most requirements are met.

Deficiencies are:

- explicit dereferencing not required
- no conversion from integer to pointer

PASCAL- Most requirements are met. Pointers are not bound to a type.

Deficiencies are:

- pointer definitions do not specify the type pointed to
- no conversion from integer to pointer

J3B- No dynamically allocated data objects are provided.

Deficiencies are:

- specification of type pointed to not required
- no dynamically allocated objects
- no conversion from integer to pointers

J73I- Pointers are really integers. Dynamically allocated objects are not provided.

Deficiencies are:

- pointers are integers
- explicit dereferencing not required
- no dynamically allocated objects
- no NULL literal

FORTRAN- There is no support of pointer data.

6.3.7 Procedure Types

Goal

The SHOL must provide procedure variables in order to support current methods of foreground task dispatching, as described in Section 5.2.8. This feature must be provided in a manner that does not increase the likelihood of programmer error nor add unneeded complexity to the language.

Supporting Concepts

3S. Procedure Type Definitions.

Specification of procedure variables in the SHOL must express clearly the intent of the programmer.

Requirements

3S-1. A procedure data type is required.

3S-2. Explicit specification of the number and types of arguments shall be required for each procedure variable definition and must be considered part of the variable's value type. [Hence, procedures having different numbers or types of arguments cannot be assigned to the same procedure variable; see 2D-2. Specifying the argument types helps to prevent error and makes programs more understandable.]

Language Evaluations

Only PL/I supports procedure variables. Explicit specification of parameters for such variables is not required, however.

3T. Procedure Operations.

Operations provided for procedure variables must meet the requirements of the task dispatching operation (see Section 5.2.8) and must be consistent with the rest of the language.

Requirements

3T-1. There must be equality and inequality operations between elements of procedure type. [These operations are required for uniformity.]

Language Evaluations

PL/I provides equality and inequality operations for procedure types.

Language Evaluation Summary

The candidate languages are ordered, according to the degree to which they meet SHOL procedure data type requirements, as follows:

PL/I- Provides procedure variables.

Inadequacies are:

- parameter specification is not part of the variable's value type

FORTRAN, J3B, J73I, PASCAL- Procedure variables are not provided.

6.3.8 Array Types

Goal

The SHOL must provide arrays, or composite data types with indexible components of homogeneous type. Arrays must be supported in a manner which contributes to program understandability and efficiency, and which meets simulator programming needs as discussed in Section 5.3.3.

Supporting Concepts

3U. Array Type Declarations.

Arrays of the various data types must be specifiable in a manner consistent with other type definitions in the language.

Requirements

*3U-1. Arrays with components of any scalar [including procedure] or composite type shall be definable. [Arrays of procedures are required to support current methods of foreground task dispatching.]

3U-2. Accuracy specifications [see 3A] shall be required for components of appropriate numeric type.

3U-3. The number of dimensions for each array variable must be specified in programs and shall be determinable at translation time. [No need for a variable number of dimensions was observed.]

*3U-4. At least three dimensions are required. [Required for 3-variable linear function interpolation.]

*3U-5. The range of subscript values for each dimension must be specified in programs and need only be determinable at translation time [i.e., specified with constant values; however, see 7E-2 for array parameters. No need for arrays with varying bounds was observed.]

3U-6. The range of subscript values must be restricted to a contiguous sequence of integers, the elements of an ordered enumeration type, or a sequence of single characters. [These types of subscripts are sufficient for simulator needs.]

3U-7. The lowest bound of a sequence of integers defining the range of subscript values must be language-defined, rather than programmer-definable. [This simplifies the language, makes subroutine interfaces more efficient, and is adequate for the simulator application.]

Language Evaluations

All of the candidate languages provide an array data type, and all allow at least the required three dimensions. All languages permit any of their scalar types as array components. (Hence arrays of procedures are supported only in PL/I.) Only PASCAL and PL/I allow arrays of arrays. J3B allows only one-dimensional arrays of records, while other languages provide more general support. All languages require the same numeric accuracy specifications as are required for scalars of the same type.

The number of dimensions is fixed at compile time for all languages. All require that subscript ranges be determinable at compile time except PL/I, which determines ranges for automatic and controlled storage at time of allocation.

Of the candidate languages, only PASCAL allows subscripts to be of enumeration type or to be single characters. PASCAL also permits Boolean subscripts, which are not desired. Only FORTRAN and J3B have a language-defined lower bound.

3V. Array Literals.

The language must support the initialization of an array with constant values. This is necessary to create such data structures as the LFI breakpoint and value lists.

Requirements

3V-1. A constructor operation [i.e., an operation that constructs an element of a type from its constituent parts] is required for array types.

Language Evaluations

None of the candidate languages provide an explicit array constructor operation. (All except PASCAL allow initialization of arrays with lists of constants.)

3W. Array Operations.

Operations must be provided to allow the use of individual array components and of subarrays of arrays of records. Operations on entire arrays representing matrices and vectors are also required, for reasons discussed in Section 5.3.3.

Requirements

*3W-1. A value accessing operation for individual array components is required. [This is a fundamental array operation.]

*3W-2. Assignment to individual array components must be permitted. [This is a fundamental array operation.]

*3W-3. Operations for value access and assignment of subarrays consisting of a complete dimension of an array of record components are required. [This allows grouping of related record data, all of which is indexible by the same enumeration type, into a single array, while still allowing vector operations on a subarray. Section 5.3.4.2.2 illustrates this concept. Note that a uniform language will provide for selecting complete dimensions of any array type as well.]

3W-4. There must be array operations for matrix addition and subtraction, multiplication of a matrix by a scalar, multiplication of a vector by a scalar, vector cross-product, and vector dot product. [Such operations are heavily used in simulators, particularly in the Aerodynamics, Visual, and Tactics systems.]

3W-5. Equality and inequality operations on arrays are required.

Language Evaluations

All of the languages provide value access and assignment for individual array components. Only PL/I permits selection of subarrays consisting of a complete dimension of an array of records. None of the languages provide equality/inequality operations on arrays.

Matrix and vector operations are supported only by PL/I, and it supports only addition and subtraction.

Language Evaluation Summary

The candidate languages are ordered, according to the degree to which they meet SHOL array requirements, as follows:

PL/I- Most requirements are met.

Deficiencies are:

- single characters not usable as subscripts
- lower bound of subscript range defined by programmer
- no array constructor operation
- no equality/inequality
- no matrix multiplication operations

PASCAL- More subscript types are permitted. Fewer operations are provided.

Deficiencies are:

- permits Boolean subscripts
- lower bound of subscript range defined by programmer
- no array constructor operation
- no subarray selection
- no equality/inequality
- no matrix and vector operations

J3B- Several desired capabilities are not provided. More restrictions are imposed.

Deficiencies are:

- only one-dimensional arrays of structures permitted
- no arrays of arrays
- single characters not usable as subscripts
- no array constructor operation
- no subarray selection
- no equality/inequality
- no matrix and vector operations

J73I- Several desired capabilities are not provided. Array lower bound is programmer-defined.

Deficiencies are:

- no arrays of arrays
- no single character or enumeration type subscripts
- lower bound of subscript range defined by programmer
- no array constructor operation
- no subarray selection
- no equality/inequality
- no matrix and vector operations

FORTRAN- As structures are not supported, there are no arrays of structures. Desired subscript types are not provided as data types. Desired operations are not provided.

Deficiencies are:

- no arrays of arrays
- no equality/inequality
- no matrix and vector operations

6.3.9 Record Types

Goals

The SHOL must provide records, i.e., composite data types with labelled components of heterogeneous type [e.g., PL/I structures]. Simulator data organization could be greatly improved by the use of records, as illustrated in Section 5.3.4.

Supporting Concepts

3X. Record Type Declarations.

Records shall be provided in a uniform and consistent manner. A variant record type facility is required for some simulator functions.

Requirements

*3X-1. Records with components of any scalar [including procedure] or composite type are required. [Restrictions on component types degrade language uniformity.]

3X-2. Accuracy specifications must be given for each component of a real numeric type. [This is for uniformity with declaration of simple variables.]

*3X-3. It must be possible to define types as alternative record structures (i.e., variants). [Variants are used in simulators, for example, in the radio station and radar emitter data files.]

3X-4. The structure of each variant must be determinable at translation time. [This is inherent in the variant record concept.]

3X-5. Each variant must have a tag field (i.e., a component that can be used to discriminate among the variants during execution). [This is inherent in the variant record concept.]

3X-6. The tag field must not be directly assignable. [Assignment to tag fields changes the type of the variant. For reliability and understandability, such assignments must be restricted.]

3X-7. The tag field must be stored in the record, and its storage position must be controllable by the programmer. [Since variants are used to describe input data records, the programmer must be able to specify the tag field position in the input record. No use of untagged record variants has been observed.]

Language Evaluations

All of the candidate languages except FORTRAN provide a record data type. All allow components of any scalar type, and all except J3B and J73I allow components of array and record types. All require accuracy specifications as for scalars of the same type. Variant record types are supported explicitly by PASCAL, and less directly by J3B and J73I through an overlay feature. (PL/I's overlay capability provides some similar functions, but less explicitly.)

Only PASCAL's variants have a tag field. However, PASCAL does not require that the tag be stored in the record, and if it is, the programmer has only partial control over its position. PASCAL does not prohibit assignment to the tag field.

3Y. Record Literals.

The language must support the initialization of a record with constant values, in order to allow the creation of data tables such as the camera/modelboard altitude limit bit map (see Section 5.3.4.4).

Requirements

3Y-1. A constructor operation [i.e., an operation that constructs an element of a type from its constituent parts] is required for all record types. [Such an operation is used to initialize record variables.]

Language Evaluations

None of the candidate languages provide an explicit record constructor operation.

3Z. Record Operations.

The SHOL must provide operations to allow the use of individual components of records.

Requirements

*3Z-1. A value accessing operation for individual record components is required. [This is a basic operation of the type.]

*3Z-2. Assignment to individual record components that have alterable values [i.e., all except the tag field] must be permitted. [This is a basic operation also.]

Language Evaluations

All of the candidate languages provide access and assignment to individual record components.

Language Evaluation Summary

The candidate languages are ordered, according to the degree to which they meet SHOL record requirements, as follows:

PASCAL- Most basic capabilities, including variant record types, are provided.

Deficiencies are:

- tag need not be stored
- lack of programmer control over tag storage position
- tag field assignable
- no record constructor operation

J3B, J73I- Variants are not supported as requested. Component types are limited.

Deficiencies are:

- no components of array or record type
- no variants (overlays instead)

PL/I- An overlay capability is provided.

Deficiencies are:

- no record variants

FORTRAN- Records are not provided.

Section Summary

Ranking the languages by how well they satisfy the data type requirements gives the following ordering:

PL/I- acceptable support for all types but Boolean and enumeration

PASCAL- major deficiencies are lack of fixed point, bit string, and procedure types; and poor character support

J3B- acceptable support for all types except Boolean, enumeration, and procedure

J73I- lacks adequate support for fixed point, enumeration, Boolean, pointer, and procedure types

FORTRAN- fails to meet most requirements

6.4 Expressions

Goal

Expressions in the SHOL should be provided in a uniform manner. Appearance and interpretation of expressions should correspond to common usage when this does not conflict with other requirements. The FORTRAN background of simulator programmers is a consideration in the design of this feature.

Supporting Concepts

4A. Side Effects.

Expression evaluation should not alter the environment of the expression (i.e., repeated evaluation of the expression should produce identical results). If not, such programs are more difficult to understand and, hence, maintain correctly.

Requirements

4A-1. During expression evaluation, assignment must not be permitted to any variable. [Note that this prohibits functions from having assignable (i.e, output) parameters.]

4A-2. A function must not be permitted to change variables that are non-local to the function. [Note this makes functions free of side effects -- two calls with the same argument values will always produce the same result. This means compiler optimizers can produce much more efficient code for programs containing function calls. If a side effect is desired, a programmer must use a procedure with input/output arguments.]

Language Evaluations

None of the candidate languages prohibit functions from having assignable parameters, and hence from altering the environment of the expression containing the function call.

4B. Allowed Usage.

Language uniformity dictates that expressions, variables, and constants be usable in the same contexts as one another (wherever such usage is sensible).

Requirements

4B-1. Expressions of a given type must be permitted wherever both constants and variables of that type are allowed. [This is a uniformity issue.]

Language Evaluations

FORTRAN fails to meet this requirement. For example, variables or constants are required as loop start, increment, and end values.

4C. Constant Valued Expressions.

Constant valued expressions support program understandability and conditional compilation, as discussed in Sections 5.5.1.1 and 5.5.2.

Requirements

4C-1. Constant valued expressions (i.e., expressions whose operands all have a constant value or Boolean expressions having a constant value independent of the value of variables contained in the expression, e.g., B OR C where C is a constant name having the value TRUE) must be permitted wherever constants of the types are allowed. Such expressions must be evaluated at translation time, with target machine accuracy. [The use of constant expressions is discussed in Section 5.5.1.1.]

4C-2. Expressions containing function calls with constant arguments need not be considered constant valued expressions. [This constraint is to make compile-time evaluation of expressions simpler.]

Language Evaluations

None of the candidate languages provide full constant expression evaluation. J3B probably provides it to a greater extent than the others. For example, only J3B supports translation time evaluation of real and pointer expressions. None of the languages provide evaluation of enumeration type or character constant expressions. Only J3B and J73I provide evaluation of constant bit string expressions.

None of the languages appear to require that constant expressions be evaluated with target machine accuracy.

Language Evaluation Summary

The candidate languages are ordered as follows, according to the degree to which they meet SHOL requirements for expressions:

J3B- Provides more constant expression evaluation than others.

Deficiencies are:

- functions can assign to parameters
- incomplete constant expression evaluation

J73I- Some constant expression evaluation is provided.

Deficiencies are:

- functions can assign to parameters
- incomplete constant expression evaluation

PL/I, PASCAL, FORTRAN- Little or no constant expression evaluation.

Deficiencies are:

- functions can assign to parameters
- expressions sometimes forbidden where constants and variables can be used
- no translation time evaluation of constant expressions

6.5 Constants, Variables, and Declarations

Goal

The SHOL must allow declaration of constants and variables in a manner supporting program understandability. These features should be designed to facilitate translation-time detection of errors and to allow generation of efficient object code. These issues are discussed in Section 5.5.1.

Supporting Concepts

5A. Declaration of Constant Names.

A constant definition facility allows programmer intent to be expressed more explicitly, enhancing maintainability.

Requirements

*5A-1. The ability to associate identifiers with constant values of numeric, Boolean, character string, bit string, array, and record types is required. [See Section 5.5.1.1.]

5A-2. Type names must be interpreted as abbreviations for their values [i.e., two record type names having the same definition shall be considered equivalent. This rule is motivated by language design considerations. It leads to a simpler use of a language.]

Language Evaluations

Of the candidate languages, only J3B and PASCAL allow constant names. Each allows them for the scalar types of the language, but neither allows them for arrays or records.

5B. Declaration of Variables.

In the interest of program understandability and maintainability, all variables should be explicitly declared.

Requirements

5B-1. The value type of each variable must be specified explicitly. [Readability is more important than writability because of maintainability considerations; see Section 6.1.]

5B-2. The value type of loop control variables must be specified as part of the loop control statement. [This also is an understandability consideration.]

Language Evaluations

All of the candidate languages provide a means to declare the types of variables, but FORTRAN and PL/I do not require explicit declarations for all variables. None of the languages allow the type of the loop control variable to be specified in the loop control statement. (PASCAL and J73I require that it be declared explicitly in the same manner as other variables.)

5C. Scope of Declarations.

Name scoping rules should not be more complex than required by the simulator application, in order to allow efficient implementation.

Requirements

5C-1. It must be possible to declare variables whose scope is at most an entire subroutine body.

5C-2. The scope of explicit declarations (except for loop control variables) is not required to be a unit smaller than a subroutine. [This simplication appears to be adequate to meet simulator needs, considering the current use of FORTRAN.]

5C-3. The scope of a loop control variable must be the loop control statement and loop body [see also 6F]. [Having this rule permits efficient loop code to be generated with less complex optimization processing.]

Language Evaluations

All of the candidate languages permit the declaration of variables whose scope is a subroutine body. PL/I provides smaller name scope units, a feature which is not required.

ANSI FORTRAN loop control variables are accessible only within the loop. This is not true of any of the other languages.

5D. Restrictions on Values.

The types of values assignable to variables should be those necessary to support simulator programming. (For example, as indicated in Section 5.2.8, procedure variables are necessary for the foreground task dispatcher.) Excess generality, at the expense of efficiency and reliability, is not desired.

Requirements

5D-1. Labels and statements must not be assignable to variables, computable as values of expressions, or usable as parameters to procedures or functions. [Having the forbidden capability encourages complex central flow, making programs more difficult to understand.]

5D-2. Procedures and functions are not required to be usable as parameters to procedures or functions, or returnable as function values. [The need for subroutines as parameters was not observed in our examination of simulator programs.]

Language Evaluations

FORTRAN allows the assignment of labels to variables in the assigned GOTO statement. J73I allows labels to be used as parameters. PL/I provides a general label variable type, with use as expression values, parameters, function values, etc., PASCAL and J3B permit none of these undesired uses of labels.

J73I, PASCAL, and PL/I allow procedure parameters, which are not required. None of the languages allow procedures as function values.

5E. Storage Classes.

As described in Section 5.1, both static and automatic storage classes are required to support simulator data organization techniques. These facilities should resemble those in other commonly-used languages and should facilitate coordination between members of the programming group.

Requirements

5E-1. The ability to statically allocate variables local to compilation units is not required. [Some simulator customers specify that static storage not be used.]

5E-2. It must be possible to statically allocate storage for variables which are external to compilation units. [Such data is required to support the 'datapool' concept used in simulator development, in which data is available to the various compilation units comprising the system; see Section 5.1.1.]

5E-3. It must be possible to have storage for variables local to a subroutine initialized (and possibly allocated) on each entry to the subroutine. [Values of such variables are not preserved from one execution of a scope to the next; see Section 5.1.2.]

Language Evaluations

All of the candidate languages except PASCAL provide static storage external to compilation units. (The FORTRAN COMMON and PL/I external data concepts are not as similar to the 'datapool' facility as is the JOVIAL COMPOOL concept.)

All of the languages also provide automatic storage local to subroutines. (FORTRAN provides this only in the sense that entities which are not initialized and which are assigned to in the subroutine are undefined on RETURN from the subroutine. The storage is statically allocated.)

5F. Initial Values.

Since knowing the initial value of a variable is often important in understanding programs, a method of specifying initial values should be provided.

Requirements

5F-1. There must be no default initial values for variables. [Default initialization can require unneeded object code.]

5F-2. It need only be possible to initialize any variable with a constant value. [Initialization with expressions whose value is only known at run-time is an unneeded capability.]

Language Evaluations

None of the candidate languages include default initialization of variables (except in a few isolated cases, e.g., PL/I AREA data). All of the languages except PASCAL provide a means of explicitly initializing variables.

5G. Operations on Variables.

It must be possible to assign and use values of variables in a uniform manner.

Requirements

*5G-1. The assignment operation and an implicit value access operation shall be automatically defined for each variable. [Note that this includes scalar, array, and record variables. This requirement is for language uniformity.]

Language Evaluations

All of the languages provide assignment and value access operations for scalar types. Only PASCAL and PL/I permit assignment to arrays. These two languages also permit assignment to record variables, which is provided in only a limited manner in J3B and J73I. (FORTRAN does not have records.)

Language Evaluation Summary

The candidate languages are ordered, according to the degree to which they meet SHOL requirements for constants, variables, and declarations as follows:

J3B- Almost all major requirements are met. Assignment to composite types is not fully supported.

Deficiencies are:

- no constant names for arrays or records
- loop variable not declared in loop control statement
- loop variable not local to loop
- no assignment to composite variables

J73I- Constant names are not provided. Label parameters are allowed. Assignment to composite types is not fully supported.

- no constant names
- loop variable not declared in loop control statement
- loop variable not local to loop
- label parameters permitted
- no assignment to composite variables

PASCAL- Constant names are provided. There is no static storage external to compilation units. There is no way to initialize variables.

Deficiencies are:

- no constant names for arrays or records
- loop variable not declared in loop control statement
- loop variable not local to loop
- no external storage
- no initialization

PL/I- Constant names are not supported. Implicit declarations are permitted. Undesired capabilities are provided.

Deficiencies are:

- no constant names
- implicit declarations
- loop variable not declared in loop control statement
- loop variable not local to loop
- label variables (though some uses of these provide a needed capability, in the absence of a CASE statement)

FORTRAN- Few of the desired capabilities are provided.

Deficiencies are:

- no constant names
- implicit declarations
- assignment of labels to variables permitted
- no automatic storage allocation
- no assignment to arrays
- loop variables not declared in loop control statement

6.6 Control Structures

Goal

The SHOL must provide control structures for conditional, iterative, and sequential control. These are required by the types of processing performed in simulators. Conditional processing is particularly prevalent, as discussed in Section 5.4.1. Control structures should be designed to support structured programming and enhance readability, and should allow programmers to express concepts in a notation which is natural to them. Each control structure should provide a single capability.

Supporting Concepts

6A. Basic Control Facility.

The set of control structures should be simple, understandable, and easy to learn to use effectively.

Requirements

6A-1. Built-in control mechanisms should be of minimal number and complexity. [This is for simplicity.]

6A-2. Each must be distinctively introduced and delimited [e.g., IF-ENDIF, CASE-ENDCASE; this tends to make the structure of programs more readily perceivable.]

6A-3. Nesting of control structures must be allowed. [This provides a natural program structure.]

Language Evaluations

All of the candidate languages provide a reasonably simple set of control structures. FORTRAN's control mechanisms are the least complex, but they do not provide the desired capabilities. FORTRAN is also the only language which does not allow nesting of control structures (except for loops).

All of the languages are deficient in the syntax used to define the lexical extent of control structures. PASCAL requires a terminator for CASE clauses, but it is not distinctive. PL/I requires a non-distinctive terminator for loops. All other control structures are defined by compound statements, which are, of course, not distinctive.

6B. Sequential Control.

The method of indicating successive statements to be executed should encourage a uniform programming style and should minimize chances for programmer error.

Requirements

6B-1. There must be explicit statement terminators [as opposed to statement separators as in PASCAL, or no statement delimiters as in FORTRAN. Statement terminators have been shown in experiments to be less error-prone than separators.]

Language Evaluations

FORTRAN does not have any statement delimiters. In PASCAL, the delimiter separates rather than terminates statements. The other candidate languages, J3B, J73I, and PL/I, all have the required statement terminators.

6C. Conditional Control.

There must be facilities for selecting among various control paths based on a condition. Such facilities should support structured programming practices and enhance program maintainability. Complex conditional assignments are needed, for reasons discussed in Section 5.4.1.1.

Requirements

6C-1. The conditional control structures must permit selection of alternative control paths depending on either:

* • the value of a Boolean expression [IF-THEN, IF-THEN-ELSE; this is the basic conditional structure].

• a computed choice among labelled alternatives [indexed CASE; see Section 5.4.1.2].

6C-2. The language must specify the control action for all values of the discriminating condition used to select alternatives. [e.g., in an indexed CASE statement, there must be a language-defined action corresponding to any possible value of the index for which the programmer provides no specific action. Specifying the action ensures standardization among implementations.]

6C-3. The user may supply a single control path to be used when no other path is explicitly selected. [e.g., in an indexed CASE statement, an alternative may be specified which is selected when the CASE index does not match the label of any labelled alternative; such an alternative can contribute to program readability.]

6C-4. Index values may be of an exactly representable scalar type [integers, enumeration elements, character strings, or bit strings] and must be constant values. [The use of enumeration types as CASE indices is discussed in Section 5.2.3.3.]

6C-5. Alternatives may be associated with several index values or with a range of index values. [This capability is often convenient and contributes to program readability.]

Language Evaluations

None of the candidate languages provide all of the required conditional control facilities. Only J73I and PASCAL have a CASE statement. PL/I and J3B have no CASE statement, and FORTRAN has neither a CASE statement nor an ELSE component for IF-THEN statements. The indexed CASE statement of PASCAL meets the requirements more closely than that of J73I by requiring explicit labelling of alternatives with the index value. Only the J73I CASE statement allows specification of a control path to be taken if no other CASE path is explicitly selected, but such a specification is not required.

The PASCAL CASE statement permits an index value of any exactly representable scalar type, whereas J73I's version does not. Specifically, character values are not permitted as indices in J73I. Both languages allow an alternative to be associated with several index values, but only J73I permits index ranges.

6D. Conditional Expressions.

It must be possible to clearly and efficiently select alternative operands within arithmetic expressions, based on a condition. Simulator design and documentation involves extensive use of such a feature, as discussed in Section 5.4.1.1.

Requirements

*6D-1. Conditional expressions, allowing selection of alternative expression values based on the value of a Boolean expression, are required. [See Section 5.4.1.1.]

6D-2. The language must require the specification of the expression to be selected for all values of the discriminating condition [i.e., IF-THEN-ELSE].

6D-3. Nested conditional expressions are not desired [e.g., IF (IF...THEN...ELSE) THEN...ELSE...; such expressions quickly become unreadable].

Language Evaluations

None of the candidate languages allow conditional expressions.

6E. Conditional Compilation.

As discussed in Section 5.5.2 and in Sections 4.6 and 4.7, it should be possible to specify inclusion or exclusion of sections of code based on information available at translation time.

Requirements

6E-1. When the selected case for any conditional statement is determined by a constant expression [see 4C], it is required that only the selected path be compiled. [This is a means of obtaining conditional compilation capability.]

6E-2. When the selected alternative of a conditional expression is determinable at translation time, it is required that only the selected alternative be compiled.

*6E-3. A method of conditionally compiling declarations is required. [This supports program portability; see Section 4.7.]

Language Evaluations

Some form of conditional compilation is provided by three of the candidate languages -- J3B, J73I, and PL/I. Only J3B supports conditional compilation as specified in the requirements, i.e, by normal conditional expressions with alternatives determinable at compile time. J73I and PL/I provide special compile-time features for this purpose. However, the J3B facility does not allow conditional compilation of declarations, while those of J73I and PL/I do.

6F. Iterative Control.

An iterative control (e.g., loop) facility is necessary to support the general iterative processing requirements of simulator programming, and to provide a complete set of structured programming mechanisms.

Requirements

*6F-1. There must be an iterative control structure that permits a loop to be terminated before or after each execution of the loop body. [Termination at other points may be useful but is not required; the need for this control facility is derived from general language design considerations.]

*6F-2. There must be a control structure that iterates over enumeration types or over subranges of integers [e.g., the FORTRAN DO-LOOP; such a structure is used quite commonly].

6F-3. The value of the control variable must be accessible only as a constant within the control structure. [This makes it easier to optimize loops and avoids errors dependent on knowledge of the control variable's value when the loop is exited.]

6F-4. The control structure must permit zero iterations to be specified [e.g., DO FROM 1 TO N, where N is less than one; not providing this capability is a significant FORTRAN failure].

Language Evaluations

All of the candidate languages provide iteration over subranges of integers, and all but FORTRAN provide an indefinite iteration facility (e.g., WHILE or UNTIL). Only J73I and PASCAL have an enumeration data type, so only these languages include iteration over enumeration types.

Only PASCAL and FORTRAN loop control variables are read-only within the loops. In other languages, the control variable may be assigned to explicitly. All candidate languages except FORTRAN permit zero loop iterations to be specified.

6G. Explicit Control Transfer.

A "go to" statement is necessary, but its use should be restricted to encourage programs with an understandable control flow. Other types of explicit control transfers are not desired, for reasons of language simplicity.

Requirements

*6G-1. There must be an explicit mechanism for control transfer [i.e., the "go to"; the need for this feature is derived from general language design considerations].

6G-2. The "go to" must not permit transfer into loops or out of procedures. [Permitting such transfers is error-prone.]

6G-3. The "go to" must permit transfer from one case constituent to another. [This can make it easier to get efficient object code.]

6G-4. Control transfer mechanisms in the form of switches, designational expressions, label variables, label parameters, or alter statements are not desired. [These are considered to be error-prone in their use and contribute to complex program control flows that are hard to understand.]

Language Evaluations

All of the candidate languages include a "go to" statement. None restrict its use to the extent required. In particular, J73I, PASCAL, and PL/I allow transfers out of procedures, and FORTRAN and J3B allow transfers into loops.

Both languages with CASE control structures (J73I and PASCAL) allow transfers from one case constituent to another.

Only PASCAL and J73I limit explicit control transfer mechanisms to the "go to." FORTRAN and J3B allow switch or indexed "go to" constructs, and PL/I has label arrays. These features are provided to supply the capability which is supplied by the CASE statement in PASCAL and J73I, so they are not redundant.

Language Evaluation Summary

The candidate languages are ordered, according to the degree to which they meet SHOL control structure requirements, as follows:

J73I- All major functions except conditional expressions are provided.

Deficiencies are:

- non-distinctive syntax
- explicit labelling of CASE alternatives not required
- CASE indices of character type not allowed
- no conditional expressions
- conditional compilation not implemented in required manner
- assignment to loop control variable permitted
- transfer out of procedures permitted

PASCAL- All major functions except conditional expressions are provided. Conditional compilation is not supported.

Deficiencies are:

- non-distinctive syntax
- statement separators rather than terminators

- no way to specify 'ELSE' control path in CASE
- CASE alternatives cannot be associated with index ranges
- no conditional expressions
- no conditional compilation
- transfer out of procedures permitted

J3B- CASE statements and conditional expressions are not provided. (Switches provide a capability similar to the CASE statement.) Conditional compilation is supported by the compile-time determination of alternatives in regular conditional control structures.

Deficiencies are:

- non-distinctive syntax
- no CASE statements (switches instead)
- no conditional expressions
- no conditional compilation of declarations
- iteration over enumeration types not available (because there are no enumeration types)
- assignment to loop control variable permitted
- transfer into loops permitted

PL/I- CASE statements and conditional expressions are not provided. (Label arrays provide a capability similar to the CASE statement.) Conditional compilation is not provided in the required way.

Deficiencies are:

- non-distinctive syntax
- no CASE statements (label arrays instead)
- no conditional expressions
- conditional compilation not implemented in the required manner
- iteration over enumeration types not available (because there are no enumeration types)
- transfer out of procedures permitted

FORTRAN- CASE statements, conditional expressions, and IF-THEN-ELSE structures are not provided and the object of IF-THEN may only be a single statement. Conditional compilation is not supported.

Deficiencies are:

- non-distinctive syntax
- no statement delimiters
- no CASE statements (indexed 'go to's' instead)
- no ELSE component in IF-THEN
- no conditional expressions
- no conditional compilation
- no indefinite iteration facility
- iteration over enumeration types not available (because there are no enumeration types)
- specification of zero loop iterations not permitted
- transfer into loops permitted

6.7 Functions and Procedures

Goal

The SHOL must allow the specification and use of subprograms (i.e., functions and subroutines). This feature is necessary to support modular programming and is required by simulator system organization, as discussed in Section 5.2.9. It should be provided in a manner which contributes to program understandability and which facilitates the production of efficient object code.

Supporting Concepts

7A. Function and Procedure Definitions.

The subprogram capabilities provided should support current practices of simulator organization.

Requirements

*7A-1. A means of defining and invoking functions (which return values to expressions) and procedures (which can be called as statements) shall be provided.

7A-2. Neither recursion nor the nesting of function and procedure definitions is required. [Such usage was not observed and not considered necessary.]

7A-3. Reentrant procedures are required. [These are used by certain system subroutines in simulators, e.g., conversion routines.]

Language Evaluations

All of the candidate languages provide function and procedure capabilities. All except FORTRAN provide more capability than is required. In particular J73I, PASCAL, and PL/I allow nesting of definitions; and J3B, J73I, PASCAL, and PL/I allow recursion. (In J3B, only procedures designated as reentrant may be called recursively.) Only J3B, J73I and PL/I have reentrant procedures.

7B. Function Declarations.

Functions need only return those value types required in *simulator programming*, and should not add unneeded implementation complexity to the language.

Requirements

7B-1. If a function result is a composite value or a bit string, then restricting its size to a constant value is sufficient to meet SHOL requirements. [This significantly simplifies a language.]

*7B-2. Function results of all scalar types except character string and procedure are required. [Functions returning character string or procedures present some implementation problems. The need for such functions is not significant in simulator programming.]

7B-3. Function results of label or procedure types are not desired. [Languages permitting such types are significantly more complex in their semantics and use of these functions can easily lead to programs that are hard to understand.]

Language Evaluations

All of the candidate languages except J73I allow function results of all scalar types in the language (except character string and procedure). (Of course, not all of the languages have all of the scalar types required by the SHOL.)

PL/I allows function results of label type, which are not desired. Many of the languages provide result types not specifically required (e.g., character string).

7C. Formal Parameter Classes.

Read-only and read/write parameters should be distinguishable from one another, in support of program readability and reliability. The language should permit implementations to provide efficient calling sequences for common cases.

Requirements

7C-1. Two classes of formal procedure parameters are required:

a) input parameters, which act as constants that are initialized to the value of the corresponding actual parameters at the time of the call [i.e., assignment to such parameters is not permitted; this helps to reduce errors and can contribute to object code efficiency.]

b) input-output parameters, which enable access and assignment to the corresponding actual parameters.

7C-2. For input-output parameters, the corresponding actual parameter must be determined at time of call and must be a variable or an assignable component of a composite type. [This is to reduce errors and ensure efficient implementations.]

7C-3. The class of a parameter must be distinguishable in the form of the call statement. [This is to enhance understandability.]

7C-4. The language must permit input parameters to be safely passed either by value or reference, depending on which method is determined to be most efficient by an implementation. [This means that even when procedures are separately compiled, it must be possible to determine whether the value of an actual input argument can be modified by assignment directly to the variable serving as the input argument.]

Language Evaluations

Only J3B, J73I, and PASCAL allow specification of formal parameters as read-only or read/write (i.e., input or input-output). In PASCAL, the distinction is not apparent

in the call statement, however. Only J3B permits the implementation to select between call by value and reference for input parameters, but it does not do this safely.

7D. Parameter Specifications.

Relationships between actual and formal parameters should be expressed readably in the language. Parameter matching rules should agree with the typing philosophy of the language.

Requirements

7D-1. Procedure parameters must be positional [i.e., correspondence between formal and actual parameters is determined by position in the parameter list. Optional and keyword parameters are not required].

7D-2. The syntax for declaring types and attributes of formal parameters must be essentially the same as that for variable and constant declarations [to promote uniformity].

*7D-3. Parameters may be of any type, but procedure parameters are not required. [See 5D-2.]

7D-4. The accuracy of each formal parameter of appropriate numeric type must be specified. [This is uniform with the requirement for accuracy specification in other contexts.]

7D-5. The value type of each actual parameter must match that of the corresponding formal parameter. [This implies that the language must be designed so that this check can be performed at compile-time, since type interface errors are difficult to discover during program development and maintenance.]

Language Evaluations

All of the languages have positional rather than keyword parameters, and none allow optional parameters. All require a declaration format for parameters which is similar to that required for variable and constant declarations. Similar accuracy specifications are also required.

In general, the languages allow parameters to be of any type available in the language. All languages except J3B allow procedure parameters, which are not required.

All of the languages require some degree of correspondence between the types of formal and actual parameters. PL/I, which allows numerous implicit conversions between parameter types, diverges most widely from the requirements in this area. J73I also allows such implicit conversions. FORTRAN and J3B essentially require exact matching between formal and actual parameters. (The PASCAL language specification does not define parameter matching requirements.)

7E. Formal Array Parameters

It should be possible to pass array parameters efficiently as long as the flexibility necessary for simulator programming is supported.

Requirements

7E-1. The number of dimensions for formal array parameters must be specified in programs and fixed at translation time. [See 3U-3.]

*7E-2. The language must allow the determination of the subscript range for formal array parameters to be delayed until execution time, and to vary from call to call. [This is required for Linear Function Interpolation, as discussed in Section 5.2.9.3.2.]

7E-3. Subscript ranges must be accessible within function and procedure bodies without being passed as an explicit argument. [To avoid errors.]

Language Evaluations

All of the candidate languages require that the number of dimensions for formal array parameters be specified and fixed at translation time. Only FORTRAN and PL/I permit the determination of the subscript range to be delayed until execution time. Only PL/I makes the subscript range accessible within the procedure (through the HBOUND and LBOUND functions) without requiring that it be passed as an explicit parameter.

Language Evaluation Summary

The candidate languages are ordered, according to the degree to which they meet SHOL function and procedure requirements, as follows:

FORTRAN- Parameter access restrictions cannot be specified. Array parameter subscript range can vary. Exact parameter matching is required (though many types required by the SHOL are not supported). Excess capabilities are minimal. Reentrancy is not provided.

Deficiencies are:

- no parameter access restrictions
- subscript range of array parameters must be passed as a parameter
- no reentrant procedures

J3B- All major requirements except array parameters of execution-time determinable subscript range are satisfied. Exact parameter matching is required. Excess capabilities are minimal.

Deficiencies are:

- array parameter subscript range is fixed at compile time
- does not permit safe selection between value and reference parameter passing by the implementation

J73I- Array parameter subscript range is not determinable at execution time. Implicit conversion occurs in parameter passing.

Deficiencies are:

- implementation cannot select between call by value and call by reference for input parameters.
- implicit conversion in parameter passing
- array parameter subscript range fixed at compile time

PASCAL- Array parameter subscript range is not determinable at execution time. Parameter matching rules are undefined. Parameter access restrictions are not determinable in the call statement. Reentrancy is not provided.

Deficiencies are:

- parameter access restriction not apparent in call
- implementation cannot select between call by value and call by reference for input parameters
- parameter matching rules undefined
- array parameter subscript range fixed at compile time
- no reentrant procedures

PL/I- Array parameter subscript ranges can vary. Implicit conversions occur in parameter passing. Parameter access restrictions cannot be specified. Excess capabilities are provided.

Deficiencies are:

- labels permitted as function results
- no parameter access restrictions
- implicit conversion in parameter passing

6.8 Input-Output Facilities

Goal

Simulator programming requires file-level I/O as well as low-level, primitive I/O. Section 5.3.2 discusses simulator file usage; Sections 5.5.6 and 5.7 deal with low-level I/O requirements. File I/O should be provided through the SHOL, but low-level I/O is probably best provided by the development of appropriate assembly language library subroutines.

Supporting Concepts

8A. File Level Input-Output Operations.

Operations for manipulating logical files must be provided in a manner supporting program portability.

Requirements

*8A-1. Standard library subroutines for logical file I/O must be provided. These must include operations for

creating, deleting, opening, closing, reading, writing, and positioning logical files. [The need for all these operations was observed.]

8A-2. Library subroutines for formatted I/O must be provided. [Formatted I/O is useful in offline work; see Section 5.3.2.]

8A-3. Binary record files of types sequential, indexed, and direct are required. [Use of all these file types was observed.]

8A-4. Blocks of fixed or variable length are required. [The need for variable length blocks is consistent with the need for variant records.]

8A-5. Files must be accessible in read-only, write-only, or update mode. [Use of these modes was observed.]

8A-6. Shared file operations are not desired. [Unneeded complexity.]

Language Evaluations

J3B and J73I provide no file I/O capability. The PASCAL file I/O feature is a primitive one which does not really meet any of the requirements.

FORTRAN and PL/I both provide file I/O facilities. FORTRAN supports only sequential files, not indexed or direct. Both languages support formatted I/O.

FORTRAN supports only fixed length blocks and does not allow specification of file access restrictions (i.e., read-only, write-only, update). PL/I supports shared file operations, which are not desired.

8B. Operating System Independence.

In support of program portability, the SHOL must not assume the presence of an operating system.

Requirements

8B-1. The form and meaning of built-in and standard library definitions shall not be restricted to any given operating system's capabilities, if one is present. [Note that functions and operators of the language can be implemented as operating system calls where the operating system is compatible with the function or operator definition.]

Language Evaluations

None of the candidate languages I/O features require the presence of a particular operating system.

Language Evaluation Summary

The candidate languages are ordered, according to the degree to which they meet SHOL I/O requirements, as follows:

PL/I- All required functions are provided.

Deficiencies are:

- support of shared file operations

FORTRAN- Some required functions are provided.

Deficiencies are:

- no indexed or direct files
- no variable length blocks
- no file access restrictions

PASCAL- Little file I/O support is provided (though the primitives could be used to build the desired functions).

Deficiencies are:

- file I/O support is too primitive

J3B, J73I- No I/O is provided.

6.9 Parallel Processing

Goal

The SHOL must support the use of multiple processors, as this is necessary to achieve the execution speed required for simulation. Simulator executives, which handle inter-CPU communication and data sharing, should be programmable in the SHOL. However, since this is an evolving area of language design with little consensus on how SHOL requirements are best supported, satisfying both these specific requirements and the general requirements (Section 7.1) may be beyond the current state-of-the-art.

Supporting Concepts

9A. Inter-CPU Communication.

The language must support control flow between CPUs necessary for simulators, as described in Section 5.4.2.

Requirements

*9A-1. It must be possible to initiate execution of a specified procedure on another CPU, to halt another CPU and to release another CPU from a wait state [see Section 5.4.2].

Language Evaluations

PL/I is the only one of the candidate languages to provide any parallel processing primitives. They are, however, perhaps too high-level to meet the specific requirements.

9B. Mutual Exclusion and Synchronization.

Processes executing on different CPUs must be able to access system data in a non-conflicting manner. There must be support for synchronization of processes executing on different CPUs, as discussed in Section 5.4.2.

Requirements

*9B-1. There must be mechanisms for mutual exclusion and synchronization of processes executing in parallel. [These are the HOL forms of primitives currently defined in assembly language.]

*9B-2. During specified portions of its execution, a parallel process must be able to seize and release certain program declared objects. [This is to ensure that variables are read and updated in a consistent state.]

*9B-3. The mechanisms provided must be sufficiently general to permit user construction of more specialized mechanisms that exploit knowledge of the overall behavior of the system being programmed [e.g., that pre-empting an executing process may not be required because interrupts are treated on a polled basis; this requirement is to ensure that the necessary level of executive efficiency can be obtained].

Language Evaluations

Again, only PL/I provides any support in this area, through its EVENT variables and associated SIGNAL and WAIT statements.

9C. Real-Time Clock.

Access to a real-time clock is necessary to support the cyclic operation of simulator programs. Access should be provided in a machine-independent manner.

Requirements

*9C-1. There must be means of accessing a real-time clock. [Real-time clocks are used for various purposes, as discussed in Section 5.4.2.]

9C-2. There must be translation-time constants to convert between the implementation units and the program units for the clock [supports program portability].

Language Evaluations

The PL/I TIME function returns the time in machine-independent units (hours, minutes, seconds, and milliseconds). However, this function presumably accesses a time-of-day clock, rather than a real-time clock (which can be set to a specific time and will interrupt on completion) as desired in simulator programming. The PL/I DELAY statement, allowing a task to be delayed for a specified time interval, provides more nearly the desired capability.

Language Evaluation Summary

The candidate languages are ordered, according to the degree to which they meet SHOL parallel processing requirements, as follows:

PL/I- All requirements are supported to some extent.

Deficiencies are:

- support may be too high-level
- real-time features may not provide the desired control

J3B, J73I,
PASCAL,
FORTRAN- No parallel processing support is provided.

Deficiencies are:

- no support for inter-CPU communication
- no support for mutual exclusion and synchronization
- no real-time clock access

6.10 Specification of Object Representation and Optimization

Goal

The SHOL must provide programmer control of and access to the object code representation of programs. Control of object representation allows the programmer to make trade-offs between time and space efficiency, as discussed in Section 4.4. Access to object representations facilitates the production of portable programs, as described in Section 4.7.

Supporting Concepts

10A. Packing of Composite Types.

Control over packing of composite types must be provided, in order to allow the programmer to make time-space trade-offs. This should be machine-independent when possible, to allow program portability. The logical grouping of record data components should be independent of the record's physical structuring.

Requirements

*10A-1. The language must permit, but not require, programmer specification of degree of packing [e.g., tight, dense, medium, unpacked] in a machine-independent manner for composite data types [arrays and records].

*10A-2. For record types only, the language must permit, but not require, machine-dependent packing specifications [i.e., by actual bit positions. This is necessary to allow description of simulator I/O data.]

10A-3. It must be possible to specify the order in which components of record types are sequenced in storage, independent of the order in which the components

are listed in the record declaration. [This can contribute to understandability by permitting logically related components to appear close together in program text even though they might be physically separated.]

10A-4. For two objects to be of the same value type, they must have the same physical representation. [Thus packing changes the type of a variable for purposes of parameter passing or assignment, i.e., a formal and actual parameter must have identical physical representation specification.]

10A-5. The default size of numeric data must be dependent on the range specification, if given, and otherwise must be implementation-dependent. [This is a consequence of making range specifications optional.]

Language Evaluations

Of the candidate languages, only J3B, J73I, and PL/I provide programmer control over packing of composite types. Of these, only J73I and PL/I permit specification of array packing. Only J3B and J73I provide machine-dependent packing of record types. Neither permits the specified packing to be separated from the logical structure of the record, though serial or parallel organization may be specified.

Both J73I and J3B require that composites have the same packing to be of the same type. PL/I does not meet this requirement.

10B. Translation Time Constants and Functions.

Environmental enquiries, providing programmer access to characteristics of the object program representation, are needed to allow the development of more portable programs.

Requirements

10B-1. The language must permit the specification of object machine configuration constants indicating, for example, machine model, peripheral equipment, memory size, word length, etc. [These are used to state what environment a program is intended to execute in.]

10B-2. The language must supply translation time constants and functions which access implementation information including:

- maximum and minimum integer values
- negative number representation
- fixed point accuracy
- floating point precision, radix, and exponent range
- maximum string length
- default character set
- bits per character

Language Evaluations

Only J73I and PASCAL provide any environmental enquiry capabilities. Specifically, J73I provides

- word length
- memory size
- bits/word
- bits/character
- bits/pointer

PASCAL provides only the maximum integer value.

10C. Code Insertions.

Assembly language insertions are necessary to implement machine-dependent simulator functions (see Section 5.7) and sometimes to achieve the necessary efficiency in certain areas. Such insertions should be easily isolated from other code, in order to support program portability.

Requirements

*10C-1. The language must permit the definition of subroutines in assembly language. [See Section 5.7.]

10C-2. Other assembly language insertions are not desired. [Restriction of assembly language to subroutines allows control over the data accessed within the assembly code.]

10C-3. The language must minimize the need for code insertions [by providing sufficient flexibility and power in the HOL].

Language Evaluations

None of the candidate languages provide any assembly language insertion facility.

10D. Inline Procedures.

In order to support programmer control over space-time efficiency trade-offs, the language must allow subroutines to be either expanded inline or called as actual subroutines. Section 5.2.9.2 discusses the value of such a feature in simulator programming.

Requirements

*10D-1. The language must permit subroutines to be defined as 'inline' -- that is, the code is to be inserted directly into the program at the point of call, rather than called through a subroutine call mechanism. [See Section 5.2.9.2.]

10D-2. The 'inline' specification must be part of the definition or in a separate declaration rather than part of the call. [Identical calls for the two kinds of subroutines facilitate tuning for the desired time-space trade-offs, as only the definition need be changed.]

10D-3. Inline substitution must not change the logical effect of a program, but where substitution of actual for formal parameters permits conditional compilation of inline code, this must be done. [e.g., if F(X) is defined as IF X > 3 THEN...ENDIF, then F(2) would result in no code being compiled. This encourages the modularization of programs and support reusability; see Section 6.6.]

Language Evaluations

Only J3B provides inline subroutines. The 'inline' specification is not in the definition or the call, but in a separate 'inline declaration.' This serves essentially the same purpose as specification in the subroutine definition. J3B performs conditional compilation of inline code when parameter substitution permits, as required.

10E. Optimization.

The language shall permit efficient code optimizations.

Requirements

10E-1. Range specifications, when given, shall be assumed to be satisfied when performing code optimization. [This will encourage the use of range declarations.]

Language Evaluations

The only candidate language with any range specification capability is PASCAL, with integer subranges. The effect of such specification on optimization is not defined by the language.

Language Evaluation Summary

The candidate languages are ordered, according to the degree to which they meet SHOL requirements for specification of object representation and optimization, as follows:

J3B- Most major requirements, including that for inline procedures, are met. Array packing cannot be specified.

Deficiencies are:

- physical structure of records not specifiable independently of logical structure
- array packing not specifiable
- no environmental enquiries
- no assembly language subroutines

J73I- Major requirements, except that for inline procedures, are met.

Deficiencies are:

- physical structure of records not specifiable independently of logical structure
- all desired environmental enquiries not provided
- no assembly language subroutines
- no inline procedures

PL/I- No machine-dependent packing or inline procedures are provided.

Deficiencies are:

- no machine-dependent packing
- physical structure of records not specifiable independently of logical structure
- records with different packing are of same type (i.e., implicit packing conversions are performed)
- no environmental enquiries
- no assembly language subroutines
- no inline procedures

PASCAL- Packing specifications are not provided. Most other functions are also not provided.

Deficiencies are:

- no packing specifications
- little environmental enquiry capability
- no assembly language subroutines
- no inline procedures

FORTRAN- Few of the required functions are provided.

Deficiencies are:

- no packing specifications
- no environmental enquiries
- no assembly language subroutines
- no inline procedures

6.11 Libraries and Separate Compilation

Goal

As discussed in Section 4.1, the SHOL must support development of simulator systems by large groups of programmers.

Supporting Concepts

11A. Library Entries.

The language must provide for the use of common (or global) data definitions, subroutines, etc. by the various individuals developing the system.

Requirements

*11A-1. The language must support the use of an external library. [Such libraries are customarily in use today.]

11A-2. Library entries must include input-output packages, common pools of shared declarations, application oriented software packages, other separately compiled segments, and machine configuration specifications.

Language Evaluations

The J3B and J73I COMPOOLs meet this requirement *most closely. Other languages* provide some similar capabilities through their support of separate compilation (see 11B). FORTRAN's COMMON facility provides sharing of common data definitions.

11B. Separately Compiled Segments.

The language must allow separate compilation of programs, and must support their integration.

Requirements

*11B-1. The language must support the integration of separately compiled program segments into an operational program.

*11B-2. The language must allow definitions made in one segment to be used in another. [This supports the data-pool concept; see Section 5.1.1.]

Language Evaluations

J3B and J73I support the use of separately compiled segments via the COMPOOL facility. The other candidate languages provide explicit 'external' declarations to access external names (though in PASCAL this is an extension to the standard language).

11C. Restrictions on Separate Compilation.

In support of program integration, the language must prohibit inconsistencies among the segments being integrated.

Requirements

*11C-1. Separate compilation must not change the meaning of a program. [This simplifies the language, making separate compilation merely a development aid.]

11C-2. Translators must be responsible for the integrity of object code in affected segments when any segment is modified.

11C-3. Translators must ensure that shared definitions have compatible representations in all segments. [This is of considerable value in program development and maintenance.]

Language Evaluations

J3B's and J73I's checking of procedure parameters in the COMPOOL and PASCAL's external reference checking provide more enforcement of compatibility than do the FORTRAN and PL/I facilities. It does not appear that separate compilation affects meaning of programs in any of the languages. It also does not appear that any of the languages require that translators guarantee integrity of object code in affected segments when a segment is modified.

Language Evaluation Summary

The candidate languages are ordered, according to the degree to which they meet SHOL library and separate compilation requirements, as follows:

J3B, J73I- COMPOOL facility provides most of the required functions.

Deficiencies are:

- integrity of affected segments not guaranteed when a segment is modified

PASCAL- There is no COMPOOL-like facility. Parameters are checked in external procedure declarations.

Deficiencies are:

- inadequate facility for sharing common definitions
- integrity of affected segments not guaranteed when a segment is modified

PL/I, FORTRAN- There is no COMPOOL-like facility. Parameters are not specified in external procedure declarations.

Deficiencies are:

- inadequate facility for sharing common definitions
- integrity of affected segments not guaranteed when a segment is modified
- external procedure parameters not checked

6.12 Language Evaluation Summary

A summary of the language evaluations is contained in Table 1. For each area of the requirements, each language is given an overall rating based on how well it satisfies the requirements. Languages that satisfy the essential (i.e., the starred) requirements are rated higher even if they do not satisfy the non-essential requirements, since the purpose of this evaluation is to decide which unmodified language best satisfies simulator requirements. The Table indicates that PL/I and J3B are most suitable, although no language is perfectly suited (a perfect language would have a score of 90). Of the two languages, PL/I is the more widely known, although neither PL/I nor J3B is significantly supported by manufacturers of computers used in training simulators. Also, neither language is approved by DoD for use in new embedded computer application efforts. Only FORTRAN and JOVIAL J73I are approved languages. From an Air Force viewpoint, JOVIAL J73I would therefore be the best choice if it were more widely available. On technical grounds alone, however, PL/I or J3B are somewhat superior to J73I. The only language that is clearly inferior is FORTRAN.

Since all of the languages have some important deficiencies, in the next Section we will discuss what language is most suitable for modification and how well the modified language would meet simulator requirements.

TABLE 1

EVALUATION SUMMARY

	PL/I	FORTRAN	PASCAL	J3B	J73I
General Syntax	*	***	***	*****	***
Numeric Type	*****	***	***	*****	***
Enumeration Type	.	.	*****	.	*
Boolean Type	.	*****	*****	.	.
Character Type	*****	.	*	*****	*****
Bit String	***	.	.	*****	*****
Pointer	*****	.	*****	***	*
Procedure	***	.	.	.	.
Array	*****	***	***	***	***
Record	***	.	*****	***	***
Expressions	*	*	*	***	***
Declarations	***	*	***	***	***
Control Structures	***	*	*****	***	*****
Procedures	***	*****	***	***	***
I/O	*****	***	*	.	.
Parallel Processing	***	.	.	.	.
Object Program Cntrl	***	*	*	*****	***
Library Facilities	***	***	***	*****	*****
	54	29	47	51	46

***** = Good, *** = Medium, * = Poor, . = Nonexistent

Section 7

LANGUAGE MODIFICATION GOALS

Since none of the languages are entirely suitable for simulator programming, we will consider what language is best suited for modification and what modifications should be made. In selecting a language for modification, in determining the modifications to be made, and in actually modifying the language, several goals must be given consideration. These goals are discussed in the following subsections. Specific modifications to each of the candidate languages are discussed in Section 8.

7.1 Minimal Cost

It is, of course, desirable to minimize the costs associated with language modification, i.e., the design, implementation, and retraining costs. This consideration dictates that possible modifications be evaluated with respect to their value and necessity for simulator programming vs. their cost before they are recommended. Each modification selected adds to the cost of producing the SHOL. In addition to the individual cost of implementing each modification, as the number of modifications increases, the complexity of the overall modification task multiplies. This is because of the effect of each addition or deletion on the remainder of the language. There are many interrelationships between the features of a language which must be carefully considered when deleting a feature and which must be defined when adding features. With a large number of modifications, interactions can become so complex that it might no longer be cost effective to modify an existing language as opposed to simply developing a new one.

Another area of cost consideration is the availability of existing support for the language selected as a basis for modification. If translators for the chosen language for the desired target machines (or some of them) already exist, the cost of implementing a set of SHOL translators is greatly reduced. (Even if no such translators are available, however, the cost of implementing the SHOL through modification of an existing language should be less than that of developing a new language. This is because language design should be less difficult and because existing knowledge about implementing the language can be employed.)

Quality of existing documentation for the selected base language is another cost factor. In establishing the SHOL as the language used by simulator programmers, a significant retraining effort will be required. This will require tutorial and user documentation of excellent quality. The degree to which existing base language documentation can be adapted to this purpose has a significant impact on cost. Another consideration involving documentation is the availability of a detailed and accurate language specification for the chosen base language. Such a document facilitates the design (and specification thereof) of the language built on that base, thereby reducing language design costs.

In view of these factors, we have considered only language modifications satisfying either of the following criteria:

a. They are essential to satisfy functional requirements of significant importance in simulator programming, e.g., fixed point arithmetic. These are the requirements asterisked in Section 6.

b. They are relatively easy to provide and are of significant benefit, even through not absolutely essential to meet simulator needs.

A language that is modified according to these criteria will be optimal in the sense that the benefits of the modifications probably outweigh the difficulty of making them.

7.2 Syntactic Integrity

It is necessary when modifying a language to conform to its existing syntactic conventions, i.e., added features must employ a syntax which is compatible and consistent with that of existing features. For example, if conditional expressions (e.g., x = IF condition THEN y ELSE z) are added to a language, their syntax should conform to the language's existing IF-THEN-ELSE construct as much as possible. This would not be at all possible in FORTRAN, which does not have an ELSE component in its IF statement. In general, the more the base language differs from the desired language, the more difficult it is to maintain syntactic integrity.

When selecting a language for modification then, it is important to consider how closely the syntactic conventions of the language correspond to those considered desirable for the SHOL. Furthermore, the goal of consistency with existing syntax must play an important part in the actual design of the SHOL.

7.3 Non-Interference with Existing Language Features

A similar goal to that described above is the avoidance of complex or undesirable interactions between modifications and existing features. As discussed previously, this problem is compounded as the extent of modification increases. Deletion of features considered to be undesirable can have a severe impact, since the feature may be needed in the semantic definitions of other aspects of the language, perhaps in a manner which is not immediately apparent. For example, deletion of a data type can have an impact on the implicit conversion algorithms employed in the language.

Additions must also be evaluated with respect to their interactions with the rest of the language. For example, if file I/O is to be added to PASCAL, it should be added in a manner consistent with PASCAL's existing I/O capability, which is very low-level. Perhaps the low-level primitives would be used to build the file I/O feature.

Additions should not introduce excessive redundancy into the language. For example, if a CASE construct is to be added to PL/I, it might be desirable to eliminate label arrays, which now serve somewhat the same purpose. On the other hand, it might be preferable to conclude that PL/I's label arrays are adequate for the purpose of simulator programming, thus avoiding both the addition and the deletion costs.

7.4 Upward Compatibility

A possible goal in developing the SHOL is upward compatibility with the base language. This means that the base language is a proper subset of the new language. (This, of course, rules out the deletion of features from the base language.)

A requirement for upward compatibility increases the difficulty (and hence the cost) of modifying the language. It is sometimes quite complex to extend the language syntax to incorporate desired additions without altering syntax of existing constructs, especially if any degree of syntactic integrity is to be preserved. As an example of this problem, consider the difficulty of adding to FORTRAN an IF-THEN-ELSE construct which allows groups of statements as objects of the THEN and ELSE, and which still accepts such FORTRAN statements as "IF (J. LT. 10) Q = R + S".

The primary advantages to upward compatibility are that programs in the base language will be accepted (and correctly translated) by the translator of the new language and that programmers trained in the base language can convert more readily to the new language. These advantages, however, are only realizable if there is a significant body of existing code in the base language which is to be reused in systems built with the new language and if programmers are already experienced with the base language. (Of course, it may be that programmers skilled in the base language would show resistance to the new features, and hence take longer to become proficient in the new language than those previously unfamiliar with the base.)

The major impact of the issue of upward compatibility is on the actual task of language modification. In general, it increases costs and detracts from the uniformity of the resulting language and should only be required if significant benefits will be obtained.

Section 8

LANGUAGE EVALUATION AND MODIFICATION SUMMARIES

Since all of the languages would benefit from modifications to make them more suitable for simulator programming, in this Section we discuss the modifications considered most cost effective. For each language, we will cite its major advantages and then discuss the modifications that are recommended. The modifications have been selected based on the analysis in Section 6. In general, modifications needed to make a language satisfy essential SHOL requirements are specified. Other modifications that are relatively simple to make and that would be of significant value are also proposed.

Each modification is evaluated in terms of its design and implementation complexity. Design complexity is increased if the modification requires changes to many parts of a language, i.e., if it affects the syntax and/or semantics of a significant proportion of constructs in the language. Design complexity is decreased if a modification is localized with respect to the capabilities a language provides and if the modification does not entail discarding existing language features. Implementation complexity is concerned with the amount of effort needed to modify or create a compiler that supports the modification. The design and implementation complexities are not always the same, for reasons that will be noted in discussing the modifications. After recommending modifications to all the languages, we will summarize the estimated design and implementation complexities of the modifications and discuss which language is best suited for modification and subsequent use.

The design and implementation complexities are evaluated on a scale of 1 to 5, with 1 indicating that the modification is simple and 5 indicating the modification is complex. A modification of the form:

(3, 5)* add fixed point

means that the design complexity factor is 3, the implementation complexity is 5, and the requirement for fixed point was considered essential in Section 6.

FORTRAN Modifications

The major advantages of FORTRAN are that it is available on many simulator target machines, many simulator programmers are experienced in its use, it is well documented (many tutorial publications exist), and it is on the DoD list of approved languages (DoDI 5000.31).

Its major disadvantages are its lack of fixed point and record types, its lack of control over data representations and its weak control structures.

The recommended modifications to FORTRAN are:

(1, 3) increase identifier length

The current length is inadequate to provide readable and meaningful identifiers. Making identifiers longer is a simple language change, but would require reorganization of a basic part of a FORTRAN compiler, the symbol table.

(3, 5)* add fixed point

Fixed point arithmetic capabilities are complex to design and complex to support because of optimization requirements.

(3, 2)* add enumeration types

As new data types are added to FORTRAN, the methods of declaring variables gets more awkward, necessitating perhaps more design effort than in other languages. The implementation of enumeration types is relatively straightforward, however.

(3, 3)* add character data type and operations

This should be straightforward in detail, although the number of details to be designed rates in complexity as 3.

(2, 1)* unprintable characters in strings

Deciding how to do this in a way that is consistent with the rest of the language will require some thought, but its implementation should be easy.

(3, 2)* ability to define new character sets

The design is difficult (it has not been done except in the DoD Common Language designs) but since the design decisions are localized to one type in the language, we rate it a 3 in language complexity. The implementation ramifications are potentially no more complex than handling packed arrays, a capability required for other reasons as well.

(3, 3)* add bitstring type and operations

We generally rate adding a completely new type and operation as complexity level 3.

(2, 3)* add pointer type

The syntax and usage of pointers can be relatively straightforward to design as a modification.

(1, 3)* add dynamic allocation

Providing simple dynamic allocation operators poses no difficulties, but the run-time support is more of a complication.

(4, 3)* add record data type

The record type is complex and so we rate it 4.

(3, 3) add constant expressions evaluated at compile time

Modifying syntactic rules so constant expressions are permitted in every context currently permitting literals makes this modification more complex than might otherwise be expected. The implementation is complex because of the need to simulate target machine arithmetic, potentially, or at least to compile or interpret expressions being evaluated at compile time.

(2, 1) correct non-uniformities in use of expressions

The constraints on use of expressions as subscript indices can be easily removed, improving the uniformity of the language.

(2, 2)* add constant names

This is relatively straightforward.

(4, 3) add automatic storage class

This is a significant change to FORTRAN but is worthwhile in simplifying storage allocation problems for what are currently treated as temporary common data locations.

(4, 2) permit nested IF-THEN-ELSE statement forms

This is a significant change to FORTRAN and requires deleting the current FORTRAN form to avoid duplication of capabilities.

(2, 3) add CASE statement

This is a significant change to FORTRAN concepts, since it adds the concept of a compound statement to the language, but given that the concept is already needed for IF-THEN-ELSE statements, the additional effort for a CASE statement is not great insofar as the language design goes.

(2, 2)* add conditional expressions

The semantic rules are straightforward if both arms of the conditional are required to be of the same type, e.g., if IF B THEN 3.0 ELSE 2 is forbidden.

(2, 2) add conditional compilation

Given that IF-THEN-ELSE statements are in the language, this is not a significant amount of work to add and it is very useful for reasons discussed in Section 4.7.

(2, 2)* add indefinite loop iteration

Adding DO WHILE loops is fairly straightforward.

(2, 1) add constant parameters

This is an easy change to the language specification even though it is a significant change to the capabilities of FORTRAN.

(2, 4) add indexed and direct access files

These can be added as library procedures and, indeed, have been made available in this way in some FORTRAN implementations.

(2, 3)* add parallel processing support

Simple primitives can be added as procedural extensions.

(3, 3)* permit programmer packing specifications

The variety of packing specifications required and their use throughout the language is complex because of the widespread impact on the language.

(1, 2)* add assembly language and inline subroutines

This is straightforward once the decision about how to do it is made.

(3, 2)* add facility for sharing common datapool definitions.

The FORTRAN COMMON facility is usable for sharing data locations, but a more reliable method of sharing definitions similar to JOVIAL COMPOOL facilities would be a worthwhile change.

The recommended modifications have an estimated design complexity of 61 and an estimated implementation complexity of 63.

PASCAL Modifications

The major advantage of PASCAL is that it is designed with simplicity and reliable programming in mind. The number of PASCAL translators is increasing, but they are not well controlled, leading to translator-dependent PASCAL dialects. Very little must be deleted from PASCAL to make it acceptable as a SHOL, but many capabilities must be added. The most significant PASCAL disadvantages are its lack of fixed point arithmetic, its lack of separate compilation facilities, and its lack of provision for controlling data representations.

Recommended modifications to PASCAL are:

(1, 1) add a break character in identifiers

Break characters permit more readable identifiers to be used in programs. The change is a simple one to make.

(1, 1) add MAX/MIN and trigonometric functions

Providing these capabilities in a library is straightforward.

(3, 5)* add fixed point (see FORTRAN discussion)

(1, 1) permit duplicate names for enumeration elements

This is a relatively simple change requiring that some method of resolving ambiguities be provided. Several language design options are possible.

(2, 1)* unprintable characters in strings (see FORTRAN)

(3, 2)* permit new character sets to be defined (see FORTRAN)

(1, 1)* incorporate a string data type

This is just a syntactic change, since the representation of such a type will be the same as the current string representation, namely, arrays of characters.

(1, 1)* provide base 2 or base 8 literals

This is a simple syntactic addition.

(1, 1) type-safety of variant records

PASCAL currently permits tag fields of variant records to be assigned directly. This capability is not required in simulator programming. Removing tnis capability is simple.

(3, 3) add constant expressions evaluated at compile time (see FORTRAN)

(1, 1) add constant records and arrays

Since PASCAL already supports constant names, extending the capability to records and arrays is straightforward.

(3, 3) add external static storage class

Since PASCAL does not support separate compilation, this is a significant change.

(1, 1) add initialization of variables

This is a simple language change. The implementation complexity is also straightforward.

(1, 1) add an ELSE alternative in CASE statements

(3, 2) permit ranges in CASE statements

Permitting ranges in CASE statements will require altering the CASE statement syntax in a way that requires some thought. Adding an ELSE clause is straightforward, however.

(2, 2)* add conditional expressions (see FORTRAN)

(2, 2)* add conditional compilation (see FORTRAN)

(3, 3)* permit array parameter subscripts to be non-constant

This is a significant change to the language capability. Its integration with the rest of the language requires careful design.

(1, 1) define parameter matching rules

This corrects an oversight in the current language specification.

(2, 4)* add direct and indexed access files (see FORTRAN)

(2, 3)* add parallel processing support

Simple primitives can probably be added as procedural extensions.

(3, 3)* permit programmer packing specifications (see FORTRAN)

(1, 2)* add assembly language and inline subroutines (see FORTRAN)

(4, 4)* add facility for sharing common datapool definitions

PASCAL currently has no separate compilation capability. Designing one for PASCAL and implementing it is therefore more complex than for the other languages.

The recommended modifications have an estimated design complexity of 46 and an implementation complexity of 49.

J73I Modifications

The major advantage of J73I is that it meets most of the essential SHOL requirements and it is on the DoD list of HOLs approved for use in developing new DoD software. Its major shortcoming is the lack of a fixed point data type and variant records.

The recommended modifications to J73I are:

(3, 5)* add fixed point (see FORTRAN)

(1, 1)* add MAX/MIN and trigonometric functions (see PASCAL)

(4, 3)* add enumeration type and operations

This addition is more difficult than for the other languages because the current status type capability must be modified, i.e., the modification requires both a deletion and an addition to the language.

(3, 2)* add Boolean data type and operations

Adding the Boolean data type will require removing the present method of computing Boolean results from the language (namely, the use of bitstrings to get the effect of Boolean operations). Although adding the Boolean type to a language not having it would be rated a 2 in design complexity, since this change to J73I requires modifying an existing capability, we rate the design complexity as 3. Implementation complexity is 2 because the Boolean operations are already present in the language; it's just the syntactic and semantic constraints that must be changed.

(3, 2)* ability to define new character sets (see FORTRAN)

(3, 3) make pointers a distinct data type

Pointers in J73I are considered integers. This use of integers does not contribute to reliable programming. Adding pointers as a distinct data type is therefore equivalent to adding a new type to the language.

(1, 3)* add dynamic allocation (see FORTRAN)

(4, 4) permit record variants to be defined

Adding this capability requires modifying J73I to forbid the use of OVERLAYs to obtain record variants and so this change is somewhat more complex than simply introducing record variants.

(4, 1) permit record components of array or record type

This capability requires a significant redesign of the JOVIAL record capabilities.

(2, 3) expand constant expression evaluation capabilities

No constant expression capability is currently provided except for bitstring expressions.

(2, 2)* add constant names (see FORTRAN)

(1, 2)* permit assignment of arrays and records as a whole

J73I already permits such assignments for records that are elements of arrays. Extending the capability to complete arrays and records is not difficult.

(3, 2)* add conditional expressions (see FORTRAN)

(3, 3) require explicit labeling of CASE alternatives

Requiring that CASE alternatives be labelled explicitly will significantly increase the reliable and understandable use of CASE statements. However, it is a more complex change than might be thought, since ranges of labels must be provided as well as simple constants.

(3, 3)* permit array parameter subscripts to be non-constant (see PASCAL)

(3, 5)* provide an I/O capability

Even though I/O can be supported by defining procedures, deciding what the routines should be and implementing them is a significant task.

(2, 3)* add parallel processing support (see FORTRAN)

(1, 2)* add assembly language subroutines and inline routines (see FORTRAN)

The recommended modifications have an estimated design complexity of 46; the estimated implementation complexity is 49.

J3B Modifications

The major advantage of J3B is that it has already been proved suitable for use in applications requiring efficient object code. Also, it meets most major SHOL requirements.

Recommended modifications to J3B are:

(1, 1) add MAX/MIN and trigonometric functions

(3, 2)* add enumeration types and operations

(3, 2)* add Boolean data type and operations (see J73I)

(2, 1)* unprintable characters in strings (see FORTRAN)

(3, 2)* ability to define new character sets (see FORTRAN)

(1, 1)* add base 2 and 8 literals

(1, 1)* permit bitstring equality/inequality

(1, 3)* add dynamic storage allocation (see FORTRAN)

(4, 4)* permit record variants to be defined (see J73I)

(4, 1)* permit record components of array or record type (see J73I)

(1, 2) extend expression evaluation

Change the language so relational comparisons (e.g., A = B) are considered constant expressions if A and B are constants.

(1, 1) add constant names for arrays and records

Since J3B already supports constant names, this is a simple modification.

(1, 2)* permit assignment of arrays and records (see J73I)

(3, 3) add CASE statement and remove SWITCH

This modification will improve the control structure capabilities of the language.

(2, 2)* add conditional expressions (see FORTRAN)

(3, 3)* permit non-constant array parameter subscripts (see PASCAL)

(3, 5)* provide an I/O capability (see J73I)

(2, 3)* add parallel processing support (see FORTRAN)

(1, 1)* permit specifiable array packing

(1, 1)* permit assembly language subroutines

The recommended modifications have an estimated design complexity of 41. The estimated implementation complexity is also 41.

PL/I Modifications

The major advantages of PL/I are that it is the most widely used of the candidate languages other than FORTRAN, it is well documented, and it provides most of the features required. The major shortcoming is that PL/I is more complex than is required.

Specific recommendations for modifications are:

(3, 2)* add enumeration types (see FORTRAN)

(3, 2)* add Boolean data type and operations (see J73I)

(2, 1)* unprintable characters in string literals (see FORTRAN)

(3, 2)* ability to define new character sets (see FORTRAN)

(1, 1)* add base 8 and 16 bitstring literals (see J3B)

(1, 2) add parameter specification for procedure variables

Since PL/I permits procedure variables, it is important to improve the reliability of this capability by permitting the types of the parameters to be specified.

(4, 3) permit record variants to be defined

Fitting the concept of record variants into PL/I will probably require a complete redesign of the PL/I record type, but the usefulness of variant records is sufficiently great to make this change worthwhile.

(3, 3) add constant expression evaluation (see FORTRAN)

(2, 2)* add constant names (see FORTRAN)

(3, 3) add CASE statement and remove label arrays (see J3B)

(2, 2)* add conditional expressions (see FORTRAN)

(2, 1) add constant parameters (see FORTRAN)

(3, 3)* permit programmer packing specifications (see FORTRAN)

(1, 2)* add assembly language and inline subroutines (see FORTRAN)

The recommended modifications have an estimated design complexity of 33 and an estimated implementation complexity of 29.

Overall Summary

The estimated design and implementation complexities for the recommended languages are summarized below:

Language	Design	Implementation
FORTRAN	61	63
PASCAL	46	49
J73I	46	49
J3B	41	41
PL/I	33	29

After the recommended modifications are made, each of the languages will be approximately equal in suitability for programming flight simulators. Since PL/I is the simplest to modify, it is a good choice as a base, especially since it was also evaluated as a suitable unmodified language.

Of the two languages approved by DoD for use in implementing new systems, namely, J73I and FORTRAN, J73I is clearly more suitable in both modified and unmodified form. More programmers are familiar with PL/I than J73I, however, and there are also more training materials for PL/I. On technical grounds, then, PL/I is the optimal choice for modification. However, the choice of J73I would not be unacceptable and would be more likely to be accepted within the Air Force environment.

In the next section we discuss implementation considerations for a standard simulator HOL based on modifying PL/I.

Section 9

IMPLEMENTATION CONSIDERATIONS AND RECOMMENDATIONS

The following subsections discuss various approaches to developing a workable SHOL facility to support simulator programming. Previous sections have justified the selection of PL/I as a base and have described the modifications recommended to develop the SHOL.

Self-Hosting vs. Cross-Compilation

Flight simulator systems have traditionally been based on a variety of different target machines -- generally commercially-available computers of moderate size. It is assumed that this practice will continue once the SHOL is in use. It has also been the custom to develop all software directly on the intended target machine. That is, language processors (assemblers and compilers) run on the target machine itself, and all program debugging is carried out on this machine. The main advantages to this approach are:

- there is no additional hardware cost over that required for the actual simulator
- programs may be modified (and reassembled or recompiled) at field locations where only the target machine is available

Another approach to simulator software development would be to use a single large-scale host computer for translation and for much of program checkout. This computer would have cross-compilers and debugging support tools for the various target machines. There are many advantages to this approach, including:

- the greater power of the host computer can be used to advantage in the translation and support programs
- much of the code in the cross-compilers can be reused in the various target machines
- more sophisticated debugging support can be provided
- more powerful facilities for editing, file maintenance, time-sharing, etc. are available for program development

These advantages, of course, must be weighed against the disadvantages of the added cost of the host computer and its unavailability for onsite modification.

The decision made with respect to self-hosted compiling vs. cross-compiling will have considerable interaction with other aspects of SHOL development, which will be discussed in later subsections. It is recommended that the cross-compilation approach be adopted for the SHOL. The language to be implemented is sufficiently complex that a more powerful computer is desired to support translation. This will allow development of a more sophisticated translator which can produce superior object code (in terms of space and time efficiency) to that produced by a translator operating on a small machine. This is an important advantage since efficiency is vital in this application area.

Another major reason for cross-compilation is the decreased cost of translator development. Compilers for each of the desired targets can share the same machine-independent portions (front ends) and will require only the development of new code generators. Code generator development is significantly less costly than total compiler development. As will be discussed later, this approach also facilitates language standardization by increasing the likelihood that all translators accept the same language.

A major concern with this approach is the initial cost of the host computer and of development of the first cross-compiler. Customarily simulator purchasers have not had to pay for separate development computers or for the production of language translators. It would not be reasonable to assume that the first purchaser of a simulator written in the SHOL should bear all of these initial costs. If the recommended cross-compilation approach is to be adopted, it will probably be necessary for simulator developers to obtain some specific support for the creation of a SHOL facility. Another problem is that onsite simulator modification cannot be readily supported except through appropriate time-sharing interaction with the host facility. Both of these problems are significant, but the current trend of DoD thinking, as reflected in the common language effort, is to provide such centralized support to DoD programming efforts.

Language Development

We have recommended that the SHOL be developed by modifying the PL/I language. Modifying an existing language leads to a simpler design effort than developing a new language. Reasons for this include:

- Most needed features are already available in the base language and need not be designed explicitly.
- Syntactic and semantic definitions of features already in the language are available, and are (hopefully) free of undesirable interactions.
- Existing language-defining documents can be expanded to define the new language.

Implementation of such a language is also simpler than implementing a new language, as discussed in the next subsection.

The modifications to PL/I which have been recommended were discussed in Section 8. All the modifications specify additions to PL/I. This indicates that design of a SHOL which is upward-compatible with PL/I might be reasonable to attempt, though it is not clear whether this is a worthwhile goal. The usual motivation for upward-compatibility is the reuse of existing code in the base language. As there is probably little or no existing simulator code in PL/I, this would not be a significant concern.

However, it may be desirable to develop a SHOL which is upward-compatible with PL/I for reasons of economy. As discussed in Section 7, the cost of development increases when features are deleted or when their syntax is altered, as well as when they are added. Thus it is probably most cost-effective to leave existing PL/I features as they are. This is particularly true if use can be made of existing PL/I translator code in developing the new translator.

If a more extensive language design effort is to be undertaken, a language satisfying more of the SHOL requirements can be developed. Many of the features of such a SHOL require significant changes to the basic syntactic and semantic conventions of the PL/I language (e.g., strong typing), and incorporation of such features into PL/I would not be practical. PL/I also contains many features which are superfluous for the SHOL. If a language exactly meeting the stated requirements were to be developed, it would probably be preferable to design an entirely new language rather than attempting such drastic modification to PL/I. Such an approach has not been recommended, however, because it is very costly and so is not an optimal way of providing a useful SHOL.

Translator Development

The recommended approach to SHOL development requires adding features to PL/I. This indicates modifying an existing translator is a possible approach. Factors influencing this decision include:

- whether PL/I compilers for the desired host and/or target computers are available
- whether the language to be implemented requires relatively few changes to the base
- whether the compiler considered for modification is well-documented and is implemented in an appropriate language in a readable and modifiable manner.

We have recommended that SHOL translators execute on a single large-scale host machine. This requires that the part of the compiler which is independent of the target machine be implemented only once, while machine-dependent portions (code generators) will be developed for each target computer. This is a more cost effective approach to the development of a set of SHOL translators for all intended targets than is the development of a self-hosted compiler for each target. Furthermore, it allows programming of the compiler in any language available on the host computer rather than requiring implementation in assembly language or FORTRAN, which are likely to be the only choices available on most simulator target machines. If a host is selected which already has a compiler for the base language (PL/I), the SHOL compiler can be implemented in the base language. (If the SHOL is designed to be upward compatible with PL/I, the compiler would then effectively be written in the SHOL, and could compile itself.)

Thus, if the compiler is ultimately to be written in the SHOL itself and/or if it is deemed worthwhile to make use of existing translator code, it would be desirable to seriously consider selecting a host facility which already has a PL/I compiler (generating code for the host machine). It is unlikely that the selected host would have PL/I cross-compilers already available for any simulator target computers, but existing compilers could be used for SHOL compiler implementation and existing compiler frontend code could be adapted to the various cross-compilers.

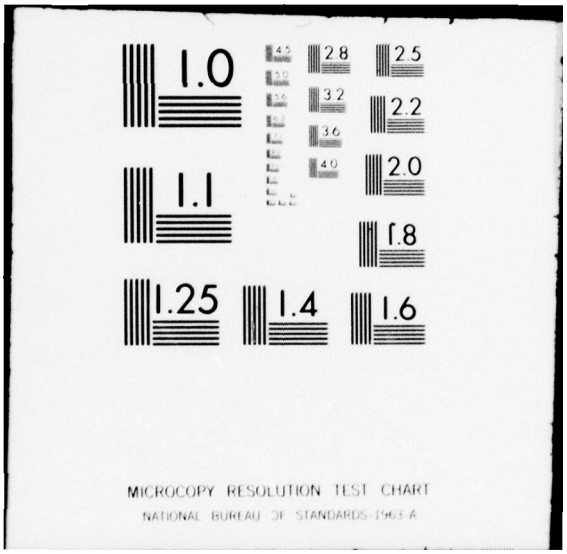
An implementation method which might also be considered is to use a preprocessor that would translate programs written in the SHOL into the base language. These programs could then be compiled using a compiler for the base language. However, this approach is most valuable if compilers for the intended targets already exist for the base language. Thus it might be reasonable if FORTRAN

AD-A058 994 SOFTECH INC WALTHAM MASS F/G 9/2
SIMULATION HIGHER ORDER LANGUAGE REQUIREMENTS STUDY.(U)
AUG 78 J B GOODENOUGH, C L BRAUN F33615-77-C-0029
UNCLASSIFIED AFHRL-TR-78-34 NL

3 OF 3
AD A058994



END
DATE FILMED
11-78
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

were the selected base, but it is probably not for PL/I. (Of course, the large number of modifications which would be necessary to extend FORTRAN to a SHOL make the use of a preprocessor unreasonable.)

Another consideration in developing a SHOL facility is the possible initial implementation of a "quick and dirty" SHOL translator to use in testing the feasibility of the language for simulator programming. Such a translator would translate and execute SHOL programs for test purposes only. This might be of some use in determining whether the language includes the constructs necessary for the programming of simulators. However, it would not test the single most important requirement of the SHOL translator -- whether it generates object code which is efficient enough for the application. It is likely that a SHOL based on the suggestions in this report will meet the functional requirements of simulator programming, and basing the SHOL on a widely-used language such as PL/I should guarantee its overall usability, so the test translator approach is probably not justified.

SHOL Programming Support

In addition to a language translator for the SHOL, certain support tools are necessary to facilitate the development of simulator systems. These tools generally aid in the integration of individual simulator programs into a total system, and in the debugging and validation of individual programs and of total systems. Section 5.5 discusses some such tools currently in use at Singer-Link.

Support programs may be divided into those which operate on the the program development (host) facility and those which operate on the actual simulator (target) facility. (Even if the same machine is used for both, such a conceptual division is reasonable.) It is intended that the SHOL be usable for the programming of all target-based support tools as well as for the programming of the simulator application programs.

Support tools which operate on the host machine should be considered part of the overall SHOL facility. In some cases, they may be incorporated in the SHOL translators, though they are not properly considered part of the language itself. Tools which might be developed as components of the SHOL facility include:

- editors
- program statistics collectors (e.g., instruction usage counts, time estimates for designated intervals)
- documentation aids

- set/used file capabilities
- link editors (for creation of target machine load modules)
- program maintenance tools
- application libraries
- program optimizers
- target machine simulators

Target machine instruction-level simulators are a particularly valuable program checkout tool made possible by the use of a separate host computer. Many debugging features which are difficult to provide on the actual target computer can be implemented easily in a target machine simulator. Examples of features such a simulator can provide are:

- mnemonic tracing
- interval timing
- interrupt modelling
- display of values of specified variables at specified time or location in the program
- trapping at a specified time or location
- setting of values of specified variables
- loading of test data sets

Debugging tools should encourage debugging in terms of symbolic program entities rather than machine values and addresses. Variables to be set or displayed should be referenced by name rather than by machine address, and values should be entered or displayed in units appropriate to the variable, rather than in octal or hexadecimal form. Recognizing that machine-level debugging is sometimes necessary, however, some support for this should be included.

Though some of the host-based support tools will differ from one target machine to another (e.g., machine simulators, link editors), user interfaces to the tools should be consistent. This is dictated by the goal of creating a unified SHOL development facility, rather than a miscellaneous collection of support programs.

Language Standardization

One of the major reasons for the development of a SHOL is the desire for program portability. As discussed in Section 4.7 there is particularly high potential for program portability in the simulator application area. Though there are some difficulties in attaining this goal (also discussed in Section 4.7), it can only be approached if the SHOL is standardized. That is, all SHOL translators (i.e., for all target machines) must accept the same inputs and must produce equivalent results for identical inputs.

Complete language standardization is difficult to achieve. The major prerequisite for language standardization is a complete and rigorous language specification document. An appropriate syntactic definition can be developed fairly straightforwardly, but, as indicated, full semantic specification is difficult. Standardization of the SHOL will be facilitated by selecting a base language with good defining documentation, and by limiting modification to that language. As mentioned earlier, use of cross-compilers with the same machine-independent part (front end) guarantees syntactic equivalence.

If language standardization is to be useful, it must be possible to validate that translators do in fact conform to syntactic and semantic specifications. This requires development of a rigorous set of acceptance programs to be used with all translators. The tests must not only ensure that programs compile without syntactic errors, but also that their semantics is as specified. This set of tests must be developed as a part of the SHOL design and specification effort.

Establishing SHOL Usage

Clearly there is little to be gained by developing a SHOL facility unless it will then be used. Also, though it will decrease development and maintenance costs on any single effort for which it is used, the full benefits of the SHOL will only be realized if it is used in all simulator programming. Only then will program reuse be a possibility, and only then will programmers become skilled in the use of the language.

While it is possible to guarantee use of the SHOL by simply requiring its use when procuring simulators, it is desirable to back up this requirement by making the SHOL facility a sufficiently attractive alternative that programmers will prefer to use it. Once programmers become conversant with the language, the increased ease of programming in the SHOL should be adequate motivation for its continued use. Initially, however, other factors will encourage the transition. These include:

- superior program development and debugging tools
- application library programs available through the SHOL facility

- high-quality documentation oriented to programmer background
- well-planned training effort

The importance of training and of user documentation in easing the transition to SHOL usage cannot be overemphasized, particularly in view of the large number of programmers involved. In addition to acquainting programmers with the use of the facility, a serious training effort assures them of management commitment to the changeover.

All tutorial material for the SHOL should include numerous examples illustrating how common simulator functions can be programmed in the SHOL. This will not only make SHOL usage easier to learn; it will also discourage excessive dependence on the assembly language subroutine capability. (It may be necessary, at least initially, to attempt to limit the use of this feature to functions for which it is really required. Section 5.7 discusses these requirements.)

Relation to the DoD Common Language Effort

The DoD is currently conducting an effort which will result in a Common Language to be used for the programming of embedded computer systems, including flight simulator systems. The IRONMAN specification defines the functional requirements to be met by the Common Language. The IRONMAN satisfies most of the essential SHOL requirements. The significant discrepancies between the IRONMAN and SHOL requirements are:

- IRONMAN does not require conditional expressions.
- IRONMAN does not require procedure variables and arrays.
- A non-exact fixed point representation is preferred to the exact representation required by IRONMAN.
- IRONMAN requires garbage collection of dynamically allocated storage, which adds unacceptable overhead. Explicit allocation and deallocation are desired and would have to be supported by extension to the IRONMAN language.

- IRONMAN does not require multiple fixed point precisions (which allow space-accuracy trade-offs).
- IRONMAN does not restrict assembly language use to subroutines.
- IRONMAN I/O and parallel processing features may not provide the required functions.
- IRONMAN extensibility and encapsulation features are considered unnecessarily complex for simulator needs.

A language satisfying the IRONMAN, however, will probably be usable for programming most simulator functions and will satisfy more of the SHOL requirements than any of the modified candidate languages considered in this study. Furthermore, the DoD backing should ensure the development of the support facilities and training efforts recommended for a smooth transition to the SHOL in previous sections of this report.

As of May 1978, work on two Common Language designs was in progress. Some modifications to the July 1977 IRONMAN were being made, based on the results of four preliminary design efforts completed in February 1978. Although final designs are scheduled for test and evaluation beginning in April 1979, it is currently unclear how suitable these designs will be for embedded computer system programming. Assuming that further redesign will be needed, it is unlikely that production compilers for flight simulator computers will be ready for use in less than 5 years.

Section 10

SUMMARY AND CONCLUSIONS

The main objective of this study was to define higher order language requirements for programming flight training simulators. A subsidiary objective was to develop a general approach for determining HOL requirements in a given application area and then to apply this approach to the simulator area. The approach we devised analyzes three sources of language requirements--the programming environment, the functions to be programmed, and language design principles. Requirements pertaining specifically to flight simulators were determined by analyzing a variety of simulators developed by the Link Division of the Singer Company. Using this information, we developed a generic model of the programming tasks relevant to simulator development. A detailed analysis of language requirements was keyed to this model.

Based on this analysis, we prepared a detailed specification of simulator HOL requirements, using the requirements structure of the IRONMAN (a specification of HOL requirements for a common DoD programming language). We then analyzed PL/I, FORTRAN, JOVIAL J3B, JOVIAL J73I, and PASCAL to see how well each satisfied the simulator HOL requirements we had developed. Our analysis showed that PL/I and JOVIAL J3B were best suited for simulator programming, although only FORTRAN was clearly the least suitable language.

Since all the languages failed to satisfy some of the simulator language requirements, we considered what language modifications would make them significantly more useful as simulator programming languages. Our analysis of the difficulty of modifying each language indicated that PL/I was the most easily modified, and recommended modifications were described.

Appendix A

SIMULATOR MODEL

This Appendix describes the various programming tasks pertaining to flight simulators. The tasks to be performed and their relationships are described using SADT notation [Ross, 1977]. An index to the model is presented in the following pages. In this index, the notation A331, for example, represents a task composed of the tasks A3311, A3312, and A3313. A page number is indicated for those tasks that are decomposed into more detailed tasks. The number indicates the page on which the decomposition will be found.

Appendix A

SIMULATOR MODEL

Section								Page
A0	Simulate an Aircraft							196
	A1	Build Simulator						197
		A11	Create System Data Base					
		A12	Code Modules					
		A13	Compile Modules					
		A14	Link Modules					
	A2	Test Simulator						198
		A21	Test Executive					
		A22	Test Simulation Programs					
		A23	Test Whole Simulator					
	A3	Simulate						199
		A31	Monitor Execution					200
			A311	Control Initialization				
			A312	Cycle Through Simulator Tasks				201
				A3121	Select Frame			
				A3122	Select Cockpit			
				A3123	Schedule Tasks for Frame & Cockpit			
				A3124	Sum Task Times for Frame			
				A3125	Sum Frame Times for Cycle			
			A313	Compute Spare Time for Cycle				
		A32	Initialize Data Base					
		A33	Model Aircraft Functions					202
			A331	Model Flight				203
				A3311	Model Aircraft Flight Controls			
				A3312	Model Aerodynamics			204
					A33121	Process Atmospheric Data		
					A33122	Compute Weight & Balance		
					A33123	Compute Aerodynamic Coefficients		
					A33124	Compute Ground Reactions		
					A33125	Compute Equations of Motion		
				A3313	Model Accessory Systems			205
					A33131	Model Fuel System		
					A33132	Model Electrical System		206
						A331321	Compute Electrical System Power	
						A331322	Compute Bus Loads	

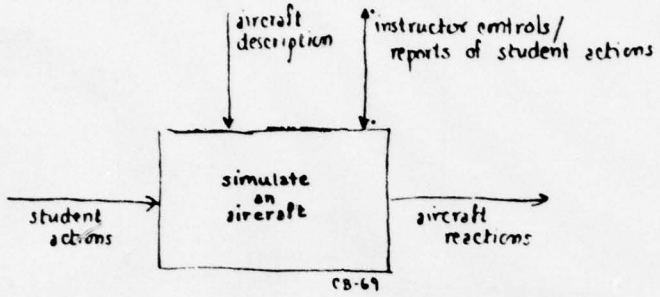
Section			Page
		A33133 Model Hydraulic System	207
		A331331 Compute Hydraulic Flow & Pressure	
		A331332 Model Secondary Hydraulic Controls	
		A331333 Model Landing Gear	
		A33134 Model Engine System	
		A33135 Model Miscellaneous Accessories	208
		A331351 Model Engine Fire & Overheat System	
		A331352 Model Ice/De-Ice System	
		A331353 Model Canopy & Ejection Seat System	
		A331354 Model Oxygen System	
	A332 Model Navigation & Communications		209
	A3321 Model Communications		
	A3322 Model Navigation Equipment		
	A3323 Model Navigation Radios		210
	A33231 Model Compasses		
	A33232 Model Attitude System		
	A33233 Model Radio Stations		
	A3324 Look Up Radio Station		
	A333 Model Motion		
	A334 Model Tactics		211
	A3341 Model Airborne Radar		
	A3342 Model Armaments		
	A3343 Model Electronic Warfare		
	A3344 Model Weapon Delivery		
	A3345 Model Tactical Environment		
	A3346 Model Avionic Displays		
	A335 Model Visual		212
	A3351 Process Flight Data		213
	A33511 Compute Attitude		
	A33512 Compute Position & Velocity		
	A3352 Drive Gantry		214
	A33521 Process Gantry Feedback		
	A33522 Control Gantry		

Section				Page
		A3353	Drive Probe	215
			A33531 Compute Altitude Limits	
			A33532 Control Probe	
		A3354	Produce Visual Image	216
			A33541 Process Instructor & Student Controls	
			A33542 Determine Image Focusing	
			A33543 Produce Cultural Lighting	
			A33544 Produce Visibility Effects	
A34	Do I/O to Simulated Cockpit			
A35	Communicate with Instructor			217
	A351 Record/Playback Mission			
	A352 Set Initial Conditions			
	A353 Set Malfunctions			
	A354 Display/Update Datapool Values			
	A355 Display Terrain Map			
	A356 Plot Aircraft Position and Track			

SADT® DIAGRAM FORM ST008 9/75

Form © 1975 SofTech, Inc., 460 Totten Pond Road, Waltham, Mass. 02154, USA

USED AT:	AUTHOR: CB PROJECT: simulator model NOTES: 1 2 3 4 5 6 7 8 9 10	DATE: 6/16/77 REV:	X	WORKING	READER	DATE	CONTEXT:
				DRAFT			none
				RECOMMENDED			
				PUBLICATION			



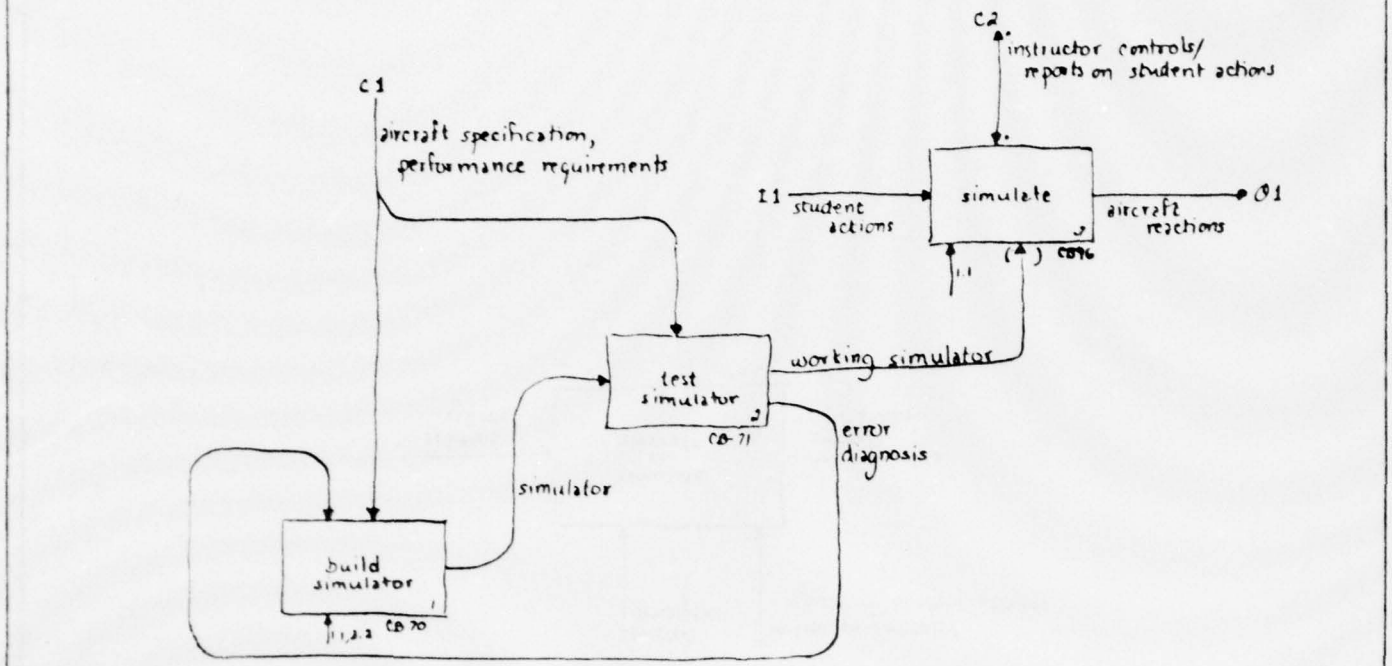
NODE:	TITLE:	NUMBER:
A-0	simulate an aircraft (context)	CB-6X

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

SADT® DIAGRAM FORM S1008 9/75

Form © 1975 SoftTech, Inc. 460 Totten Pond Road, Waltham, Mass. 02154, USA

USED AT:	AUTHOR: CB PROJECT: simulator model NOTES: 1 2 3 4 5 6 7 8 9 10	DATE: 5/16/77 REV:	X	WORKING	READER	DATE	CONTEXT: top
				DRAFT			
				RECOMMENDED			
				PUBLICATION			

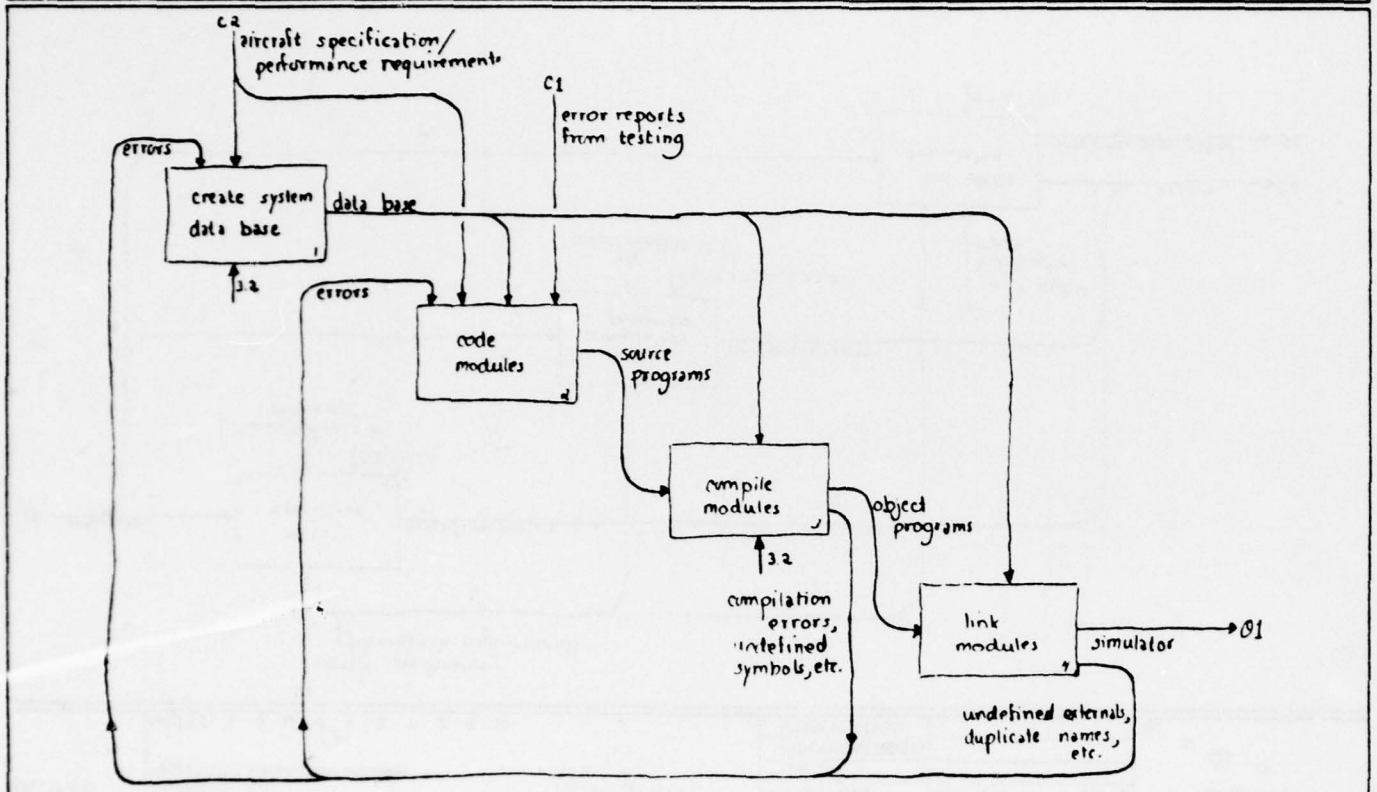


NODE:	TITLE:	NUMBER:
A0	simulate an aircraft	CB-69

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

SADT® DIAGRAM FORM ST098 9/75
Form © 1975 SofTech, Inc., 460 Totten Pond Road, Waltham, Mass. 02154, USA

USED AT:	AUTHOR: CB PROJECT: simulator model NOTES: 1 2 3 4 5 6 7 8 9 10	DATE: 5/16/77 REV:	X	WORKING	READER	DATE	CONTEXT: A0
				DRAFT			
				RECOMMENDED			
				PUBLICATION			



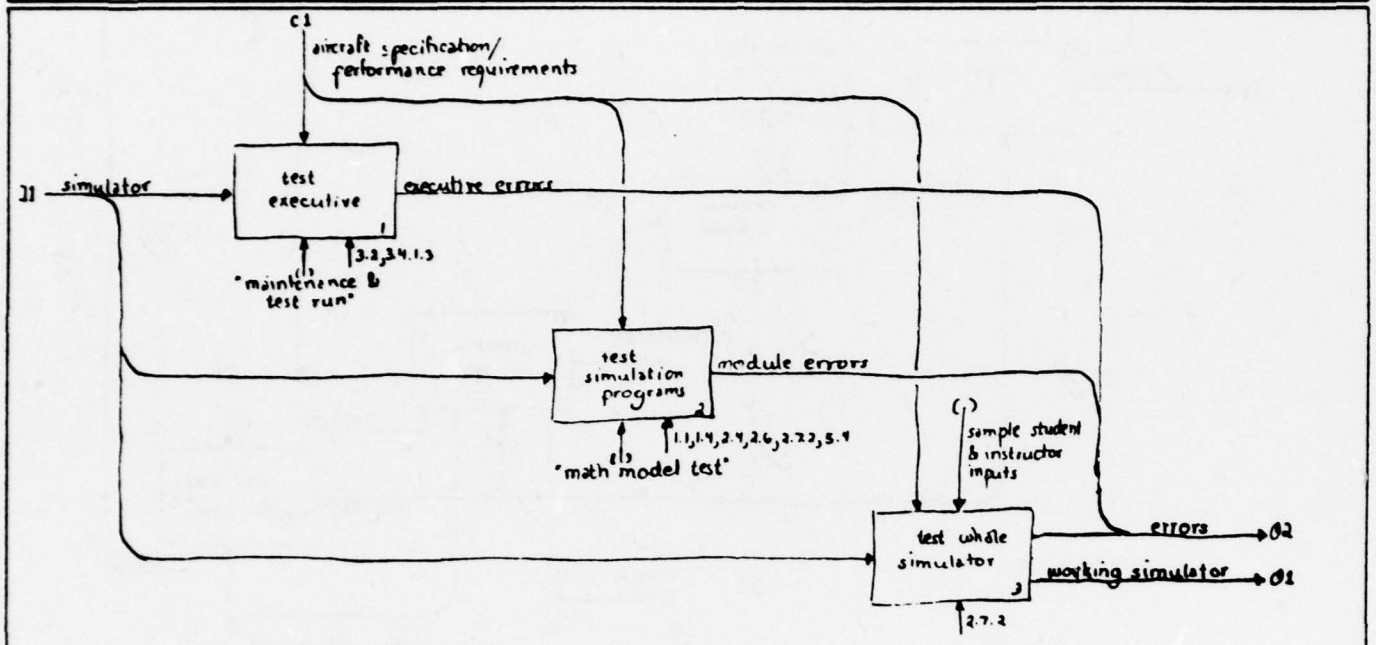
NODE:	TITLE:	NUMBER:
A1	build simulator	CB-[redacted]70

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

SADT® DIAGRAM FORM ST098 9/75

Form © 1975 SofTech, Inc., 460 Totten Pond Road, Waltham, Mass. 02154, USA

USED AT:	AUTHOR: CB PROJECT: simulator model NOTES: 1 2 3 4 5 6 7 8 9 10	DATE: 5/16/77 REV:	X WORKING DRAFT RECOMMENDED PUBLICATION	READER DATE	CONTEXT: A0



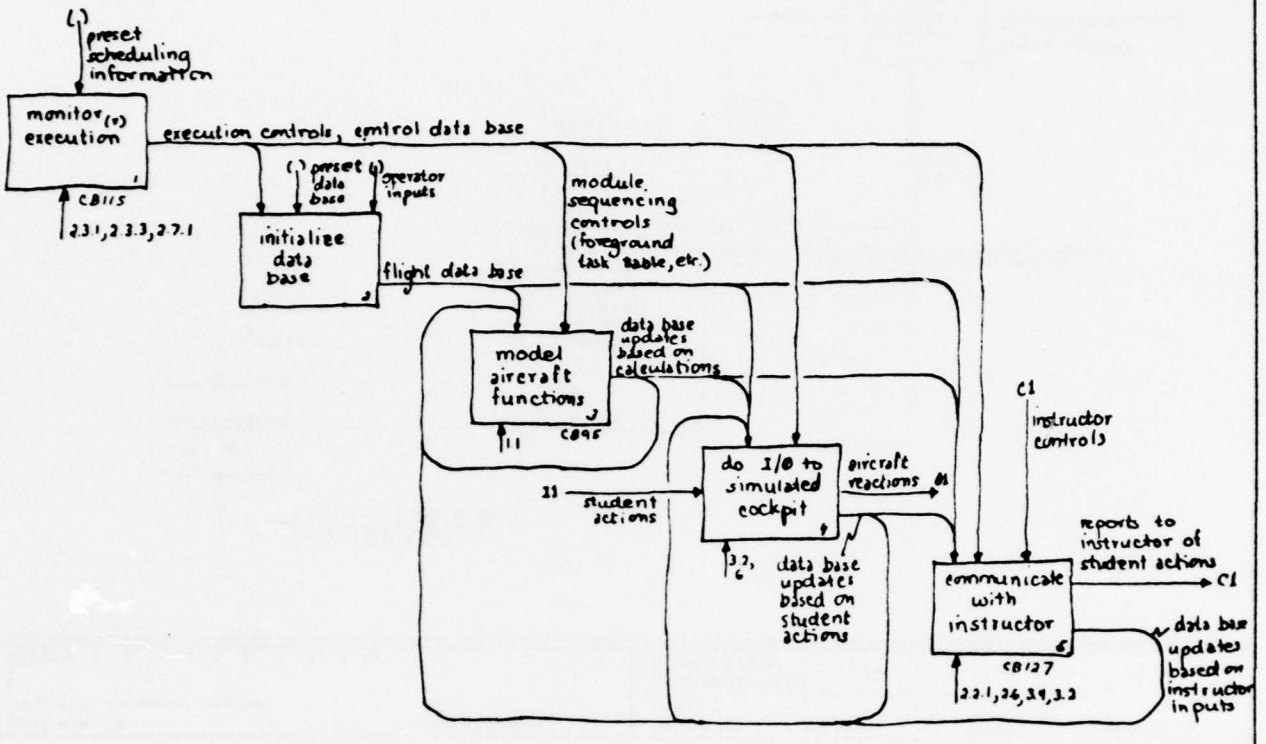
NODE:	TITLE:	NUMBER:
A2	test simulator	CB-71

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

SADT® DIAGRAM FORM ST098 9/75

Form © 1975 SofTech, Inc., 460 Totten Pond Road, Waltham, Mass. 02154, USA

USED AT:	AUTHOR: CD	DATE: 5/31/77		WORKING	READER	DATE	CONTEXT:
	PROJECT: simulator model	REV:		DRAFT			
				RECOMMENDED			
	NOTES: 1 2 3 4 5 6 7 8 9 10			PUBLICATION			A0

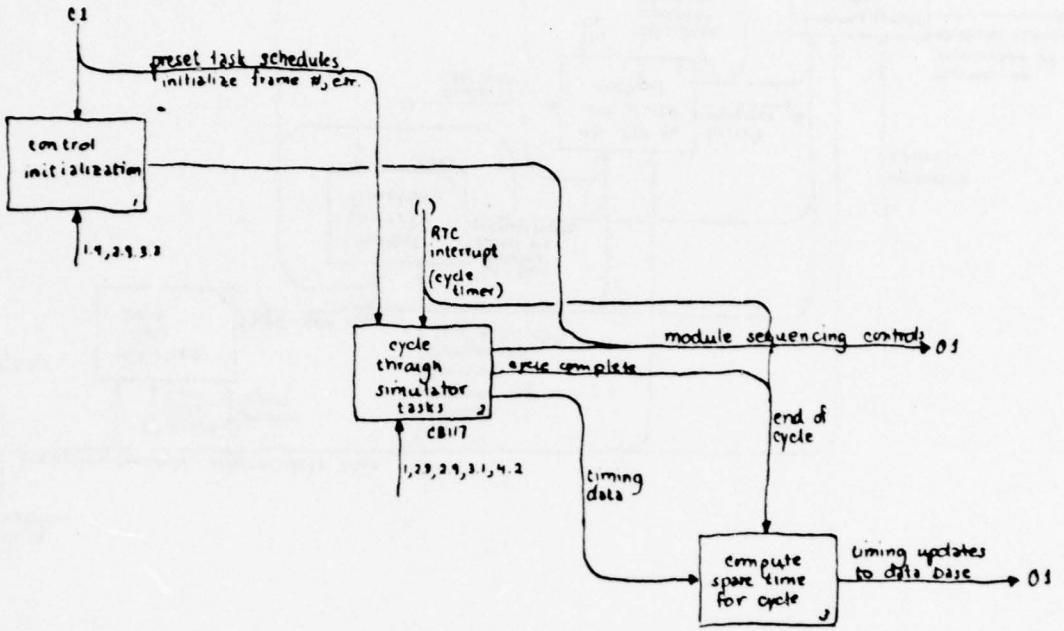


NODE:	TITLE:	NUMBER:
A3	simulate	CB96 (CB74)

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

SADT® DIAGRAM FORM ST0998 9/75
Form © 1975 SofTech, Inc., 460 Totten Pond Road, Waltham, Mass. 02154, USA

USED AT:	AUTHOR: CB PROJECT: simulator model NOTES: 1 2 3 4 5 6 7 8 9 10	DATE: 6/3/77 REV:	X WORKING DRAFT RECOMMENDED PUBLICATION	READER DATE	CONTEXT: A3



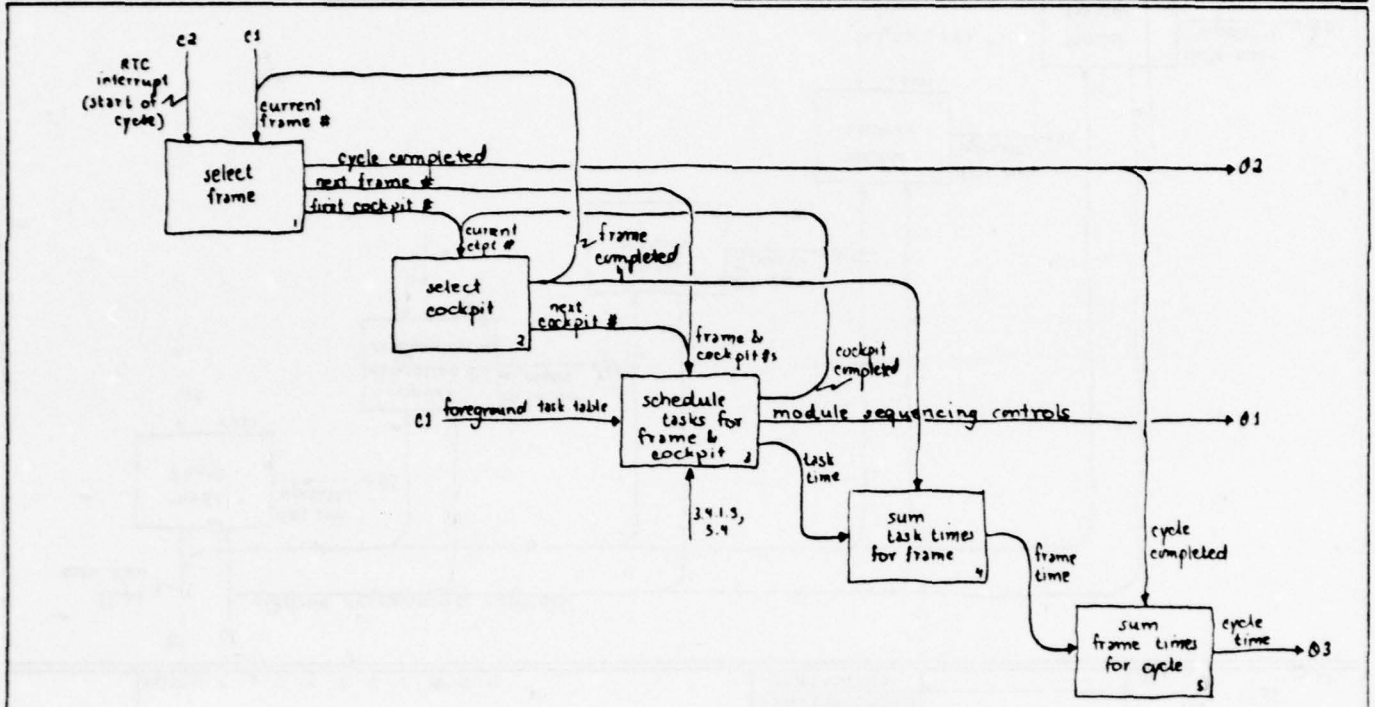
NODE: A31	TITLE: monitor (.) execution	NUMBER: CB115 (CB114)

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

SADT® DIAGRAM FORM ST098 9/75

Form © 1975 SofTech, Inc., 460 Totten Pond Road, Waltham, Mass. 02154 USA

USED AT:	AUTHOR: CB PROJECT: simulator model	DATE: 6/3/77 REV:	WORKING	READER	DATE	CONTEXT:
			DRAFT			
	NOTES: 1 2 3 4 5 6 7 8 9 10		RECOMMENDED			A31
			PUBLICATION			



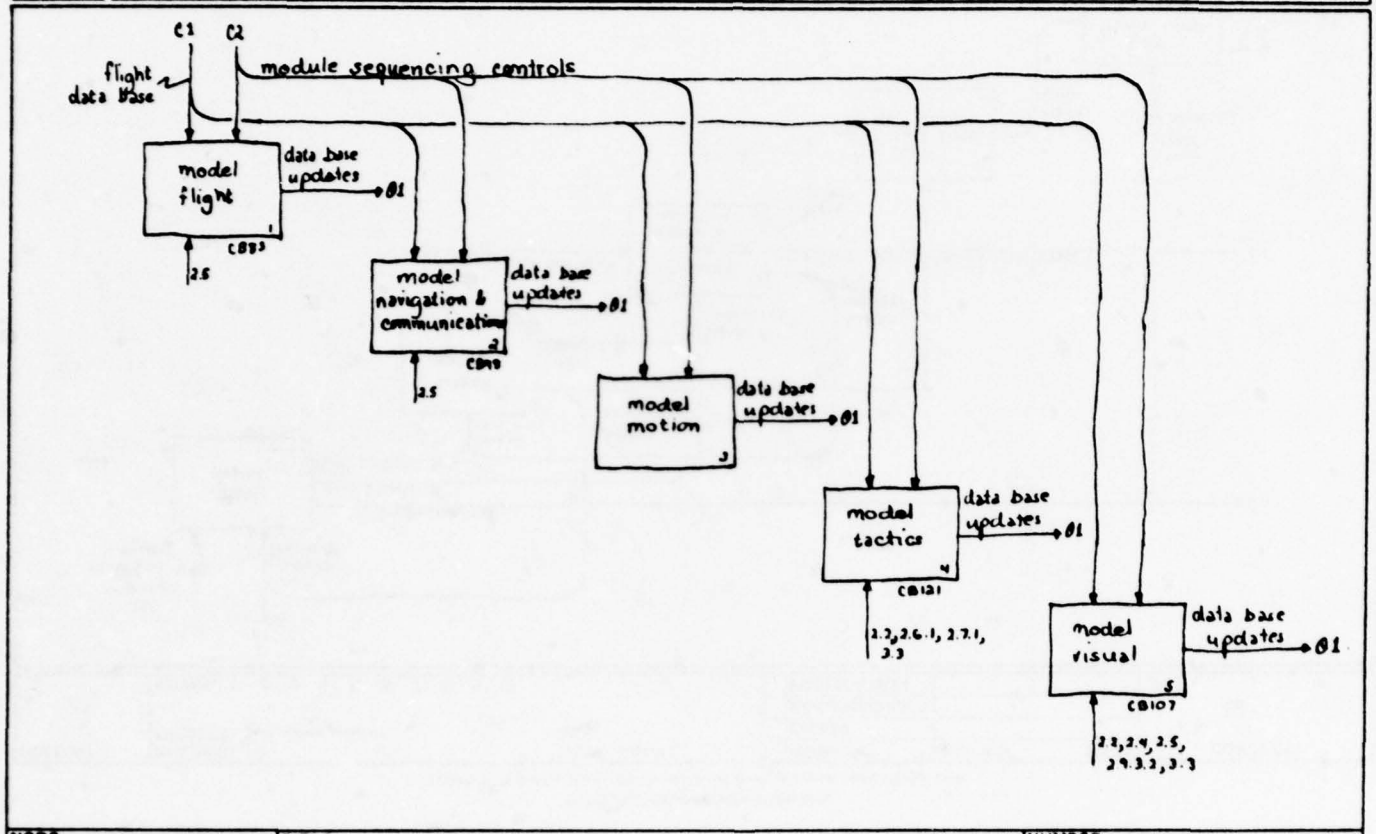
NODE: A312	TITLE: cycle through simulator tasks	NUMBER: CB117 (CB116)

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

SADT® DIAGRAM FORM ST098 9/75

Form © 1975 SofTech, Inc., 460 Totten Pond Road, Waltham, Mass. 02154, USA

USED AT:	AUTHOR: CB PROJECT: simulator model NOTES: 1 2 3 4 5 6 7 8 9 10	DATE: 5/31/77 REV:	X	WORKING	READER	DATE	CONTEXT: A3
				DRAFT			
				RECOMMENDED			
				PUBLICATION			



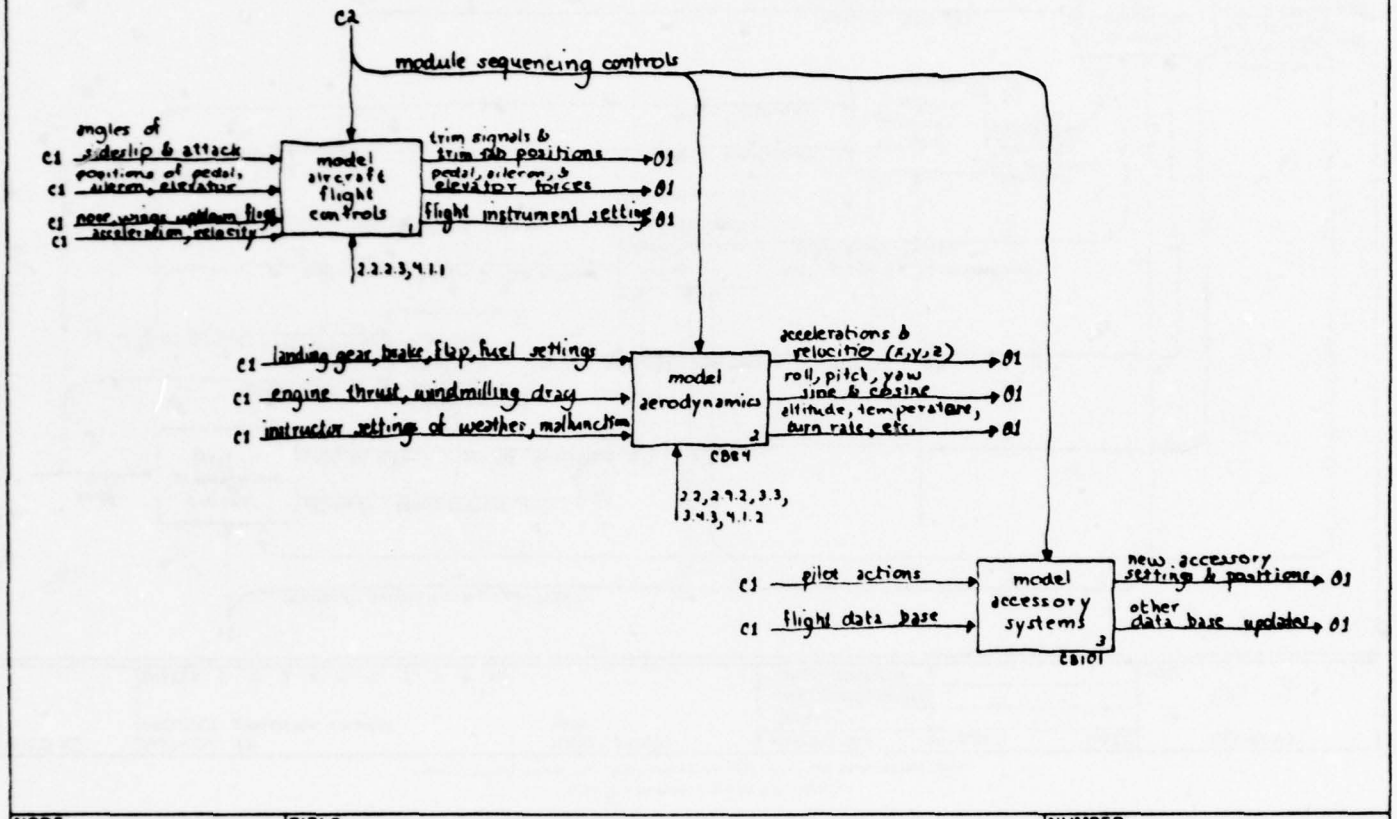
NODE:	TITLE:	NUMBER:
A33	model aircraft functions	C095 (CB77)

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

SADT® DIAGRAM FORM ST098 9/75

Form © 1975 SofTech, Inc., 460 Totten Pond Road, Waltham, Mass. 02154, USA

USED AT:	AUTHOR: CB	DATE: 5/10/77				CONTEXT:
	PROJECT: simulator model	REV: A	x	WORKING	READER DATE	[1]
				DRAFT		
	NOTES: 1 2 3 4 5 6 7 8 9 10			RECOMMENDED		A33
				PUBLICATION		



NODE:	TITLE:	NUMBER:
A331	model flight	CB83 (CB78)

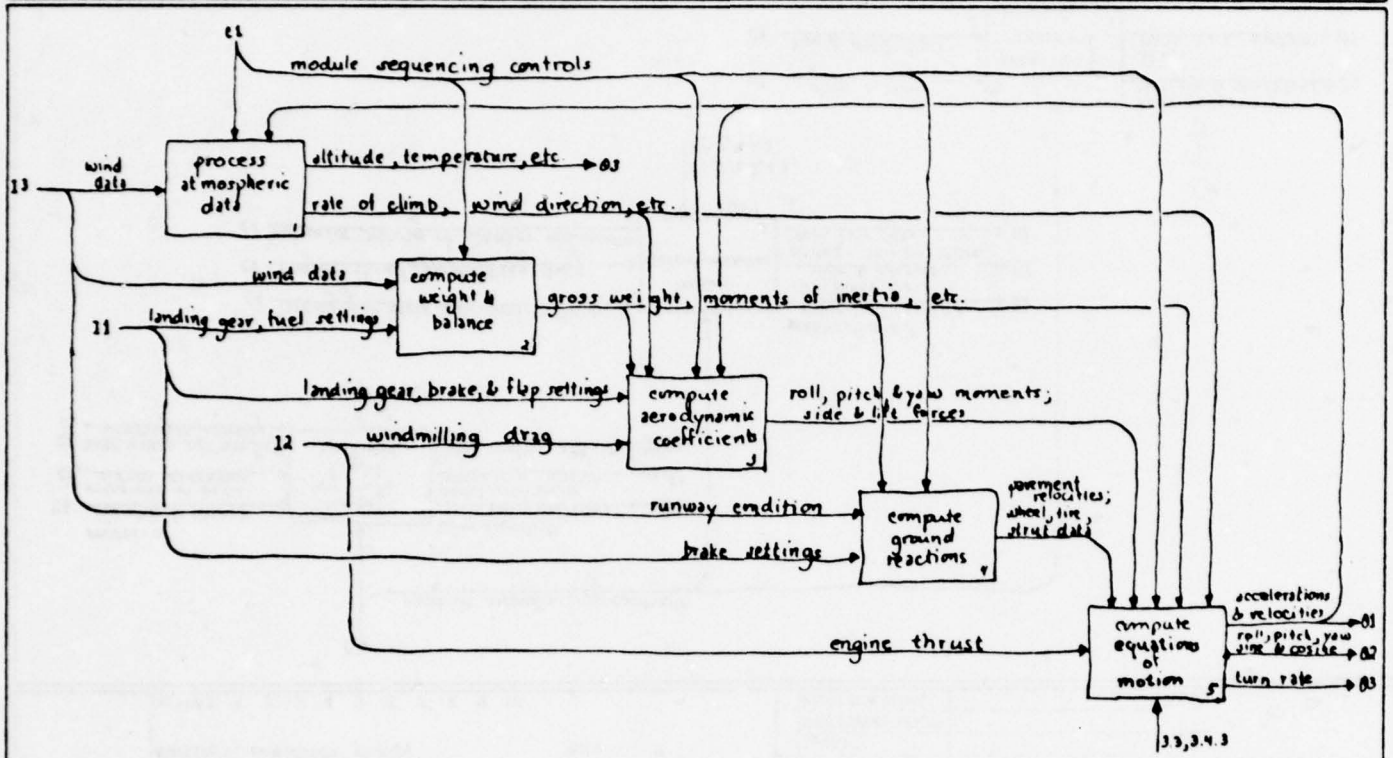
THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

SADT™ DIAGRAM FORM ST098 9/75

Form © 1975 SofTech, Inc., 460 Totten Pond Road, Waltham, Mass. 02154, USA

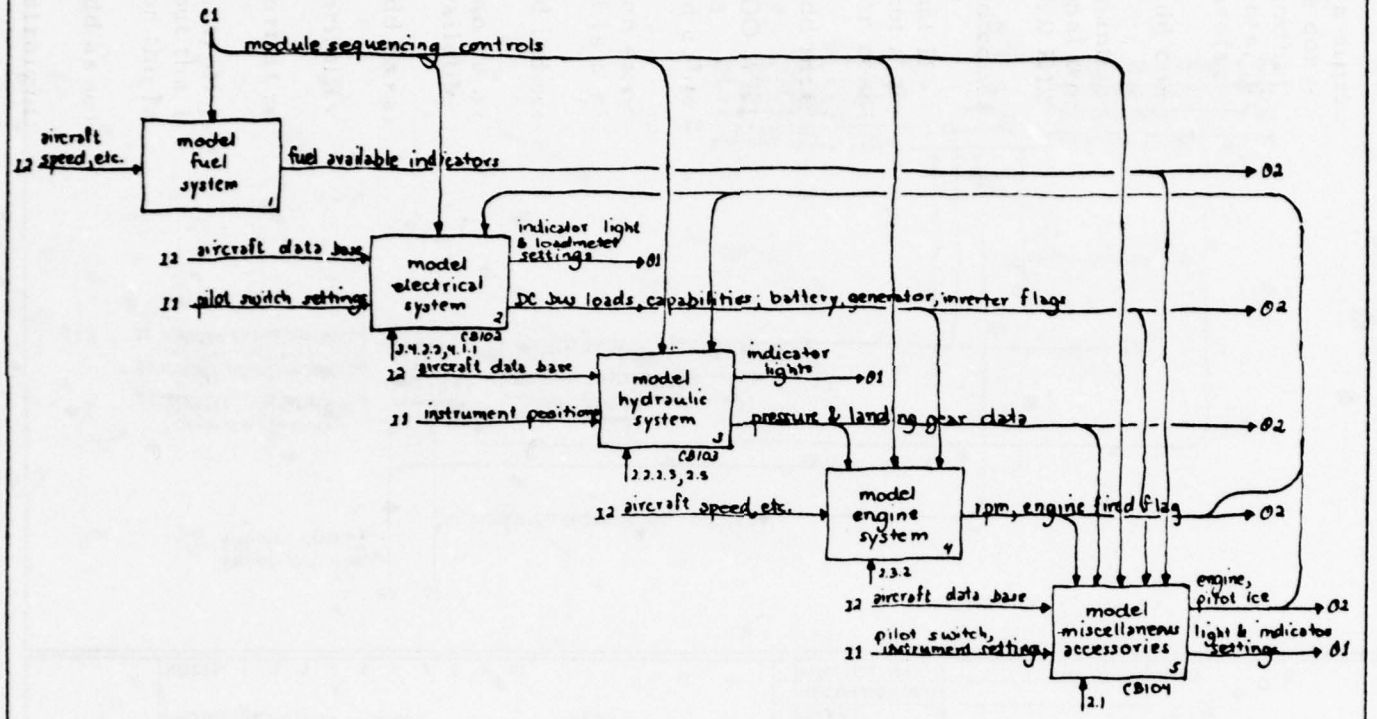
USED AT:	AUTHOR: CB PROJECT: simulator model NOTES: 1 2 3 4 5 6 7 8 9 10	DATE: 5/19/77 REV:	X WORKING DRAFT RECOMMENDED PUBLICATION	READER DATE	CONTEXT: 0 [2] 0 A331



NODE:	TITLE:	NUMBER:
A3312	model aerodynamics	CB84 (CB82)

SADT® DIAGRAM FORM ST098 9/75
Form © 1975 SofTech, Inc., 460 Totten Pond Road, Waltham, Mass. 02154, USA

USED AT:	AUTHOR: CB PROJECT: simulator model NOTES: 1 2 3 4 5 6 7 8 9 10	DATE: 6/1/77 REV:	X	WORKING	READER	DATE	CONTEXT: A331
				DRAFT			
				RECOMMENDED			
				PUBLICATION			

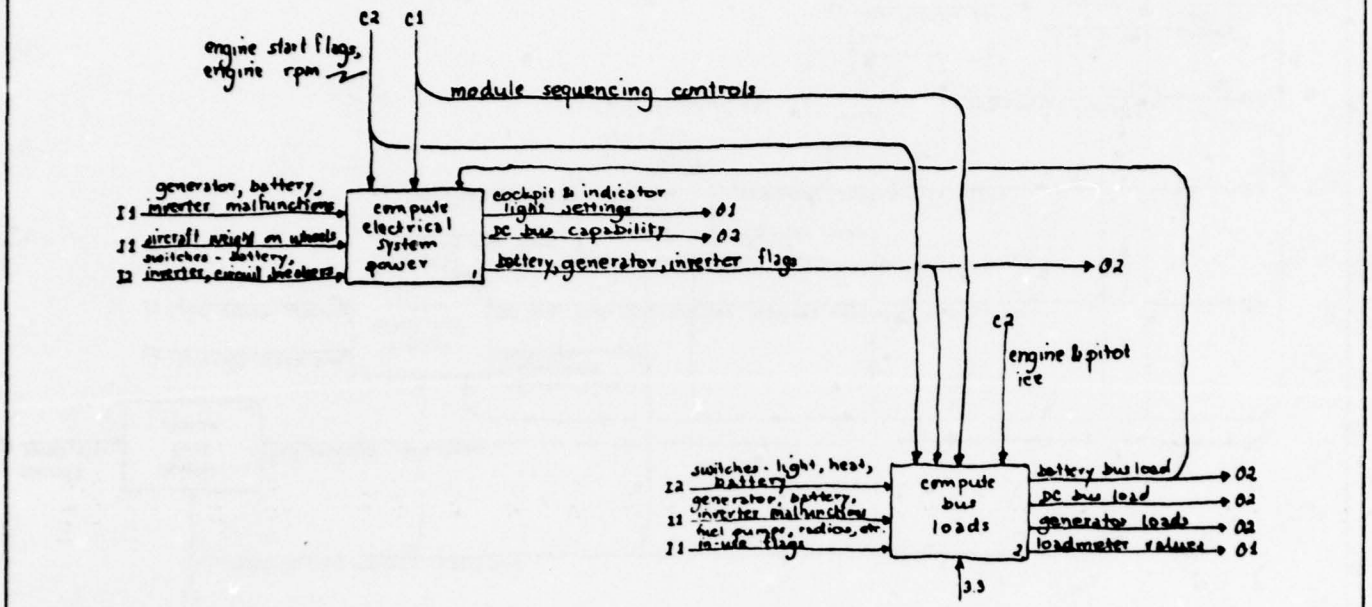


NODE:	TITLE:	NUMBER:
A3313	model accessory systems	CB101 (CB86)

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

SADT® DIAGRAM FORM ST098 9/75
Form © 1975 SofTech, Inc., 460 Totten Pond Road, Waltham, Mass. 02154, USA

USED AT:	AUTHOR: CB PROJECT: simulator model NOTES: 1 2 3 4 5 6 7 8 9 10	DATE: 6/1/77 REV:	X	WORKING	READER	DATE	CONTEXT: A3313
				DRAFT			
				RECOMMENDED			
				PUBLICATION			



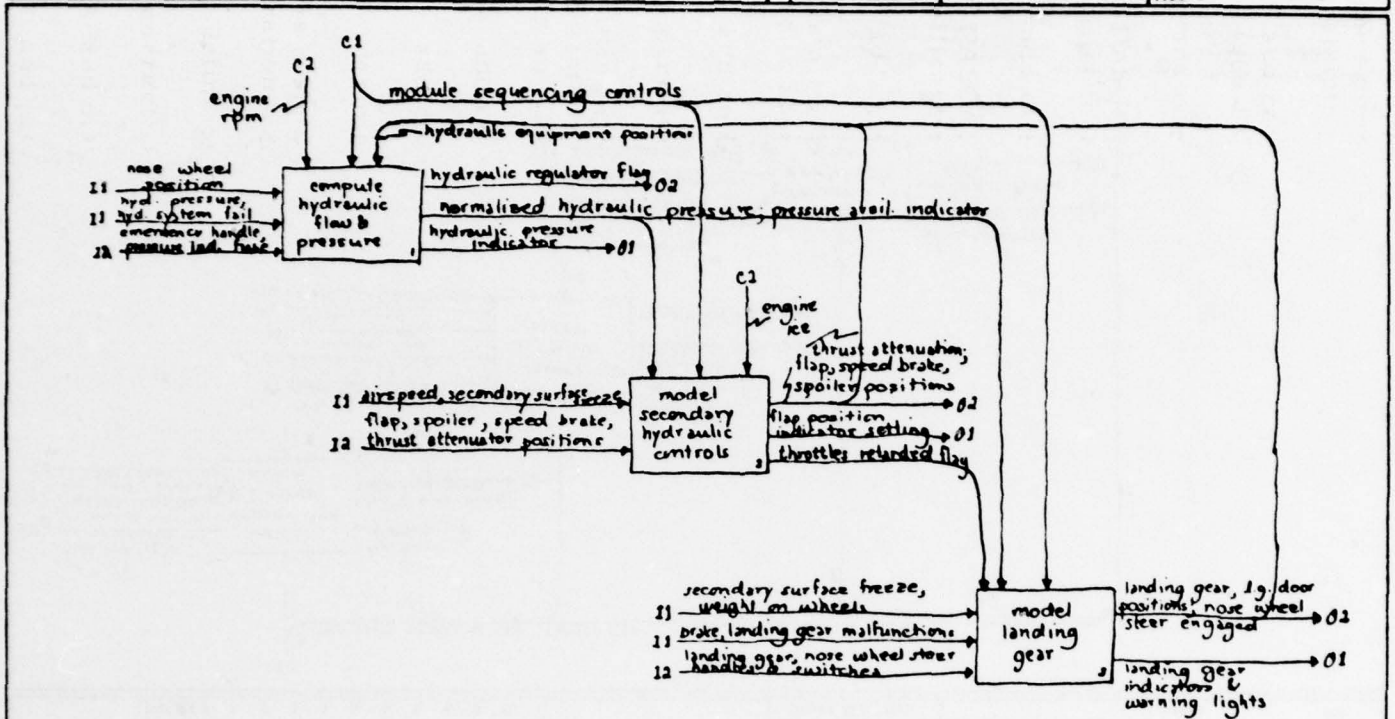
NODE:	TITLE:	NUMBER:
A33132	model electrical system	CB102

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

SADT® DIAGRAM FORM S1098 9/75

Form © 1975 SofTech, Inc. 460 Totten Pond Road, Waltham, Mass. 02154, USA

USED AT:	AUTHOR: CB PROJECT: simulator model NOTES: 1 2 3 4 5 6 7 8 9 10	DATE: 6/1/77 REV:	X	WORKING	READER	DATE	CONTEXT: A3313
				DRAFT			
				RECOMMENDED			
				PUBLICATION			

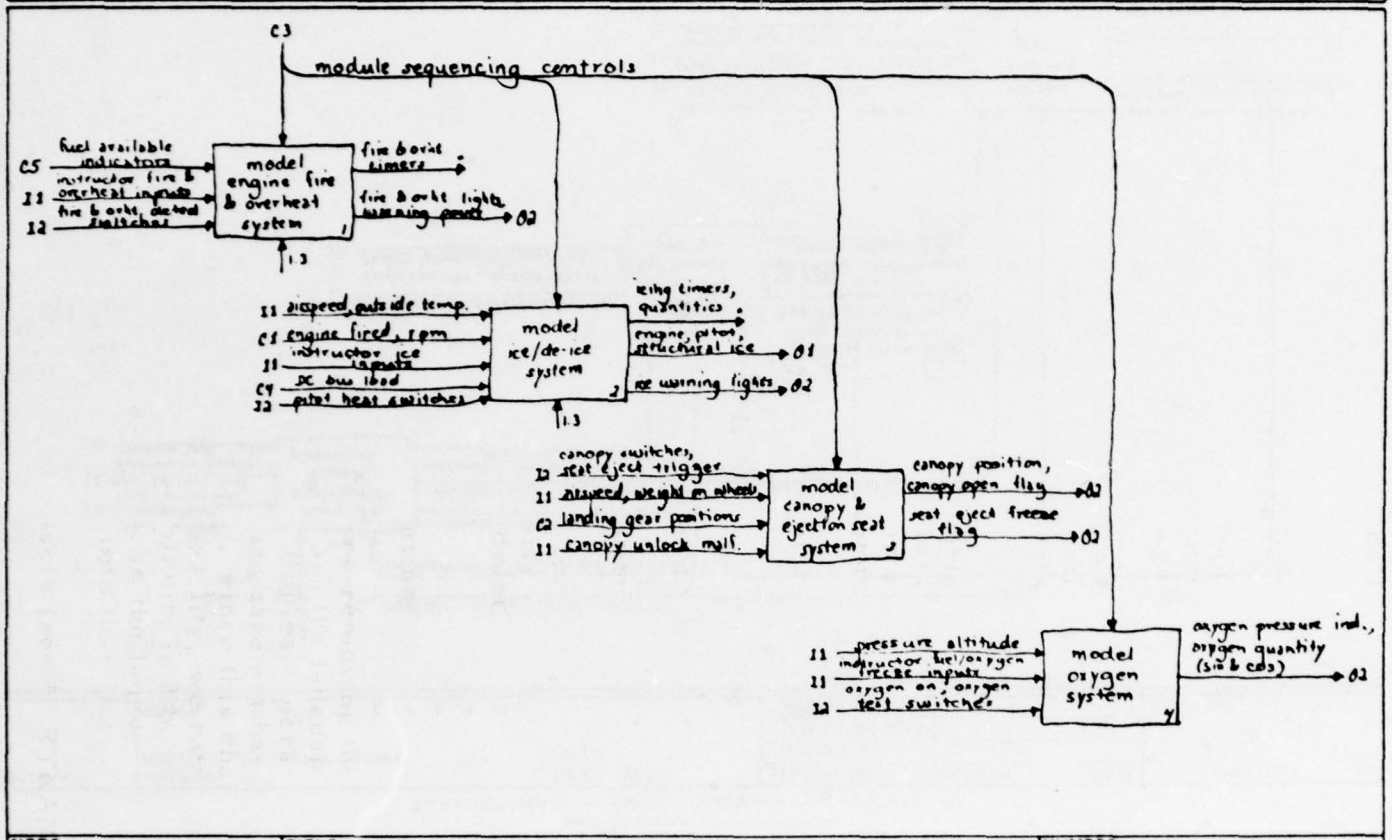


NODE: A33133	TITLE: model hydraulic system	NUMBER: CB103

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

SADT® DIAGRAM FORM ST098 9/75
Form © 1975 SofTech, Inc., 460 Totten Pond Road, Waltham, Mass. 02154, USA

USED AT:	AUTHOR: CB PROJECT: simulator model NOTES: 1 2 3 4 5 6 7 8 9 10	DATE: 6/1/77 REV:	WORKING / DRAFT / RECOMMENDED / PUBLICATION	READER	DATE	CONTEXT: A3313



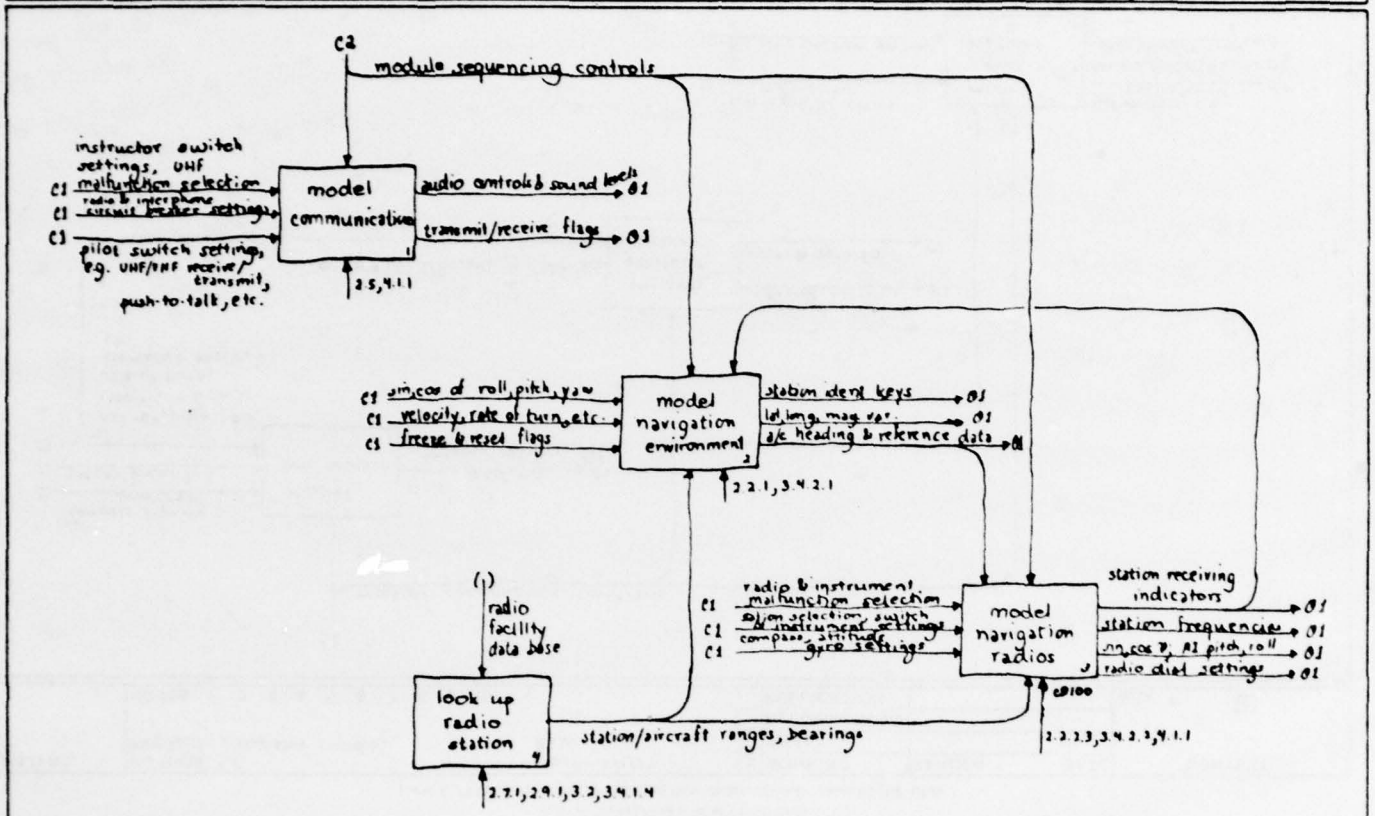
NODE	TITLE	NUMBER
A3313S	model miscellaneous accessories	CB104

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

SADT® DIAGRAM FORM ST098 9/75

Form © 1975 SofTech, Inc., 460 Totten Pond Road, Waltham, Mass. 02154, USA

USED AT:	AUTHOR: CB PROJECT: Simulator model DATE: 6/1/77 REV: NOTES: 1 2 3 4 5 6 7 8 9 10	Y	WORKING	READER	DATE	CONTEXT: A33
			DRAFT			
			RECOMMENDED			
			PUBLICATION			

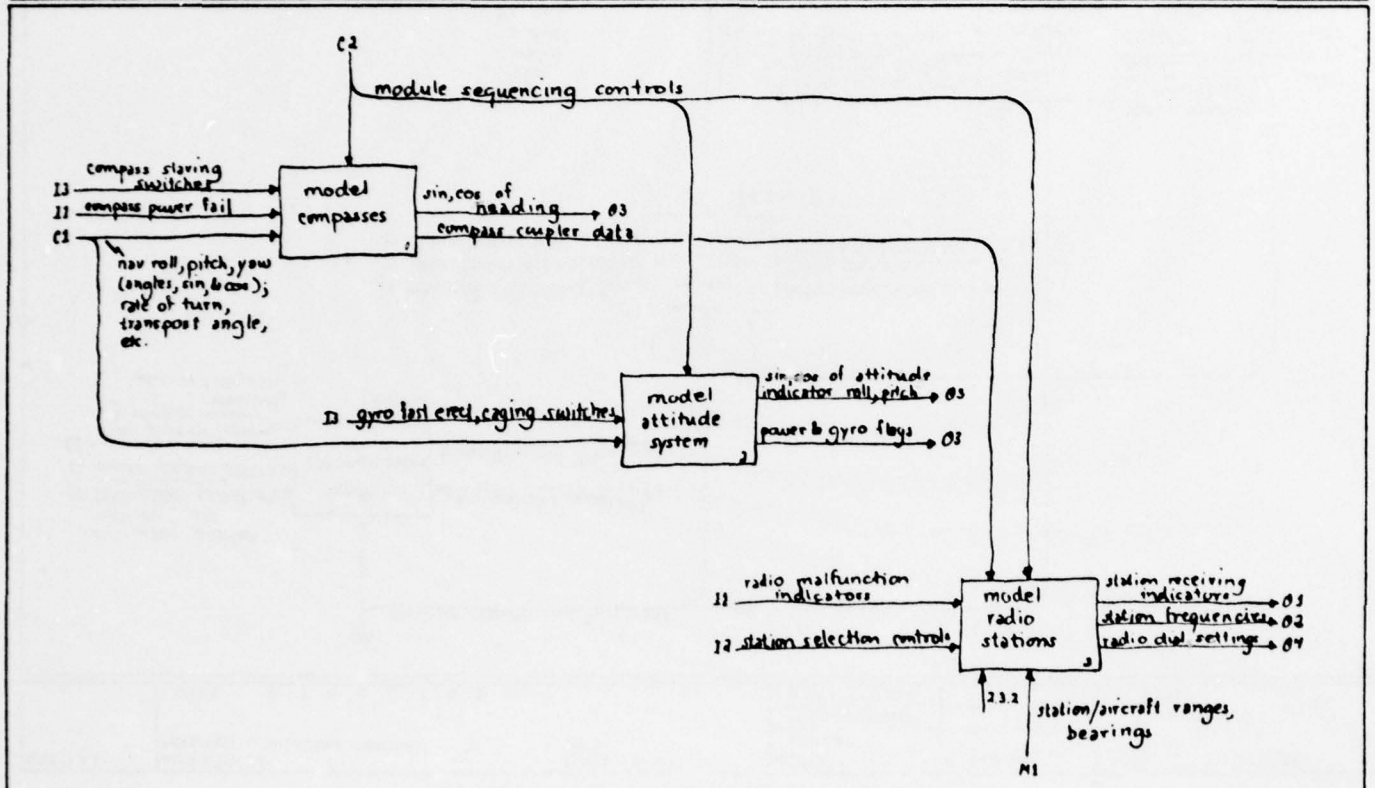


NODE:	TITLE:	NUMBER:
A332	model navigation & communications	C898

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

SADT® DIAGRAM FORM ST098 9/75
Form © 1975 SofTech, Inc., 460 Totten Pond Road, Waltham, Mass. 02154, USA

USED AT:	AUTHOR: CB	DATE: 6/1/77		WORKING	READER	DATE	CONTEXT:
	PROJECT: simulator model	REV:		DRAFT			
				RECOMMENDED			
	NOTES: 1 2 3 4 5 6 7 8 9 10			PUBLICATION			A332

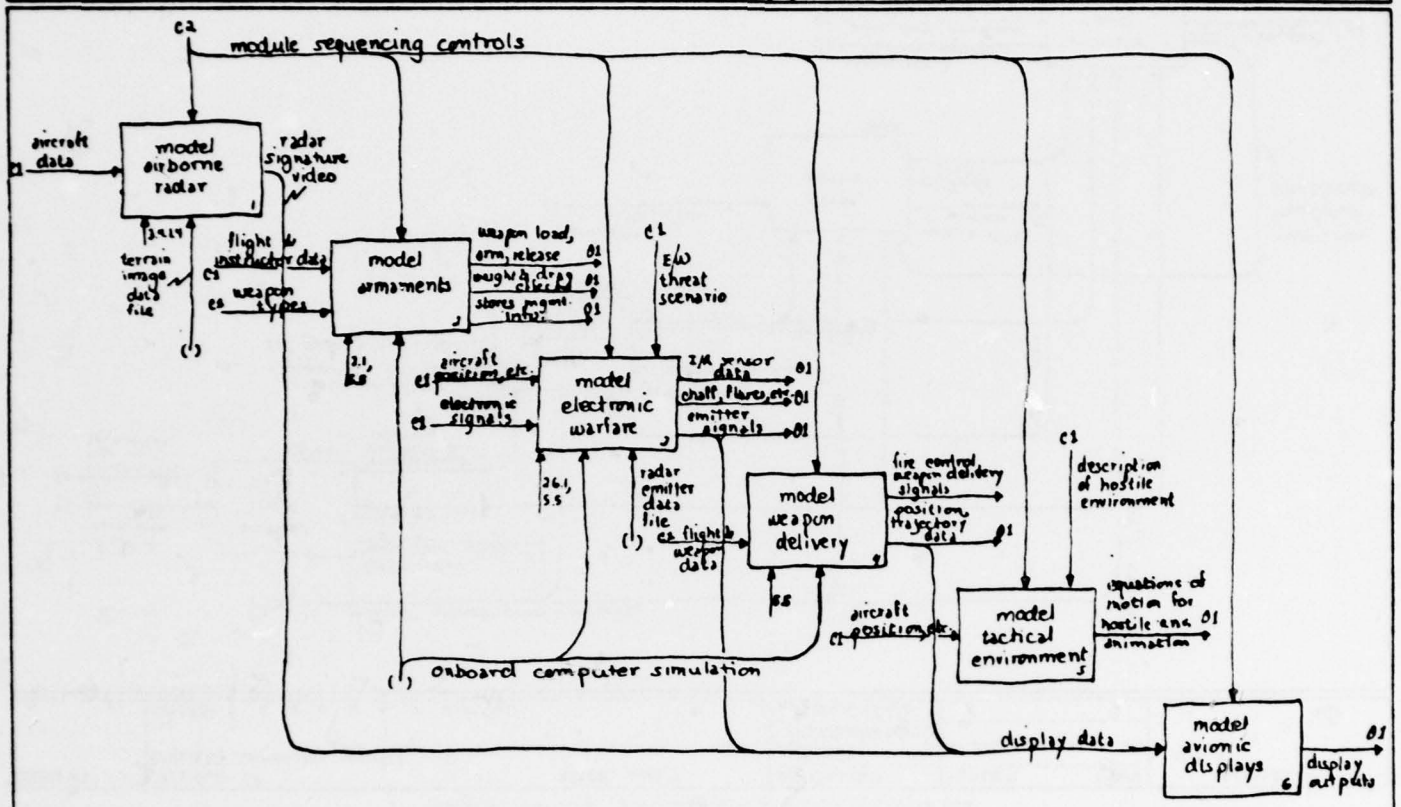


NODE:	TITLE:	NUMBER:
A3323	model navigation radios	CB100 (99)

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

SADT® DIAGRAM FORM ST098 9/75
Form © 1975 SofTech, Inc., 460 Totten Pond Road, Waltham, Mass. 02154, USA

USED AT:	AUTHOR: CB PROJECT: simulator model	DATE: 6/7/77 REV:	X	WORKING	READER	DATE	CONTEXT:
	NOTES: 1 2 3 4 5 6 7 8 9 10			DRAFT			A33
				RECOMMENDED			
				PUBLICATION			



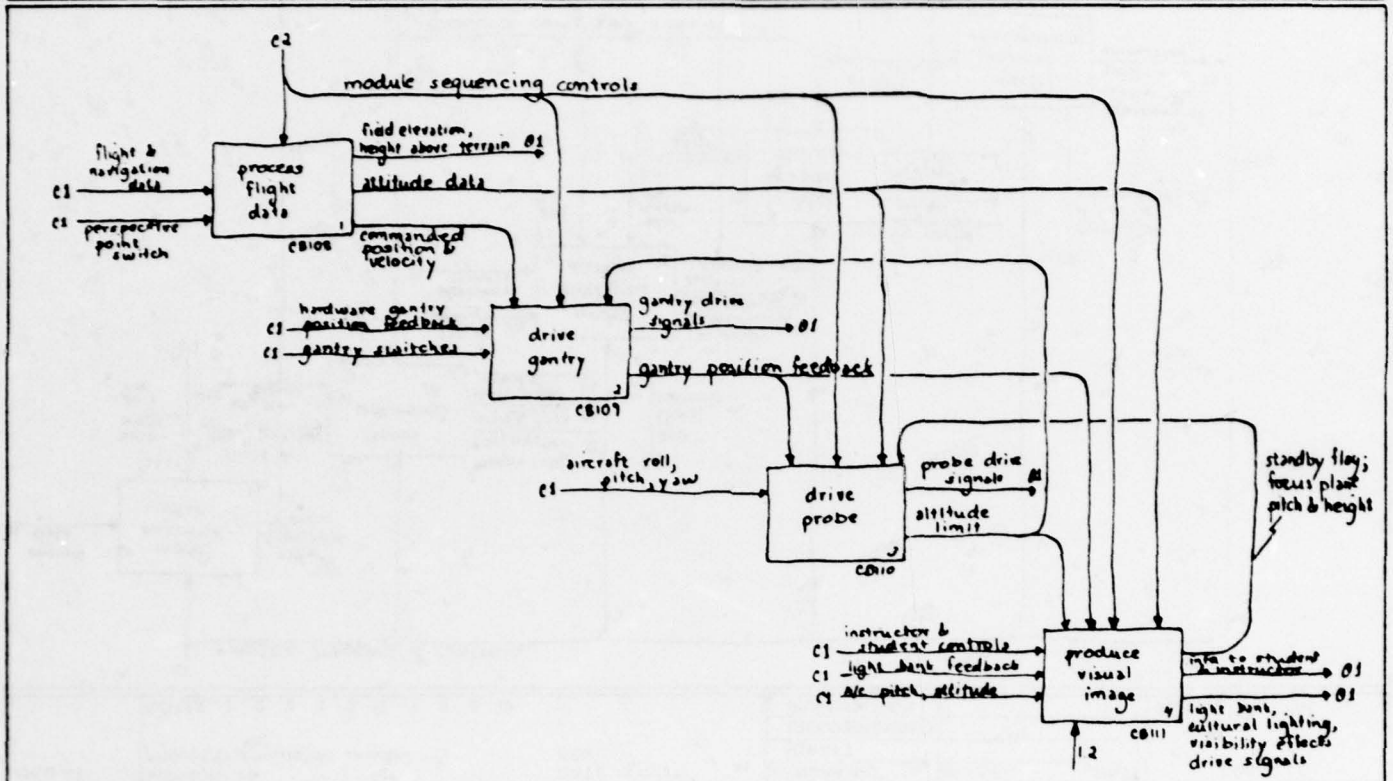
NODE:	TITLE:	NUMBER:
A334	model tactics	CB121

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

SADT® DIAGRAM FORM ST008 9/75

Form © 1975 SoftTech, Inc. 460 Totten Pond Road, Waltham, Mass. 02154 USA

USED AT:	AUTHOR: CB PROJECT: simulator model NOTES: 1 2 3 4 5 6 7 8 9 10	DATE: 6/2/77 REV:	A WORKING DRAFT RECOMMENDED PUBLICATION	READER DATE	CONTEXT: A33

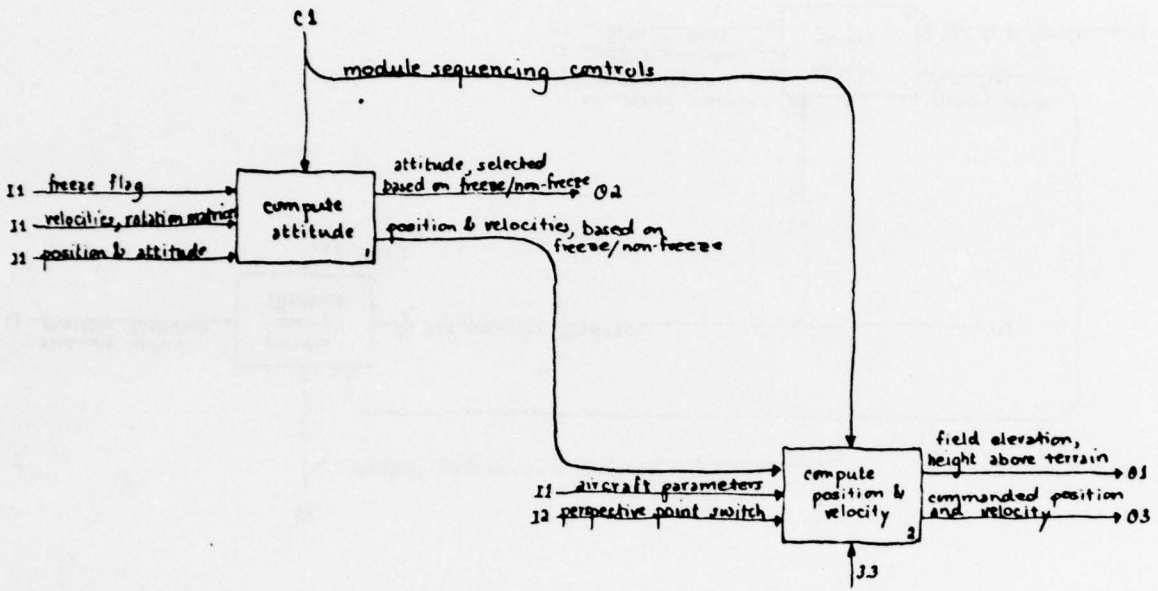


NODE: A335	TITLE: model visual	NUMBER: CB107

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

SADT© DIAGRAM FORM ST098 9/75
Form © 1975 SofTech, Inc., 460 Totten Pond Road, Waltham, Mass. 02154, USA

USED AT:	AUTHOR: CB PROJECT: simulator model NOTES: 1 2 3 4 5 6 7 8 9 10	DATE: 6/2/77 REV:	x	WORKING	READER	DATE	CONTEXT: A335
				DRAFT			
				RECOMMENDED			
				PUBLICATION			



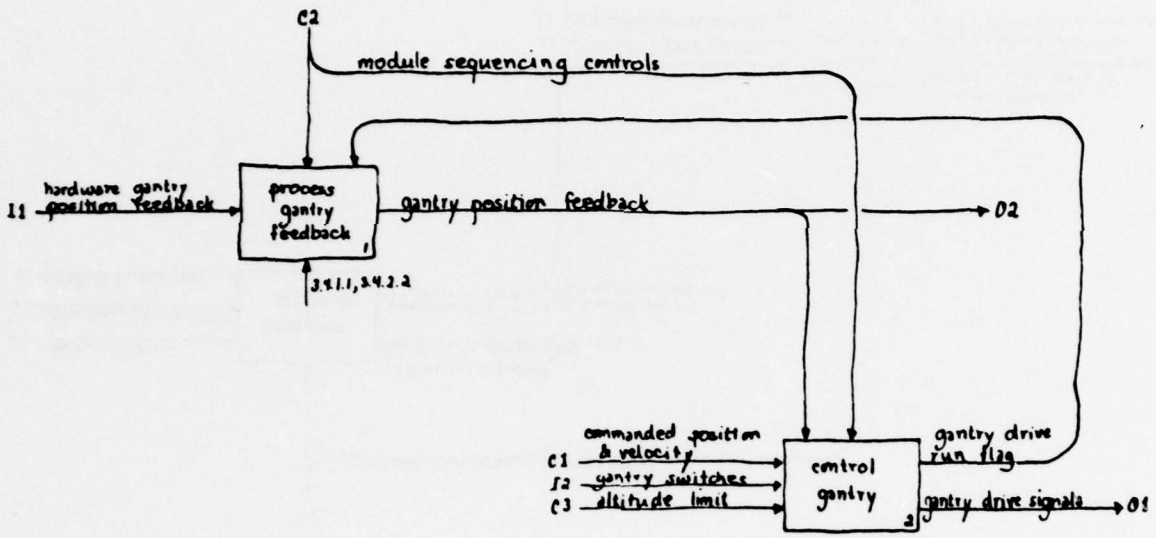
NODE:	TITLE:	NUMBER:
A3351	process flight data	C8108

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

SADT® DIAGRAM FORM ST098 9/75

Form © 1975 SofTech, Inc., 460 Totten Pond Road, Waltham, Mass. 02154, USA

USED AT:	AUTHOR: CB PROJECT: simulator model NOTES: 1 2 3 4 5 6 7 8 9 10	DATE: 6/2/77 REV:	X WORKING DRAFT RECOMMENDED PUBLICATION	READER DATE	CONTEXT: A335



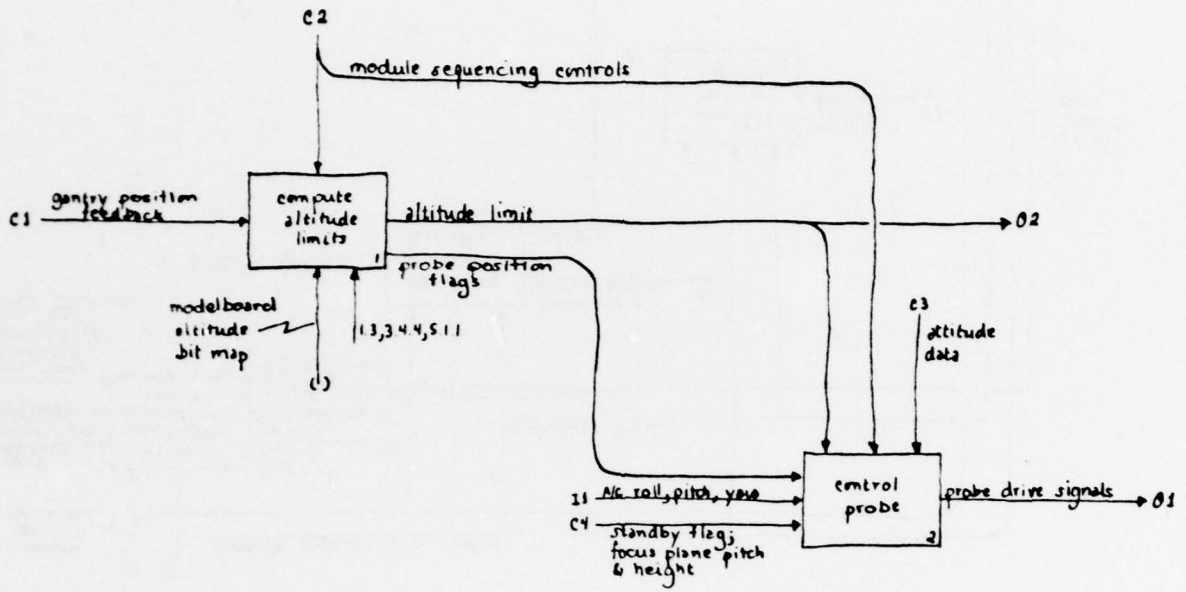
NODE:	TITLE:	NUMBER:
A3352	drive gantry	CB109

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

SADT® DIAGRAM FORM ST098 9/75

Form © 1975 SofTech, Inc., 460 Totten Pond Road, Waltham, Mass. 02154, USA

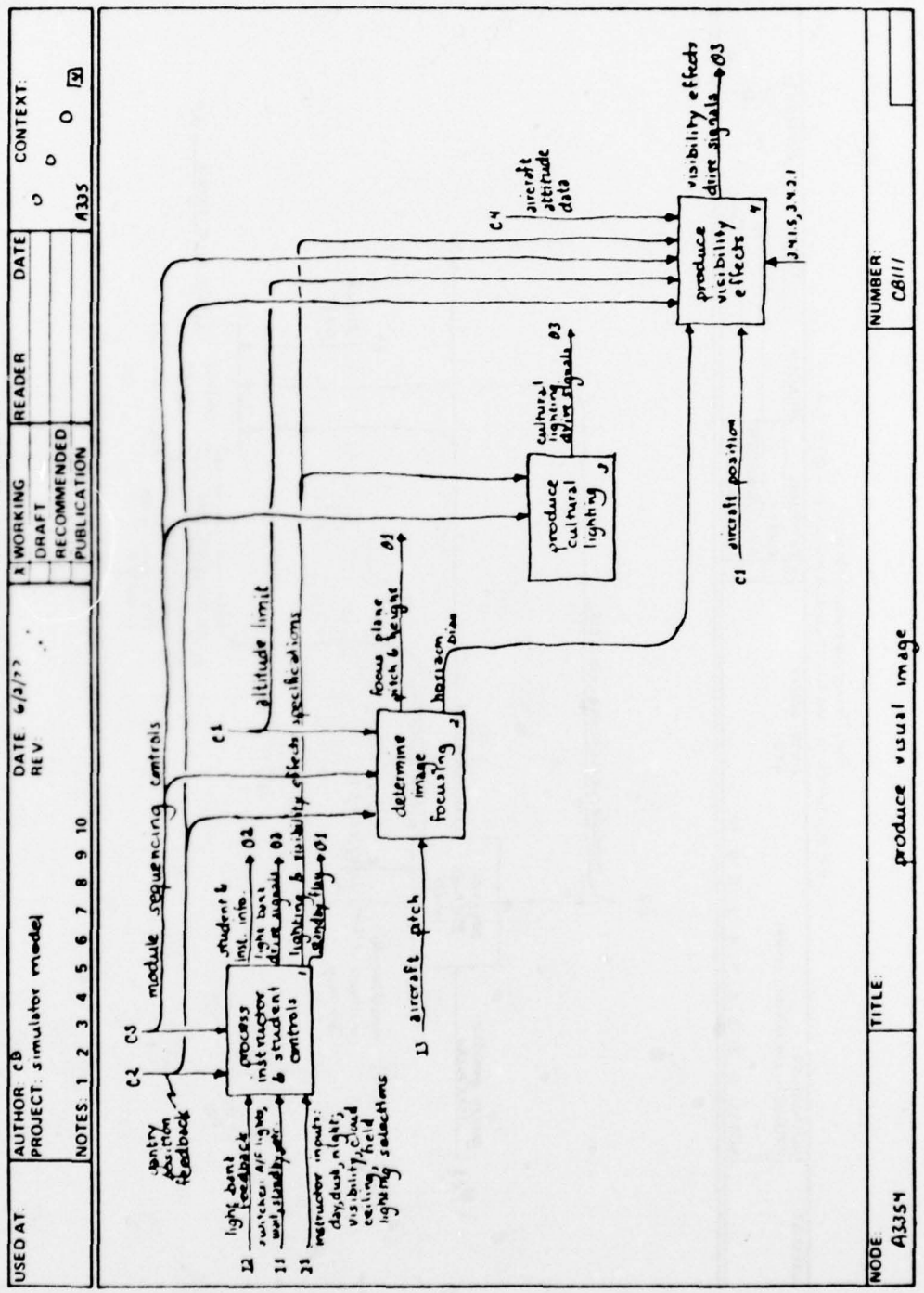
USED AT:	AUTHOR: CB PROJECT: simulator model DATE: 6/2/77 REV: NOTES: 1 2 3 4 5 6 7 8 9 10	X	WORKING	READER	DATE	CONTEXT: A335
			DRAFT			
			RECOMMENDED			
			PUBLICATION			



NODE: A3353	TITLE: drive probe	NUMBER: CB110

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

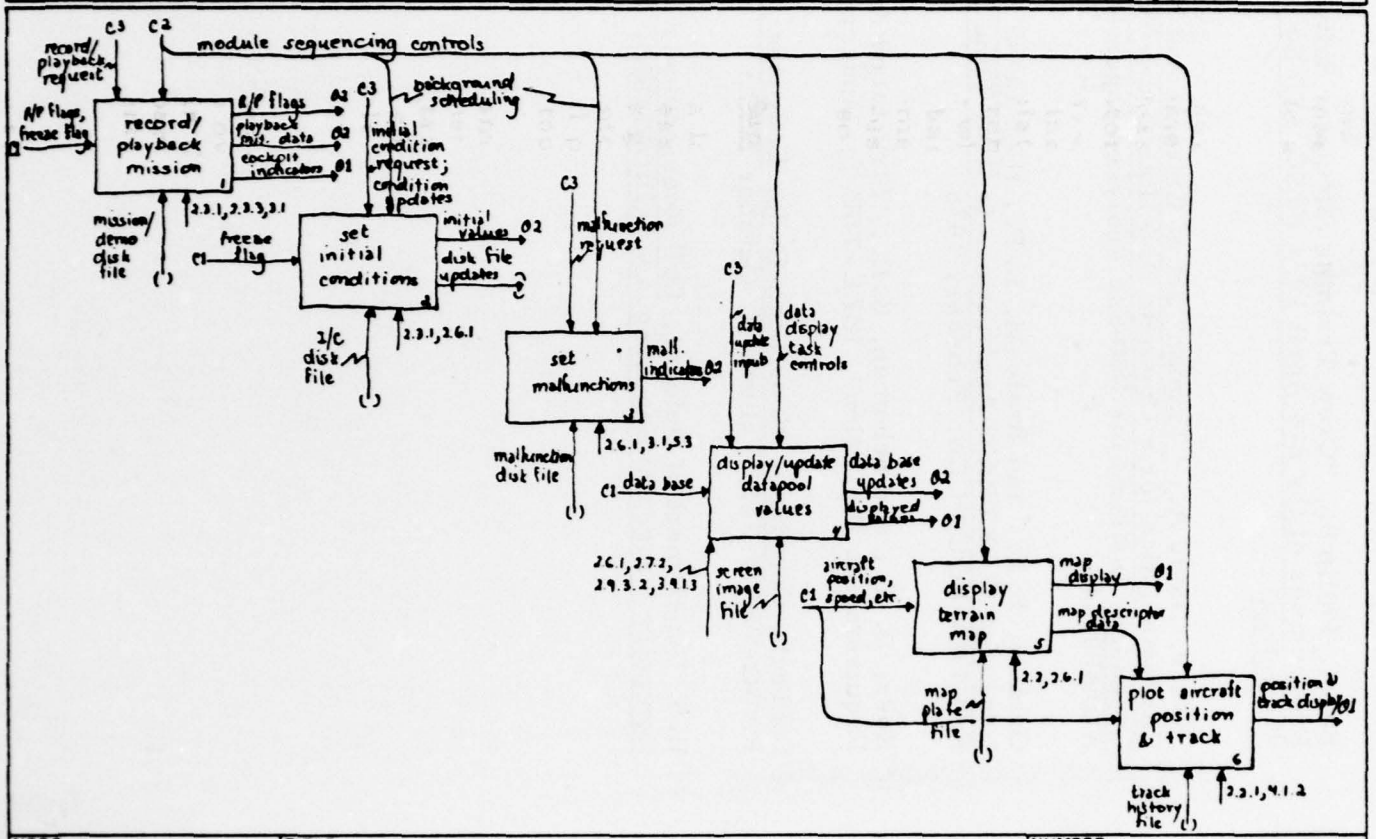
THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC



SADT© DIAGRAM FORM ST098 9/75

Form © 1975 SofTech, Inc., 460 Totten Pond Road, Waltham, Mass. 02154, USA

USED AT:	AUTHOR: CB PROJECT: simulator model NOTES: 1 2 3 4 5 6 7 8 9 10	DATE: 6/15/77 REV:	X	WORKING	READER	DATE	CONTEXT:
				DRAFT			
				RECOMMENDED			
				PUBLICATION			



NODE:	TITLE:	NUMBER:
A35	communicate with instructor	CB127

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

REFERENCES

Babel, Philip S., "Crew Training Simulator Software", Proceedings of the 2-4 April 1974 AFSC/ASD Software Workshop.

Fisher, David A., "Tinman" Set of Criteria and Needed Characteristics for a Common DoD High Order Programming Language, Institute for Defense Analyses paper P-1046, AD/A004841.

Goldiez, B. F., and Braleski, L. P., High Level Programming Language (FORTRAN) Applications in Real Time Simulation, NAVTRA-EQUIPCEN, February 1976.

Ross, D. T., and Schoman, K. E., "Structured Analysis for Requirements Definition", IEEE-TSE-3, January 1977.

SofTech, Communications High-Order Language Investigation, RADC-TR-77-341, Volume II, October 1977.

U.S. Department of Defense, DoD Requirements for High Order Computer Programming Languages (IRONMAN), July 1977

BIBLIOGRAPHY

Anderson, Christine M., Aerospace Higher Order Language Processing, Technical Report AFAL-TR-73-151, AFAL, Wright-Patterson Air Force Base, Ohio, June 1973.

Archdeacon, John L. and Wessale, William D., "Real-Time Digitally Driven Graphic-Display Systems in Aircraft Simulation", NAECON '77, May 1977

Babel, Philip S., Considerations in High Order Language Compiler versus Assembler for Programming Real-Time Training Simulators, TM ASD/ENCT-75-2, ASD, Wright-Patterson Air Force Base, Ohio, February 1975.

Dahl, O. -J., and Hoare, C.A.R., "Hierarchical Program Structures" in Structured Programming, Dijkstra, E. W., Dahl, O. -J., and Hoare, C.A.R., Academic Press, New York, 1972, pp. 175-220.

Engelland, J.D., et.al., Operational Software Concept (Phase Two) Final Report, AFAL, Wright-Patterson Air Force Base, Ohio, July 1975.

Epps, Robert, "Automated Instructional System for Advanced Simulation in Undergraduate Pilot Training (ASUPT)", NAECON '77, May 1977.

Francis, J. W., La Padula, L. J., and Mott-Smith, J., High Order Language Standardization: Perspective, Summary, and Annotated Bibliography, MTR-3331, Mitre Corporation, September 1976.

Goodenough, John B., and Shafer, Lawrence H., A Study of High Level Language Features, SofTech, Inc., ECOM-75-0373-F, Vol. I and II, February 1976.

Goodenough, John B., An Exploratory Study of Reasons for HOL Object Code Inefficiency, SofTech, Inc., ECOM-75-0373-F, August, 1976.

Hansen, Gilbert J., and Lindahl, Charles E., Preliminary Specification of Real-Time PASCAL, Technical Report NAVTRAEQUIPCEN 76-C-0017-1, July 1976.

La Padula, L. J., A Recommendation to Promote Programming Language Standardization in Range Support and Simulator and Trainer Applications, Mitre Corp. WP-20433, October 1975.

La Padula, L. J., and Loring, P. L., Air Force Programming Languages: Standards, Use, and Selection, Mitre Corp. MTR-3169, January 1976.

Liskov, B. H., and Zilles, S., "Programming with Abstract Data Types", SIGPLAN Notices 9, April 1974.

Moyen-van Slimming, Capt. Otto, "A Flexible Approach to On-Board Computer Simulation", NAECON '77, May 1977.

Purdue Workshop on Industrial Computer Systems, Significant Accomplishments and Documentation - Part II, The Long Term Procedural Language, January 1977.

Ross, D. T., Goodenough, J. B., and Irvine, C. A., "Software Engineering: Process Principles, and Goals", Computer 8, May 1975.

SofTech, Evaluation of Algol 68, Jovial J3B, Pascal, Simula 67, and TACPOL vs. TINMAN Requirements for a Common High Order Programming Language, SofTech, Inc., 1021-14, October 1976.

U.S. Air Force, Military Specification - Digital Computational System for Real-Time Training Simulators, MIL-D-83468, December 1975.

U.S. Army Research and Development Group (Europe), Implementation Languages and Real-Time Systems, European Research Office (England), Tech. Rpt. No. ERO-2-75, Vol. 1, 2 and 3.

U.S. Department of Defense, DoD High Order Language Program Management Plan.

Warnier, John Dominique, Logical Construction of Programs, Van Nostrand Reinhold, New York, 1974.

Weinberg, G. M., The Psychology of Computer Programming, Van Nostrand Reinhold, New York, 1971.

 ☆U.S. GOVERNMENT PRINTING OFFICE: 1978- 771-122/53