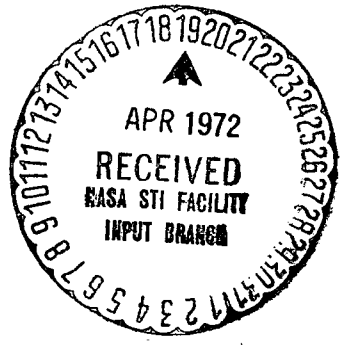
CR 115513

(NASA-CR-115513) ADVANCED SOFTWARE TECHNIQUES FOR DATA MANAGEMENT SYSTEMS. VOLUME 1: STUDY OF SOFTWARE ASPECTS OF THE PHASE B SPACE F.H. Martin (Intermetrics, Inc.) Feb. 1972 263 p CSCL 09B G3/08

N72-21204

Unclas 24025



INTERMETRICS

OFFICE OF PRIME RESPONSIBILITY

EDS

263

CAT. 08

Volume I Final Report

Contract NAS-9-11778

ADVANCED SOFTWARE TECHNIQUES FOR DATA MANAGEMENT SYSTEMS

February 1972

STUDY OF SOFTWARE ASPECTS OF THE PHASE B SPACE SHUTTLE AVIONICS SYSTEM

Intermetrics Technical Report #12-72

INTERMETRICS INCORPORATED • 701 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

Foreword

This document is the final report of a study entitled "Advanced Software Techniques for Data Management Systems". The study was focused on an evaluation of key aspects of software for the Phase B Space Shuttle avionics system including the executive system, software verification, programming languages and computer features. This work was sponsored by the NASA Manned Spacecraft Center in Houston, Texas, under Contract NAS-9-11778. It was performed by Intermetrics, Inc., Cambridge, Mass. over the period of 16 June 1971 to 1 February 1972, under the technical direction of Mr. Joseph A. Saponaro. The technical monitor for the Manned Spacecraft Center was Mr. Donald Barron, EB5.

This final report is presented in three volumes:

Volume I: Software Aspects of the Phase B Space Shuttle Avionics System

Volume II: Space Shuttle Flight Executive System: Functional Design

Volume III: Programming Language Characteristics and Comparison Reference

The publication of this report does not constitute approval by NASA of the findings or conclusions contained therein.

ACKNOWLEDGEMENT

Several Intermetrics personnel contributed to the preparation of material and the publication of this report:

Dr. Fred H. Martin
Mr. Alex L. Kosmala
Mr. Daniel J. Lickly
Mr. Thomas A. James
Dr. James T. Pepe
Mr. Woodrow H. Vandever
Mr. James H. Flanders

Table of Contents

INTERMETRICS INCORPORATED • 701 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

Chapter 1

Objectives and Summary

1.1 Introduction

Past experience has shown that software is a significant factor in the costs associated with development of manned spacecraft systems. Since software is an integral and important part of the total system it is essential that it be given consideration during the system definition.

Design concepts for the reusable Space Shuttle vehicle recommended during Phase B include a centralized integrated avionics system approach. The baseline avionics system designs incorporate a quad redundant central computer system which performs all primary processing functions for the entire Shuttle mission. Various levels of redundant avionics equipment, required to achieve multiple failure tolerance requirements, are interfaced to the computer system via a high speed, time multiplexed data bus system.

1.2 Study Objectives and Scope

This study is concerned with the evaluation of several key software aspects of the Space Shuttle Phase B avionics system design. Five primary software areas were addressed:

a) Flight software executive system. The primary objective of this study was to perform a top level functional design of the flight software executive system. Executive functions of task management, scheduling and control, I/O management and configuration management were emphasized in the design. Several key aspects of the overall executive

structure were analyzed,such as: synchronous versus asynchronous control structure, memory allocation and sharing schemes, central I/O control, interrupt handling and high speed data bus I/O. The design is based on the application software requirements of the Phase B avionics system. This task was given major emphasis during the study.

b) Software verification. Since the major cost of developing aerospace software be attributed to verification, a secondary study objective was to devise a comprehensive approach to software verification, including a definition of verification levels and the roles of facilities to support them. The emphasis was directed at defining an overall approach toward more reliable Shuttle software development in order to lower the high costs of testing and validating flight software. This task was given secondary importance during the study.

c) Phase B baseline avionics system review. The objective of this task was to review the Phase B baseline avionics systems design and identify the major software implications. The scope was limited to a general review of the designs with emphasis on the architecture of the computer configuration, redundancy management, onboard checkout functions. The purpose was to evaluate their impact on the organization of flight software and its verification. In addition, a summary of the major characteristics of the application software was obtained for use during the course of the study in terms of size, speed and general processing functions.

d) Higher order languages and compilers. The objective of this task was to investigate the role that a higher order language compiler should have in the development of flight software for the Space Shuttle. This task was limited in scope and was primarily directed at:

1) evaluating software areas for which a higher order language approach is difficult, and

2) evaluating the role of other languages such as crew language and checkout languages.

In addition, methods of improving compiler efficiency were examined.

e) Computer hardware features. Since the architecture and hardware design of the computer system have a direct impact on the software, the objective of this task was to identify those computer features which are desirable from a software point of view.

1.3 Background and Approach to the Study

1.3.1 Background

Heavy penalties in cost and time have been paid for underestimating the manpower and time necessary to produce effective qualified and documented flight software. The production of Shuttle flight software will be a complicated and lengthy process and will involve at least the following activities:

a) generation of mission requirements;

b) generation of functional software requirements;

c) generation of software design specifications for: executive and operating system, pilot display processing, telemetry, autopilots, guidance, navigation, subsystem monitoring and control (radar, power, etc.), onboard checkout, data management, configuration and sequencing control, utility and support programs (compilers, simulators, diagnostics) and operational mission programs such as targeting, rendezvous, entry and landing;

d) preparation of code;

e) debugging and testing of code;

f) modification of design;

g) verification and demonstration of software;

h) maintenance of software.

Superimposed on the software development cycle above are the requirements for adequate documentation, and management visibility and control. These procedures are necessary to measure program progress, cope with design changes and change controls, insure high quality output, and respond to the pressures of developmental and operational schedules.

In the development of flight software for Apollo, the magnitude of this task caused many problems and delays, primarily in software debugging, verification, and change control. The difficulties experienced can be related directly to the fact that the software design exhibited little overall structure and the programmers themselves had few flight computer hardware aids and little programming language help. Software verification became the single most time consuming and laborious activity because neither the avionics configuration, the computer, nor the programming techniques were designed with checkout in mind.

The MIT Draper Laboratory has reported that fully 80% of the total Apollo software effort was expended on verifying and qualifying flight software. A recent Rand Report [1] corroborates the significant cost of verification by estimating that approximately 50% of the software dollars for SAGE and GEMINI went toward verification. In addition, these factors contributed to the set of "traditional" software difficulties, namely: unpredictable schedules, poor visibility of program status, undefinable software quality, residual unreliability, and spiraling cost.

1.3.2 Approach

Accordingly the approach to this study has emphasized the selection of features, tools, and techniques which aid in software verification. The philosophy proposed is one of improving software quality through careful design and structured software rather than through an iterative search for errors during verification (i.e., "an ounce of design is worth a pound of testing").

In this final report, the Shuttle software development process, the necessary programming language(s), computer hardware features, avionics configuration and executive design are examined in great detail. All are considered with a view toward creating reliable Shuttle software at reasonable cost. Techniques such as top down structured programming, use of a higher order programming language, comprehensive computer hardware and diagnostic facilities are analyzed. Furthermore, an effort has been made to eliminate software problems which in the past imposed difficulties during verification by designing them out. The proposals concerning the structure of the executive system, the handling of interrupts, and the allocation of memory are aimed toward this goal.

Finally, the inclusion of software from the start as an integral part of the total system design will be of immeasurable help. The interrelationships of hardware and software and their effects on overall reliability, demand an integrated design effort.

1.4 Task Summary and Review

1.4.1 Executive System Design Task

The executive system design is presented in Volume II of this report. The design was based on the application software requirements derived for the Space Shuttle Phase B avionics system and is specified for implementation on the IBM 4 Pi EP computer system within the NASA breadboard data management system. Although a detailed summary is provided in Volume II, an overview of its main features is provided here.

a) Structure. The executive system structure is based on a high priority synchronous "foreground" for execution of cyclic tasks and an asynchronous priority controlled background for other applications software. The synchronous foreground is initiated by a timer interrupt at a fixed frequency, with the scheduling and sequencing of each computation in a cycle predetermined and specified via control sequencing tables. After completing execution of the computations each minor cycle, the executive dispatches the processor to one of the "ready" tasks in the executive ready queue on the basis of priority. A total of three priority levels has been established for application programs.

b) Interrupt and task dispatching. Interrupts are immediately serviced by the executive and entries are made in appropriate queues. The interrupted task then resumes and continues until it either ends or until it reaches a segment dispatch point. Only then is a higher priority background task activated by the executive dispatcher. Long duration tasks can be organized into reasonable execution segments with task swapping or interruption points being more predictable. The dispatching of the cyclic task controller each minor cycle is, however, an exception and is executed immediately at the occurrence of the minor cycle clock interrupt. This exception is made as a reasonable tradeoff to provide the timing and response characteristics needed for cyclic computations.

c) Task and event scheduling. Any executing task may request the executive to schedule another task on the occurrence of an event or a specified time. Events are system defined in scope and may be posted or deposted by applications tasks via the executive.

d) Memory organization and allocation. Programs are defined as either total mission resident or mission phase resident. Phase resident programs are loaded from the secondary storage device into their assigned portion of the operating memory by the phase initiation function of the executive. Dynamic memory is allocated to each task by the executive, when it is made ready for execution, out of a subpool of working memory established for each priority level.

   A portion of the memory is dedicated to shared data. It is organized into mission dependent resident data and an overlaid area for phase dependent data. All access to the common data is controlled through and by the executive.

e) I/O control. Control and execution of all input and output operations are performed by the executive system. Input/output services are performed in two modes: on demand via request by an executing task, or table driven as in the case of cyclic computations in the synchronous mode. Secondary memory management is under the control of the executive.

f) Configuration management error recovery. The executive responds to all system hardware and software detected error conditions and supervises reconfiguration of the system. A standard system error recovery action is defined for each error class. During execution, application tasks may invoke local recovery for a class via specification of a task reentry point.

1.4.2 Software Verification Task

An approach to the development of more reliable software is presented in detail in Chapter 2 of Volume I. The thesis put forward is that definitive statements of software quality can only be made by careful examination of the basic program structure. This structure must be carefully and explicitly created during a thorough initial software design phase.

The traditional role of software testing is shown to be one of demonstrating that the software can meet a finite set of requirements rather than verification of an error-free program. Verification, on the other hand, must be performed from the beginning as an integral part of the implementation,

and consists of a combination of visual examination ("eye-balling") and selected testing based on program structure and subroutine interfaces.

It is suggested that the verification procedure is enhanced through the discipline of structured programming. The technique proposed is best described as "top down" programming. The overall control structure of a program is implemented first, and run, calling upon dummy subroutines. After its operation is verified the subroutines are "filled-in, as you go" with actual code, always keeping the total program in operating order. Programming proceeds in this top-down fashion until all model subroutines ("dummies") are replaced by intended code. The objectives of this technique are to provide "at a glance" understanding of the functions of the program and an ability to achieve an operating status at a very early stage of development. A "top down" assembly of structured programs can be continuously exercised throughout development,providing continuous integration and thereby confidence and visibility of status. The use of a higher order programming language is promoted as an important tool in facilitating this structured approach to Shuttle software.

The facilities required to support the phases of software development are presented in Chapter 3. Four phases of software development are identified: Phase 1, software requirements, Phase 2, implementation code and test, Phase 3, verification through software, and Phase 4, verification through hardware. An all-digital software development facility is recommended to support Phases 1, 2, and 3. A hybrid test bed avionics integration facility is used for Phase 4.

The software development facility (SDF) consists of a large scale commercially available computer system augmented with disk type storage drives, printers and other data processing peripheral equipment. A higher order programming language and compiler for both the host and flight computer are presumed in support of Phases 1 and 2. The use of an interpretive instruction level digital simulator is recommended during Phases 2 and 3. Methods are suggested for improving simulation speed including an efficiently tailored simulator, simulation advance through periods of idle computer activity, and maximum use of HOL programs by direct testing on the SDF host computer.

An all digital avionics environment simulation is recommended for use during the requirements phase (Phase 1). During Phase 3, it will be augmented and interfaced to the interpretive computer

simulator providing "feedback" to the software requirements. The SDF will provide both interactive and batch operation,with limited user interaction during simulations. The avionics integration facility will consist of actual flight hardware supported and controlled by an environment computer.

1.4.3 Space Shuttle Phase B Review Task

The Phase B avionics system designs defined by North American Rockwell and McDonnell-Douglas Aircraft Corporation were reviewed. The software implications and features are summarized in Chapter 4 and Appendices A and B. Both Phase B designs have many similarities. Both employ centralized integrated avionics systems with multiple levels of redundant avionics equipment. The central software of both systems is organized around a synchronous structured executive system which includes functions of guidance, navigation, flight control, displays, checkout and configuration management. It is estimated in size to be approximately 50K of 32 bit words with maximum processing speed of up to 275K adds per second. The synchronous versus asynchronous method of control is analyzed in Volume II with a recommendation for a combined synchronous and asynchronous structure.

The multiple failure tolerance requirement of "fail operational" after failure of two critical components and "fail safe" after the third failure is identified as introducing the greatest complexity in the Phase B designs. The control and management of the various levels of redundancy in the system have significant impact on the software, particularly in the executive system. Several general implications of the centralized Phase B designs are discussed: central computer software management problems and I/O timing difficulties with a shared multiplex data bus.

Those differences in the baseline avionics configuration features, which effect software, are identified as:

a) computer organization and redundant operation,

b) operating memory and reconfiguration,

c) secondary storage and utilization,

d) subsystems redundancy management and interfacing,

e) operation of display systems.

A discussion of these features in each configuration and its influence on software is provided in Chapter 4.

A review of onboard checkout software was conducted during this task to determine its requirements and interfaces with the executive system. It was determined that checkout software has similar requirements to other flight software and can be implemented within the framework of the executive system described in Volume 2. That is, functions such as subsystem status monitoring, displays and other cyclic processing can be accommodated by the synchronous foreground structure and event driven processing such as diagnostics, recovery and crew requests, can be controlled by the asynchronous background.

1.4.4 Higher Order Programming Language Task

The role and types of languages for the Space Shuttle onboard software are presented in Chapter 5. A general purpose higher order programming language (HOL) is recommended for use in developing flight software. It is proposed as a significant step towards a more orderly and controlled production effort. It is also recommended as an essential ingredient of the structured programming approach. Supporting justification and rationale for the HOL recommendation are reviewed in Chapter 5.

A crew language is identified for use by pilots and other crew members to communicate with and command the computer. The crew language must be designed to enable insertion of data, control over program module and processing flow, and general support of crew interaction. This language is not used for software development or on-line compiling.

The features and characteristics of eight programming languages have been tabulated and are presented in Volume III including PL/1, HAL, SPL, CLASP, FORTRAN, ALGOL, MAC and JOVIAL. SPL Mark IV and HAL contain many general purpose features applicable to a wide variety of aerospace software applications, including the Shuttle.

Although it appears reasonable that most flight programming can be done with a general purpose HOL, the difficult areas of machine dependent coding such as system programming are discussed. The use of direct machine language intermixed with HOL statements is not recommended. It is recommended that if special machine dependent features are required, then they should be provided through a special subset of the general purpose development language and be restricted in use.

It is also recommended that if higher level, problem oriented type languages (such as checkout) are required, that they be linked to the general purpose HOL compiler. As discussed in Chapter 5, statements in all languages are ultimately directed into a single compiler system to facilitate standardization and automatic checking performed during compilation. It is recognized that a "single compiler" approach for both lower level system programming and problem oriented languages is not the most flexible. However, it is motivated by a goal of producing quality flight software of high integrity and reliability which may only be achieved through conformance to a highly structured and controlled environment.

In a subsequent section, the advantages provided by a procedure oriented higher order language and compiler for structure and modularity are discussed: independent compilation, compool control of shared data, block structure, access rights to shared data, and automatic checking features.

The final section of Chapter 5 discusses compiler implementation. The chief complaint regarding use of HOL compilers has been inefficient generation of machine code. The use of a "software interpreter" executing and intermediate language is discussed as an approach to conservation of memory. Microprogram implementation of the interpreter is suggested as a possible approach.

1.4.5 Computer Features Task

Several computer features and architectural characteristics are identified as desirable from a software viewpoint in Chapter 6. While it is recognized that these features have tradeoffs associated with hardware complexity, cost, and availability within off-the-shelf hardware, they are presented as valuable to the software production effort as well as a trend for future flight hardware. A summary of the major computer features discussed in the chapter is provided below.

a) It is recommended that the flight computer selected for the Shuttle should possess features to enable efficient execution in a higher order language environment. Ideally, it would be designed as a higher order language machine. The design of a machine which matches the language will not only improve processing efficiencies but will improve performance and reduce memory requirements.

b) Hardware stacks are identified as desirable for the management of nested procedures and efficient execution of arithmetic statements.

c) Microprogramming is desirable to optimize instruction sets, word length and particular processing aspects of Shuttle software. It provides significant flexibility which is desirable over the long life of the Shuttle through its ability to modify and change microprograms. Specialized spacecraft functions such as data bus servicing can be absorbed into the high speed microprogram and improve performance significantly.

d) Descriptors are desirable features to provide hardware checks of proper use of data.

e) Hardware implemented run time diagnostics features are desirable for direct execution and debugging of software, thereby reducing digital simulation requirements. These features can be provided as options in a ground based version of the flight hardware.

f) The addressing scheme is identified as one of the most important characteristics of the computer. General addressing schemes are reviewed. The "zero address" or stack machine is shown to be most efficient. Most current aerospace computers shown to be "two address" machines containing general registers.

The need for both static and dynamic addressing is recognized for the Shuttle. Indirect addressing, indexing, and base registers are all recommended as desirable for this environment.

g) Three methods of subroutine linkage are reviewed: return address in a memory, register or stack. The stack is identified most desirable to software.

h) Floating point data representation is strongly recommended for use on the Shuttle. Appendix D provides a detailed discussion of this recommendation. Appendix E provides an analysis of 32 bit single precision floating point word size as to its adequacy in navigation computations.

i) In lieu of microprogramming, unimplemented op-codes are identified as a technique of obtaining specially tailored system instructions or routines.

A summary of major flight computer features identified for Phase B is presented in Chapter 6 including: physical characteristics, processor, memory, and production availability.

Finally, higher order language benchmark programs tailored to the Shuttle application software are suggested as an additional measure of the relative performance of candidate Shuttle computers.

Reference for Chapter 1

1. Boehm, B.W., "Some Information Processing Implications of Air Force Space Missions: 1970-1980", Memorandum RM-6213-PR, Rand Corporation, Santa Monica, California, January 1970.

Chapter 2

Software Verification

2.1 Introduction

The question of how to develop reliable flight software in a cost effective manner is of overriding concern to the Shuttle Program. Although methods employed during the Apollo development were successful in that Command Module and Lunar Module computer software did the job and suffered few significant anomalies, the effort expended in manpower and program testing time, and the residual uncertainty in product reliability leads one to search for surer, more efficient ways to achieve program quality.

The production of man-rated real-time operating software for the Shuttle will be a complex process of interrelated activities: generation of requirements and specifications, design of algorithms and methods of implementation, coding and testing, verification, management, change control, operations, and field support. Each activity can bear a direct responsibility for contributing to software reliability. In an effort to better organize, expedite, track and smooth the process, a number of improvements might be suggested; thus

a) more detailed specifications

b) initial design documents

c) detailed design documents

d) detailed test plans

e) automatic decoding aids

f) automatic flow charts

g) multiple levels of testing

h) automatic diagnostic aids

i) automatic test aids

j) automatic evaluation aids

k) management review of milestones at all significant points of development

l) strict configuration control of coding and testing

m) strict procedures for change control

n) control and timely production of all necessary documentation

No one can doubt that many of the items listed above will enhance the production of Shuttle software, but still one might ask "will they insure bug-free programming?". It certainly will be more convenient and less error prone to set up test runs and conduct them using automatic aids for initialization, edits, evaluation, etc., but will these features find, or help to locate, software errors? And, further, how can it be determined that no more errors exist and the testing phase is completed?

For any large, non-trivial programming effort these questions have remained largely unanswerable. The approaches taken, for the most part, include establishing the closest control over the coding and changes to the coding, demanding comprehensive documentation and conducting elaborate and usually exhaustive test programs. While recognizing the value of these approaches, this report contends, nevertheless, that the reliability of software is best improved by proper design, structure, methods of implementation, and through the use of software and hardware techniques which can prevent errors in the first place. For example, Apollo flight software anomalies were caused mostly by errors due to erasable conflicts, scaling difficulties and imperfect restart logic [1].

Prodigious amounts of engineering and computer time were spent in diagnosis and testing for this kind of error. If the Shuttle software utilizes floating point, automatic data sharing through hardware and/or software locks, and a single instruction restart machine, then the most prevalent Apollo-type errors would be eliminated, not by controls, test plans, test facilities, etc., but by design.

In the sections that follow, it is design responsibility, design methods and careful implementation procedures that are stressed. The hope, and objective, is that both quality and cost efficiency will be gained by entering the verification phase with an inherently more reliable software product. Verification can then proceed, quickly and smoothly, to demonstrate compliance with requirements instead of having to bear a major responsibility for debugging an imperfect program.

Three principal ideas are espoused:

1) Coding must be preceded by a thorough software design effort in which the methods and procedures of implementation are carefully formulated and specified.

2) The application programmer's machine interface is built upon layers of virtual machines where each layer only uses the facilities provided by the "machine" (layer) directly below it. Access to internal layers from above is disallowed.

3) The programs are implemented in a "top-down" manner beginning with the overall control and concepts, and are gradually refined by filling in the detailed code. Programs are kept in operating condition while still under development (using models) so that problems of integration are faced and solved along the way, rather than being postponed until all coding is submitted.

In summary, more emphasis is placed on building in software quality by design, and less on the relentless search for errors after the fact.

2.2 Requirements and Specifications

The production of Shuttle software is a process which begins when the broad objectives of the Mission Requirements are formulated. Its primary objective is to produce a concrete set of correctly functioning programs for the on-board computer(s). It is an activity spanning a wide spectrum of management and engineering disciplines, during which the statement of the overall requirements undergoes successive refinement and redefinition on the way to the final product. The ultimate form of this statement is the computer program itself.

The need for commonly accepted definitions of the software requirements, at levels in between the statement of mission objectives and the computer program, has been recognized in the past by the establishment of software functional specifications, software engineering specifications, program design specifications, etc. It is Intermetrics' view that the formulation of these specifications is an important part of the software production process itself, and must be integrated with it.

A specification must describe accurately and fully the functions it defines. No questions which require recourse to an authority outside of the specification for answers must be raised in the minds of those interpreting that specification. On the other hand, the specification writer must not impose constraints on the subsequent design activity by specifying details of implementation more properly deferred to that activity. Perhaps the most insidious software error is that due to the misinterpretation of a specification requirement by the implementer. It is impossible to devise rigorous verification procedures to detect failures of human understanding. Ambiguities and omissions in the specification must be eliminated in order to achieve "correct" programs.

The assurance of this kind of quality in a software specification implies that the development of such a specification is a design task in its own right, and must be considered a part of the overall software production effort. The objective of this section is to define the number, purpose, and the characteristics of a minimal set of specifications considered to be necessary for the implementation of Shuttle software, and to indicate how the design of such a set is to be approached.

2.2.1 The Formulation of Software Specifications

A four level hierarchy of Shuttle software specifications is envisaged. Four specification documents need to be identified, to define and describe the following:

a) Mission Requirements

b) Functional Requirements

c) Software Requirements

d) Program Design

The term "specification", where used in the remainder of this section, will refer to one of these documents.

Each level constitutes a refinement of the requirements of the previous one; each specification in the sequence yields a definition of the software in increasingly concrete terms. Each set of requirements is assumed to be organized in the form of a document, which is treated as the specification and reference manual for the next activity. Each of the above specifications will now be described in greater detail.

2.2.1.1 Mission Requirements. At the highest level are the Mission Requirements. These set forth the activities and objectives of the Shuttle concept. All intended mission phases are indicated from launch to landing, with definitions of all nominal, off-nominal, contingency, backup and abort situations. The Mission Requirements indicate broadly the operational functions of the Shuttle, the range of capabilities, the expected levels of performance, and the accuracies that must be achieved. The Mission Requirements are the specification for a detailed functional design of an operational system to accomplish the mission objectives. The result of this stage of design is a set of Functional Requirements which specify the manner in which the Shuttle will accomplish the objectives.

2.2.1.2 Functional Requirements. This specification will define the various operational phases, identify the functions to be performed, determine the sequence of operations, establish the interrelationships of the functions, and provide mission time-lines. The Functional Requirements Specification will identify and determine the scope and capabilities of the various on-board subsystems, and will

indicate the nature and degree of cooperation with ground based operations. It is during the design of the Functional Requirements that definitions about the number and types of the various subsystems, their functions, and the configuration of the system begin to be made.

The next stage of software design takes the form of a detailed analysis of the Functional Requirements, with the objective of establishing techniques for their realization. The end result of this very important phase is a set of Software Requirements.

2.2.1.3 Software Requirements. This Specification contains definitions of all the appropriate equations and algorithms to be implemented, details of critical timing and sequencing as demanded by the Functional Requirements, operational procedures, and definitions of operational interfaces between the computer/software and the rest of the Shuttle system (i.e., crew,subsystems, ground). The Software Requirements Specification contains all information that is required to embark upon a detailed structural design for the Shuttle computer programs. This includes, particularly, all appropriate data that is to be used, both fixed and variable. Dynamic ranges of all specified mission variables will be established and defined. The Software Requirements will constitute a specification for the design of the actual program.

An essential part of the Software Requirements Specification is a definition of the set of tests to which the completed program must be subjected in order to demonstrate its correct operation. The task of defining these tests requires knowledge of the design concepts for the whole system, and is not one to be left to the programmer. The test requirements will include prescribed test cases with initial conditions, mission parameters such as state vectors, weights, inertias, etc., and the specified levels of performance to be achieved, together with evaluation criteria to enable a judgement to be made.

2.2.1.4 Program Design. This final stage of software specification is perhaps one that has not, in the past received the attention it demands. Too often a basic structural software design has not been possible, perhaps because there is no time for it, because the job was not completely specified at the time programming began, or because already existing segments of software had to be

pressed into use. It is our contention that the establishment of a Program Design Specification is necessary to ensure that an overall program structure and operating philosophy be established at the outset. There must be a conscious effort to design this Specification. It must be based on a comprehensive set of Software Requirements, and it must precede the start of operational program coding. Because it is followed by implementation, the final stage of refinement, the Program Design Specification must address the set of problems associated with programming, checkout and integration. Section 2.3 will review in greater detail the approach to be taken in establishing this Specification.

2.2.2 Necessary Detail Within Software Specifications

The generation of definitive software specifications is a process of successive refinement, during which the scope of Shuttle functions being considered narrows to only those associated with the details of software operation, as those details become increasingly more apparent. The primary purpose of the specification documents is to provide tangible interfaces and control mechanisms between the groups of people, of varying disciplines and working styles, who will be involved in the process. The hierarchy of specifications (i.e. the several Requirements) can also be seen as stepping stones on the route to a program design, forcing an evaluation of the design at intermediate points. One purpose of this hierarchy is to prevent selected aspects of the total system design from being taken prematurely from conception through implementation, or, conversely, from falling behind. An uneven rate of development creates inflexibility and discourages the across-the-board compromises which are always necessary as a design solidifies. This principle will be invoked again in the discussion of program implementation (Section 2.4).

In order to maintain the desirable flexibility it must be emphasized that each specification is only the starting point for the next stage of design. A set of requirements should avoid dictating, or even implying, the use of specific techniques of organization or design that might appear to be necessary to achieve these requirements. It is important to preserve the choice of design policies, so that overall compromises can be made. The tendency to become too specific is a common fault in specification design. It is difficult to resist as the system becomes progressively more clearly defined and seemingly obvious techniques of implementation begin to present themselves. For example, although the Software Requirements Specification should present all the

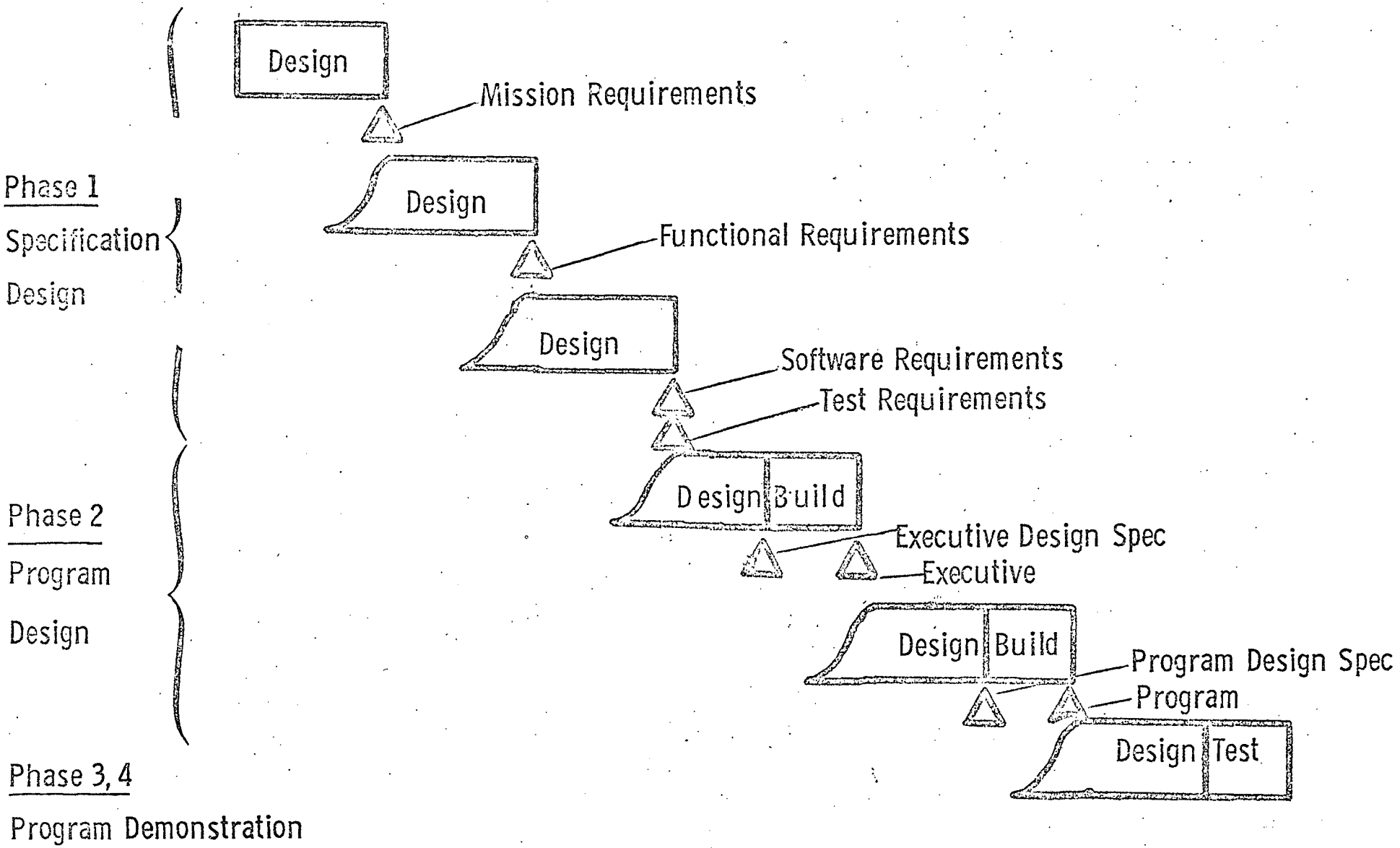
information required by a program designer to create and deliver the software, it often specifies, as well, how it is to be structured. Functions are presented as software modules with the interfaces among modules already defined. A specification that goes to these lengths not only absorbs unnecessary effort during its design, but creates further waste because it may have to be unravelled before a real program design can begin. If such over-specification is not recognized, but is promulgated in the design of the program, basic options of organization and design may no longer be available. Furthermore, the application of the principles of structured programs (to be discussed in Section 2.3) is impeded, lessening the possibility that reliable software can be produced and making modification and change procedures more difficult to apply.

2.2.3 The Phases of Software Development

Shuttle software production must be considered as a continuous process, which is initiated by the formulation of the mission requirements, and does not end until after the delivery of a validated set of flight-ready programs. The nature of the activities involved in this process naturally undergoes significant transformation as the software evolves through specification, design, implementation, testing, integration and, eventually, to operation. These activities can be grouped into three phases, characterized broadly as specification, implementation and verification activities. The disciplines of the personnel, and the techniques and facilities that are involved in these phases differ sufficiently to suggest these categories. The remainder of this section will be devoted to a brief description of each phase and its product.

2.2.3.1 Phase I: Specification. The basic characteristics and content of the Mission, Functional and Software Requirements documents generated during this phase have already been discussed in Section 2.2.2. Although shown as separate activities in Figure 2-1, the work leading to the publication of these three specifications cannot be conducted independently. Each specification must acknowledge fully the capabilities and limitations of the activities it seeks to direct. The Functional Requirements designer must be aware of equipment limitations. In generating the Software Requirements, the basic problems associated with program design and checkout must be appreciated. This may seem obvious, but in practice

Figure 2-1 The Software Process



these principles have often been violated. As a consequence a subsequent activity experiences problems that should have been anticipated earlier (e.g. inadequately defined hardware interfaces, or failures to specify the dynamic ranges of intermediate results). To minimize these occurrences, specifications must be formulated, reviewed and iterated with participation from all phases of software design.

As the definition of the software progresses towards its embodiment in the actual program, the specification becomes successively more quantitative and requires an increasing degree of analysis and computation. This is especially true in the case of the Software Requirements Specification which contains the necessary equations, algorithms, loop gains, coefficients, sampling frequencies, etc., so that no aspect of the performance of the system need be established during the program implementation phase. This implies that a thorough analytical investigation is necessary to devise the techniques by which the performance specified in the Functional Requirements may be attained. The development of this specification is therefore a major undertaking. It is our contention that specification design should rank in importance and emphasis with the design and development of the computer program itself. Too often it is left to the flight program personnel, already hampered by an unproven program and by a simulation environment ill-suited to the purpose, to fill the gaps of an inadequate specification.

An analytical specification design effort requires the support of adequate tools and facilities. The evaluation of the closed loop performance of the Shuttle thrust and attitude control systems, for example, requires an accurately modeled simulation of the vehicle's dynamics, its stabilization and thrusting subsystem, and other appropriate guidance and control functions. If such a simulation is a requirement for this stage of software design, then it is natural to ask whether it can also be used in the later stages of verification. It will be suggested in Section 2.5 that a modular simulation organized with this objective is feasible. Its modular elements can be utilized, either individually or in smaller subsets, to support the needs of Phase 1 analysis tasks. In its integrated configuration it can provide the comprehensive all-digital simulation environment postulated for the demonstration testing of Phase 3. A modular environment simulation offers two significant advantages:

a) only one simulation design effort need be undertaken to satisfy the needs of two distinct activities.

b) the simulation models that are used in Phase 1 to formulate software concepts, are themselves verified at a later stage, by comparing their performance against that of the more real equipment in the system integration facility. This concept will be developed more fully in Section 2.5.

An important aspect of software production is testing. Although it will be shown in Sections 2.3 and 2.4 that there are good reasons to change the customary view that testing is the basis of verification, software testing will always be necessary. As a consequence the task of specifying and defining all the test cases, and the problem of establishing the test criteria will have to be faced. The responsibility for test case design belongs properly to those charged with developing the Software Requirements. During this stage the basic concepts of system and software design are formulated, and the software performance characteristics are established. The design of tests that demonstrate the correct implementation of these concepts requires considerable effort from personnel who have knowledge of the overall system operation.

Special care must be taken to eliminate uncertainty in the mind of the tester when he is asked the question "was the test successful?". The performance of closed loop control system always falls short of that established by the ideal, weightless, inertia-less systems upon which concepts are based. The degree of acceptable mismatch must be established in the Software Requirements Specification so that the software tester will have to exercise an absolute minimum of judgement in evaluating program performance. This is especially true for tests conducted by the Phase 2 personnel, the programmers. Their primary objective is to deliver correct code, not to evaluate say, the performance of a minimum fuel thrusting maneuver. Test case design cannot be performed by those already overburdened with the demanding task of designing and verifying the details of program implementation. The test requirements, therefore, will form an essential ingredient of the software requirements document, to be delivered at the conclusion of Phase 1.

2.2.3.2 Phase 2: Program Design. This phase accomplishes the design, implementation and testing of the actual flight program to the Software Requirements established in the previous phase. The details of how this is to be done will be described in Sections 2.3 and 2.4. At this point only a brief summary will be given. Figure 2-1 illustrates the two major activities of this phase; the executive design, and

the design of the operational programs. Both consist of a pure design phase followed by an implementation phase. At the end of the executive design phase, which will apply the principles of "layered" structure to be described in Section 2.3, an Executive Interface Specification document will be published. To the applications programmer this becomes the definition of his interface with a "virtual" machine. It will specify how he will schedule tasks, handle I/O,control timing, etc. It will obviate any necessity for him to become familiar with the real details of machine hardware structure, and will prevent him from interfering with such basic operations as scheduling, memory allocation I/O, interrupts, etc. The publication of the Executive Interface Specification signals the start of executive program coding and test, and the start of the design phase for the applications programs.

Applications program design is perhaps the most important activity in the whole software development process. Its purpose is to establish an overall structure for the computer program before the coding and testing begins. The main objective is to create a framework that logically and sequentially interrelates the various program modules which are also identified during this phase. This represents the first opportunity to apply in an initial layout of the actual program itself some of the principles of structured programming and top-down implementation. This subject is explored fully in Section 2.3. The applications program design phase is complete when the Program Design Specification document is published. This is the last in the series of specifications that together define the software design process. From this point on program coding, integration and testing can proceed, with the confidence that no major design hurdles have still to be overcome. Effort can be concentrated on the creation of a well defined, visible, program whose dynamic characteristics can be more easily assessed from its static structure, than has been the experience of the past. This subject is analysed further in Section 2.4.

2.2.3.3 Phase 3 and 4: Verification. At the end of Phase 2 the flight program is complete and "internally" verified. It now enters the final phase of testing before being exposed to the actual flight environment. The principle objectives of this phase, which will be described fully in Section 2.5 are:

a) to demonstrate correct software operation according to the Software Requirements Specification;

b) to confirm that the concepts formulated during Phase 1 are still effective in an environment which more closely resembles that of the real world;

c) to confirm the adequacy and validity of the modeled environment used during Phase 1 and 2 to provide quantitative test and performance data.

The rationale for dividing the verification activity into two distinct phases will be presented later, together with a description of the techniques and facilities to be used.

2.3 Program Design

It is proposed to build reliability into the Shuttle software, by design, through the technique of "structuring" the flight programs. The essence of this approach is "modularity with structure". This section will explore its application under the principal headings of

a) Program structure,

b) Program modularity,

c) Influence of language,

d) Program design process.

2.3.1 Program Structure

A large, complex program must be visualized as an hierarchical organization of functions, rather than as a collection of somewhat arbitrarily defined program modules. The overall objective of a computer program is to execute a translation from the highly sophisticated requirements of the mission and environment into the relatively simple logical statements required to control the computer hardware. This translation can be conceived in a series of steps, in which each step performs a translation between intermediately defined procedural and logical interfaces. In this way the interaction between functions in the software can be limited, defined, and more easily controlled. Such a structured approach has been successfully applied to the design of operating systems for large general purpose facilities [2,3,4]. It is proposed to apply the principle to the whole of the program in the Shuttle computer, including the applications program. Because of the strong influence of the real-time

requirements, structure becomes even more desirable, in order to achieve effective separation and modularization of functions.

The hierarchical, or layered, organization can be represented as in Figure 2-2. The machine itself is depicted as the lowest or zero level. The actual hardware/software interface is therefore between Level 0 and Level 1. The "tiered cake" appearance of the figure is due to the choice of coordinates. The horizontal scale represents interface complexity, defined as the number of system variables controlled by that interface. The vertical scale indicates the level of sophistication or abstraction of the function from the interface. The higher the level the more sophisticated the "virtual machine" becomes. The highest level is ultimately the crew member, who uses the computer to accomplish the appropriate mission phases. He may have only a few options of control; for example, he may be able to select major mission modes such as "ascent", "insertion", "rendezvous", with perhaps a few variations, as specified by the Mission Requirements. Although limited in number, these are very sophisticated functions indeed.

The layers are so organized that each level interfaces only with those levels immediately above and below it. A given level is unaware of any detailed structure within the "level" below it. The functions performed by the software within that level may, in fact, regard the lower interface as a "virtual machine" [2], whose properties are defined by the requirements of that interface.

The decomposition of the Shuttle software functions into a succession of layers can be illustrated as in Figure 2-2. The function of each of these levels is as follows:

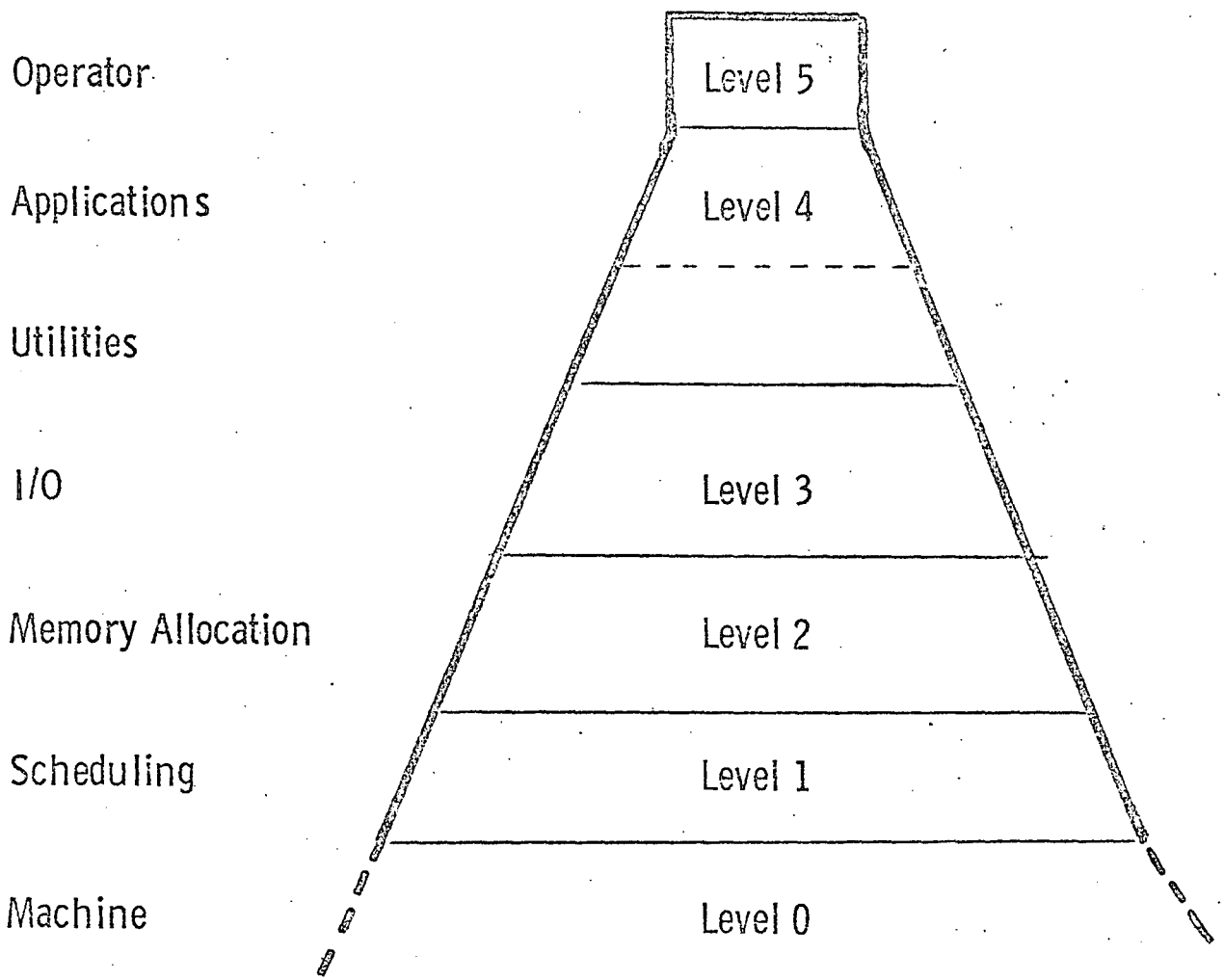
Level 0: The machine

Level 1: The control of processes. The allocation of the processor(s) to active jobs, i.e., scheduling and dispatching. Also the control of response to interrupts, and the resolution of priorities.

Level 2: The responsibility for allocating space in operating memory, and if it is provided, the control of secondary storage.

Level 3: The management of I/O operations. This might include the interfaces to the displays, manual inputs, and the Shuttle data bus.

Figure 2-2 Layered Functional Structure



Level 4: The application program functions. A sublevel of Level 4 might be a collection of program utilities which provides standard higher mathematical functions, coordinate transformations, integration algorithms, etc., not already available within the programming language.

Level 5: The crew member.

These are only broad suggestions, offered in illustration of the principle. Other categorizations have been made [2,3]. Levels 1 through 3 fall into a category normally referred to as the operating system. From a structured viewpoint, an operating system has, in the past, been somewhat arbitrarily defined. It is normally considered as an entity, rather than a subset of hierarchical levels, because the collection of executive functions is usually considered to be invariant, a part of the machine. The applications programs, and their requirements, are often quite unknown until late in the development of the system. In the Shuttle environment, the totality of functions is known and well-defined, and the layers of program functions may be optimally organized.

The first stage of program design is to identify the various levels, and to define their functions. The interfaces of the levels with each other are then determined, and will form the basis for the implementational techniques to be described in Section 2.4. The constraints to be observed in organizing the software into a hierarchy of functions will obviously be, at the highest level, the requirements defined in the Software Requirements Specification, and at the lowest, the characteristics of the computer.

2.3.2 Program Modularity

To be containable and manageable, program code must be capable of being organized into sizable modules, containing groups of logically related statements, or instructions. It is advantageous if the number of different types of such modules is kept to a minimum, so that simple rules can be devised to enforce their control. Past experience with software for real-time aerospace control applications indicates that perhaps only two types of module need be defined, and that their desirable characteristics should be as follows:

a) Data Modules

Prescribed inputs are processed in a logical or computational fashion and outputs are delivered back to the caller. Access to common data is allowed only through the rigid supervision of a common data pool (COMPOOL) control mechanism.

b) Control Modules

This module is scheduled by the executive and may call Data Modules as necessary to perform the basic processing tasks. A Control Module may schedule,via the executive, other Control Modules. An objective in defining this type of module is to bound the program control functions so that the number of executive interfaces is minimized. A well defined control module is thus independently operable, without being unmanageably complicated or large. The purpose behind this concept is to create software entities that are visible, comprehensible and testable; i.e., easy to conceive, construct and verify, to set up and run, with well-defined, limited modes of operation. Their functions are obvious by examination, and their performance under test can be readily observed and evaluated.

The software modules created by application of these definitions do not necessarily bear relationship to the functional elements of former, typical software requirements specifications. It is on this point that a departure from previously established procedure is suggested. Formerly, the software specification was organized into convenient functional portions, each of which played a visible role in the operation of the avionics system. These elements were formerly implemented with little reorganization for optimal implementation, verification, and execution, so that they became intimately and often randomly entwined with each other and with the system programs. The result was a complex structure that became very difficult to debug, verify, or modify.

During Phase 2, the program will be broken down into an assembly of Data and Control Modules. The performance and interface specifications for these Modules will be established as part of the Program Design Specification. The Module specifications will include

a) definition of all input and output parameters and their characteristics (e.g., dynamic range, scaling, precision, formats, etc.);

b) all timing details, both internal and external;

c) all priority and other scheduling information;

d) definition of interfaces that the module shares with other modules;

e) definition of all logical and computational operations to be performed within the module.

A higher order compiler language can be instrumental in achieving the first objective. A language which provides for the definition and manipulation of program blocks will allow Data Modules to be more easily defined, maintained, and their intended use to be enforced. A real-time control capability can assist the definition of the Control Modules and enforce their proper use. A specific language, HAL, will be discussed in the next section, and the relevance of its structure to the attainment of program modularity will be indicated.

2.3.3 The Role of the Programming Language

The advantages and disadvantages of the use of a higher order language for the programming of the Shuttle computer is discussed in a broad context in Chapter 5. In this section only those aspects which have relevance to the structuring of programs will be considered.

The concept of structure in software depends, initially, on the ability to perceive modularity of function and behavior. This must be followed by techniques which can exploit the modularity. Such techniques must be supported by mechanisms which can be invoked to enforce them. Higher order languages are attractive in this regard, especially those which exhibit structural characteristics such as PL/1, ALGOL and HAL. However, the principles of structured software can also be enforced by convention, or by an assembly language tailored for structure [5]. The aspects of a language which have greatest bearing on structural program design are rules defining program modularity, name scope and data usage, and those that limit and control the interface with the executive, especially in the area of scheduling. The HAL programming language will be used to illustrate the application of these rules.

2.3.3.1 Modularity. HAL imposes program modularity by recognizing only certain organizations of code. A HAL Program consists of one or more, independently compilable and schedulable, "Programs", and a common data pool [6]. A Program consists of in-line coding, except for CALL and SCHEDULE statements. For example, Program #1 might be rendezvous navigation, Program #2 autopilots, and Program #3 the onboard checkout system. A Program may call one or more Procedures, and schedule, via the executive, one or more Tasks. Tasks and Procedures may, in turn, call one or more Procedures. A Procedure may not schedule a Task. Tasks and Procedures can not be compiled independently of Programs. All these organizational blocks are self-contained. They may be entered only at a single location, and exited at a single location. With few exceptions, all HAL coding must be arranged into Programs, Tasks or Procedures. These basic elements, and the rules that govern their interaction, are illustrated in Figure 2-3.

2.3.3.2 Name Scope. The structure of HAL also provides for control and protection of variables and routines, through name scope. Name scope restricts the accessibility of program names. Variables that are required to be recognized and available globally, throughout the Shuttle computer program, are assigned to a centrally organized and controlled common data pool, or compool. The vehicle's inertial state vector is an example of a compool variable. Any other variable is known only within the scope of its name. The scope of a name is defined as the Program, Procedure or Task in which it is declared. Name scope includes all Programs, Tasks or Procedures called by the block in which it is declared. But it is not known outside of the block, and therefore a name cannot be accessed in any way from outside the region of its scope. In addition, there is no confusion between variables or routines which have identical names, but are of different scope. The application of name scope means that many separate program functions can "live" together in the same computer, and yet remain isolated and unaware of each other. They are incapable of writing over, or otherwise interfering with variables or locations that are not mutually defined.

2.3.3.3 Scheduling. In addition to constraining the choice of code organizations, and restricting the freedom of program variables, HAL also enforces rules that determine the dynamic relationships of Programs and Tasks, to each other and to

the executive. Figure 2-3 depicts all the blocks that are definable in HAL. The following rules, which are additional to those cited in Figure 2-3, illustrate how these blocks may relate to each other:

a) P_1 may schedule P_2, and vice versa.

b) P_1 may not schedule T_3, because the name of T_3 is not known outside of P_2 (by name scope rule).

c) T_1 may schedule T_2.

d) T_1 may schedule P_2, because names known to P_1 are also known to T_1.

e) T_1 may call R_2.

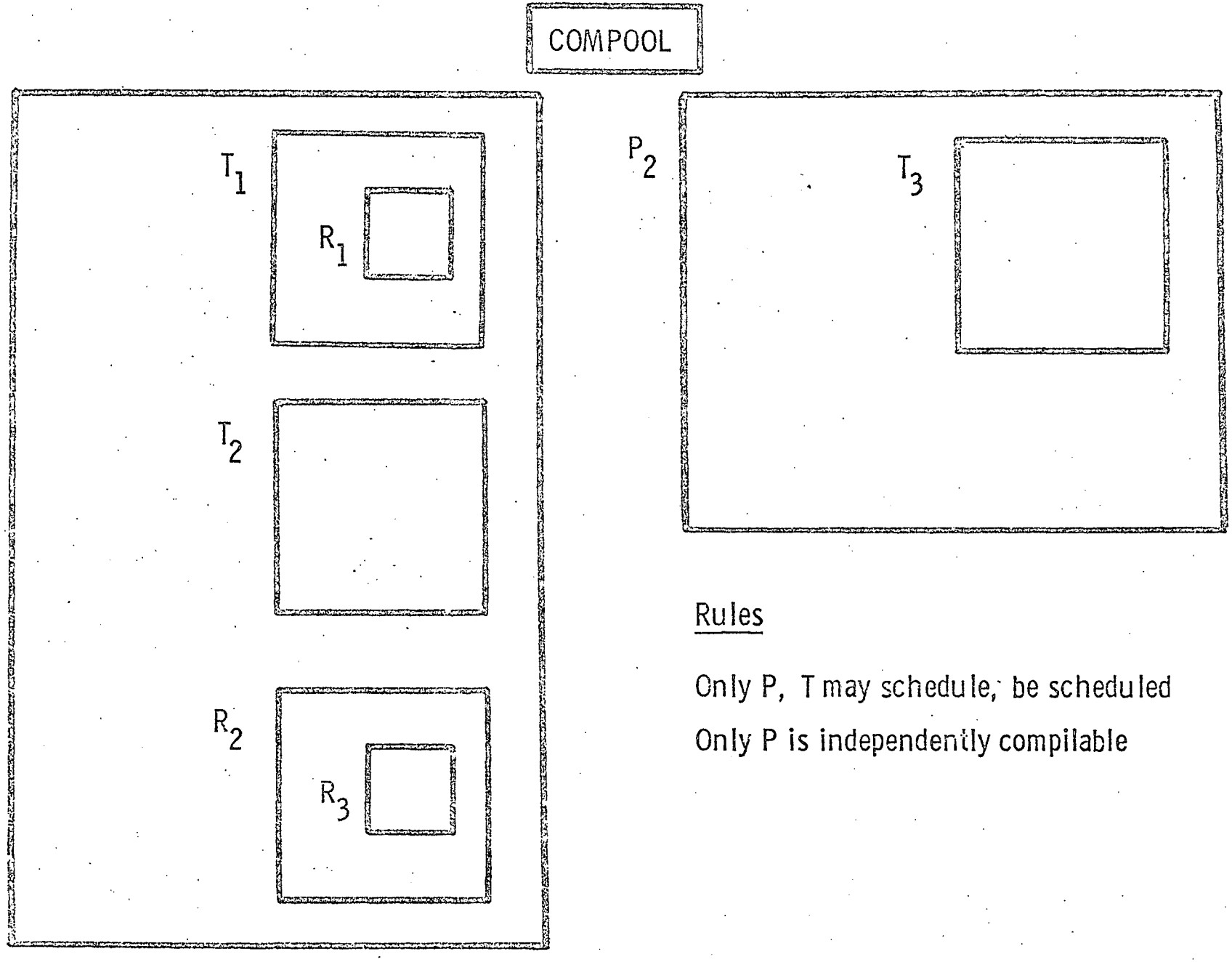
f) T_1 may not call R_3, because the name of R_3 is not known outside of R_2.

Despite these restrictions it is obvious that extended hierarchical scheduling relationships are still possible. Figure 2-4 illustrates an example. Each numbered entity is a Program or Task that is scheduled by, and in turn schedules others. The above HAL scheduling rules are complemented by rules governing the termination of dynamic activities, which are important for the following reasons. Scheduling means that the executive is requested to cause the indicated block to be executed when conditions permit. This may not happen for considerable periods of time. The activity that schedules a Task for future execution must not in the mean time die without trace, leaving the scheduled Task an orphan to be taken care of by some other activity, which is probably not prepared for the eventuality. On the other hand, a Task created somewhere near the bottom of the tree, for example Task #15 in Figure 2-4, must not be empowered to topple the whole structure by performing a summary "terminate". To prevent such phenomena and the possibility of unpredictable and uncontrollable program operation, HAL provides the following functions:

a) A CLOSE at the end of a program, defined as a wait for all the dependently scheduled activities to finish, and only then a return to the executive.

b) A WAIT in the middle of a program, defined as a wait at that juncture until all dependent, previously scheduled activities have finished, and then continue.

Figure 2-3 HAL Program Structure



Rules

Only P, T may schedule; be scheduled

Only P is independently compilable

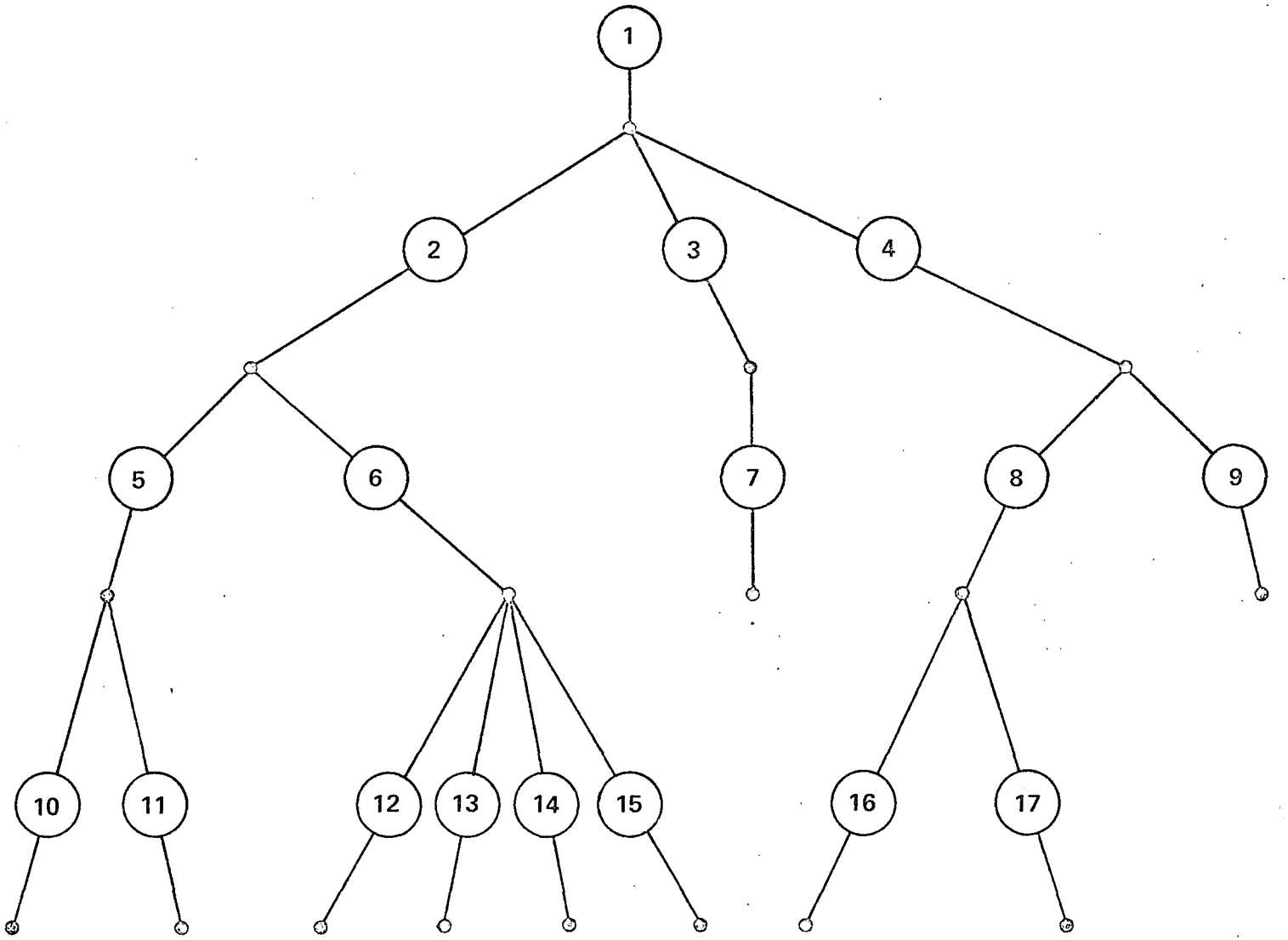


Figure 2-4 Scheduling Hierarchy

c) A TERMINATE is an unconditional termination of self, and of all dependently scheduled activities.

d) A TERMINATE EXCLUSIVE is an unconditional termination of all dependently scheduled activities, but not of self.

A TERMINATE may not refer to tasks scheduled at a higher level than the current task. It is possible for several unrelated tasks to schedule the same task. This must be done by attaching an identifying number to the name in each SCHEDULE statement. A given numbered TERMINATE can then be identified with the specific scheduled activity in the executive which is to be ended.

2.3.3.4 Conclusions. It has been shown that a suitable higher order language provides a very significant step towards an orderly structuring of the program code. In addition, it allows tight control to be exercised over the accessing of variables and subroutines, causing the dynamic behavior of the computational processes to be more apparent from their static structure.

HAL is a higher order language that enables a direct identification of the program modules proposed in Section 2.3.2 to be made. The Data Module as defined finds a correlation with the Procedure block in HAL. A Procedure, like the proposed Data Module, has well defined single entry and exit points, and may not schedule other activities via the executive. It may call other Procedures, although only through a similarly well controlled mechanism. The Control Module is realized by the Program and Task blocks of HAL, which match its requirements exactly.

2.3.4 The Program Design Process

The preceding sections have presented a number of techniques for achieving a more deterministic, manageable, and less error prone Shuttle computer program. This section will briefly indicate how some of these techniques can be integrated into an overall design and implementation process. A development of the subject matter in greater depth and detail will be presented in Sections 2.4 and 2.5.

A computer program of more than trivial complexity requires the combined efforts of many programmers, who may number into the hundreds for the larger Programs. It

is usual to break the total volume of software into a number of somewhat arbitrarily defined modules, and to assign these modules to many programmers, who then independently proceed to generate code. Eventually the many pieces of program are melded together during a scheduled integration period. This process is usually difficult and painful because of faulty interaction between the modules due to misunderstandings of their mutual interfaces and the data they share [7]. Often the only answer to the subtle errors such misunderstandings create is exhaustive testing for as many cases and logical possibilities as time and money permit.

2.3.4.1 Program Structure. A number of researchers have been investigating the possibility of a more definitive evaluation of the quality of software, by exploring techniques for actually proving program correctness [8]. The most encouraging avenue has been the development of program structures which enhance the possibility of demonstrating, in a minimal fashion, correct operation. A major premise of most investigations has been that the commonly used GO TO statement to be found in all high level languages must be considered deleterious [9]. A GO TO causes a sequence of logical operations to be transferred to another area of the program, with no guaranteed return. From the conceptual point of view of the programmer or verifier this causes a break in the train of thought, a thumbing through the program listing, a struggle to remember where the sequence came from and what it was doing. One GO TO leads to another, and soon continuity is lost.

Proposals have been made for a disciplined approach to coding in which GO TO statements are replaced by two or three branching statements which force a return. For example Mills [10] advocates a structure using only the IF THEN ELSE and DO WHILE statements to be found in PL/1, with perhaps the DO CASE from PL/360. These alternate branch mechanisms are acceptable because they return in a very visible manner to the current sequence, and restore the continuity. Any program can be proved to be implementable using only these control structures, although with existing programs the procedure can become rather tedious [11] Dijkstra has coined the term "structured programming" for conscious efforts to avoid GO TO statements and limit control mechanisms to only two or three types of statements [8].

The state of the art in program correctness proofing has not proceeded to the point where techniques can be applied to an operational problem [7]. The principles of

structured programming have, however, been applied to the design of a real-time system by Mills, and are reported to have contributed greatly to a reduced difficulty and lessened time for the testing phase.

The Shuttle program presents new difficulties for the structured approach because of its real-time control requirements. It is, however, proposed that a preliminary structural design can take into account both the logical and sequential characteristics of the task without relinquishing the advantages of structured programming, as follows.

2.3.4.2 Time-line Design. The initial structuring of the program must start with some estimate of the size and execution time required to achieve the computational and control functions specified by the Software Requirements. This first step can only be achieved by educated guess work on the part of experienced programmers, assisted perhaps by trial programming. The initial objective will be to plan out the scheduling work load of the executive during all the major programs. The importance of an initial design of the executive functions of the program functional hierarchy defined in Section 2.3.1 is obvious. Of significance here is the executive philosophy. The executive design presented in Chapter 2 proposed a synchronous foreground capability for a limited number of high priority functions, and an asynchronous background for other functions. A time-line design must ensure that the distribution of the available processor time adequately provides for the timely execution of the foreground tasks. The background tasks must of necessity be interruptable. This can be provided for by allowing interruptions by the executive only at well defined points (for example between HOL statements), to avoid hazards to the variables specifying the state of the calculations and the status of the interrupted task. The cooperation of language and executive can ensure that the integrity inherent in a structured program is not endangered.

Once the program has been defined in terms of schedulable functions, these may then be identified as Data or Control Modules.

2.3.4.3 Structural Design. The technique proposed here has been defined by Mills as "top-down" program implementation. The objectives of the technique are

a) to provide "at a glance" understanding of the function of a program;

b) to achieve an operating status with the actual program at a very early stage of development.

Each basic major program is initially written in series of statements that do not exceed one page of printout. The statements are limited to data declarations, sequential, logical and arithmetic operations, calls to Data Modules (HAL procedures), statements scheduling Control Modules (HAL programs and tasks), and simple control statements such as IF THEN ELSE, DO WHILE and DO CASE. Data and Control Modules that do not exist initially are represented by names and code simulating their time and storage requirements. (This method is further explored in Section 2.4 and 2.5.) Each of the functional program levels defined in Section 2.3.1 is begun this way, by writing its top-level functions in the form of a single page program.

As soon as a compilable entity (e.g., a HAL program block) is produced, it may be compiled and executed, without waiting for the last piece of code to be delivered. In fact, it may be run before every procedure and task has even been identified. The main advantages of this programming technique are:

a) it is easier to create programs in a structural manner, because they are easier to visualize;

b) a "top-down" assembly of structured programs can be continuously exercised throughout its development. This provides confidence and visibility of its status to management. It establishes the understanding of interfaces and program operation in a continuous manner from the start of implementation.

c) A structured one-page program can be more easily verified. It may be proved to correspond to its specification, if not by rigorous automatic techniques, then by examination. It will be shown in Section 2.4 that examination is perhaps the most powerful tool for verification available today.

2.4 Implementation: Code and Test

The "layered" design process described in Section 2.3 results in the definition of distinct machine levels, the characteristics of the level boundaries, and an overall design of the structure within each level. The design is described in a "top-down" fashion, and will be so implemented. This design hierarchy was described in Section 2.3.1, and depicted in Figure 2-2.

2.4.1 Implementation of System Programs

The design and implementation of the Shuttle computer executive and systems programs is performed during the Phase 2 described in Section 2.2.3.2. This includes scheduling, memory allocation, basic subroutines, self-test, general I/O routines, display interface, initialization and load and other services. At the end of this phase all systems programs will have been coded and tested to meet the software requirements. This is accomplished in the steps defined by the levels. Thus, the scheduling, dispatching and interrupt handling functions (Level 1) are completely coded and tested prior to the implementation of the memory allocation scheme (Level 2). Level 2 then presumes that Level 1 represents a virtual machine; i.e., is coded using only the facilities provided by Level 1 and, more importantly, coding within Level 2 never accesses or directs the machine itself. In this manner the user interface (to the applications programs) is established building only upon the layer that preceeded it. Thus, any layer, L, can be understood and debugged independently of the level higher than L and independently of the levels lower than L-1. Note that this approach clearly distinguishes between the user and the system and is specifically intended to enhance reliability by allowing the user (applications programmer) only those facilities designed for him, and not permitting alteration, by him, of the system itself.

The step-by-step development of code within each layer, and the accompanying tests, are conducted "top-down" as briefly described in Section 2.3.4.3. This process and its benefits are discussed more fully in the next section.

2.4.2 Implementation of Applications Programs

The design and implementation of the Shuttle applications program (Level 4) follows the design of the executive (Levels 1 through 3). These programs will include guidance, navigation, flight control, onboard checkout, display

processing, systems monitoring, subsystem servicing, reconfiguration logic, telemetry interfaces, and other special purpose routines. At the end of this phase, all applications programs will have been coded and tested to meet the Software Requirements.

2.4.2.1 Top-down Implementation. As a result of the Software Requirements and the subsequent Program Design process, the applications programs have been structured into Data Modules and Control Modules. The Data Modules encompass subroutines and functions which may be called in-line entities. They are distinguished in that they represent only logical and/or computational processes and initiate no real-time control statements (e.g. SCHEDULE, WAIT, etc.). Accesses of common data are made through the rigid control mechanism of the shared data pool. The Data Module is significant to the construction of reliable software in that it represents a static portion of the program code and can be verified locally without direct dependence on the dynamic run-time behavior of the program. Since the bulk of computational and logical operations are performed by Data Modules, a sizeable portion of the total applications software can be verified by this relatively straightforward process.

The rest of the software is comprised of Control Modules. These are schedulable by the executive, and may perform calculations and/or logic, call subroutines and functions, and in turn, schedule other Control Modules. An objective in concentrating real-time control statements within Control Modules is to bound the program control functions so that the number of executive interfaces is minimized. When reviewing a written program it is expected that reliability will be improved by the convention (perhaps compiler enforced) of keeping real-time control within Control Modules and totally out of Data Modules.

Implementation of code based on these modules can now begin, top-down, as directed by the Program Design Specification. Figure 2-5 illustrates how the first coded version of the Control Module for a Shuttle major burn maneuver might appear. (Actual coding shown is based on the HAL language.) This initial design is considered "top-down" because the entity MAJOR_BURN describes the overall workings of the program, indicating what subroutine functions are required, their order, and their real-time scheduling.

```
MAJOR_BURN; PROGRAM;

        CALL IMU_INITIAL;
        IF ERROR_CASE THEN TERMINATE;

        CALL VELOCITY_TO_GO ASSIGN (VG);
        IF ABVAL (VG) < 10 THEN TERMINATE;
        CALL THRUST_ORIENTATION;

        WAIT FOR KEYBOARD;

        CALL MANEUVER;

        WAIT FOR END_MANEUVER;

        SCHEDULE ENGINE_ON
                AT T_IGNITION;

        SCHEDULE STEERING
                AT T_IGNITION + 2 SEC;

        CLOSE MAJOR_BURN;
```

Figure 2-5 MAJOR_BURN Program

An interesting aspect of this approach is that this first Control Module, coded as shown, can be actually run on a bit-by-bit simulator. In order to do this with some realism, all the subroutines indicated would have to be modeled. Each model would indicate the amount of local storage it requires, and would implement a loop corresponding to a "time-to-run-through" budget. Several possible models for the routines in MAJOR_BURN are shown in Figure 2-6.

It now becomes apparent that the first step in implementing the Shuttle applications programs is to code the "upper most" module, and to model the subroutines it calls. The program can then be run in a simulation mode that exercises all dynamic executive functions. That is, the overall timing and use of storage are being exercised. The programmer also verifies that the module itself is correctly coded by concentrating his attention on the logic and computations already appearing as actual code within the module. He need not be concerned with the correctness of the called subroutine in order to verify the module, only that they are correctly called. Once this Control Module is verified, more of the actual code is written to replace the temporary models. Consider STEERING as an example. This task might be coded as in Figure 2-7. The routines ENGINE_OFF and VECOMP would be modeled. The programmer would verify the logic and computations actually coded in STEERING, and would then insert this task into the established program MAJOR_BURN.

The conception here is that the manner in which the applications programs are constructed and, for the most part, their chronological or dependency order, will have been determined during the Program Design phase. The result of the design will be a thorough, but not complete structuring of the software, and the identification of many, but not all, of the Control and Data Modules. Coding in the top-down process can begin immediately following release of the Program Design Specification. However, the design process does not stop here, but continues to precede the coding by identifying and structuring the remaining software, until all code and models are indicated. The state of the coded program as it might appear at any point in time is illustrated schematically in Figure 2-8. Blocks are indicated by A or M according to whether they are actual code or models. The captioned numbers indicate chronological order. Thus (1) might represent the direct coding of a part of the

```
1)      IMU_INITIAL: PROCEDURE;
            DECLARE LOCAL ARRAY(30);
         LOOP: DO FOR X=1 WHILE X<100;
         END LOOP;
        CLOSE IMU_INITIAL;

2)      VELOCITY_TO_GO: PROCEDURE ASSIGN(VG)
            DECLARE LOCAL ARRAY(100);
         LOOP: DO FOR X=1 WHILE X<1000;
                X=X+1;
         END LOOP;
           VE = VECTOR (10,10,10);
        CLOSE VELOCITY_TO_GO;
                :
3)      STEERING: TASK*;
            DECLARE LOCAL ARRAY(50);
         LOOP: DO FOR X=1 WHILE X<500;
                X=X+1;
         END LOOP;
        CLOSE STEERING;
```

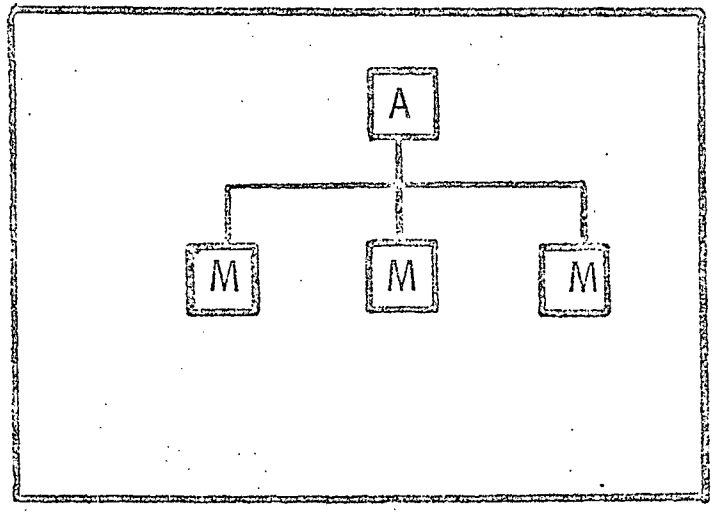
*TASK(in HAL) is a schedulable entity.

Figure 2-6 Model Routines for MAJOR_BURN

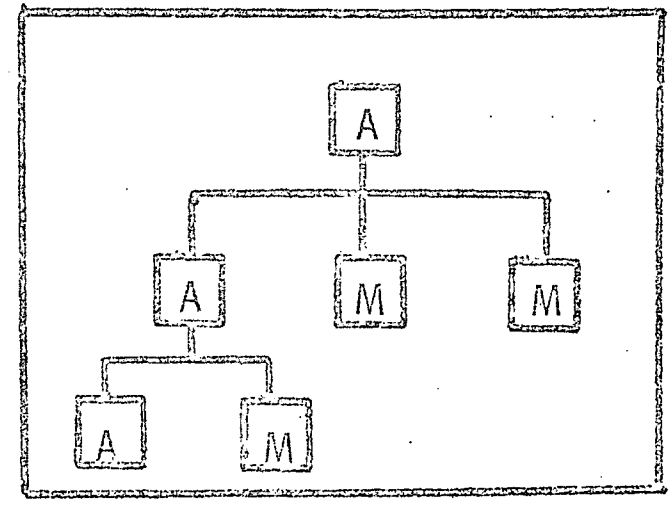
```
STEERING: TASK;
   IF TGO < 4 THEN
        SCHEDULE ENGINE_OFF AT (TIME+TGO);
   CALL VECOMP ASSIGN(VG,VGDOT);
   STEER_RATE = K UNIT(VG) * UNIT(VGDOT);
   AUTO_PILOT_RATE = *COORD_TRANS STEER_RATE;
CLOSE STEERING;
```

Figure 2-7 Steering Task

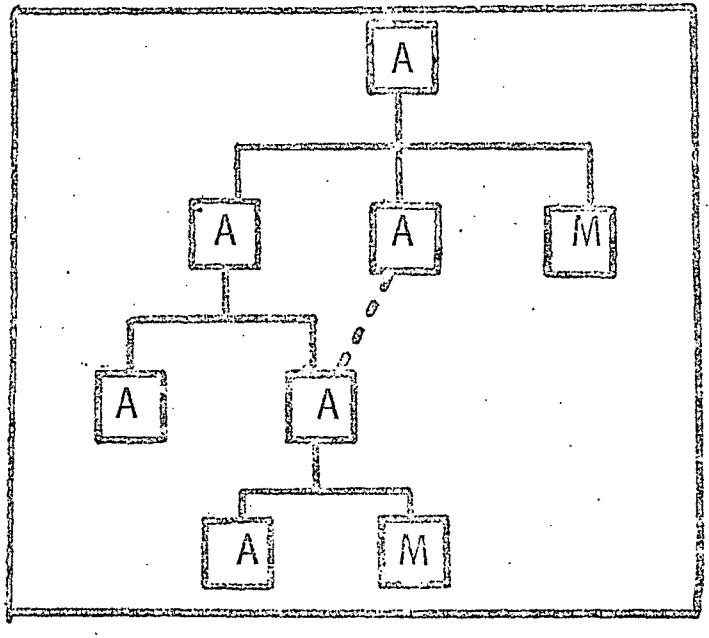
Figure 2-8 Development of Program



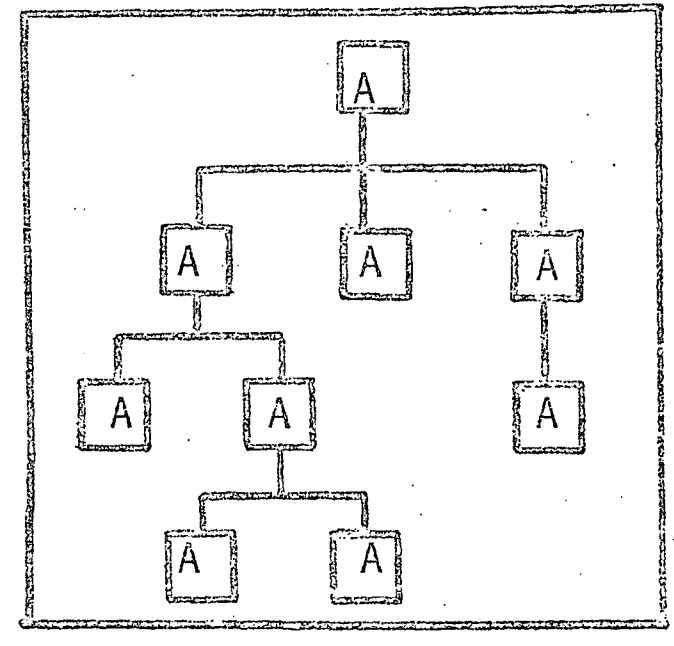
(1)



(2)



(3)



(4)

A = actual code

M = modeled code

Program Design output (e.g. MAJOR_BURN). The A would represent the actual code shown in Figure 2-5, while the M's has been filled with actual code, and another coded module and a model have been added. By (3) this process has continued, and more modules have been filled with verified actual code, and more detailed modules have been identified. By (4) the process is complete; all models have been replaced by actual code, and for this particular program, Phase 2 is finished.

It is important, and must be emphasized, that the program described above was being executed throughout its development period. In other words, a multitude of modules were not being coded and tested independently, awaiting the judgement day of integration. Instead, the program was being integrated as it was developed. Interface inconsistencies, timing difficulties and storage deficiencies were addressed and corrected at their source and were not experienced during the more traditional, protracted integration period. Herein lies the major contribution of the top-down approach. Careful and progressive construction of the program will produce more reliable and consistent software than the module stage - interface stage approach, by catching interface errors as they appear. Chronologically, coding time (i.e. Phase 2 in this context) might take longer than traditional methods, but at the end the overall program will work because it always has worked.

2.4.2.2 The Program Building Environment. Although the coding process above was illustrated for only one program, the Shuttle software development implies the concurrent implementation of many programs. It is anticipated that a number of relatively independent top-down processes will be identified during the Program Design phase. These might include the functional mission phases, checkout and service routines, as well as universally required sub-routines (e.g. state vector integration). Some examples might be:

Preflight	Approach
Boost	Landing
Orbit Insertion	Autopilots and control
Orbital Flight	Onboard checkout and monitoring
Rendezvous	Reconfiguration logic
Docking	Telemetry
Entry	

Note that many of the mission phases involve similar functions: guidance, targeting, navigation control, state vector manipulation, etc.*. The Program Design Specification must specify the manner in which the program is to be implemented. That is, by mission phase, by modular functional "blocks", by subroutines, by a combination of all these. This design activity cannot be minimized, it is essential to the overall success of the program. For the purpose of illustration, suppose that N programming activities can proceed in parallel. In the context of the HAL language, this means N independently compilable Programs. Communication among these Programs is strictly controlled through a central Compool. The structure was illustrated in Figure 2-3. The Compool and common Data Modules (Comsubs) appearing in the pool are centrally managed, and individual "Program-requests" must be evaluated at the highest management level before incorporation. The process of implementing any of the P's shown in Figure 2-9 was described before. The significant problem in a parallel development is to provide an environment for local development and test while maintaining control and consistency of the overall effort. The maintenance of only a single official source code for the entire effort will greatly assist in achieving this objective.

The following procedure is suggested to achieve parallel development. Each P-program is officially compiled with COMPOOL + COMSUBS. The collection of compilations then constitutes the official source code. A programmer working on code to replace a current model (M),(see Figure 2-8), compiles an off-line representation of the specific Program in which his code resides. (This would apply equally as well to a Comsub.) He then locally tests his new code by running the off-line compilation. If he requires the COMPOOL, COMSUBS or any other of the P's in order to run, he can only obtain the officially compiled code for these entities. This ground rule would be true for all applications programmers coding within any of the P's. When a programmer has locally tested his code and has, in addition, shown proper adherence to the functional criteria and data specified in the Software Requirements Specification, his new code can be accepted into the official version of the Program. This step must be controlled by a source control group which admits the code and recompiles the Program.

---

*See Space Shuttle Orbiter Guidance, Navigation, and Control MSC-03690 12/15/71.

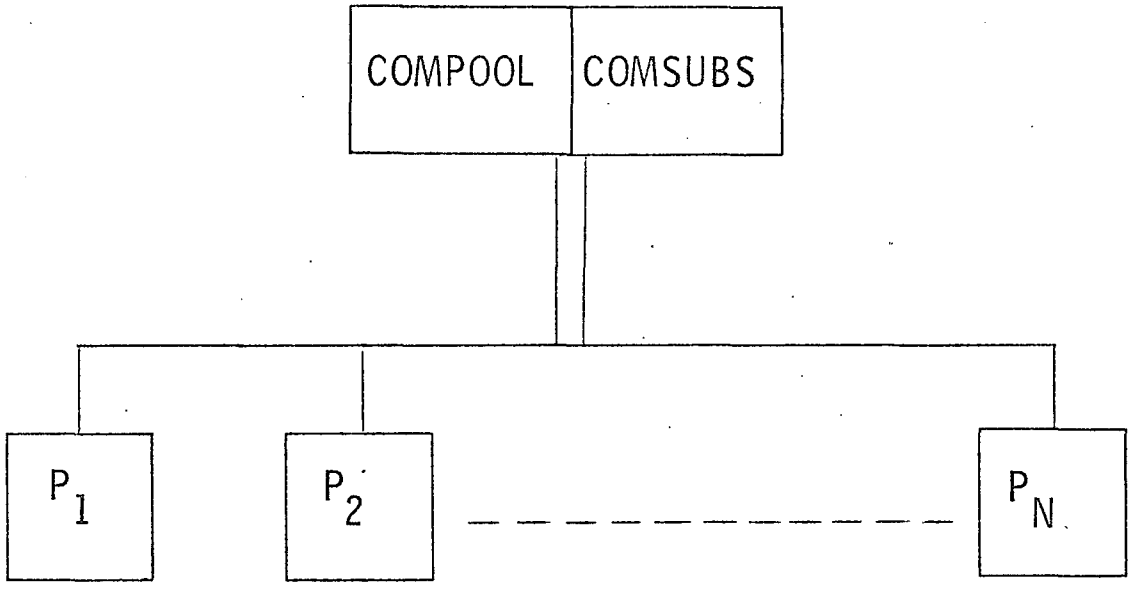


Figure 2-9 Program Organization

This procedure achieves two objectives:

1) a programmer can experiment with his own code, modify it, test it, etc. without holding others up with his own mistakes.

2) a programmer will, however, always experience the up-to-date; official versions of programs that interface with his own. The result is that then entire implementation effort progresses in parallel across a broad front. "Private" or off-shoot versions of the program which are created for local testing and soon get out of step, are thereby discouraged.

2.4.3 Program Confidence: Test Philosophy

The objective of software verification is the assurance that the program is correct, that no bugs remain, and that it will work in the manner intended. While the objective may be simply stated, its attainment implies basic difficulties. These difficulties revolve about the fact that any non-trivial program especially one expects to operate in real-time, will be exceedingly complex. The Shuttle major burn maneuver program used for illustration in Section 2.4.2.1 eventually draws into play many detailed Data Modules involving a myriad of logical branches and computational steps. The fundamental problem to be faced in verifying this program may be illustrated by the following analogy.

2.4.3.1 2^{54} Possibilities. Consider the design, construction, and testing of a binary counter* which overflows at 2^{54} pulses. The requirement is simply that the logic of the counter must count accurately up to $2^{54}-1$. The designer determines which electronic components are necessary, specifies the design and methods of construction, and the counter is built. Now the question comes: will the counter perform correctly for all cases? One approach might be for the originator of the requirements, who understands the need for a counter in the first place, and knows how it will be used, to specify a series of performance test cases for the equipment. For example, he will not accept the counter unless it can count 1; 10; 121; 3,654; 2^8-1; 1,000,000. However, these are only random tests, and their success does not imply correctness. What is the alternative? 2^{54} pulses would take many thousands of years to

---

* Example adapted from Ref. 12.

accumulate, even at microsecond speeds. The conclusion must be drawn that it is impossible to verify that the design of the counter is correct as long as one regards the mechanism as a black box. Verification has to be based upon an examination of the internal structure of the mechanism to be tested [12]. For example, the designer, knowing in detail the manner in which the counter was constructed, might conduct 54 tests. He could verify the operation of each of the 54 stages of the counter, and their propagation mechanism. Having verified the internal workings of the counter, he would pronounce it ready for service.

But what of the test cases specified with the original requirement? These are actually requested demonstrations that the counter can perform the necessary computations. The "requirements specifier" is entitled to these demonstrations, but it is the counter designer who is ultimately responsible for the proof that his counter operates correctly, for all cases. Note that it is impossible for the "requirements specifier" to devise sufficient test cases to prove the correctness of the counter. In fact, for a complicated internal mechanism the relevant cases could not even be postulated externally.

2.4.3.2 Implications for Program Testing. The reasoning above must influence the methods of building and testing Shuttle software. The Software Requirements can specify the algorithms, logic and timing of the programs, but the sets of test cases and test criteria evaluating program performance can only be regarded as required demonstrations. It is impossible at the Requirements level to specify the relevant test cases required to validate all the branches and computations within the implied structure. Test cases to prove correct program design must be based upon the programmers' implementation of the Requirements. In the discussions within Sec. 2.4.2.1 and 2.4.2.2 the implication is that when real code replaces modeled code in Control and Data Modules, the code is first locally tested for correct implementation structure. Of course, demonstrations of its operation can be conducted when called for by the Software Requirements. Thus, the correctness of the code is the responsibility of the programmer, not of the "requirements-specifier," who is unable to indicate the necessary set of relevant test cases.

How is this correctness established? Presumably the programmer understands how he has implemented a requirement, and knows what it takes to verify that he has done so correctly. His approach will be two-fold: (1) inspection, by "eye-ball", (2) relevant test case results, by benchtesting. Eye-balling the code should be emphasized as a bona-fide verification

activity. It is a mental activity in which one sees what one has done. Of course, for the greatest effect, one employs an independent eye-baller who is inserted into the programming process at a number of points. For example, independence can be achieved by an external contractor evaluating the developing code, or by the source-code controllers who scrutinize the new code prior to allowing it into the official source. Both of these methods were employed during the production of the Apollo flight computer programs, with resounding success. Experience indicated that many more bugs were caught by visual examination than as a result of simulated or actual running of the computer programs in accordance with an established test plan.

Proving correctness of code by eye-balling is not unlike checking the derivation of a mathematical formula. If the desired result turns out to be the integral of a complex expression, e.g.

$$Y = \int_{a_0}^{a_1} f(a,b,c)\, d\, a$$

one does not "prove" this equation by programming it and running it for selected test cases. Instead it is proved by induction. Demonstrations for selected cases then give credence to the derivation, but not proof.

When a Control or Data Module contains many logical branches and is quite complex, the eye-ball process is deficient and the relevant test cases must be designed by the programmer and exercised.

After the individual modules have been verified locally, by eye-ball and test, and their validity demonstrated to the source controllers, the new code is officially compiled into the program. At this point all the interfacing calls and schedule statements within the official source which had previously regarded the new code section as a temporary model, must be now exercised to demonstrate that the new additions can, in fact, be connected. Once assured, the overall program should run as before only now, one more step toward completion has been taken.

2.4.3.3 Built-in Program Reliability. Aside from the structured program organization, top-down implementation and test

procedures referred to in the previous paragraphs, program reliability can be achieved to a very large measure by built-in features. These features should include a higher order, block-oriented language and compiler, automatic prevention of conflicts over shared data, restriction of access to data and routines by designated access rights, and extensive compile-time and run-time checking. These and other automatic aids will be identified and discussed in Chapter 5 of this report. It might be useful at this point to illustrate how some of these benefits might prevent or catch programming errors. Consider the following example:

```
DECLARE AUTO_PILOT.RATE VECTOR (3);
DECLARE STEER_RATE VECTOR (3);
DECLARE COORD_TRANS MATRIX (3,3);
                .
                .
                .
                    *
AUTO_PILOT_RATE = COORD_TRANS STEER_RATE;
                .
                .
                .
```

The significant point to observe here is that through the use of an expressive higher order language the intent of the programmer, in some sense, is recognized by the compiler. Thus,

1) The listing appearance itself gives ample evidence of the intended operations.

2) The compiler will check the syntax of the code; in this case, it will test the validity of a matrix-vector product and whether or not the result of that product may be assigned to a vector.

3) Since the dimensions of the vectors and matrix are known at compile-time, the compiler will check that all quantities are consistent. That is, if STEER_RATE were of dimension VECTOR (4), compilation would be prevented and an appropriate error message generated.

4) The compiler will check to see if all names of quantities are recognizable at this particular point in the program. In block-oriented languages, variables declared locally within subroutines cannot be referenced from outside of these routines.

5) At run-time an important reliability factor is the assurance that only intended data accesses are made; i.e. unanticipated "clobbering" of data cells is prevented. Where vector, matrix and array lengths (or sizes)

are not specified at compile-time, a set of run-time checks can be inserted to evaluate indexing code to prevent the stray "write" or "read" and to maintain the consistency of dimensions. When data is intentionally shared, real-time processing demands a controlled environment for this data, as is suggested in Chapter 5.

Consider the following example:

```
b)   CALL TIME_RADIUS (RT2, VT2, (RT3 MAG-30480),MU)

                    ASSIGN (TIME_32)R̄T3, V̄T3);

                       ⋮

     TIME_RADIUS: PROCEDURE (Ā,B̄,C,D)ASSIGN(E,F̄,Ḡ);

         DECLARE VECTOR (3), A,B,F,G;
         DECLARE SCALAR, C,D,E;

                       ⋮

     CLOSE TIME_RADIUS;
```

In this example, the intention is to call the TIME_RADIUS subroutine which will accept the current position and velocity and the desired radius; then to compute and assign the time to reach the radius and to establish the resultant position and velocity (a conic transfer trajectory about the earth is presumed). The example is meant to illustrate that the data types "expected" by TIME_RADIUS are prescribed by its compilation and are therefore known at compile-time. The compiler will check to see that all calls to TIME_RADIUS match these anticipated data. It will prevent compilation, and issue appropriate error messages, if a programmer attempts to call the routine incorrectly. Of course, provision can be made for lengths and dimensions which can be specified only at run-time, in which case run-time checking is necessary.

The importance of automatic devices as described above to the generation of reliable software cannot be minimized. While local testing and overall verification must be conducted, bug-free programs are more directly approached through methods of construction and the use of aids designed to prevent programming errors. "Program testing can [only] be used to show the presence of bugs, but never to show their absence" [12].

2.5 Verification

The term verification, in the context of this report, is meant to convey the overall exercising of the Shuttle software and its performance evaluation. Before initiating the verification phases (Phases 3 & 4) two prior phases will have been completed:

Phase 1: generation of the Software Requirements. These requirements reflect a detailed design satisfying the Functional Requirements. The Software Requirements encompass analytical formulations, timing, interface design and expected performance results. The analytical concepts have been demonstrated and results obtained through the use of a modular environment simulator. This simulator represents, in modular form, the subsystem functions and equipments, vehicle characteristics, universe, atmosphere, etc. to the extent necessary in order to prescribe, with validity, the Software Requirements. (See Sec. 2.6 for further discussion of environment simulators.)

Phase 2: design, implementation and test of systems and application programs. In response to the Software Requirements, first the necessary systems programs are designed, structured and implemented. These programs are locally tested (see Sec. 2.4.1) using features of the environment simulator as needed. Completion of the executive design provides the proper "user interface" for applications programmers. The necessary applications programs are then designed, structured and implemented. During Phase 2 these programs are locally tested (See Sec. 2.4.2.1, 2.4.3.2) using features (modules) of the environment simulator as needed. Performance demonstrations may also be conducted as indicated by the Software Requirements. Completion of Phase 2 means that all the code for the Shuttle software has been generated and tested; the programmers know of no program bugs.

Following Phase 2, it is well to consider in what ways the software still might not work:

1) it might not work within the real computer operating in a real environment;

2) it might not achieve the overall performance (i.e., defined success per mission phase) expected by the Software Requirements.

These two possibilities are now explored during Phases 3 & 4 - program verification.

2.5.1 "Feedback" and "Feed-forward"

Program verification is the final phase in the production and delivery of software. Two activities are postulated, to be conducted on two different facilities, with two different objectives:

1) the software must be shown to meet the performance criteria dictated by the Software Requirements (Phase 3);

2) the software must work within the real computer, operating in an environment as close to the flight environment as practicable (Phase 4).

The roles of Phases 3 and 4 are complementary in that the latter is, in a sense, a "feed-forward" activity toward the flight environment while the former is a "feedback" process, back to the Functional and Software Requirements. This concept is illustrated in Figure 2-10. Both paths are necessary, and fulfill different objectives. Phase 3 establishes the performance of the software with respect to the environment presumed by the Software Requirements. For this purpose, the environment models used in determining the Software Requirements Phase 1, then for the implementation/code/test Phase 2, and now for Phase 3 should be the same, or at least controlled and based upon the same model environment specifications. Phase 4 moves the software out of the realm of simulation and into the real world of equipment, and human and electrical interfaces. Design flaws can no longer be masked by model presumptions, approximations or inaccuracies. Satisfactory operation here is the prerequisite for flight.

As a logical progression of testing, the results of the two phases may be interpreted as follows:

1) Satisfactory performance within Phase 3 means that the Software Requirements are met. The original design concepts appear to be valid and confidence is established in the software implementation. Unsatisfactory performance can be due to three factors:

   a) the original design concept is faulty, even though it was correctly interpreted and correctly implemented;

   b) the Software Requirements have been misinterpreted;

   c) the code implementation contains errors.

   These factors imply that the facilities needed for Phase 3 must permit rapid diagnosis of the fault to

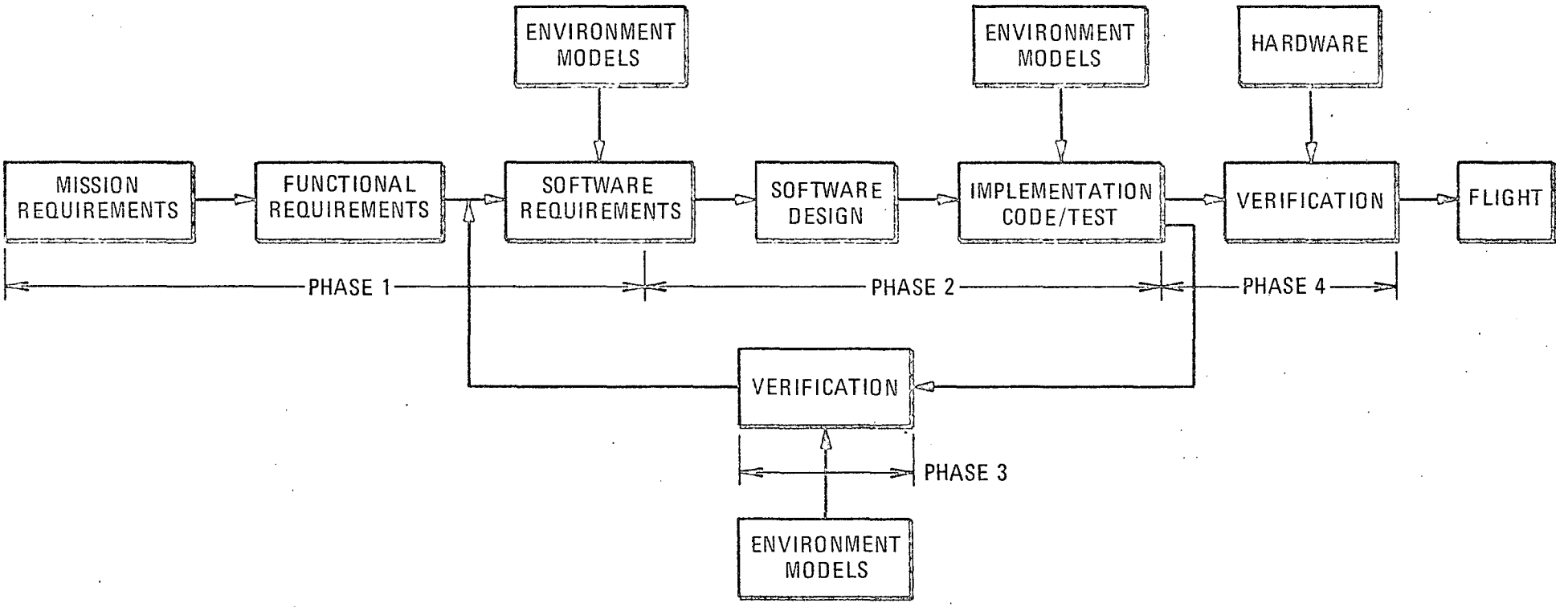


Figure 2-10 Software Development Phases

to determine which of these contributing factors is the culprit.

2) Satisfactory performance within Phase 4 means that the Mission Requirements can be met independently, to a large extent, of the models presumed in establishing the Software Requirements. That is, either the models were, in fact, valid or their inaccuracies were of no consequence. Successful completion of Phase 4 increases confidence in the design concept and the software implementation. In addition to demonstrating the software, Phase 4 serves to establish the hardware-software integrity of the entire avionics system and its ability to perform the real mission. Unsatisfactory performance can be due to four factors:

   a) the hardware-software interface design is faulty;

   b) the code implementation is incorrect;

   c) the models presumed in the Software Requirements were inadequate;

   d) the integration of the hardware subsystems is causing problems.

The important point to recognize is that by conducting both Phase 3 and Phase 4 with different objectives, classes of errors, created by failures of specification, interpretation, code, modeling, and hardware can be separated and identified.

2.5.2 Phase 3: Verification through Software

During this phase a series of performance tests will be conducted on an all-digital simulation of the avionics systems. The simulation will include a bit-by-bit simulation of the airborne computer as well as models of the environment and subsystem equipment. The test plan should be designed to produce the results necessary to satisfy the Software Requirements. This will encompass all phases of the Shuttle mission and will include nominal, off-nominal and contingency cases. At the end of Phase 3, the performance of the software, based on the environmental models used to derive the Software Requirements, will have been established and documented.

Phase 3 can take advantage of the flexibility, reliability, and repeatability of the general-purpose digital computer facility. Thus, a large number of multiple users with multiple objectives can be accommodated through rapid reconfiguration

of models, initialization of test data, and standard batch or time-shared utilization. With a single facility providing all computation machine "down time" can be kept to a minimum, and convenient user procedures, and adequate computing capacity can assure adequate turn-around times.

The evaluation of digital runs presents a problem because of the mass of data that potentially becomes available. Therefore, it is important to understand the objectives of these tests and to display only that information necessary for evaluation. More elaborate recordings of data may accompany any run, for troubleshooting or reestablishment ("roll-back") of test conditions, but the generation of printed (or otherwise displayed) data must be disciplined. A library of data-compressing edit programs is an essential tool.

The all-digital simulator enables two views of the operating software:

1) macroscopic, i.e. demonstration of overall adherence to the Software Requirements;

2) microscopic, i.e. isolation of detected anomalies to the software module and even to the detail code itself. When required, and at the test-engineer's discretion, the ability to stop, start, trace, edit and dump computer memory, based on a wide range of conditions is easily invoked. And, in addition, runs may be repeated, or "rolled-back," in an exactly repeatable manner in order to re-establish conditions which might have caused an anomaly.

Thus, even though the debugging of coding errors is an explicit activity of Phase 2, Phase 3 also permits fine grained analyses of the working software.

2.5.3 Phase 4: Verification through Hardware

The Avionics Integration Facility (see Section 2.6) provides the test bed for this phase of activity. The primary purposes here are to iron out hardware interface problems, and to demonstrate the performance of the entire avionics system (hardware and software) in the face of the real (not simulated) computer and subsystem equipment hardware, and the real (not simulated) electrical interfaces. A set of performance test cases must be defined which exercises all of the subsystems, and is representative of the mission phases. The tests should encompass boost, rendezvous, entry, landing, etc.

With respect to the software, special attention during Phase 4 should be paid to those program areas where presumptions have been made concerning the operation, procedures, and timing of external (to the computer) equipment. It is in these areas that one might expect to find a higher incidence of anomalous behavior. Coding errors might, in fact, still exist but it is more likely that the misunderstandings of subsystem operation, timing and moding have resulted in inappropriate designs (perhaps even at the Software Requirements level) which will now cause faulty system operation. "Playing" the software in a close-to-real* environment is an indispensible step before committing the system to flight. Prior to flight, it is only here that the analyst's theories and the programmer's implementation are truly demonstrated. The software is transformed from an abstract paper listing into a component of the avionics computer.

2.5.3.1 Detailed Evaluation of the Software During Phase 4. Even though Phase 4 serves to establish system-wide compatibility, and utilizes standard-procedure techniques for the diagnosis and troubleshooting of hardware and hardware interface anomalies, the detection of software errors and the evaluation of performance over a large set of mission situations (including non-nominal) is more properly the domain of Phase 3. The principal reasons for this are:

1) inability to repeat, exactly, results from run to run;

2) multiple users are not easily supported;

3) "up time" for a hybrid computational facility is characteristically low.

The avionics integration facility, comprised of actual subsystems, interface equipment, digital and analog computing elements and man-in-the-loop displays and controls will suffer from random uncertainties. These uncertainties will be due to electrical noise, inaccurate initialization, temperature and power supply sensitivities and other vagaries. System malfunctions and degradations in performance may be due to errors anywhere in the system. In the case of software errors, or improperly operated software, run repeatability is a necessity. The speed of the real computer (2 microsecond add approximately) precludes real time troubleshooting which requires the ability to stop the machine at any instruction, or at least trace

---

* Some models will still be required, e.g. the atmosphere, experienced specific forces, vehicle characteristics, etc.

machine operations instruction-by-instruction, at the discretion of the test engineer. The difficulties are compounded, in the avionics laboratory, because there is the additional requirement for stopping or tracing the avionics environment and equipment as well as the computer. It becomes evident that debugging of this sort and the features it implies, are best conducted on an all-digital simulator (see Section 2.5.2).

The question of multiple users becomes important when the avionics integration facility is proposed for primary performance evaluation. The combination of equipment set-up time (i.e. checkout and initialization) and the low percentage of up-time for hybrid facilities (typically 25%), is a discouraging fact. Once set-up and running, the best use a hybrid facility can be put to is evaluation of a particular mission phase and demonstration of the system integrity for that phase (or variations). Rapid reconfiguration of equipment, environmental models, initial conditions, etc. are not so easily accomplished, even when aided by a general-purpose digital computer within the loop.

In view of the foregoing discussion, Phase 4 of software verification is recommended as a hardware-software compatibility check which will demonstrate whether the presumptions built into the software concerning I/O and subsystem operations are valid. Although simulations of mission phases will be used for this purpose, and their performance must match that expected by the Mission Requirements, Phase 4 is not designated as the medium for extensive evaluation of software performance.

2.5.4 Independent Verification

An additional measure of software reliability may be gained by subjecting the software process to critical review and independent evaluation. Independent activity, i.e., by another contractor or "outside" group, can include evaluation of the Software Requirements and Program Design, eye-balling and testing of the code, and verification through separate all-digital and hybrid facilities. New points of view, additional tests, and independently developed models will all contribute to increased confidence in the software product.

In spite of these benefits, legitimate questions remain. Is a full independent "verification" program justified? Will the considerable cost reduce the number of residual software errors? These questions are most difficult to answer quantitatively. However, it is suggested here that independent effort may be applied cost effectively at two specific points in the software process:

1) during development of the Software Requirements;

2) during implementation of the code.

The Software Requirements constitute the algorithms and methods by which the Shuttle software will achieve the functional objectives. As such the Requirements are the primary software specification and, if faulty, guarantee an imperfect end product. Independent verification of the concepts upon which the Software Requirements are based will help to eliminate errors at the ultimate source, i.e., before the coding process even begins. The independent contractor must fully understand the Functional Requirements and then proceed to review all the fundamental assumptions and derivations leading to the design of algorithms and/or procedures. For example, for each mission phase (boost, rendezvous, etc.) the pertinent mathematical formulae will be rederived and then exercised to substantiate performance based on presumed subsystem and vehicle control error models. In this manner, all the data and engineering design postulated by the Software Requirements will come under scrutiny, establishing their validity where necessary appropriate redesign can be indicated.

The implementation phase is characterized by the coding and testing of the program as described in Section 2.3. The testing consists of internal verification based on program structure. It is at this point that an independent effort would be of most use. Starting with a full understanding of the Software Requirements, the independent contractor would review the Software Design, eyeball the coded and conduct test as required. The contention here is that the process of "overlooking" the code at the detailed level (i.e., examination of its internal structure) is a powerful device for catching bugs. In fact, Apollo experience indicates that many more software errors were detected by an independent "reading" of the code than were discovered through formal planned test activities.

The independent verification activity could be extended to encompass the performance evaluation detailed in Phase 3 above. Its validity would dictate independent model development and independently derived test plans. While this activity might increase confidence in the software, its effectiveness is to be questioned. Satisfactory performance would indicate that the programs work only with respect to the new models; unsatisfactory performance might be caused by software or environment modelling errors. In view of the fact that Phase 3 is already based on an environment simulator, and is complemented by the actual hardware interfaces of Phase 4, an additional independent Phase 3 will not provide a well defined test bed for error detection and therefore is not to be recommended.

2.5.5 Special Testing

Two important categories of software testing not previously mentioned are stress testing, and "user" (or random testing). These activities can uncover errors that might be missed because they exercise the programs in non-standard and, in fact, non-approved ways.

2.5.5.1 Stress Testing. Stress testing involves procedures which stress the capabilities of the computer and/or software in an effort to expose the margins inherent in the design. The most effective stresses are those which have blanket properties. For example, Apollo flight software was subjected to "time-loss" testing, in which the computation speed of the computer was slowed (via the simulator) while environment timing was held constant. The effect was to increase the computer duty cycle and potentially cause difficulties for marginal timing and priority designs. The following types of stress testing are suggested for Shuttle flight software:

1) time-loss,

2) reduction of available dynamic memory,

3) increase in segment time between points at which executive job swaps are permitted.

2.5.5.2 "User" Testing. Additional testing of the Shuttle software will come about as a by-product of program usage outside the software development process itself. Crew training, ground controller exercises, launch facility activities, etc. will all need and utilize the flight software. These personnel will interface with the computer strictly as users, and as a result, many non-standard and even inappropriate sequences will be attempted. In most cases a software performance will be predictable, producing either normal operation or pre-programmed indications of error. On occasion, anomalous behavior will be experienced which must be recorded and reported. Very often software designers and programmers, being close to the code and understanding how it should be operated, fail to anticipate the multitude of possible keyboard (or other input) activities and thereby neglect important sources of errors. Normal computer and program usage will help to uncover some of these errors.

REFERENCES

1. STS Software Development (Study Task 5),C.S. Draper Laboratory Report E-2519, MIT, Cambridge, Mass., July 1970.

2. Dijkstra, E., "The Structure of THE-Multiprogramming System", CACM, May 1968, pp. 341-346.

3. Rossiensky, J.P., "A Kernel Approach to System Programming SAM", Software Engineering, Jon, J.T., editor, Academic Press, New York, 1970, pp. 205-224.

4. Organick, E., Guide to MULTICS for Subsystem Writers, MIT Press (in preparation).

5. Wirth, N., "PL360, A Programming Language for the 360 Computers", Journal of ACM, 15, 1968, pp. 37-74.

6. HAL, the Programming Language - A Specification, NASA/MSC Document #MSC-01846, Intermetrics/NASA Contract #NAS-9-10542, June 1971.

7. Liskov, B.H. and Towster, E., "The Proof of Correctness Approach to Reliable Systems", AF/Mitre Corp. Contract #F19(628)-71-C-0002, July 1971.

8. Dijkstra, E.W., "A Constructive Approach to the Problem of Program Correctness", BIT 8, #3, 1968, pp. 174-186.

9. Dijkstra, E.W., "GO TO Statement Considered Harmful", Letter in CACM, March 1968, pp. 147-148.

10. Mills, H., "Top-down Programming in Large Systems", Debugging Techniques in Large Systems, Rustin, R., editor, Prentice-Hall, New Jersey, 1970, pp. 41-55.

11. Knuth, D.E., and Floyd, R.W., "Notes on Avoiding 'GO TO' Statements", Inf. Proc. Letters 7 (1971) North Holland Publishing Company.

12. Dijkstra, E.W., "Notes on Structured Programming", Private set of notes (EWD-249), June 1971.

Chapter 3

Software Verification Facilities

3.1 Introduction

In the previous chapter, four phases of software development were identified: software requirements, implementation code and test, verification through software, and verification through hardware. The objective of this chapter is to identify the verification facilities for each phase of software development and summarize significant aspects of capabilities.

There are two primary facilities suggested for flight software development and verification. For purposes of this report they are termed the Software Development Facility (SDF) and the Avionics Integration Facility (AIF).

The software development facility is an "all digital" facility consisting of a large scale commercially available computer system. It will be equipped with a large operating memory, adequate random access secondary storage files, I/O peripherals (printers, tapes, cardreader/punch, and other equipment required in a data processing center). Its operating system will support both batch and interactive operations. The SDF will be used to support Phase 1, 2, and 3 of software development. Other support software required in the facility for these phases is discussed in subsequent sections.

The avionics integration facility is a "hybrid" type of test bed facility consisting of actual flight hardware including the flight computer configuration, data bus system and redundant avionics equipment. The AIF will support a simulation of vehicle dynamics and environment. The position and velocity information as well as operator control will be provided by an AIF control computer.

The avionics integration facility is used for integration of flight equipment, testing the total avionics system, and demonstrating its ability to meet mission requirements.

The detailed specifications of these facilities are not within the scope of this report. However, subsequent sections discuss the support software required in the SDF and its capabilities during phases of development.

3.2 Facilities Versus Levels of Development

Figure 2-10 illustrates the software development process. A brief discussion of the facilities used during each phase is provided below.

3.2.1 Phase 1 Requirements

During this phase the software requirements will be defined, and the analytical concepts formulated and demonstrated on the SDF. The facility will be utilized by the analysts at least on a functional basis to exercise and validate the specified software requirements. The SDF will require a digital avionics environment simulator during this phase to enable performance evaluation of software requirements. The environment simulator will be structured as a "modular" environment to enable evaluation of localized algorithm concepts as well as full mission segment analysis. The environment must contain models of the vehicle and avionics equipment to a functional level sufficient to validate software requirements. Subsystems models should be structured to simulate functional dynamics of sensors sufficient to verify the computer interface and for closed loop performance verification.

Other utility software required in the SDF during Phase 1 include a higher order programming language(s) and a compiler(s) such as PL/1, Fortran or HAL. The use of the HOL selected for the flight computer during this phase could facilitate the transferring of some code applicable for the flight computer. In addition, a standard debugging software package for the HOL should be available to the analyst.

The SDF should support interactive operations during this phase of development as discussed in Section 3.5.

3.2.2 Phase 2: Implementation Code and Test

The SDF is used during this phase for development of the flight computer code. The SDF must provide support software to enable flight computer programs to be coded, assembled and debugged. A compiler for the flight computer HOL will exist for the SDF computer. The compiler must provide significant automatic features for compile time checking as described in Chapter 2 as well as provide detailed information on the static structure of flight code, e.g., cross reference listings and compool reference information (i.e., maximum statis verification features).

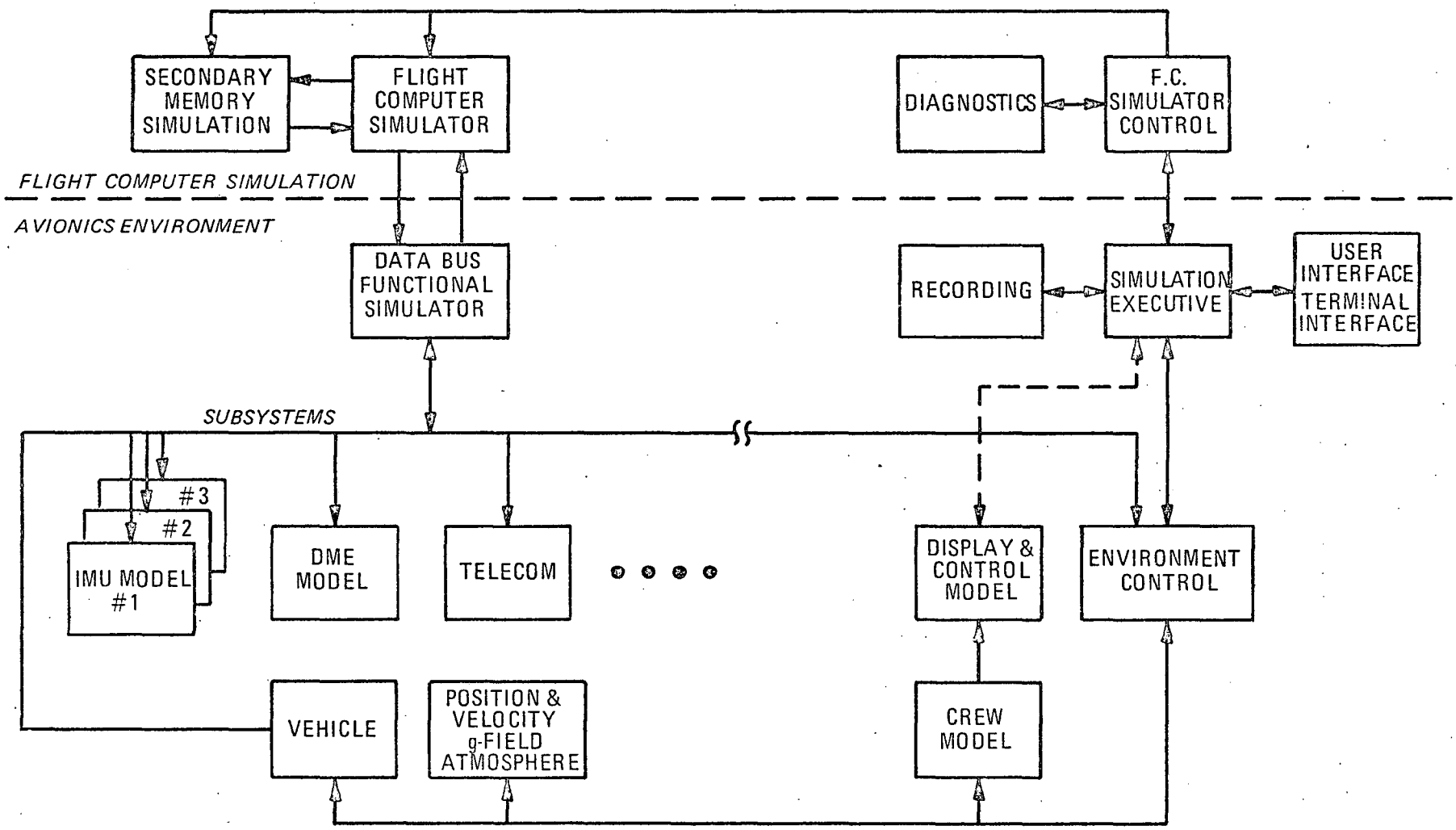
The flight computer compiler will be interfaced to an interpretive instruction level digital simulator for the flight computer to enable execution of the code. The characteristics of the flight computer simulator are discussed in Section 3.3.1. A simplified version of the environmental simulator may be necessary for supporting bench test type debugging.

The SDF should be capable of supporting both interactive and batch operations during this phase. A small number of users located at terminals should be able to compile and execute flight software interactively, particularly during early debugging stages. During this phase, comprehensive debugging aids are indispensable, e.g., conditional instructions, statement traces, variable traps, watch dog timers, a "coroner function" for post run analysis, stress testing options and the collection of dynamic statistics of flight code during execution. Finally, a source language maintenance system is required for the management of flight code including text editors, compool control and system library.

3.2.3 Phase 3: Verification through Software

During this phase performance of the software will be demonstrated as specified by the software requirements. The complete flight code will be available and exercised in both open and closed loop using the flight computer and environment simulator within the host computer. The test plan includes nominal mission segments (or perhaps "mission-like" segments), as well as a variety of off-normal and contingency runs. The SDF will require the same utility software as in Phase 2 with the addition that the avionic environment simulator will be integrated with the flight computer simulator. A full digital simulation capability will be available for closed loop operation of the software as pointed out in preceding sections. These provide a feedback to the requirements, through the use of this environment simulator. Figure 3-1 shows an overview of the all-digital simulation.

Figure 3-1 Phase 3 All Digital Simulation System



The simulations of Phase 3 require flexible editing and environment module control. Run initialization may become complex and should be automated by libraries of stored data.

The debugging tools alluded to above for the previous phase should be augmented by a "rollback" capability. The run is rolled back to a predefined breakpoint prior to the occurrence of an error, and simulation can proceed from the breakpoint. There should also be a selection of environment models, structured in a modular fashion, for use with the flight computer simulator. These models should be both fast and slow, where the speed is determined by the type of environment and the level of detail that is included.

Provision must also be made for automatic recording.

3.2.4 Phase 4: Verification Through Hardware

Phase 4 constitutes evaluation of the total avionics system, both hardware and software. As much of the real subsystem equipment as is practical, including redundancy, should be interfaced to the computer system configuration. Modeling of vehicle characteristics, atmosphere, g-field, and specific forces will be required and accomplished by a general purpose digital control computer.

Although the details of the facility are beyond the scope of this report, a suggested organization is illustrated in Figure 3-2. A digital computer will provide overall control and monitoring of the integration facility. It will contain utility software and debugging features as necessary support to the operators of the facility.

An operators' console is suggested to control the facility. A limited amount of interactive capability should be provided, such as the ability to stop, monitor, and restart. All subsystems including the flight controls will be interfaced to the computer over the data bus, as in the flight configuration. The control computer may be interfaced to the bus enabling it to be used in simulating subsystems not implemented in the facility.

A significant amount of design effort should be expended in defining the capabilities of this facility. Of primary importance to software, however, are capabilities for recording data and status sufficient to permit detailed debugging on the SDF in the event of error(s).

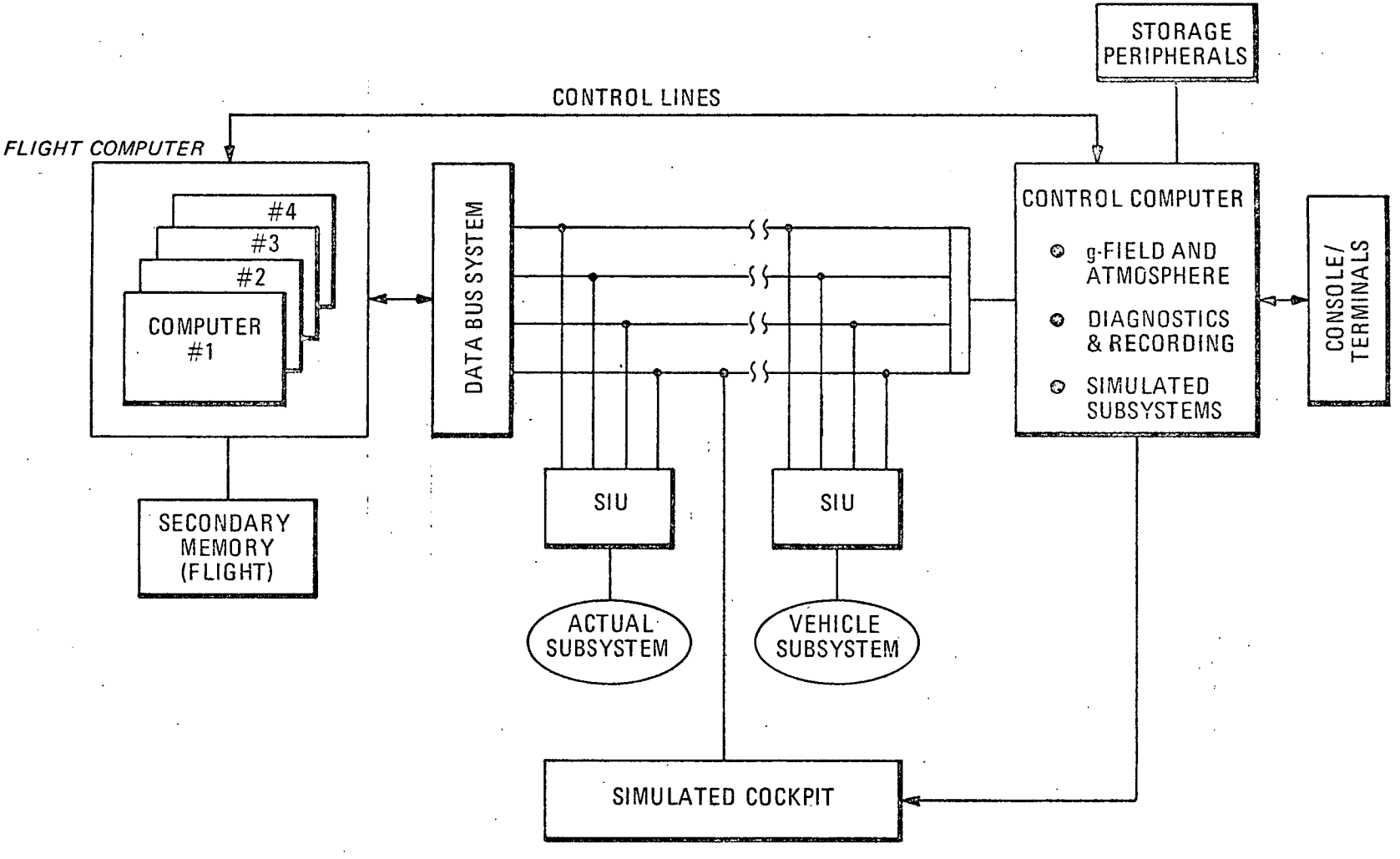


Figure 3-2 Phase 4 Avionics Integration Facility Structure Possible Approach

3.3 Software Development Facility

Software debugging and verification for computers have been characterized by two extremes: one, by running sections of machine-language coded programs on actual hardware, usually with the assistance of special monitoring equipment; and the other by a complete software simulation of the machine and its environment. Because of the extensive scope of the Shuttle computer's software, the first technique is probably not adequate itself due to the lack of adequate debugging aids. Furthermore, the full simulation of the second technique can be time-consuming since the performance of today's aerospace computers already approaches the capabilities of the typical third generation commercial computer. In determining the proper approach to facilities, one must consider:

a) the specific test requirements of the individual phase of development verification procedure;

b) the characteristics of the actual flight computer (or computers), especially the existence of special features to aid the verification of software;

c) the configuration of the total avionics system, especially the levels of redundancy and the philosophy of failure detection and reconfiguration.

Naturally, other practical factors such as cost, ease and visibility of operation, and the ability to define and control the simulation itself, are also driving forces in determining the optimum test facility.

3.3.1 Flight Computer Simulation

Historically aerospace software has been debugged and verified using digital simulation. The flight computer and its environment are simulated on a ground based commercial host computer. The object code of the flight software is interpreted, and a simulated execution at the instruction level is performed by a software

simulator on the host machine. This approach was used due to a number of reasons among which were the unavailability of the computer, the lack of an operating system, and the lack of debugging software for execution on the flight computer. Most importantly, however, was the inability to perform "closed loop" testing of the unaltered flight software interacting with its environment in a "real time" mode. An interpretive "bit by bit" computer simulation provided a means of controlling a simulated clock to enable this real time operation in addition to providing a mechanism for detailed debugging features.

3.3.2 Advantages of Interpretive Simulation

Interpretive simulation provides a powerful debugging tool for software. It offers the following advantages.

a) It provides a means for microscopic evaluation and debugging of software. Since the simulator controls the simulated clock and instruction sequencing, the programmer may request diagnostics to be performed by the simulator during execution of the code. Comprehensive code traces, monitors dumps,and traps can be requested by the programmer with conditional options for enabling and disenabling them. A "coroner" capability can easily be implemented in which the sequence of execution of instructions and the state of the simulated computer,that existed prior to an error, can be recorded and printed for analysis subsequent to an error aborting the run. This can save time in resubmitting aborted runs with appropriate diagnostics to localize an error.

b) Actual timing and performance measures of the computer can be obtained during a simulation run. The activity and duty cycle of the computer can be measured as a function of time and be recorded for load analysis. In a real time computer system such as the Shuttle, the analysis of the load on the CPU, backlogs, the state of jobs in the executive queues, and executive overhead are important to system design verification (i.e., the dynamic flow of the software). This type of information can be obtained through the simulator and presented for analysis.

c) It provides the facility for implementing stress testing features for software evaluation. Features, such as reducing the speed of the computer or the amount of memory available to examine the behavior of the software under this reduced capability, can be incorporated into the computer simulator.

This will be useful in determining the critical design parameters of the software structure and their limits. Similar features can be provided through the simulator, e.g., introducing random,yet controlled error conditions that the software is designed to withstand. When the software is subjected to these conditions, the results can be recorded and presented to the programmer for analysis.

d) It can be built to support multiple users operating in an interactive fashion. Since the simulator is essentially software, it can be developed to allow multiple simulations to occur concurrently, improving facility throughput. This cannot be accomplished as easily using the actual flight computer or a hardware emulator.

e) It provides a deterministic tool enabling a run to be repeated with exactly the same conditions occurring. Since debugging an error is usually an iterative process this is an important aspect of a diagnostic facility.

f) A simulator eliminates the need for altering the flight code with debugging software. In addition to stress testing, the simulator can provide a means of performing "run time" checks on software. It can provide automatic checking for execution dependent errors such as indexing out of an array, watch dog type timing checks on maximum execution time, improper addressing, and control transfers. These are provided without modification to the flight code.

Compiler diagnostics generally insert trap code into the actual code as a method of implementing traces and other diagnostics. This has the disadvantage that actual size and executive timing statistics include this debugging code. In addition, there is the possible introduction of flight software errors caused by this code. However, this possibility is eventually removed as the code reaches flight ready state.

3.3.3 Interpretive Computer Simulation Speed

The primary concern in using interpretive simulation for the Shuttle flight computer is its slow running time and the effective speed of the simulation. The speed of an interpretive instruction level simulation for a Shuttle type flight computer on a host machine is effected by the following factors.

a) The number of instructions executed on the host computer for each flight computer instruction. Depending on the architectural differences between the two computers (in terms of instruction set, work length, etc.) this figure ranges from 10-25 instructions on the average. There will always be some overhead even if the order codes were nearly identical. This limit would basically represent an overhead for simulation control functions, such as incrementing the location counter, checking for diagnostics, and updating the environment. The upper limit,which can exceed 25 is effected by several features, such as the complexity of the instruction set and the size and complexity of the memory of the flight computer.

b) Speed of the host computer and flight computer. The speed of the flight computer for the Shuttle (having a 2-3 microseconds add time) approaches that of some large ground based computers (having 1 microsecond times or less). If this ratio of computer speeds were 1, and if the average number of instructions in the host per flight computer instruction were 10, then the average simulation speed would be 10:1 (discounting other factors such as diagnostics, overhead and environment). However, an important point often overlooked is that this assumes a 100% duty cycle for the flight computer. That is, although the simulator is executing flight computer instructions at a rate 10 times slower than they would be executed on the actual machine, the flight computer should not execute application software instructions 100% of the time. There should be periods of "idle activity" in the flight computer (particularly if 50% speed margins are enforced). The simulation could be advanced through these idle periods and improve overall speed. It is recommended that the capability for advancing the simulation be incorporated into the simulator.

c) Diagnostic options. Simulation speed is affected by the number of diagnostics. A full instruction trace, in which the host computer records the contents of registers for every instruction, can cause an order of magnitude increase in simulation time. Speed is of course dependent on the number of diagnostics and options requested by the programmer for each run.

d) Simulator overhead. The overhead in the simulation associated with instruction execution has been previously mentioned. However, several simulators built by manufacturers for flight computers have been coded in Fortran to enable transferability of the simulation tool to a number of customer host machines. Fortran coding usually introduces an overhead.

Clearly a simulator for a particular flight computer can be optimized for a host machine. This optimization can be achieved in areas, such as set up and initialization, memory fetching, and instruction sequencing.

e) Environment. One of the important factors influencing overall simulator speed is the external environment of the computer. Typically the computer simulator is interfaced to a model of its external environment for the purpose of simulating computer I/O. For the Shuttle configuration this includes: secondary storage, display and controls, data bus and avionics equipment. Although this environment simulator is discussed in a subsequent section, it must be pointed out here that the host computer time consumed in updating and maintaining the environment can be a significant percentage of the overall time. In fact, depending on the complexity of the models and the I/O occurring in a particular code sequence, it can dominate the simulation by consuming the major part of the host computer's time. The run times for lunar landing simulations on the Apollo digital simulator were influenced considerably by the time consumed in updating the environment. It is roughly estimated at requiring 75-9)% of the overall run time.

3.3.4 Experience

Interpretive instruction level simulators are available for most off-the-shelf aerospace computers and are implemented on at least one host computer. A limited amount of information is available, however, concerning the run time simulation speed. The Apollo Guidance Computer with a 12 μs cycle time is simulated on the IBM 360 model 75. In a mode with no environment this operated at a ratio of better than 1:1 (i.e., .1 sec of 360 time for 1 sec of AGC time). With a full environment the most time consuming digital simulations were lunar descent and landing for the lunar vehicle and rendezvous operations in the command module. However, other simulation speeds for new aerospace computers have been estimated and operated at significantly slower speeds*, in fact, by orders of magnitude such as 100:1 or greater.

---

* The CDC Alpha simulated on the CDC 3300 (slower machine) operates at 500:1 with no traces or dumps.

3.3.5 Avionics Environment Simulator

The environment simulator for the SDF will be implemented entirely in software. It will be modularly structured to include the "environment" external to the computer. The avionics subsystems will be represented by mathematical models to a degree sufficient to represent their interaction with the computer and to verify the software. More than one module for each subsystem may be provided. Since the environment consumes a significant portion of the overall simulation time, the capability of providing variable fidelity in subsystem models can be a significant factor in improving simulation time requirements.

During Phase 3 the avionics environment simulator will be coupled with the interpretive computer simulator through a functional simulation of the data bus. The centralized integrated avionics system utilizing a data bus system is described in Chapter 4. The data bus system will receive input/output commands from the computer and can be used to control computation by the environment simulator. When enabled, the simulator updates the environment to the current time in the flight computer and passes the required sensor input data to the ICS. This process continues for the duration of the closed loop system simulation. The interface module should prevent excessive updating of the environment. The amount of computer time and memory space required for such "all up" simulations can be large. From Apollo data, this fact should limit the number of closed loop system simulation type jobs that are simultaneously supported by the SDF.

Data recording mechanisms are easy to provide as a part of the simulator system. The recorded data then can be automatically processed by data reduction programs which provide publishable records of flight software performance. Programmers can develop and tailor the data reduction programs to system specifications.

A representative list of the type of functions required in the avionics environment simulator based on the Phase B Space Shuttle design are:

A. Physical environment

1) gravitational field
2) atmospheric model
3) star, earth, moon, sun ephermerides

B. Vehicle

1) thrust-rocket and jet engines

a) propulsion
b) fuel slosh, flow

2) airframe

a) geometry
b) bending modes
c) aerodynamics and coefficients
d) thermodynamics and coefficients
e) mass properties

3) mission phases

a) prelaunch
b) ascent
c) on orbit
d) rendezvous
e) entry
f) aerodynamic flight and landing

C. Vehicle subsystems

1) guidance, navigation and control

a) inertial measurement unit
b) flight control system
c) star trackers

2) communications and navaids

a) distance measuring equipment
b) radar
c) microwave landing system
d) S-band links

3) data management

a) computers
b) bus structure
c) interface units
d) control units

4) mechanical airframe

5) displays and controls

6) crew or pilot simulation

3.3.6 Redundancy Simulation

3.3.6.1 Computer System Redundancy. The Phase B computer configuration employs contralized quad redundant computers, each executing "identical" software. The necessity of providing interpretive simulation for up to four computers in the software development facility is questionable, since the control of the computer system error detection and switching is external to the software (as described in Chapter 4). The added complexity of redundant interpretive computer simulation in the SDF does not appear cost effective. The function of verifying Phase B redundant computer operation can be more easily accomplished in the avionics integration facility since it contains the actual hardware with redundancy. If a significant portion of the software were dedicated to inter-computer communication and control,then multiple computer digital simulation should be considered.

3.3.6.2 Subsystem Redundancy. The executive software system will control switching of redundant avionics equipment as described in Volume 2 of this report. Digital simulation of redundant avionics equipment may not be required in the environment simulator for all subsystems. It is, however, desirable to include this capability in the environment simulator for those subsystems which utilize software control over the redundancy. At a minimum the environment could contain request options for equipment malfunctions (e.g., out of limits, status error, etc.) to enable testing of flight software or redundancy switching.

The initial configuration status should be a control option; for example, specifying active and standby units and their states at the beginning of runs and a time line specification for change of status. In addition, subsystems which employ software voting on multiple inputs may require an environment simulation with multiple redundant units operating. The same fundamental model of the subsystem can be used by the environment simulator to accomplish this function.

Although these features add some complexity to the environment simulator,as well as increase overall simulation time, they are considered necessary to support the testing of software redundancy management functions and onboard checkout software.

3.3.7 Space Shuttle Simulation Speed Improvements

Several recommendations are now made for the interpretive simulator used for Shuttle software development.

a) It should be efficiently tailored to the host computer and flight computer for maximum performance. An average of 10 instructions per flight computer instruction should be a design goal with minimum overhead.

b) The simulator should include features to advance through "idle" CPU periods of the flight computer and thus reduce overall simulation time. The simulator should also advance to the next predicted environment event when the CPU is idle. This could improve overall simulation speed by a significant percentage (30-50%) particularly in longer mission simulations.

c) An interpretive "statement level" simulation technique for direct execution of higher order language statements on the host computer instead of on the instruction level should be investigated. The key to this approach is to devise methods of working around the maintenance of real time requirements and occurrences. If the software were structured to be less dependent on interrupt occurrence, it could be executed directly on the host computer.

d) Modularize and improve the avionics environment to improve speed.

e) Partial development of flight software on the host computer.

3.4 Direct Use of Higher Order Language on the Host Machine

The use of a higher order programming language and compiler in previous flight software enables some code to execute directly on the host machine, minimizing simulation requirements. The availability of a compiler with multiple code generators for both the flight computer and host machine allows the same source code of a flight program to be compiled and executed on the flight and host machine. Since the syntax analysis phase of the compiler is machine independent, source code may be prepared by the programmer, compiled, and executed directly on the host machine. Subsequent to some debugging, the same source code can then be compiled using the code generator for the flight machine. This approach may be extremely useful in providing rapid debugging of certain modules of flight software and minimizing the time consumed in executing the code in the flight computer simulator. Although the code generated for each computer may be significantly different as a result of differences in the host and flight computers, a reasonable amount of testing of the program algorithm logic at the statement level may be accomplished directly on the host machine, particularly for data modules. For example, syntactical coding errors, parameter passing, computational errors other than precision and logic flow can all be executed and checked on the host machine which should eliminate the need for using the flight computer simulator for these purposes.

This approach is limited in that it cannot be used for evaluating timing and memory aspects of the code. Its use, therefore, should be restricted to evaluation of code at the statement level and not as a substitute for implementation on the actual flight computer. If the word length and number representation of the two computers were similar, this would aid in verifying numerical computations on the host machine. Although a reasonable amount of testing may be accomplished particularly in Phase 2, the code must ultimately be compiled and verified within the flight computer. If applied properly, a HOL can, however, be instrumental in improving production, reducing turnaround, and lowering simulation requirements on the host machine.

3.5 Comparison of Interactive and Batch Computer Facilities for Shuttle Software Development

The purpose of this section is to discuss the advantages and limitations of a user interactive environment for the production of Shuttle software and to compare this with the traditional batch environment. The term "batch" takes its name for historical reasons from the fact that input cards were combined together into a group or batch which was then submitted to the computer. It was typical to sort these cards into similar job types and then to submit them (e.g., a Fortran batch, a Cobol batch, etc.). The name still persists, denoting the form of operation in which the jobs to be run are fed into a single job stream and queued for subsequent execution. A further distinction can be made as to whether the batch facility is multiprogrammed or not. A multiprogrammed system permits the computer to operate on more than one job at a time (interleaved execution) in order to increase the computer utilization efficiency, or in the jargon to increase the "throughput". Examples of multiprogrammed operating systems are OS/360 MVT and Univac 1108, EXEC 8. Conceptually it is immaterial to this discussion as to whether the batch facility is multiprogrammed or not. It is of direct concern only to the computer operations staff in their quest to provide better service.

The fundamental issue that is germaine here is the turnaround time. Turnaround time is the elapsed time from job submission until results are available. Turnaround times of one hour or less from a batch facility are remarkable and rare. A four hour time is considered good service, and one day is a typical turnaround time in a busy batch environment. A turnaround time of longer than 24 hours is considered excessive and poor service. The advantages of a batch computer facility are:

1) it is easier to run a batch facility, to control the jobs that are being submitted, and to exert some influence over the mix and priority of the jobs to be executed via the scheduling algorithm.

2) A well run batch facility can be extremely efficient. A high level of output per unit of time can be maintained within the available system resources.

An interactive facility provides hands-on programming for the user. It consists of direct communications between a central computer doing job execution and a user via remote terminals. The user submits commands to the central computer and then examines the results at his terminal. He is then at liberty to stop the execution, fix up errors, look for bugs, redirect the computer's activities, or permit it to run to completion. In this situation turnaround is not meaningful. The user is not delivered the entire output from a run or sets of runs but is able to read the results as they are generated, or to interact with the computer. Thus, the name interactive facility. The figure of merit in this case is the response time. Response time is a measure of the period between the initiation of a stimulus at a user terminal and the indication of some event, perhaps observed by the user. Response time is a function not only of the amount of CPU time that a particular job step requires, but also of how busy the central computer is kept servicing other users, the efficiency of the computer to support the time sharing mode of operation and a number of other factors. In general, however, the response time of an interactive computing facility can be measured in seconds.

The precise details of the computer systems necessary to support the time shared environment are not particularly relevant. Suffice it to say that there is an overhead associated with swapping the users' programs in and out of the central computer's memory. The size of this overhead is far from negligible; it can become overwhelming if too many users are present. In fact, nearly all time sharing services are forced to limit the number of users so that the quality of the service does not deteriorate beyond acceptable limits.

The Shuttle software development facility should be implemented to support multi-user operations during Phases 1 and 2 but to operate "full up" simulations of Phase 3 in a batch mode.

The advantages of an interactive facility are generally obvious.

1) Foremost is the ability to make rapid corrections and changes to program and job decks. This is especially beneficial during initial program development while still weeding out control card errors and trivial coding errors. Fast detection of out-right blunders which might take a week on a batch system can be debugged in a single terminal session on a time sharing facility.

2) The hands-on appeal tends to promote individual source file preparation by each programmer rather than submission to a central key punching and verifying operation. Fewer errors are made by programmers typing their own inputs. They are more familiar with the meaning of the material than a mechanically functioning key punch operator.

3) Debugging capability is enhanced in two ways:

   a) The interactive techniques employed in the trial and error nature of hunting for solutions.

   b) The immediacy of the results. It is much easier to search for the trouble when the objective and the purpose of the test are still fresh in one's mind.

4) A more orderly test process can be followed in an interactive environment. Good results from a single test precede and give confidence to the quality of the results before a whole spectrum of tests are conducted. In a batch environment, it is not at all uncommon to see individuals submitting multitudes of tests that all fail for the same reason. Poor turnaround forces users to gamble and submit a plethora of test cases when these yield no useful product, and only waste more machine time and worsen the turnaround time problem.

5) The user's natural trains of thought flow more consistently in such an environment. Notes need not be scribbled on scraps of paper for incorporation into tomorrow's run. The user can interact with job execution in one continuous session of updating, compiling, testing, and debugging.

Both interactive and batch capabilities are seen as desirable for the Shuttle software development facility, as previously discussed.

Chapter 4

Space Shuttle Phase B Design Review

4.1 Introduction and Scope

An initial task in this overall study involved viewing several Phase B avionics configuration designs with emphasis on determining their impact on the software development effort. The objective of the task was to review the general architecture of each configuration and to assess the impact on the software development of features within each of the avionics configurations. The scope of the study did not include defining a configuration or analyzing functional subsystem requirements associated with the avionics system. In addition, no attempt was made to perform a detailed hardware/software trade-off analysis or to critique the design itself. Instead, the primary purpose was to gather and organize information on software design parameters which were derived during Phase B and to use these as a basis for the rest of the tasks in the study. Key parameters included functional requirements, memory size, and operating speed. Orbiter vehicle requirements were emphasized over the booster vehicle in the belief that the requirements of the former are the more difficult to satisfy.

In Section 4.2 below, the Phase B avionics designs of North American Rockwell (NAR) and McDonnell-Douglas Aircraft Corporation (MDAC) are summarized. In Section 4.3, there is a discussion of software implications that follow from features of the avionics designs common to both the NAR and the MDAC designs. In Section 4.4, on the other hand, the software implications of the differences between the two technical approaches are set forth. Finally, in Section 4.5 the overall problem of checkout software is discussed.

4.2 Summary of Phase B Designs

Although a number of avionics system designs were identified during the course of Shuttle Orbiter Phase B, this study was focused on a review of configurations derived by two prime contractors: North American Rockwell (NAR) and McDonnell-Douglas Aircraft Corporation (MDAC). A more detailed exposition of these two candidate avionics system designs is presented in Appendices A and B.

Both Phase B configuration designs have many similarities with respect to the impact on flight software. The concept of integrated avionics and a centralized computer imposes certain general requirements on software functions from the beginning. A summary of the major features of each baseline system is depicted in Table 4-1, and the configurations of the primary computer system for the NAR approach and the MDAC approach are depicted in Figures 4-1 and 4-2 respectively. Both avionics configurations consist of centralized data management computer systems. Both systems contain a high speed time multiplex serial data bus system which provides a communication path between the avionics equipment and the prime computer complex. The bus system provides a capability of interfacing with redundant electronic subsystems via a remote interface unit. Each computer system is also interfaced to a redundant secondary storage unit. Each bus line is assumed to be physically separated onboard the vehicle for reasons of reliability and carries serial digital data at a rate of 1 MBPS.

The central computer is the sole authority on the bus, and all communications with equipment are initiated and directed by it. The central computer computation functions encompass almost all aspects of operation of the total Shuttle Orbiter mission, including flight control, guidance and navigation, displays and controls, on-board checkout, and configuration management. The significant exception is the dedicated processing associated with the main engine systems.

The Phase B design ground rules for system failure tolerance was the primary factor in influencing the design of the configuration; that is, "fail operational" after the failure of the two most critical components and "fail safe" after the third failure. In a practical system, failure tolerance requires that each major element of the system possess internal functional redundancy and a highly effective technique for failure detection to allow quick reconfiguration in the event of a failure. It is this requirement that has introduced the greatest complexity into

Table 4-1 PHASE B BASELINE AVIONIC SYSTEM FEATURES

	System Feature	MDAC Baseline	NAR Baseline
I.	Computer System	• Centralized four computer organization No memory sharing	• Centralized modular organization of 4 central processor and hardware interconnect of 12 memory modules Configure as 2 computers of 5 memory modules and 2 processors with standby units
	• Operating Memory	• Each computer has 65K, 32 bit words	• "Active computer" and "checker computer" have 40K 32 bit words each 2 spare 8K memory modules 2 spare processing units
	• Central Processor Characteristics	Not available off the shelf • 2 μs add time • Floating point • Half and full word instructions and data • 8-16 general registers 4 arithmetic registers • Large instruction repitoire with macro instructions for math functions, sine, cos.etc. • Programs separate from data	Available IBM 4PI AP-1 • 2 μs add time • Fixed point only • Half and full work instructions and data • 8 general registers • 77 instructions (IBM 4PI AP-1)
	• Interrupts	External 4 Internal Power fail Illegal I/O Memory protect violated Fault detect External clock	External 4 Internal 11 As per IBM 4PI AP-1
	• I/O	Minimum of 2 channels: One to SCU One to Input/Output control unit with DMA access to memory	Minimum of 2 channels: One to DATABUS One to Mass Memory

System Feature	MDAC Baseline	NAR Baseline
⊛ Other	Indirect addressing and full indexing	Indexing and direct and indirect addressing
II. Secondary Memory	2 Units Tape 1.2×10^7 bits Via Data Bus	Triple redundant Drums 400,000 words (32 Bit) Direct to Computer At transfer rates of 3.2 MBPS
III. Data Bus	⊛ Single bus line for command and data ⊛ Quad redundant ⊛ Time division multiplexing Biphase modulation 10^6 BPS	⊛ Two bus lines for separation of command and data ⊛ Five bus line redundancy ⊛ Same
⊛ Bus Control	Remote unit addressing Via command response	⊛ Same
⊛ 8NB System Interface	⊛ One bus line interfaces to one interface unit ⊛ Option for memory at interface unit	All 5 bus lines interface to one interface unit
IV. Software Executive Structure	Synchronous Structure 20 msec minor cycle 1 sec major cycle	Synchronous Structure 40 msec minor cycle o Event handling o Program reload for mission phases o Internal self test
Software Size Estimates	50K 32 bit words Max	36K 32 bit words for in orbit phase Max
Software Speed	462 ms/sec Duty cycle during landing	284 KAPS max for landing

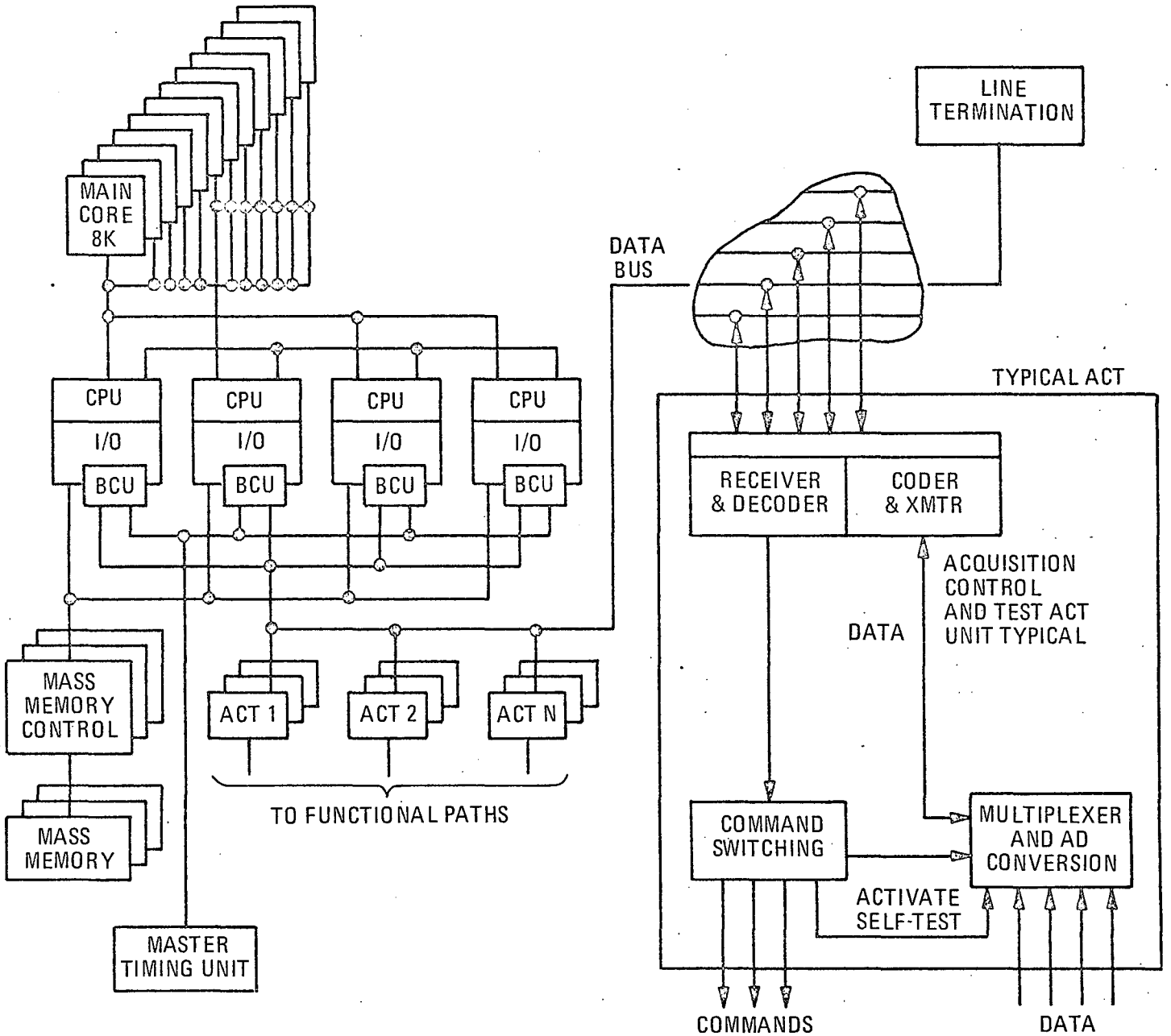


Figure 4-1 NAR Baseline Computer Configuration

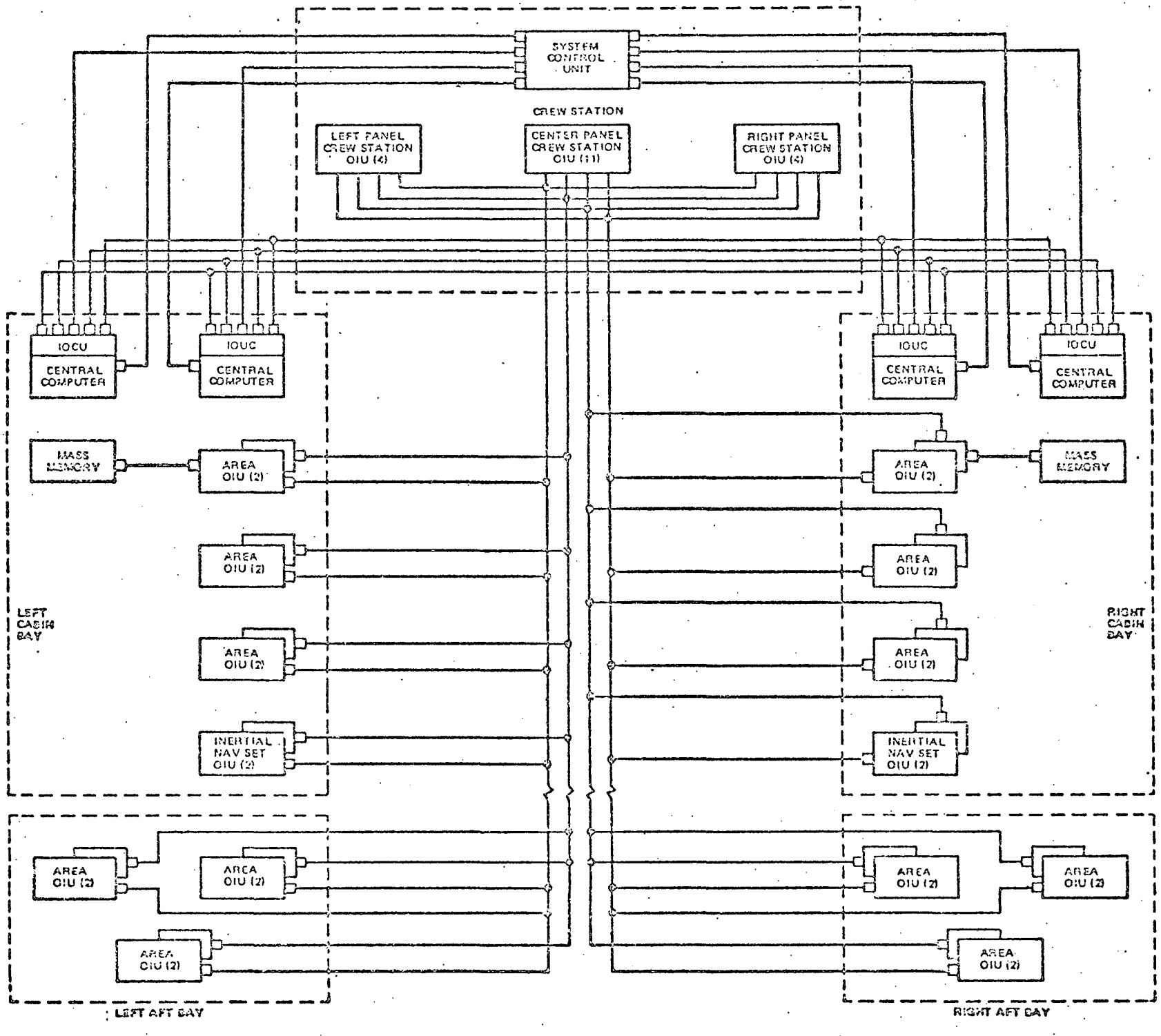


Figure 4-2 MDAC Baseline Computer Configuration

the integrated Shuttle avionics system. The high level of redundancy that is required to achieve a multiple failure criteria encourages application of voting and comparison techniques especially to systems which generate output data; for example, the computer. One of the penalties that must be paid for the voting approach to failure protection is that all redundant copies of a given piece of equipment must be powered up and operating identically. Accordingly, the designs devised during the course of the Phase B study encompassed significant and costly levels of redundancy. Furthermore, the requirement for largely autonomous operation of the Shuttle vehicle and the two week turn-around time between missions results in designs which incorporated: 1) onboard checkout and fault isolation, and 2) computer controlled switching of redundant equipment.

4.3 General Software Implications of Phase B Designs

4.3.1 Centralized Computer Software Management Problem

The software in the central computer services all sensors connected to the bus and performs all of the functional requirements of the system. The boundaries of a functional subsystem tend to disappear, and exist only as shared software in the computer. For example, the stabilization and flight control system will consist of redundant sensors connected to the bus operated by programs which are allocated a portion of the central computer resource. The total flight control subsystem (including software) is therefore not a visible separate entity but becomes, in fact, part of an integrated avionics system.

It is important to observe that the size of the central system has direct bearing on the ability to manage it from a project point of view. It would be obvious to most that the central concept is clearly controllable if the size of the software were small enough. If it were small, then partitioning the system functionally via distributed hardware would not be desirable since it would probably create more management problems than it would solve. The converse would also appear to be true. That is, a large software effort brings with it numerous management problems (e.g. storage and time budgets, priority allocations, complexities in configuration and change control, definitions of shared variables, software interfaces). If the software were large enough, for example 200,0000 to 300,3000 words, then partitioning the job into several functional computers might alleviate these problems.

The Shuttle software has been estimated at approximately 40-50K words with a processing requirement of 200K OPS. Many centralized ground-based systems with much larger software have been successfully developed, particularly the Air Force systems: 425L NORAD, 496L Spacetrack, BMEWS, and others. These systems, however, have not had the difficult FO-FO-FS reliability requirement nor have they been totally free of management problems. They have, however, been successfully developed. Other flight systems have been built or are being developed which are as large if not larger, than the Shuttle, such as the Navy's A7 and AADC. It is, therefore, reasonable to assume that the software effort for a Shuttle central computer is of the size which can be successfully managed and developed.

4.3.1.1 Advantages. The following are some potential management advantages of the centralized computer system approach. Admittedly, these advantages express a subjective point of view.

a) The centralized system promotes standardization of approach to system design problems. Centralized software and a central bus provide the means to impose this standardization.

b) The systems integration task becomes intimately involved in the development of the central computer system and software. This minimizes the number of organizations during the design, integration, and testing of the system.

c) The system can consist of computers, software, the bus system and line replaceable units (LRU). LRUs will be accepted from suppliers based on their performance to functional specifications and their interface to the bus; for example, an RCS quad. On the other hand, the stabilization and control functional system (which is not an LRU) cannot be accepted until performance tests are run using the flight control software, central computer, and all required sensors.

4.3.1.2 Disadvantages.

a) The key problem with the centralized system is that there appears to be no straightforward way to partition functional subsystems around which project organizations may be formed. It is an inherent characteristic of the centralized approach that, in most cases, functional organizations will overlap and cause conflicts in defined responsibilities. For example, the G&C subsystem supplier

in a centralized concept could perform analysis, design logic, specify sensor requirements, and provide software specifications (equations). However, integrating these to perform the GNC functions requires developing the software for the central computer and interfacing the systems to the central bus system. One suggested approach might be for a functional subsystem supplier to develop the software, constrained by certain standards and budgets, using a higher order language and deliver to the integrator the completed software as well as the sensors. This, however, is probably not a workable nor attractive arrangement for either the subcontractors or the integrator. The division of responsibilities is unclear. Does the subsystem supplier remain responsible for the system during the test and acceptance phases or is this responsibility transferred to the integrator? Can the integrator specify subsystems (including software) sufficiently so as to be in a position to accept "independently" designed subsystems and make it work? These are difficult questions and they are unanswerable at this time.

It is more reasonable to assume that the organization for a centralized system must consist of an avionics integration contractor, perhaps a software support contractor, a computer system and data bus contractor, and a number of suppliers of sensor equipment. The integrator would be forced to organize the software and the analysis effort functionally, seeking outside support for analysis and design where required.

b) The centralized approach places responsibility on the integrator for specifying the sensor requirements for all functional subsystems. Needed expert support on particular functional subsystems may require numerous subcontracts.

c) A centralized system does not provide isolation or localization of changes. Changes made to a particular system such as electrical power distribution, may not be easily or absolutely isolated from the rest of the system.

d) It does not provide a hardware independence of functions. That is, security of subsystems is only achieved through the bus system design.

e) Incremental delivery of the avionics system must be achieved through software. For example, in order to

support early horizontal flight tests of the vehicle, a preliminary software package must be prepared containing those functions required. Even though an "all-up" avionics system is not required, a central computer and a bus system to operate the sensors are necessary.

In summary, a central computer configuration does provide some significant advantages in design and configuration control, standardization, and minimum organizational interfaces. Yet its drawbacks are also apparent. It is the most difficult to partition functionally and requires sophisticated, but achievable, software and bus control designs to ensure subsystem isolation.

4.3.2 I/O Timing Difficulties

A class of system problems exists in the operation of a time-shared bus which is associated with the correlation of data and commands with "time". For example:

a) Correlation of data and absolute time. Several system computations demand the acquisition of data from separate subsystems at the same time. For example, a navigation measurement combines sensor data with attitude information, correlates both with the same absolute time, and updates the navigation data. With a synchronously controlled data bus, in which sampling is performed only at fixed minor cycle intervals, time may only be established with a granularity of the sampling period. That is, all samples taken during one minor cycle are associated with the same time tag. If a finer time reference is required it must be provided by a local clock. In an asynchronously driven bus system a finer reference time quantization may be obtained because a specific I/O command may be serviced within approximately 100 μs (depending on the I/O queue backlog).

b) Local precision timing. Another problem that may arise concerns the precision timing of events at geographically separate and remote subsystems, for example, the timing and coordination of firing commands to the RCS jet thrusters. From a system point of view, it is desirable to design such subsystems to receive a message which contains not only the command but also the firing interval. The impact on I/O complexity, bus traffic, and response due to separate transmissions to command the thruster on and then off could be considerable if this type of bus activity predominates. The capability for local precision timing may be incorporated into the subsystem or terminal.

4.3.3 Failure Identification, Isolation, and Reconfiguration

Probing deeper into the problem of failure identification, isolation, and reconfiguration, it is clear that this is one of the most difficult technical problems to be overcome in the Phase B designs submitted by either NAR or MDAC.

Control of multiply-redundant inertial subsystems (ISS) epitomizes the extreme aspects of this problem. The I/O timing problem discussed above has a particular bearing on the ISS failure identification and isolation problem. A sick inertial measurement unit (IMU) among a set of IMUs will exhibit a gradual drift of its state vector relative to two others. In a dynamic situation, precise timing of the comparison is required. This drift can be detected only by having two other systems as a standard with which to compare. (If one failure has deleted one out of three IMUs, then there is no present method for clearly distinguishing gradual degradation of one of the remaining pair.) Finally, reconfiguration in a dynamic situation requires another timing achievement so that the bit stream from one IMU is shut off and that from the substitute IMU is turned on, using the best possible estimate of the true state vector.

4.4 Software Implications of Differences Between Two Phase B Contractor System Baselines

4.4.1 General

Although there are many differing aspects of the North American Rockwell (NAR) and McDonnell-Douglas (MDAC) configurations, this section reviews only those features pertinent to impact on software, since it was the intention of the study to concentrate on the differences in each baseline system and how those differences effected software. Five principal differences are addressed:

a) computer organization in redundant operation,

b) operating memory,

c) secondary storage and utilization,

d) redundancy management and subsystem interfacing, and

e) operation of display systems.

In evaluating these configurations, features which directly impact the complexity of the software system have been identified. In summary, the Phase B configurations were compared in an overall sense and those features identified which affect software design. Software costs were not traded-off against the associated hardware costs.

Several difficulties were encountered during the course of the review of Phase B, principally due to the scale of information presented in the reports made available during the course of the study. In attempting to evaluate subsystems and their modes of operation, either the information was too detailed or too gross for adequate software load analysis. Selected redundancy operational modes were discussed at top level only. Total impact on software requires lower level design to evaluate the "iceberg effects". In addition, information pertaining to software requirements was not easily found within the reports, because it was distributed primarily among the functional subsystems areas. A definition of various application software packages in terms of the functional inputs and outputs and frequency of operation was either incomplete or too vague or undefined in some cases. As a result the information presented in this chapter is based solely on Intermetrics' evaluation of the impact on software as interpreted by the description of the configuration in this document.

4.4.2 Computer Organization in Redundant Operation

A key difference in the baseline systems is the method used for computer error detection and reconfiguration. Both baseline systems appear to be designed to incorporate the comparison of data transmitted by the computer over the bus system as a means of error detection of the central computer. The NAR design incorporates a synchronized two-computer environment where both computers are performing functions simultaneously and cross-checking calculations prior to transmitting data onto the bus. One computer is considered the master and the other the slave. All data transmissions on the bus are made via the master computer. Both computers input identical data via the data bus utilizing the same serial channels. The MDAC system, on the other hand, can have up to four computers performing the same computations. One computer is designated as active by the crew and is the only computer transmitting on the data bus system as in the NAR design. Each of the operating standby computers, through its IOCU, performs a comparison of data being transmitted by the active computer. After performing this bit-by-bit

comparison it transmits a binary signal to the system control unit. The system control unit based on the results of the comparison sent by each standby operating computer determines if all computers agree or if an error has been detected. The system control unit is designed so that it can select the controlling computer during the time critical phases of the mission,and/or the crew can manually override the selection at any time as well as select the active computer directly.

The problems of recovery, via software, after the detection of a computer failure can be severe. Error detection by voting on and/or comparing the outputs of two or more redundant operating computers is utilized in the Phase B avionics designs. Some techniques can be made less difficult to implement if the elements being compared are complete computer units, each consisting of the full complement of memory processor and I/O controls. A detected failure results in the removal of a complete computer and its replacement by a standby unit. However, if redundancy error detection and recovery are taken to the level of the memory unit (which is then considered as an element of the system independent of the processor), the complexity of the reconfiguration problem increases. The recovery from a memory module failure requires either the replacement of the failed module by an identically loaded copy or the regeneration of its state prior to the hardware failure. This involves a continuous updating of spares or an initial load with consequent delay in system operation.

Failure detection by computer comparison poses a problem of determining, in the event of a comparison failure, which of the processes is defective.

In the NAR approach it is necessary,subsequent to detecting the error,to run diagnostic routines in each computer and then to reconfigure once the safe computer is identified. However, reconfiguration in this style poses the following questions:

1) What happens to the time critical processes that may have been active at the time?

2) If the active computer is the one that failed, how does it hand-off control to its back-up?

3) What is the next step if both computers indicate failure?

These questions do not imply that the problems are insolvable but they do underline the impact of placing the recovery and error detection responsibilities in the software.

The MDAC approach, on the other hand, has the error detection and the reconfiguration completely independent of software. A voting mechanism decides on the basis of a majority of comparative results whether the active computer is operating correctly. It may also determine which of the inactive computers has developed failure. In the event of a failure, the active computer is replaced by one of the standby computers. There is, however, a possibility that a split vote situation will arise with these binary comparison outputs and a greater likelihood of identical multiple failures.

A characteristic of both approaches is, of course, that neither one can detect errors in the software. For the purposes of comparison and voting, the software in each of the redundantly operating computers must be virtually identical. It is, therefore, inherently non-redundant. A software fault would produce data, which being identically erroneous, will appear to compare correctly. This condition must be classed as a design error, which along with a similar logical hardware fault, must be prevented by careful design and adequate verification rather than by complicating the system in an effort to make it immune to conceptual errors.

4.4.3 Operating Memory

The NAR baseline system incorporates twelve memory modules of 8192 words each, which can be accessed by any of the four processing units. Although this memory organization may offer a lower power and weight requirements from a hardware viewpoint and a degree of expandability, it has a direct impact on the software executive system. The executive system must provide the necessary logic to configure and assign the physical memory modules to the "active" and "checker" computers. The operating memory for each computer must be established via configuration management and assigned from the available set of twelve memory modules. Furthermore, the executive must be capable of reconfiguring modules in the event of a failure by using the two backup spare modules. Replacement of a failed memory module requires that the spare contain a copy of the information that the failed module contains. Then it can be switched into either the active or checker computer configuration to replace the failed module.

As a consequence, the executive system must be designed to insure that spare memory modules are updated at appropriate

intervals and with the proper information. Software associated with the management of this type of memory system is clearly more complex than if each process were connected to a dedicated operating memory.

The MDAC baseline incorporates such a design, i.e., four computers each with their own operating memory. Since the memories are not shared among the processors, there is no need for software to manage their switching. The apparent cost savings in hardware gained through the use of a modularized interconnected memory system must be traded-off against the increased complexity and cost of the software necessary to manage this type of system.

4.4.4 Secondary Storage and Utilization

Incorporating a secondary storage device into the architecture of the onboard computer configuration offers several advantages from a software viewpoint. Traditionally, memory in aerospace computers has been at a premium. As a consequence, very careful planning and utilization of the operating memory has been a major part of the software effort in aerospace programming. Typical software approaches to this type of programming involve such things as tricky coding to conserve space and overlaying techniques for sharing memory among operating tasks of the computer system.

Both Phase B designs have incorporated a form of secondary storage. The NAR baseline design utilizes a redundant drum system. The drum system is used during system operation to reload the operating memory with programs for each major mission phase. The MDAC design incorporates a tape system primarily intended for backup copies of the program. It does not appear to be used for dynamic loading of programs during the mission.

Although there is an additional increase in the complexity of the executive software for incorporating dynamic loading of the program storage during flight, utilization of secondary storage in this method appears to have several advantages. First, it lessens the impact of incorrectly sizing the operating memory. No matter how carefully software functions are initially sized, including budgeting allowances, the requirements will change somewhat for the project and eventually fill the operating memory. Second, because programs are loaded for each phase of the mission and exist in the computer only when required, there is some inherent

separation in preventing sharing problems. It maximizes the independence of these functions and provides a certain visible separation of software.

4.4.5 Management of Redundant Subsystems by the Central Computer

One of the most complex functions of the software system associated with the Phase B designs is that associated with configuring and reconfiguring sets of redundant avionics subsystem equipment. A high degree of redundancy for avionics subsystem equipment demands complex software for its maintenance and operation. The cross-strapping of avionics equipment through the redundant bus system establishes a multiplicity of paths for the computer system to control.

The two Phase B designs have different approaches to the redundant interfacing between the bus system and the terminal. The MDAC system has in effect no cross-strapping between the bus and the terminal. Each DIU, in effect, is connected to one and only one bus. A separate address in the bus control word format must be provided for each DIU located within the configuration. Software must be used to address a new terminal in the event of a failure somewhere along the functional paths. The NAR system on the other hand has its remote terminals (ACT) cross-strapped to all five buses. Five bus lines are interfaced to each ACT with a single address associated with each ACT. This feature to some extent limits the necessity of the software of re-establishing a new address in the event of failure along the functional path.

4.4.6 Display Subsystem Interface to Computer

Both baseline designs incorporate some form of display subsystem in order to provide integrated mission information to the crew. The display system provides the crew with, among other things, continuous visibility of trajectory performance and progress during critical mission phases. The display and control systems are interfaced to the central computer via the data bus system. The characteristics of both display systems vary. However, they both incorporate some form of CRT display with alphanumeric capability. Flight control graphic displays are also available on CRTs. The crew requires access to the computer memory for parameter call-up, monitoring of parameter updates, and requesting computer processing functions.

The MDAC system incorporates a display system which includes a composite of microfilm-stored, fixed format data and a TV raster presentation. Information generated by the computer may be sent to the display system superimposed on a stroke-written presentation. The keyboard or computer-inserted data is written against a background raster format. The fixed background format data includes operator instructions, parameter names and units or other descriptive data to enhance crew interpretation and increase crew confidence in command of inputs. Local TV raster presentations include microfilm, stored procedures, text, flow charts, schematics, etc. This has the distinct advantage from the software organization point of view that only updated data for variable displays need be sent over the data bus to the display system.

In the NAR baseline design, fixed data for the display called frame formats is stored on the mass memory unit. Read/write memory functions are included in the display electronic unit (DEU) which permits complete format flexibility. New formats are sent to the local electronics unit via computer from the mass memory unit. All formatting, including vectoring, pattern rotation, and scaling, is accomplished within the DEU. There is, however, software associated with the selecting and reading of these frame formats from the secondary storage into the computer and then retransmitting them over the data bus to the display electronics units. Once the local memory has been initiated with the frame format the display system is updated and refreshed from the local memory. Only variable data recomputed by the computer in real time is then transmitted into the display units for updating purposes. This added software feature of transmitting the frame formats from the mass memory through the central computer over the data bus not only increases the traffic on the data bus but also adds to the computation load of the central computer. It would seem more desirable to have the fixed formats stored locally in the display unit via some type of mass memory tape system (as in the MDAC system). Formats would then be selected by command from the central computer.

4.5 Onboard Checkout Software

4.5.1 Overview

One of the objectives of this Phase B design task was to review the requirements for onboard checkout software. The emphasis of this part of the review task was not to define checkout software requirements since they are subsystem-dependent but to evaluate such requirements only to the level of determining structure and impact on the executive system design.

The Phase B requirements for autonomous operation and minimal "airplane type" ground test operations have resulted in both ground and inflight checkout. The ground checkout is applicable to the development phase and acceptance testing as well as pre- and post-flight checkout during the operational life of the Shuttle. Because of the need to determine checkout software executive requirements from Phase B study documents, the work in this area was focused on the review of inflight checkout software. Inflight checkout software, for purposes of this report, will be defined as checkout functions operated during the mission under the control of the flight executive. It includes functions such as subsystem monitoring, error detection and reconfiguration, status displays, and recording.

The inflight checkout concept and philosophy is defined as follows:

1) Inflight status and monitoring will provide system/subsystem status by utilizing redundancy voting, built-in test status monitoring, and critical measurement limit checks.

2) Upon detecting a failure of a functional path, time critical functions will be automatically switched to an alternate path. The capability for automatically or manually calling up a diagnostic/reasonableness test will be provided. (Redundancy voting, built-in test parameters, and measurement limit checks may indicate a sensor failure rather than a subsystem functional failure.)

3) Malfunctions detected by the status and monitoring tests will be displayed to the crew along with an indication of the results of switching and/or diagnostic evaluation performed by the computer. For non-critical functions, the display may only indicate the nature of the malfunction and the crew options available. All malfunctions, switching, and results of diagnostic routines will be recorded to aid ground maintenance operations.

A review of the MDAC baseline system approach to inflight checkout for the major avionics subsystems was made and a summary of the review presented in Appendix C. It was used primarily as a reference guide in determining the scope and structure of the inflight checkout software.

It is clear, based on this review and other information, that checkout approaches are not completely defined for the Phase B baseline system. The concepts employed for control and operation of redundant equipment are not fully defined. The methods of comparison and voting of redundant subsystem inputs as a means of error detection may be difficult to implement particularly for subsystems with a high repetition rate input such as the inertial subsystem.

Ideally, each subsystem would have an array of bit discretes which would, in combination, determine unequivocally the ability of that subsystem to support upcoming mission phases. However, the inertial subsystem (ISS) consisting of three or four inertial measurement units (IMU) is an example of a failure detection and isolation problem for which no easy technique exists. Accelerometer digital outputs are required to have uncertainties less than 50 to 100 micro-g's. Gyro drift stability is required to one or two thousandths of an earth rate. Given three candidate IMUs for voting, detection of differences to this level is marginal. If only two IMUs are left, crucial data on degradation of one IMU cannot be obtained during the critical mission phases where such degradation must be detected and isolated.

Subsequent to this review, it was determined that inflight checkout software functions will be directly dependent on the equipment characteristics and the inter-connection of said equipment to form the flight configuration. However, the general functions of inflight checkout, such as subsystems monitoring, status displays, functional path reconfiguration, and diagnostics, do not impose any significant requirements on the executive control system. The requirements on the flight executive for scheduling and dispatching of the CPU and other computer system resources are similar to the other operational requirements software for the executive in the computer. The subsystem monitoring functions are cyclic tasks executed at high priority as flight control but at a slower frequency. Diagnostics and fault isolation routines may be executed asynchronously as background computations. Caution and warning displays are maintained by procedures similar to other pilot displays and controls. Some of the general problems in these areas are discussed below.

4.5.2 Subsystem Monitoring

Subsystems are defined as the operational hardware and are catalogued functionally, i.e., flight control, inertial measurement unit, etc. They are built up from line replaceable units (LRU). Communication is via the data management system (DMS). The DMS inflight checkout software performs continuous or periodic evaluation of subsystem performance (all LRUs) and reconfiguration in event of detection and isolation of a failure.

Continuous evaluation is based on the basic system clock. For example, a subsystem may require evaluation every 20 ms. Periodic evaluation is based on subsystem and mission phase requirements. For example, a subsystem may be evaluated prior to a specific event to assure operational readiness. Some subsystems will be monitored continuously; others periodically.

4.5.2.1 Data Acquisition (Input). Subsystems must be sampled at some given rate over a given interval. These figures are determined by the individual subsystems, and these in turn by the mission phase. The input process is started by transmitting commands to the bus control unit (BCU). These commands are in the form of large tables which the BCU processes, independent of the central computer. These tables inform the BCU of the specific subsystem, type of data, address, and location in which to store the data. The bus requests data from the identified subsystems. The subsystems respond by transmitting data to the BCU and then to memory.

Since the bus commands are issued in a serial format, the time associated with the sampling of data from redundant subsystems must vary. Therefore, some form of time tagging must be available at the subsystem level. Otherwise, it is difficult to correlate readings from redundant subsystems and impossible to detect small differences between large numbers.

The bus is byte oriented (8 bits plus parity) so the BCU can transfer subsystem data to memory in several ways. Each byte can be justified, left or right, in the whole word (32 bits plus parity). Bytes can be packed four to a word from left to right or from right to left. Within the bytes the most significant bit can be ordered left to right or right to left. The parity bits should be removed by the hardware or the software. Status discretes may be "ored" into the data words so the software must mask them out. The data formatting affects the programming for each subsystem. One user may require unpacked data right justified and another require packed data left justified. Another may have

left-justified significance in one word and right-justified in another. Parts of one byte may have to be extracted and "ored" with another byte.

Certain subsystems transmit coded data. An example would be the keyboard associated with the CRT display. Some character strings must be converted from ASCII code to binary and then selectively processed for display in alpha inputs. Another example might be that system timing data exists as a four bit BCD code. The time codes must then be converted to binary and then processed. Each type of conversion requires separate software that consists of logical operations, shifting, merging, and masking.

If subsystems can operate in closed loop test modes, stimuli would have to be transmitted to test points, processed, and returned for comparison. Subsystems may also require calibration type tests which can be quite complicated.

Each type of conversion and test requires separate software and logical operations to format the data for each subsystem.

4.5.3 Fault Detection

Before the redundant units are enabled on-line, the onboard software must be aware of failures in the present on-line units. There are several techniques that can be used to detect errors. The following describes some of these techniques.

4.5.3.1 Status Discrete Checking. Subsystem hardware contains built in test equipment (BITE) which provides a signal describing the condition of the subsystem. The signal is the result of much internal test logic and is either high or low. The high condition could signify either good or bad. The low condition also could signify either good or bad. The selection is usually subsystem dependent. The program must know the good/bad states for all the status test points for all the LRUs. Naturally, uniform signals from all the LRUs are easier to process. The program logic could then establish the rule that all status test points in the zero or low state signify a good condition and in the one or high state signify a bad condition. The alternative is to construct status discrete tables which are initialized with the proper status values and then exclusively "ored" thereafter to check for changes. A change then means the LRU is in a bad condition.

Subsystem status discretes could be "ored" with or attached to the operational data and save I/O traffic on the bus. The alternative is to request the subsystem status via separate bus commands. If the status discrete is in the data it must be accessed, checked and masked off. If the discrete arrives via a separate command, the bit is simply checked. Current data bits could be associated with subsystems, set by the subsystem,and reset by the read command. If the bit is not set, the data is not current or not being generated.

4.5.3.2 Limit Checking. Subsystem test can be divided into two categories, critical and non-critical. Critical test limits can be further divided into warning and danger categories. Predetermined low and high values are established for both warning and danger limits. Non-critical test points have predetermined low and high values. The limits types may be fixed, scheduled or historical. The fixed limit remains unchanged unless changed by the crew via a keyboard request. A method related to the fixed limit is to check a test point for change from a previous value. Test points that are staircase voltages changing with specific events are good to evaluate with this technique.

The scheduled limit is varied as a function of mission conditions and phase. For example, the primary bus voltage might be 28 ± 1 volts at no load, but 26 ± 1 volts at full load. The historical limit is based on the previous behavior of a parameter and is used where a large variation is common among LRUs, but where any one LRU is expected to remain stable. Combinations of checks can be used on various test points (scheduled and historical). An example might be fuel flow rate, which could vary between engines at a given thrust but also varies with thrust for a given engine. The limit checking algorithm must keep track of all the various limits, the type of limit check,and all the limit checked test points. It also has problems as test points approach limits, exceed limits and return back within limits. For example, assume a test point gives a value of 6. The limits are set to a low limit of 1 and a high limit of 7. In this case the software does nothing. If the next value is 7 it has reached the high limit, but still nothing happens. The next value is 8 and the high limit is exceeded which causes an executive request for a CRT display of a limit excession. The LRU might be switched off-line and a redundant LRU enabled. If the test point was critical and the warning limits had been exceeded,it might continue to be limit checked. As long as the value remains at 8 nothing further happens. If the value drops to 7, a notification may or may not be required. In fact, it may not be desirable because small random changes in data at the limit value may cause repeated requests to the executive for CRT display of limit excessions. When the value drops below the high limit (7) an executive request should be made to display the fact that the value is proper.

4.5.3.3 Comparison with Redundant Units. Data acquired from redundant units can be compared or processed by software-implemented state adaptive voting. This algorithm compares bit for bit the outputs of up to four subsystems. If the values compare, there is no problem. If they do not compare, the software must resolve the discrepancy. If three compare and one does not, then an executive request is made to switch the failed unit off-line. The crew is always informed via CRT display of reconfiguration. If two compare and two do not compare, a simultaneous double failure occurred. More information must be acquired before the algorithm knows what to do. The crew may be informed via an executive request for a CRT display message. The crew then may decide what to do. Reasonableness calculations may be performed on the data to estimate which is the best pair. Other alternatives may be used, but the fact remains the problem is difficult to solve. If after one unit has been switched off-line and two new subsystem values compare and the third does not, then another executive request is made to switch the failed unit off-line. Later on, if the two remaining subsystem values do not compare then the algorithm must have more information before it knows which unit to switch off-line. The use of software voting for error detection and recovery in the Shuttle is a complex subject requiring further analysis of each subsystem to determine the validity of its application. This is beyond the scope of this study.

4.5.3.4 Reasonableness Calculations. Subsystem measurements can be used in conjunction with a predetermined algorithm to establish whether or not the subsystem is failing. Actually, the equipment may not be failed, but simply out of specification, and the reasonableness calculation may keep track of this problem. Another program could then be scheduled to further analyze the equipment at a later time. A very detailed subsystem analysis is required to generate the proper algorithm. Associated with these algorithms are common calculations such as formatting the measurements, converting them to engineering units, and checking on the accuracy and resolution of the measurements. Allowances must be made for data input timing differences unless time tags are provided. Since the algorithms are usually complicated and time consuming, the subsystem should be able to tolerate the time laps. Subsystem software that uses this technique would be low priority and executed at extended time intervals. Many subsystems cannot use reasonableness calculations for fault detection.

4.5.4 Reconfiguration Management

The fault detection software triggers the reconfiguration software. The reconfiguration software depends largely on the design of the DMS. In the MDAC Phase B baseline system the software modifies the I/O tables. The modification consists of changing all references to the failed subsystem and replaces them with references to the alternate subsystem.

4.5.5 Display and Controls

The onboard checkout software must keep the crew current as to the status and configuration of the Space Shuttle. A display of vehicle status would probably occur periodically or when the status changed. The crew could manually make vehicle status requests through the keyboard. Error displays for detected faults and reconfigurations must be provided. Error summaries may be provided to give the crew information about errors and error rates. Displays may occur that require crew action concerning critical subsystem reconfigurations. The logic may transmit, via telemetry, to mission control or tracking stations, all of the above mentioned display data. The software for this task must keep vehicle process tables, generate complicated CRT displays, process keyboard requests, and format data for downlink transmission. This processing involves conversions, character handing, bit manipulation and executive requests.

4.5.6 Flight Recording (Electromechanical Mass Storage [EMS])

Bus traffic during critical mission phases should be recorded for historical and maintenance purposes. If failures occur, data concerning the failure should be saved. Subsystem data can be recorded for postflight analysis. The volume of this data can be reduced with data compression techniques. Basically two methods are used: one where the criterion is a limit and the other where the criterion is a tolerance. The limit method compares the value with an upper and lower limit. If the sample is between the limits, it is discarded. If the sample exceeds the upper limit or falls below the lower limit, it is passed to the EMS. In the tolerance method, the rule states that if a new value varies by more than a predetermined amount or tolerance, the value is discarded. The tolerance band is shifted each time an important sample occurs. Each technique is useful for various types of subsystems. The data may be written on the EMS until it is full at which time it is written over by new data. This continues throughout the mission. An alternative is to store data at a given duty cycle.

Chapter 5

Higher Order Programming Languages

5.1 Introduction

The use of a higher order programming language (HOL) is currently under consideration for the development of flight software for the Space Shuttle. Several contractors are recommending a HOL over the more typical machine language approach because of the expected benefits of lowered software production costs, and improved management control during long term maintenance, which are traditional problems associated with any large aerospace software effort. The principal criticisms of the HOL approach that still remain based upon the inefficiencies in code generation with its increased memory requirements, the increased execution time introduced by the HOL compiler, and the lack of experience in utilizing this approach in similar aerospace applications. Although considerable interest has been demonstrated by the Air Force and other governmental agencies in supporting the design and development of higher order languages for programming aerospace computers, there has been, to date, no wide spread application of them in actual practice.

In the opinion of Intermetrics, a general purpose procedure oriented higher order programming language should be used in the development of flight software for the Space Shuttle. It will be a significant step toward a more orderly and controlled software production effort, toward a useful analytical tool for the designer, and toward a convenient straightforward technique for the programmer. Furthermore, it will be an essential ingredient in the effective production of highly reliable flight software, used extensively as part of the top down structuring process described in Chapter 2.

The areospace software industry as well as other governmental agencies are devoting a great deal of attention to the development of common higher order programming languages for use in such applications. In the 1973 to 1980 time frame of the Space Shuttle, programming languages will most likely become commonplace for use for aerospace computers of that generation just as they are with the large third generation ground based computer systems of today. Consequently, they should be included in the planning of a major space project of the 70's such as the Shuttle.

Intermetrics recognizes that a HOL approach may not be applicable or cost effectively applied to all aerospace computer systems, particularly small dedicated systems. However, the size and complexity of the Space Shuttle software posed in Phase B design appears to be of sufficient magnitude to effectively apply the use of a HOL

The objective of this task, as part of the overall study, was to determine the role that higher level compiler languages should have in programming the flight computer on the Space Shuttle. This chapter discusses those features of the language compiler which will aid in structuring and verifying software. Those areas traditionally difficult to code in a HOL, such as system programming,are discussed, as well as the role and interaction of other special languages, e.g., the crew language and checkout language.

5.2 "Languages" on the Space Shuttle

A distinction must be made between the classes of "languages" used on the Shuttle. For purposes of this report, only those languages utilized within the onboard data management computer system are considered. There are, of course, others which will be used in conjunction with other facilities involved with the Space Shuttle; e.g., those for test and ground checkout operations, simulation facilities, and other computer operations. An aim of this study was to distinguish between those languages used to control and operate the computer system onboard the vehicle, and those used to develop the software for the onboard computer system. Both are referred to as "languages" but will be distinguished as the crew languages and the software development language.

5.2.1 Role of the Crew Language

Pilots or other crew members will require a language to communicate with the computer system. They must be able to insert information, control the flow of processing, and receive information from the computer. This language will be

referred to as the crew language (CL). The CL will depend to a great extent on the capabilities of the display and control software system. A CRT type display system with an alphanumeric keyboard input is most likely for the Shuttle avionics system. The syntactic structure of the CL can range from simple numeric function control, as was used in Apollo, to English language statement commands entered through the alpha keyboard. Alphanumeric and graphical outputs will be used for communication from the computer to the crew.

The Apollo Guidance Computer display and control system transmitted commands and requests with a limited vocabulary of 99 nouns and 99 verbs. To command the computer the astronaut depressed the verb (operator) key followed by two decimal digits, and then the noun (operand) key also followed by two decimal digits. Then when the function key was depressed, the computer began to take action on the request. For example, verb 16 noun 20 meant display and monitor spacecraft attitude. Verb 16 meant "display and monitor" (continuously update), and noun 20 identified what to display; in this case, spacecraft attitude. Moreover, major mission programs were selected by Verb 37 with a program number identified by the noun.

This type of crew language has a disadvantage in that the operator must learn the coded list of nouns and verbs and the operational procedures associated with using them. However, once learned, it is very efficient. A crew language similar to the Apollo type, has been recommended within the MDAC Phase B baseline system.

The use of English language commands for a CL entered through a keyboard could be employed within the Shuttle. This type of language would consist of a finite set of keywords and elements defined with syntactic properties which would be entered by the crew. They would be decoded and translated by display and control software in the flight computer. For example, an on-line control language is defined as part of the breadboard fault tolerant data management system at NASA, Houston. It is used in conjunction with a checkout and data management language and allows the systems operator to exercise manual control over the system while it is operating. It includes English language text entered through a keyboard which enables it to initiate, control and display information while the software is executing. Typical commands are DISPLAY, LOAD and CALL.

Other general purpose languages of this type have been designed for the control and operation of software: a) executive job control languages such as the OS 360 operator language and CRBE; b) information retrieval languages such as ADAM and

AESOP; and c) test editing languages such as DATATEXT. Although these languages have been tailored for specific needs they contain some basic features needed in a CL such as the ability to retrieve and manipulate data and displays. These languages are of course more flexible, but they are slower to use, and rapid crew interaction with the computer during critical flight phases is needed.

5.2.2 Crew Language Requirements

A summary of requirements for a command language for the Space Shuttle is presented below. It is not meant to be exhaustive. The ultimate structure capabilities in the on-line command language will be a significant factor in the design of the total system and is only presented here to indicate the type of capabilities that are expected.

For purposes of the Space Shuttle avionics system, pilot commands should be entered from a display and control device, and then decoded and executed on-line. The language should not be compiled by the computer system but rather interpreted as on-line commands before the appropriate action is taken. When the English language statements or numeric coded functions are used, the language should provide the following functional capabilities. It must provide the crew with capability to:

a) select and control software functions for all phases of the mission;

b) control and configure avionics equipment;

c) request display of pertinent mission and trajectory information;

d) enter data pertinent to the mission programs;

e) control system priorities and options;

f) initiate and control checkout of subsystems.

5.2.3 Role of the Software Development Language

As previously stated, it is recommended that a general purpose, procedure oriented,higher order programming language be used in developing the flight computer software. The role of this language will be primarily for the preparation of code for all software in the flight computer. It is also expected that the language can be used for developing other related non-flight software, particularly in the areas of

mission planning and design analysis. This fact will facilitate standardization and communication among organizations working on the project.

Associated with the HOL will be a compiler with a machine independent syntax analyzer and machine dependent code generators for several computers, including the flight computer, development computer and others as applicable. The requirements from such a programming language have been derived and are documented in Reference 1. The language should be capable of supporting the programming of all Shuttle software applications: navigation, guidance, control, data management, onboard checkout and systems monitoring, communications, displays and controls.

5.3 Justification for Using a Higher Order Programming Language

In the past, manned space flight computers have been special purpose machines performing tasks, principally for guidance and control. The computer was provided with a restricted instruction set, small working memories, no secondary storage capability, and established interfaces to a limited number of output devices. For the most part, programming was accomplished in basic machine language.

The architecture of aerospace computers is now maturing to a close functional similarity to ground based computers. General registers, modular word lengths, and larger memories are already in evidence. Years of initial programming and making programming changes are becoming more important as these computers assume multipurpose use. The use of higher order programming languages which had practically no utilization in the aerospace community in the past, can now be reasonably considered. The lowering of costs associated with memory and hardware in the aerospace computers has changed tradeoff factors. In addition, the increased computational tasks required in the manned space environment have required use of larger, more general purpose computer systems and corresponding software to support them.

Flight computer software developments will certainly continue to suffer schedule pressures. In spite of careful planning, the software effort will often be disrupted by additional requirements to perform functions that were inadequately specified at the outset.

Programming languages have been effectively used in large scale ground based military systems. There are a number of standard arguments in favor of using a higher order language approach.

a) Ease of communication with the program

   1) The program becomes self-documenting, and therefore reduces the cost of and need for separate documentation at different levels of management (e.g., mission definition, analysis, program specification).

   2) In any large project, the problems of maintainability are aggravated by the inevitable turnover of personnel. Not only must different people be able to maintain the program, but they must also be able to easily modify, add, and redesign sections of the software.

b) The HOL is chosen because it is oriented to the problem being solved and uses languages more natural to the programmer. The concise formulation of the problem is therefore enabled. This leads to:

   1) fewer errors due to conceptual difficulties and different ways of stating a problem;

   2) shortened program design and development time.

c) The programmers need be less concerned with the following traditional machine features and problems:

   1) scaling and precision problems,

   2) base register allocations,

   3) general register considerations,

   4) initialization problems, particularly in loops,

   5) data protection.

d) The HOL allows program transferability from one machine to another. It eases debugging and reduces checkout problems due to problem oriented modularity and separation from hardware.

e) Carey and Sturm [2] present some interesting facts concerning the costs of existing space software and the projected cost savings of a compiler for aerospace programming. In particular they are concerned with the compiler. The following information is extracted from the above reference to indicate the software cost for aerospace missions.

   1) The cost of software for manned space missions is two to four times the hardware cost.

2) The Apollo Saturn V's Instrument Unit software was produced at a rate of 2.5 instructions per man-day.

3) As much as 1-2 months was needed to make a 500-1000 instruction change in the Titan III computer.

4) Software checkout is very expensive and not perfect. A single error in a 2000 instruction space program might require 50-100 validation runs on a simulated ground-based machine. Extrapolation to a 25,000 instruction program indicates 1000 to 1200 runs.

5) Typically 100 instructions in new unvalidated machine code written by a senior programmer may contain 3-8 errors. Carey and Sturm estimate up to 70% of these errors can be avoided by the use of a compiler.

6) By hand, machine code typically is produced at a rate of 270-350 instructions per man-month. With a compiler, 500-540 instructions per man-month are possible.

7) Writing a JOVIAL compiler for an IBM 4 Pi computer would cost between $300,000 and $500,000.

5.3.1 Higher Order Programming Language Experience

In the past several years there has been an effort to develop higher order procedure oriented programming languages for use in spaceborne software development efforts. Among those specifically aimed at spaceborne programming are SPL (Space Programming Language) developed by the Air Force under the sponsorship of the Space and Missile Systems Organization (SAMSO); CLASP (Computer Language for Aeronautics and Space Programming) developed under contract to NASA Electronics Research Center, Cambridge, and the HAL language developed under contract to NASA MSC, Houston.

Other military agencies have similar efforts to develop such programming languages. The Army has funded a survey to determine the most appropriate procedure oriented language for its TAC-FIRE system and selected a subset of PL/1 designated as TACPOL for the job. The Navy utilizes a programming language termed CMS/2 for the development of software for shipboard and airborne applications. In addition, the Navy is pursuing development of an advanced programming language based on CMS/2 for the advanced avionics digital computer system. This language, designated CMS/3, will be a problem oriented language which will express avionic missions and requirements in terms which are pertinent to a commanding officer.

CLASP and SPL MK2 are primarily directed at small fixed point aerospace computers. Heavy emphasis is placed on code optimization, scaling operations, and limited data manipulation. SPL MK4 and HAL encompass more general purpose features applicable to the wide variety of aerospace programming tasks. The characteristic features of eight programming languages have been tabulated and presented in Volume III of this final report. These languages include PL/1, HAL, SPL, CLASP, JOVIAL, FORTRAN, ALGOL, and MAC.

5.4 Single Compiler Approach

It is the thesis of this section that a single compiler be used for generating code for the flight computer. The basic concept is that to assist in the approach to verification described in Chapter 2, all code generated for the flight computer should be subjected to standardized automatic checking within the compiler. The system specification, design, documentation, and verification are all built around the unified idea: the HOL. Furthermore, the Phase B centralized approach with a single computer having integrated software does not readily lend itself to multiple compilers generating code without requiring linking of code.

It is reasonable to assume that the statements provided within applicable programming languages do provide most of the capabilities necessary for the Shuttle application. If however, a separate special purpose language is necessary, the proposed solution is to express source language statements in the general purpose higher order language. For example, a checkout language becomes a special application which is "grafted" onto the general language at a higher level. It appears as a collection of procedures and subroutines to the compiler.

This approach however, does not necessarily bar the use of other languages. Rather, it forces others to link at either a high level, by producing outputs which are the source languages for general purpose programming languages, or at a low level, by accepting the standardized operating procedures and conventions established for the general purpose programming language. Moreover, it recognizes that there may be a need for programs to be prepared using statements tailored to a specific application. At a high level such applications are subsystem checkout or hardware interfacing; at a low level, systems programming. However, each set of statements is directed into the single compiler system to facilitate standardization and commonality of checks which are performed on the

software during compilation. This standardization, not unlike that experienced by other industries, will help to produce a higher quality, more reliable software product.

Other options are also available to extend the general purpose programming language to meet these needs. For example, through macros the language can be extended to incorporate special features for certain problem applications.

5.4.1 Systems Programming

Generally there is a small section of the coding which is difficult to accomplish in the higher order language. This involves machine dependent coding, such as I/O, address constants, machine registers. Usually, the basic machine language is used for these functions, but more recently system implementation languages have come into usage. The justification for the special treatment is

a) the need for efficiency,

b) the need to get at special registers, I/O channels, and absolute memory locations.

While the efficiency question is often nothing but a hollow fear, there is no doubt that at some point the coding must come to grips with the actual machine that it will run on. However, the number of places in the Shuttle program where an I/O channel needs to be directly addressed is certainly minimal. I/O requests should generally be funnelled through well-defined localized areas in controlled subroutines. In any case, the need to do system programming and machine dependent operations is recognized.

On the other hand, the need for a system language could be minimized or totally eliminated if the computer were designed to go with the language and to execute its constructs directly and efficiently. It is then unnecessary to operate in a "lower level" language since there are no machine dependent features outside the scope of the language. Additionally, all application programs written in the higher language are executed far more efficiently both in terms of the speed and especially the core size they require. Burroughs has been structuring its computers to higher order languages for many years. When a machine is constructed in this fashion,it is easy to efficiently accomplish system programming tasks. Burroughs writes its operating system (ESP), its scheduler, and all its compilers

in extended ALGOL, the language its computer is designed around. In fact, the computer does not have an assembly language. Since the computer is designed around a higher order language, there are no addressable special registers to be dealt with by the programmers. There are special registers, of course, but they are automatically updated by the hardware using higher order language instructions. In addition, the computer is stack oriented, which makes it easier for a higher order language compiler to generate efficiently executed code for it.

5.4.1.1 Approach to Systems Programming. If a currently off-the-shelf computer is selected for the initial Shuttle application, then some degree of machine dependent coding will be required. There are two ways that this might be accomplished. The first approach is to extend the scope of the higher order language to include more low level features even though they might be machine specific. However, the ultimate in direct, hands-on, programmer control is the capability to switch from compiler code into direct or in-line machine language. There are several drawbacks to this approach.

1) This kind of capability jeopardizes program integrity. Once address constants, pointers, and register manipulations are available to the programmer, the possibilities for creating errors is significant. The entire structure that was so carefully contrived within the compiler to ensure program standardization and reliability can easily be circumvented. The introduction of such hazardous programming practices can hardly enhance program reliability.

2) Readability and understandability goals can be jeopardized when obscure machine dependent code appears with the listing. In-line basic assembly language code is particularly unfathomable and obfuscates the meaning of entire sections. These are fundamental reasons for using higher order languages.

3) Neither of the above two drawbacks would be of so much concern if their use could be confined to areas where it was essential. However, even if sensible groundrules for their use and control were established, it is a virtual certainty that nearly every programmer will advance persuasive arguments as to why his task is special and needs to use machine language coding to produce highly tuned efficient code.

Another approach is to keep all low level language capability, such as options for direct machine code, out of the general purpose language. When the need arises for a task or procedure to be programmed that cannot be accomplished in

the regular language, it is assigned to a special implementation group that programs it in another language, usually the assembly language for the specific flight comptuer. These experts tailor the code so that it is compatible with the higher order language environment that exists in the running computer, and conforms to the accepted standards and conventions, while meeting its functional specifications. Thus, the usage of this powerful but hazardous capability is isolated and controlled. Applications programmers must either accomplish coding in the higher order language, or else it is developed by a special group after interfaces and specifications have been negotiated and defined. This seems superficially to be attractive but has two drawbacks, besides the obvious one of dependence on "experts".

1) It isolates the low level activity to machine language subroutines which are not readily visible or easily understood even when located.

2) It is still quite possible for the programmer to engage in a great deal of "trickery". He can, for example, call an assembly language subroutine that returns a variable purported to be an integer but which is actually a memory address value computed in the subroutine. It is then arithmetically manipulated and used as an index in fetching other data. The achieved effect is a program that superficially accomplishes one thing, but when examined closely, is doing something entirely different. This sort of "trickery" is commonplace in Fortran usage of assembly language coding.

The proposed solution is basically to define a selected subset of the programming language with added features to improve its deficiencies. The proposal is that there be established a special language to accomplish low level and machine dependent tasks. But rather than use the completely separate assembly language, it is proposed that this low level language be incorporated and integrated into the main language compiler as a restricted subset of the language. That is, those given access rights to the "lower level" language can use the special statements and data types, and also freely intermix these with the higher level language statements. All are compiled together so that standard interfacing and data type checking is performed by the compiler. This effectively prohibits the "trickery" of (2) above. In addition, it is possible to intermingle both types of language statement when it is natural to do so. This removes the restriction of the forced and sometimes artificial dichotomy objected to in (1).

This approach should yield a program listing that is more readable and understandable even when computer specific. Applications programmers are in general, prohibited by the compiler from using these low level,relatively unsafe statements. Their

use is granted to a select few who have the authority of the project manager. For the purpose of ease in use, it is recommended that these lower level language routines be available not only as callable procedures and subroutines but also as in-line parameterized macros, or the equivalent. This provides a convenient method for using commonly required low level functions in a carefully controlled manner.

The intent of this somewhat cumbersome and laborious process should be "made perfectly clear". It is not the intent to put obstacles in the paths of applications programmers or to thwart their efforts to get the job done. It is proposed only as an additional technique to assist in the production and maintenance of quality flight software of high integrity and high reliability. This goal is accomplished by insisting on conformance to a highly structured and controlled environment. These constraints are not meant to hamper the programming effort but to place sensible limitations and bounds so that the overall result is of uniform high quality.

5.5 Advantages of the HOL and Compiler to Software Modularity

A factor in recommending a HOL and compiler for Shuttle software development is its direct application to the "top down" structuring processes discussed in Chapter 2. The benefits derived from modularizing the static software structure and the automatic checking features offered by the compiler will be a significant contribution to high quality software. This section discusses some of the advantages which result from using the HOL and compiler.

5.5.1 Apollo Experience

In a sense, the primary Apollo computational facility was concentrated in a "centralized data management system" - the Apollo Guidance Computer (AGC). This single computer was responsible for guidance (i.e., steering), automatic control, navigation, I/O processing (e.g., radar, IMU, optics, engines, keyboard, etc.), hardware compensation (e.g., for gyro and accelerometer inaccuracies), and a set of miscellaneous tasks including self-check, system test (onboard and pre-flight), crew communications, status monitoring, and up- and down-link telemetry. The Shuttle data management system will have to include all these functions while expanding the self-check, test, and system monitoring capabilities. It must also have the logic necessary to monitor and control the onboard environmental system and to reconfigure any or all of the subsystems based on the FO-FO-FS criterion.

5.5.2 Software Modularity

The Shuttle data management tasks promise to be more extensive and complex than that of Apollo. In addition, the reconfiguration logic associated with FO-FO-FS reliability presents software challenges not previously encountered. In order to accommodate all programs in a single computer, or substantial portions in distributed computers, it is imperative that systems be introduced which effectively isolate programs from one another except at controlled and visible interfaces. This isolation should prevent the unrestricted access of common data and the arbitrary transfer of control to any location in the instruction logic.

Software techniques now exist which allow many programs, designed to do various related and unrelated functions, to be written and incorporated in a single computer without conflict. The apprehension that the Shuttle DMS might be a bigger and more complicated Apollo-type effort with even more erasable conflicts and control interferences is relieved by the introduction of effective software modularity through language and compiler. The following features have been incorporated in the HAL compiler and provide significant capabilities toward handling a large, complex, cooperative programming effort.

5.5.2.1 Independent Compilation and the Compool. Figure 5-1 illustrates a suggested program organization. The individual numbered programs represent independently compilable units. Thus, for example, Program #1 might be rendezvous navigation, Program #2 - autopilots, Program #3 - environmental system monitoring. Independent compilation permits divergent groups to contribute to the whole and yet progress at varied paces with measures of local management control.

The communication between programs is provided through a common data pool (compool). The compool is a centrally defined and centrally maintained group of definitions. Variable names and location labels in the compoool are potentially known to all programs and, in fact, provide the only means of communication between programs.

The Shuttle's many tasks can be apportioned into programs which are managerially or functionally convenient. Information interfaces among programs then become visible at the compool level and can be monitored with respect to definition and usage by a central authority.

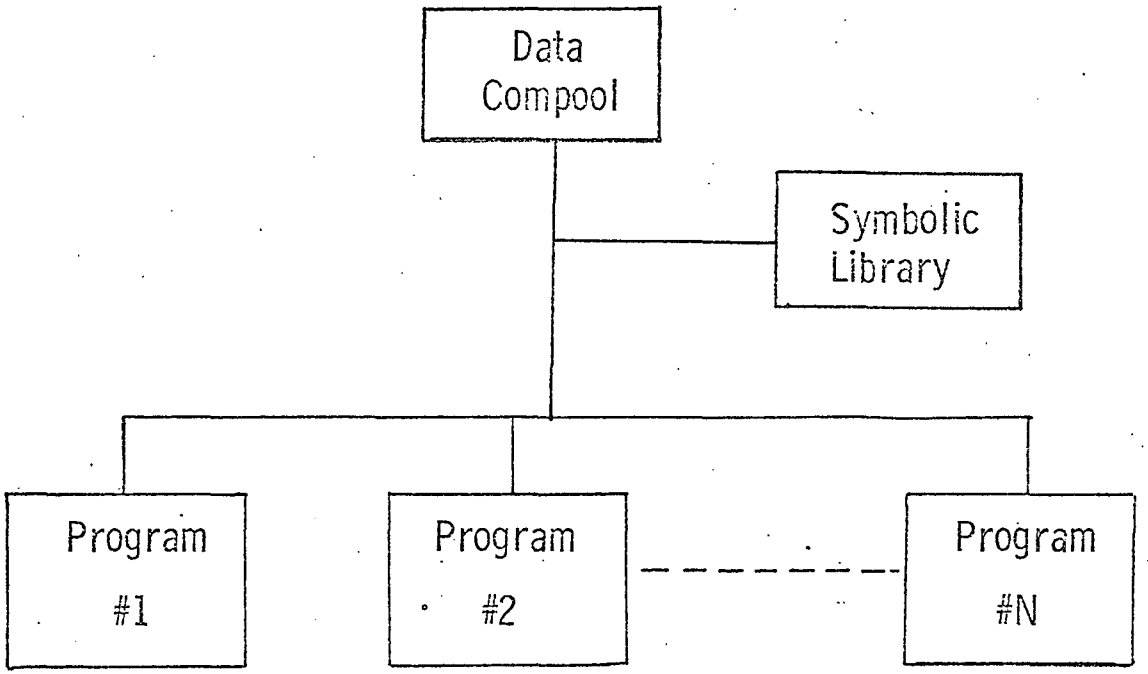


Figure 5-1 Program Organization

Note that, except for the necessity of communication among programs, the complete separation (or isolation) of programs within a single computer is commonplace today in a time-sharing environment. That is, to each programmer the machine appears to be a dedicated facility,and the probability of his conflicting with another user is remote.

5.5.2.2 Blocks Structure (Name Scope). Figure 5-2 defines the nested structure of name scope. For the purposes here, tasks, procedures, and functions may be considered as subroutines (or blocks). Thus, names defined in the compool are potentially known in every program. Names defined at the program-level are potentially known within all included (or nested) subroutines, and so on. The region in which a name is known because any particular name can be declared again in an inner block and then its scope would become all the nested blocks within this block. An example may help to illustrate these principles (see Figure 5-3).

Two desirable effects of the scope rules are:

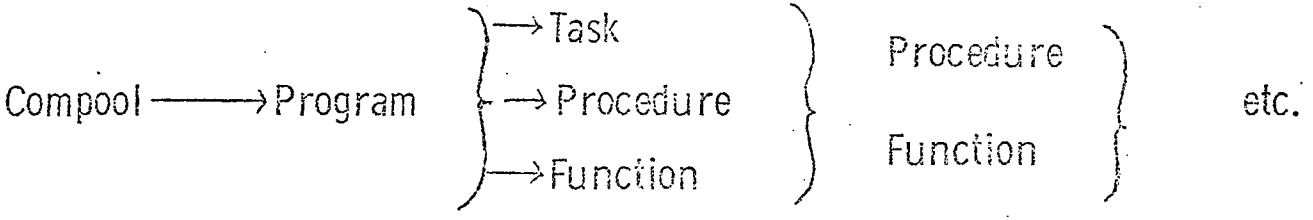
1) common data must be delcared at the highest level and only once. This contributes to more direct management control and better visibility.

2) Local variables may be defined within inner blocks and remain unaffected by outside definitions. For example, a programmer declaring X in procedure CHARLIE (Figure 5-3) need not fear that any other program will overwrite his quantity. That is, this particular X is not addressable from outside this block. In fact, the X in GRAB (Figure 5-3) must refer to different memory cells.

For the Shuttle application, a name scope or block-oriented language means that many programs and subsections of programs (i.e., subroutines) can "live" in the same computer, isolated, and unaware of each other. They are incapable of writing-over or otherwise interfering with variables or locations that are not mutually defined.

5.5.2.3 Control of Shared Data. The erasable memory conflict, along with restart and scaling problems, provided most of the Apollo software anomalies. To illustrate the problems, in a general way, that can arise because of sharing data, consider the examples shown in Figure 5-4.

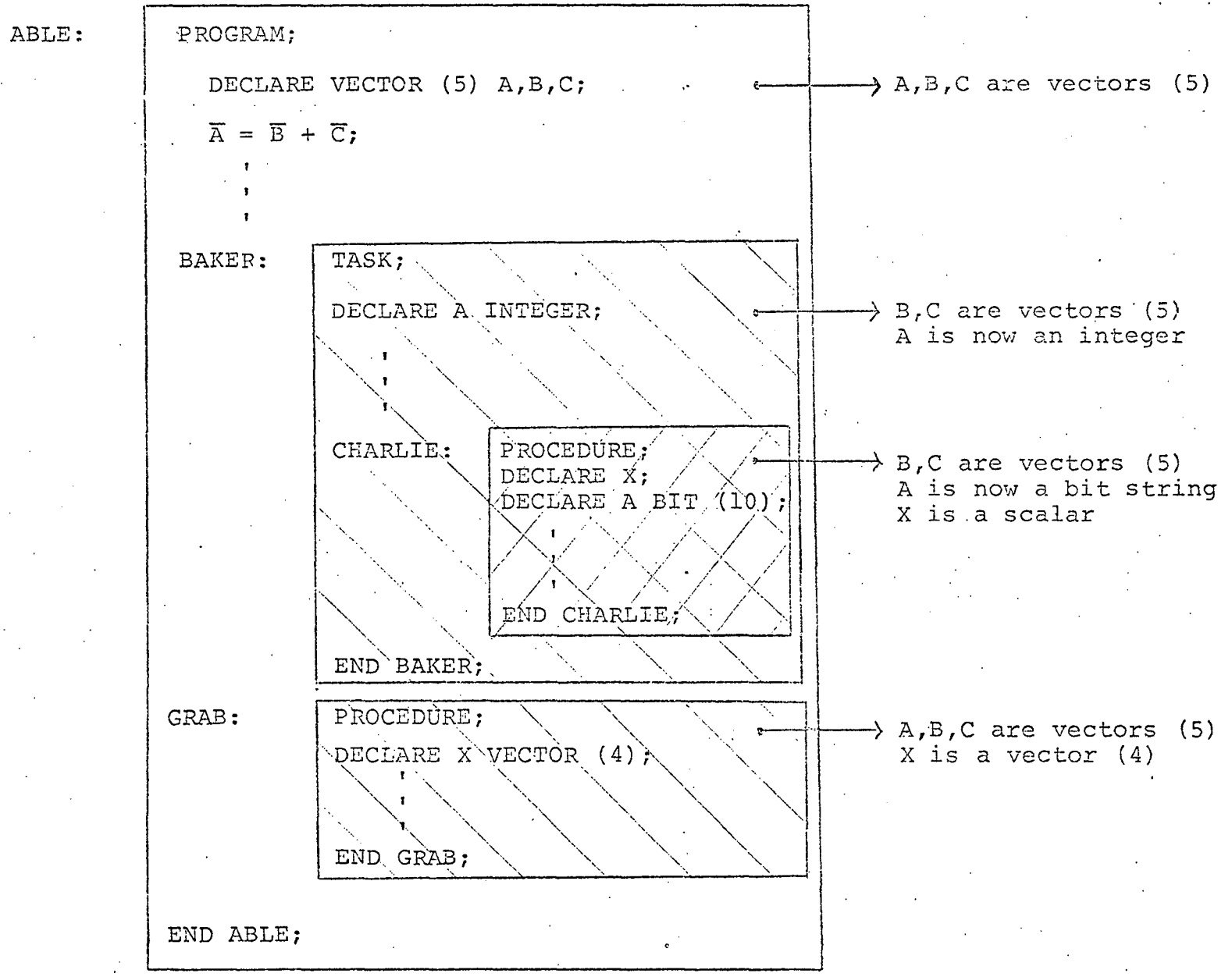
- Scope is the region in which a name is recognized.
- Scopes are defined from the outermost block toward the inner; i.e.,



- Names defined in an inner block are <u>never</u> recognized in an outer block. Inner blocks effectively isolate locally defined variables.

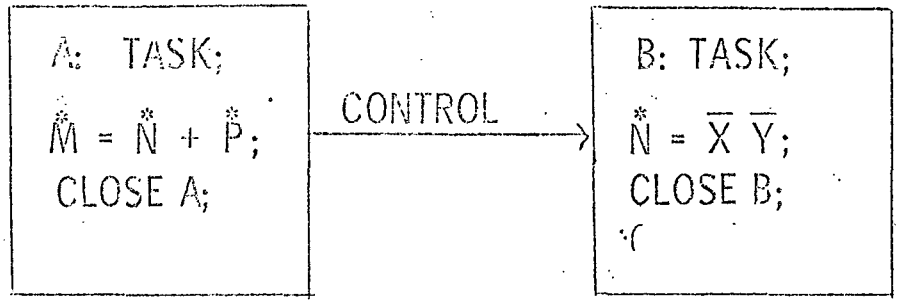
Figure 5-2 Scope of Names

Figure 5-3 Example of Name Scope

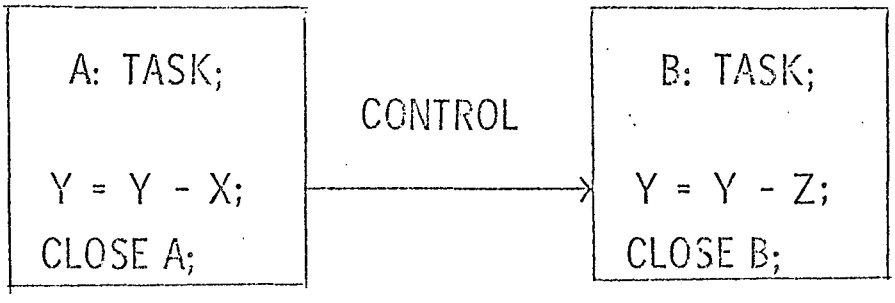


- PROBLEM IS THE CONFLICT OVER UTILIZATION OF COMMON DATA ELEMENTS BY EXECUTING TASKS.

EXAMPLE 1: READ AND WRITE CONFLICTS



EXAMPLE 2: UPDATE CONFLICTS



NOTES

1. B "INTERRUPTS" A IN BOTH CASES
2. #1 TASK A RESUMES USING OLD AND NEW VALUES FOR N*
3. #2 TASK A RESUMES "CLOBBERING" THE VALUE FOR Y SET BY TASK B

Figure 5-4 Background in Problems of Controlled Shared Data

In both examples TASK B interrupts TASK A during the execution of a statement. The interruption may be caused by a hardware or software interrupt or by a "job swap" based on priority. In either case, the interruption of TASK A causes a conflict in common data usage.

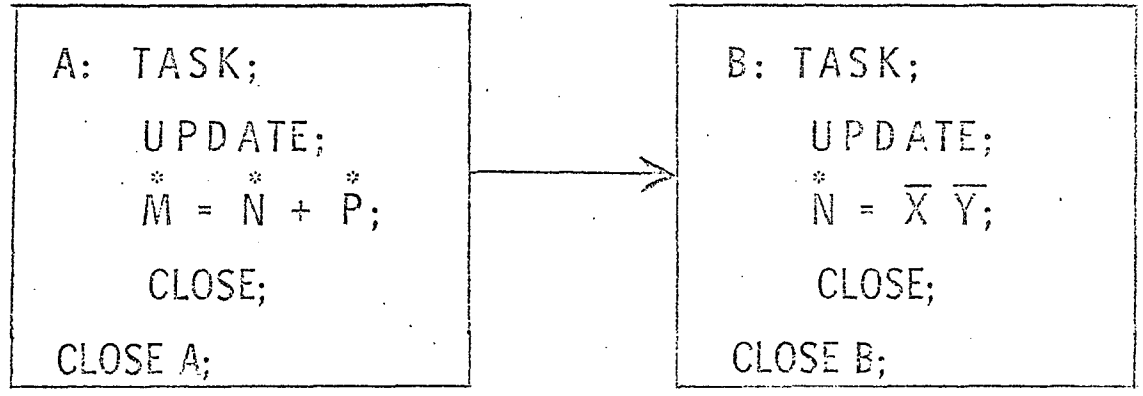
The approach taken, in HAL, towards solving these problems is to confine the read and write accesses of shared variables to identified update blocks. The compiler assigns a locking control variable to each shared variable. The value of "lock" is examined at run-time and only consistent (i.e., safe) accesses are permitted (see Figure 5-5). Volume II of this report presents a more detailed explanation of update blocks.

The use of an update block is not a simple solution to the data sharing problem and presumes a sophisticated compiler; and yet the goal is worth the effort. The problem of sharing common data in a real-time flight environment always exists. The Apollo "solution" was to attempt to arrange memory so that conflicts did not occur. This proved to be a time-consuming process, at best, requiring extensive verification with inconclusive results.

For the Shuttle, data sharing will be a necessity regardless of which avionics configuration is selected. A unified approach through a compiler, as outlined above, will permit safe operation in multiprogram and even multiprocessor environments.

5.5.2.4 Access Rights. The sharing of compool variables among several programs may be restricted and controlled by the issuance of access rights. These rights are attached to the data declarations within the compool. Each program is identified by number and permitted to access only those variables which have been declared with corresponding identification numbers. An illegal reference to a compool variable will prevent successful compilation of the program. For example, on the Shuttle access rights might be employed to allow only those programs comprising guidance and control to address compool variables associated with main and reaction control jet engine performance.

5.5.2.5 Automatic Checking. Besides being expressive and enforcing programmer conventions, additional major advantages of a compiler language are the ability to perform extensive checking at compile time and the opportunity to structure and modularize programs. Compile time checking can verify that subroutines are called with proper data; that dimensions (i.e., the units) of variables and constants are consistent; and that array variables (vectors, matrices, etc.) are not referenced out of range. In addition, the compiler can perform other static cross-checks on the intent of the programmer.



IF $\overset{*}{N}$ IS DESIGNATED AS A SHARED VARIABLE THEN:

1. UPDATE IN A CAUSES $\overset{*}{N}$ TO BE 'READ-LOCKED'
2. UPDATE IN B CAUSES $\overset{*}{N}$ TO BE 'WRITE-LOCKED' AND A COPY MADE OF ALL 'WRITE-LOCKED' VARIABLES
3. CLOSE IN B WILL NOT ALLOW UPDATE OF $\overset{*}{N}$ IF VARIABLE IS STILL 'READ-LOCKED' (TASK B WILL STALL)
4. POTENTIAL CONFLICT IS AVOIDED

Figure 5-5 Use of Update Block to Avoid Data Conflicts

5.5.3 Additional Advantages of the HOL Approach

5.5.3.1 Management. The technical management of the software for the Space Shuttle faces problems of visibility and control. Design changes, short production times, and pressing operational schedules would demand flexibility in software design and organization. Clearly, an overall management and control plan is required which will define the procedures for developing software design requirements, interface specifications, documentation requirements, testing requirements, change procedures and organizational responsibility. Presumably, a higher order language should provide features which support the software production environment in general. It would be self-documenting to a maximum extent, provide ease in program modification, and provide mechanisms for enforcement of management rules and programmer conventions.

5.5.3.2 An Improvement in Communications. In this context communications are meant to include requirements, specifications, descriptions, all forms of documentation, methods of configuration and change control, management visibility and technical exchanges (written and oral) that must occur among engineers, analysts and programmers. Traditionally, the engineer designs and expresses his algorithms using conventional mathematics, or perhaps Fortran-like statements, and the programmer translates these into his language, usually a basic assembly language appropriate to the particular computer. The programmer must then explain his efforts by using other media, e.g., detailed functional charts, user-guides, or other apparently helpful devices. Unfortunately, in many projects the coding language has isolated the programmers from everyone else associated with the effort. The programmer becomes too busy to learn the physics and objectives of the mission and is too busy to explain to others how the code works. He, therefore, is forced to assume an increasing share of the total responsibility. Small indispensable groups of experts direct and shape the code and become the overworked "authorities".

A properly designed higher order language could be a useful analytical tool for the designer and a convenient, straightforward technique for the programmer. The specific format of the language should promote the ability to read, write, and understand the language quickly and easily, and to document results in a clear and unambiguous manner.

5.5.3.3 Prevention of Errors by Readability of Code. A higher order language can be instrumental in preventing errors. The truth of this assertion can be seen simply by comparing the probability of error when using assembler versus compiler coding techniques. A programmer using a higher order compiler language can express his problem in a problem-oriented manner. For example:

$$\overline{W} = \overset{*}{M}\ \overline{V}$$

indicates that the product of the matrix M and the vector V be assigned to the vector W. The programmer does not have to express how he wants the machine to handle his statement; e.g., where the variables are in memory, what base or index registers to use, what basic machine instructions to employ, or how to set up, and call an assembly language subroutine. The single statement above will generate many assembly language instructions automatically. If these had to be hand-coded, the probability of programmer error would greatly increase. It also is an aid to visual inspection or "eyeballing" of the code for correctness as pointed out in Chapter 2. The HOL enforces standard code and discourages a "handcrafting" that invariably leads to subtle errors.

5.5.4 Summary

In considering methods of implementing Shuttle software, hardware and software techniques are available to insure program modularity. For a centralized avionics configuration, this means that effective isolation can be insured among programs performing different functions, and that the interferences and potential memory conflicts of Apollo need not occur. For a decentralized system, the enforced hardware separation of the several functional computers adds a measure of safety in that, assuredly, a program operating in one cannot cause a memory conflict with a program operating in another. However, even in this case, the computational load in a single computer, e.g., guidance and control, can be sizable (perhaps 40% of the total) and modular programming techniques and aids should be utilized. Once these techniques and aids have been provided, it makes little difference from a programming point of view, whether the total software is centralized or decentralized.

5.6 Checkout Languages

Several higher level languages have recently been developed for purposes of system checkout. Examples of these are GOAL and ATLAS. These languages have, however, been primarily directed at integrated ground checkout and subsystem test operations.

It is reasonable to assume that checkout software for the flight computer can be developed using the general purpose HOL. It can be operated and controlled interactively by the crew using the crew language as any other flight software. The crew language may require a special subset to accommodate all memory options and control functions required to perform checkout and maintenance. The capability to select diagnostic and subsystem checkout programs and to control their options must be provided.

The software environment of the Shuttle is one in which software will be assembled and loaded prior to the flight. No on-line compiling of software and program generation is assumed. Accordingly checkout software must be constructed to allow modes or crew options for accommodating the variety of fault isolation and diagnostic requests.

In the event that the general purpose HOL cannot be extended to satisfy the needs of this type of software, it is recommended that the single compiler approach be used as discussed in Section 5.4.

5.7 HOL Compiler Implementation

5.7.1 Compiler Problem

The chief complaint about higher order languages has been that HOL compilers are inefficient generators of machine language code, in terms of both quantity of code and in execution time. Secondary factors are 1) that compiler design is a very significant effort if it has to be considered in line with the operational software task, and 2) that the indirect and unclear relationship between a program written in the HOL and the resulting machine code impedes the correction of program errors discovered at the machine language level. The reason for the compiled code's stigma of inefficiency is that compiler systems have not evolved with the conservation

of machine resources as a primary design criterion, but have concentrated on isolating the programmer from having to worry about the machine characteristics. Since it is difficult, perhaps even impossible, to serve both the programmer and the machine interfaces equally well within the mechanism of a single translation, the tendency has always been to incur object code inefficiencies rather than decrease the programming effectiveness.

It should be noted however, that with the continual decrease in hardware costs, and corresponding increases in cost for software, the conservation of memory may no longer be the prime objection to a HOL and compiler. Certainly, if the software is sized with higher order language considerations initially and a secondary memory system is used for loading mission phase programs to lessen the impact of operating memory size as recommended in Chapter 4, the software cost savings of the HOL approach may well exceed the increased hardware costs.

The penalty of an increased memory capacity, however, will always be considered when the use of a HOL is contemplated. A competently written compiler can be almost as efficient as an average programmer. The MIT experience with PL/1 on MULTICS has demonstrated this. But compared to the highly efficient machine code customarily produced (at considerable cost) for military aerospace computers, a compiler may be less economical.

Since compiler efficiency is still an important consideration, the purpose of this section is to describe some possible approaches to improve compiler efficiency. Higher order language machines, interpreters, use of microprogramming, and high speed memories are all approaches that aid in achieving more efficient code generation if it is required.

5.7.2 Approaches to Efficient Code Generation

An approach that circumvents the drawbacks of compilers, is the construction of special higher order language machines that decode and execute the HOL operations directly within the logic of the hardware. Although a number of these has been reported in the literature [3,4,5,6] it is not a widely applied principle.

There is another approach that appears to solve a number of the previously identified problems and whose drawbacks show promise of being eventually diminished by current trends in computer hardware design. It involves the establishment of the program in a coded form intermediate between the HOL and

machine language. The translation from the HOL to the intermediate form is accomplished off-line in an operation that can be made much simpler, faster, and cheaper than the traditional compilation of machine code from the HOL. The translation of the intermediate form into machine operations is done at execution time in an "interpretive fashion". This concept appears to offer the following benefits.

1) For a given application the computer memory requirements can be made less by up to a factor of two compared with the direct translation compiler approach.

2) It allows the choice of HOL to be uncoupled to a great extent from the problem of satisfying the machine characteristics, and it is unaffected by consideration of machine to machine transferability.

3) The intermediate form of code provides a very convenient, visible "stepping stone" between the machine and the HOL, which would greatly assist the problems of debugging.

4) Current trends of computer design offer the possibilities of (a) higher performance using this approach than can be obtained by hand-crafted assembly language programming, and (b) a reduction in the amount of machine-dependent coding that is required whenever a new computer is being considered.

5.7.2.1 The Concept of an Intermediate Language. It is feasible to formulate a medium which lies intermediate between the problem and the machine, which enables a concise enough description of the problem's characteristics, and yet accommodates sufficiently to the limited word format and instruction repertoire of the computer. Such a medium would possess a high information content and would be storable in the computer's memory. The basic concept, however, is the translation of the operational program (expressed in a language highly appropriate to the problem it seeks to solve), into a compact intermediate form (or language) which, when stored in the computer memory, maximizes the density of the information.

For the condensed information of the intermediate language to perform any operation, its basic instructions must be decoded and executed by some mechanism within the computer. This process involves a number of logical operations which will consume a certain amount of time. For an individual instruction, it need not be changed (unless, of course, the instruction is modified).

5.7.2.2 Characteristics of Compact Form. The structure and notation of the compact form of the program must be defined in a formal code or language. A basic set of more elementary instructions can always be derived to mechanize all the basic HOL statements [3]. The proposed intermediate language (IML) will be based on this set of elementary instructions. The processing of the HOL into the IML becomes a more direct, less complicated, faster operation than compilation into machine code. This is attributable in part to the fact that a good deal of the decoding task is done at execution time, relieving the translator of some of the burden. Furthermore, since the translation is less difficult, it becomes natural to contemplate fairly sophisticated and universal HOLs like HAL, SPL, or PL/1 for the application programming.

5.7.3 Implementation Factors

An important constraint on the design of the IML is the method of decoding and execution by the machine. The more concise and compacted the language, the higher becomes the potential economy in memory. However, the full impact of its advantages will be realized when the current trends in microprogramming achieve operational status. The IML design must remain cognizant of this trend. Experience with and acceptance of the language today will then constitute a firm foundation which will provide continuity into future programming.

5.7.3.1 Software Interpreter. The majority of today's aerospace computers possess a fixed internal logic which defines their basic operating modes. The IML program would exist in memory in encoded form produced by the machine section of the HOL-to-IML translator. The decoding and execution of the individual instructions of the IML program must be performed by the standard instruction set of the computer under the direction of an interpreting routine written in the assembly language of the machine. Instruction by instruction software interpreters have been used in aerospace applications for the purpose of storage efficiency [7], but they are more usually employed in commercial applications where their ability to decode and execute individual statements can be used to advantage in on-line programming and debugging.

The usual complaint against a software interpreter, which is well earned, is that because it repetitively performs the redundant operations of decoding and dispatching for each statement, it is considerably slower than the object code of a compiler, which is analyzed and translated prior to execution.

However, the example of interpretive programming in the case of the Apollo Guidance Computer demonstrates that the penalty is quite acceptable. An equivalent instruction, for example a double precision add, was 20-30 times slower in the interpretive mode than in the machine language. Although the computer with its 12 μs cycle time was ten times slower than a typical small machine of today, interpretive routines were used to implement guidance and control loops with periods of less than 1 second. The use of the interpreter enabled 50% more interpretive programs to be accommodated in the memory than if a pure assembly language approach had been taken.

With the higher performance computers available today, it should be possible to do at least as well; and with a more sophisticated interpretive language than was used for Apollo, a much higher ratio of IML to assembly language programming should be achievable. With this level of performance less than half as much memory is needed to contain a HOL program translated into the interpretively executed IML than one in machine code generated by a regular 25% inefficient compiler. The cost savings come with all the advantages of comprehensive HOL programming.

5.7.3.2 Hardware Implementation and Use of Microprogramming. The use of special logic circuitry within a computer to assist the interpretation of a higher order problem-oriented language has been reported in the literature. Some of these attempts have mechanized subsets of Fortran directly with specially designed logical hardware [3,5]. Other more promising approaches have applied the concepts of microprogramming. A very relevant example is reported by Webster [4], in which a machine independent interpretive language is decoded by microprogramming on a modified IBM 360/30. The original programming is done in a higher order language, and a relatively short compiler generates an "intermediate text" or middle language for storage in the machine. The string language interpreter reduces storage, and the microprogramming feature allows special instructions which actually improve the run time over standard assembly language techniques.

It is true that microprogramming brings with it its own problems of language and design. However, a microprogram instruction is generally more powerful than a basic machine instruction. The microprogrammer is given greater scope to optimize the sequence of operations required to decode and execute an IML statement. Once it is set up, the microprogram storage resides in a read-only memory, which is generally capable of higher speeds than main read-write memory. We do

not suggest that the technique of microprogramming is without characteristic problems of its own, but for the short fixed logical sequences associated with decoding a set of IML instructions, it offers a higher efficiency than the software approach and is far more flexible than advanced logic. The application of microprogramming is discussed in Chapter 6.

5.7.3.3 An Interesting Example. Although the following description of an aerospace programming application is not an example of HOL usage, its significance lies in its conscious attempt to economize on memory requirements. Since this is the central objective of the concept described in this section, and because of the relationship of the techniques, the application will be briefly considered here.

The example in question is the software interpreter [7] used in the Apollo command and lunar module computers: the CMC and LGC. The computer is a 36,000 word 16-bit machine with a 12 microsecond memory cycle time. The requirements placed upon the onboard computer grew with the development of the total program. For the lunar landing mission, Apollo 11, each computer had less than a hundred or so unused memory registers. The coded interpreter implemented 127 double precision arithmetic, vector and matrix algebraic operations, and many trigonometric functions. Yet it took less than 1600 16-bit registers of computer memory. The command module program used approximately 16,000 interpretive instruction registers.

If this effort had been done in basic assembly language it may be presumed that instead of all in-line coding, a number of subroutines would have been written to conserve storage. Some 75% of the interpreter would have to remain as basic language subroutines, i.e., 1200 words. This represents a saving of 400 words. Of the 16,000 words about one-half are instructions and one-half are addresses. The assembly language approach would retain the addresses and would require, on the average, about two instructions for every one interpretive instruction. The net result is that without an interpreter the Apollo computer would have required approximately 8,000 additional words of memory to accomplish the job. This represents a saving of 33% over efficient assembly code.

References for Chapter 5

1. Requirements Analysis for a Manned Spacecraft Programming Language and Compiler, MSC 01845, April 1970, prepared by Intermetrics, Inc.

2. Carey, L. and Sturn, A.A., "Space Software: At the Crossroads", Space/Aeronautics, December 1968.

3. Kerner, H., and Gellman, L., "Memory Reduction Through Higher Level Language Hardware", AIAA Paper No. 69-693, Aerospace Computer Systems Conference, Los Angeles, California, September 8-10, 1969.

4. Weber, H., "A Microprogrammed Implementation of EULER on the IBM System 360 Model 30", Comm. ACM, Vol. 10, No. 9, September 1967, pp. 549-58.

5. Baskow, T.K., Sasson, A., and Kronfeld, A., "System Design of a FORTRAN Machine", IEEE Trans. Elec. Comp., Vol. EC-16, No. 4, August 1967, pp. 485-99.

6. Melbourne, A.H., and Pugmire, J.M., "A Small Computer for the Direct Processing of FORTRAN Statements", Computer Journal, Vol. 8, No. 1, April 1965, pp. 24-27.

7. Muntz, C., "Users Guide to the Block II AGC/LGC Interpreter", R-489, MIT Draper Laboratory, Cambridge, Mass., April 1965.

8. Bostrom, F.D., Higher Order Language Study for Avionics Programming, TR AFAL-TR-71-154, June 1971, IBM, Owego, New York.

9. Graham, R.M., "Use of High Level Languages for Systems Programming", MIT Project MAC Tech. Memo 13, September 1970.

10. Corbator, F.J., "Sensitive Issue in the Design of Multi-Use Systems", MIT Project MAC Memo M383, December 1968.

11. Corbato, F.J., "PL/1 As a Tool for System Programming", MIT Project MAC Memo M378, July 1968.

12. Hess and Martin, "TACPOL - A Tactical C&C Subset of PL/1", Datamation, Vol. 16, No. 4, April 1970, pp. 151-157.

13. Hopkins, Martin, "SABRE PL/1", Datamation, Vol. 14, No. 12, December 1968, pp. 35-48.

14. Preliminary Functional Design of the SPLM as Developed and Procured by the Air Force, Attachment 2, RFQ F04 701-71-Q-0145.

Chapter 6

Flight Computer Features

6.1 Introduction

The architectural features of the flight computer selected for the Shuttle will be of signficant influence on the software. Historically, inadequate computer hardware has increased the scope and cost of software; it inevitably must be implemented to accomplish mission requirements. Computer systems for airborne applications have been tailored for the requirements of a particular mission application. Typically, mission requirements are underestimated and result in inadequate computer speed, memory size, and instruction set capability.

More recently, there has been an increase in the role and functions performed by spaceborne computers, particularly in manned space applications as the Shuttle. As a result of and due to advances in hardware technology, manufacturers have introduced more general purpose aerospace computers which include features contained within most third generation conventional ground based computers, such as: general registers, more comprehensive instruction sets,and indirect addressing. However, several features desirable to software are not readily available, e.g., floating point, microprogramming, and hardware stacks. Although aerospace software emphasizes high reliability and is characterized by the significant costs for software verification (perhaps an order of magnitude greater than commercial software), few if any computer hardware features have been provided to assist in software production.

6.2 Scope and Objective

The primary objective of this task was to identify those computer hardware features which are desirable from a "software viewpoint". While it is recognized that these features have tradeoffs associated with hardware complexity and costs, as well as general availability within the aerospace computer market, this chapter identifies those architectural features considered desirable and effective in producing high quality software for the Space Shuttle.

Furthermore, it is not meant to imply that these features are "hard requirements", mandatory for the Shuttle flight computer. It is agreed the Space Shuttle software may be accomplished within the capabilities of an "off the shelf" aerospace computer. However, the impact of the costs on software over the life of the Shuttle can be significant. Accordingly, features are identified which, based on past experience, can help lower software costs. In addition, the presumed use of a higher order programming language and compiler in developing software has been an important aspect of identifying "desirable features".

The chapter first presents a review of the flight computer current generation, followed by a section discussing advanced features such as higher order language processing, stacks, microprogramming and run time diagnostics. An additional section on general computer features discusses addressing, subroutine linkage, floating point and short form instructions.

A method of using "benchmark" programs is discussed in a subsequent section as a criteria for measuring performance of candidate systems.

6.3 Background to Computer Features

6.3.1 Flight Computer Generation

Denning [1], has reported computing "generations with approximate dates of:

a) First - 1940-1950,

b) Second - 1950-1964,

c) Third - 1964-present, and

d) Late Third - 1968-present.

The principal properties of the generations are listed in Figure 6-1 which apply not only to hardware technology but to the total hardware/software system. By examination of Figure 6-1 it appears that the technology of current aerospace computers is predominantly third generation with the exception of floating point hardware, and that too, is beginning to make an appearance. In quest for absolute minimal cost, floating point hardware was always classified as a luxury and not a necessity. It shares that characteristic with the mini-computer market where floating point is just beginning to appear on the scene.

As discussed in Chapter 5, lack of higher order language capability for aerospace computers has always been a question of efficiency. However, current language developments seem to indicate that this too is about to change. The flight software environment however, has been reasonably sophisticated because of the real time aspects of the application. Multi-programming, data sharing and locking, task synchronization and error recovery have been commonplace. Support software for aerospace computers has also been extensive. High-fidelity, bit-by-bit computers and environmental simulators and high quality debugging and verification tools, such as traces, dumps, diagnostics, edits, rollbacks, and performance measure-ment programs have been used in the development of the aerospace software.

In summary, at the current state of the art of computer design, flight computers and minicomputers resemble each other markedly. Both tend to be short word length, limited instruction repertoire and restricted memory size machines. The reason for this in the case of the minicomputer is apparent. The manufacturers are attempting to produce extremely low cost items. However, on a cost scale, flight computers reside at the opposite end of the spectrum. They tend to be extremely expensive. The reason for the stinginess of the flight computer design and features has been based upon size, weight, power limitations, and stringent reliability requirements. At least, these have been the traditional reasons given. With the advent of LSI technology and the so-called computer-on-a-chip, it is anticipated that the capabilities of flight computers will expand enormously. Planned flight computer developments, e.g., the Navy AADC work, will produce airborne computers that rival the largest commercial computers in their computational power.

Figure 6-1 A Summary of Characteristics of the Generations of Computers

Characteristic	Generations			
	First	Second	Third	Late third
Electronics:				
Components	vacuum tubes	transistors	integrated circuits	same as third
Time/operation	0.1-1.0 msec	1-10 μsec	0.1-1.0 μsec	same as third
Main memory:				
Components	electrostatic tubes and delay lines	magnetic drum and magnetic core	magnetic core and other magnetic media	semiconductor registers (cache)
Time/access	1 msec	1-10 μsec	0.1-1.0 μsec	0.1 μsec
Auxiliary memory	paper tape, cards delay lines	magnetic tape, disks, drums, paper cards	same as second, plus extended core and mass core	same as third
Programming languages and capabilities	binary code and symbolic code	high level languages, sub-routines, recur-sion	same as second,plus data structures	same as third, plus extensible languages and concurrent pro-gramming
Ability of user to participate in debugging and running his program	yes (hands-on)	no	yes (interactive and conversational programs)	same as third
Hardware services and primitives	arithmetic units	floating-point arithmetic, inter-rupt facilities, microprogramming, special-purpose I/O equipment	same as second, plus: microprogramming and read-only storage, paging and relocation hardware, generalized interrupt systems, increased use of parallelism, instruction lookahead and pipelining, datatype control.	
Software and other services	none	subroutine libraries, batch monitors, special-purpose I/O facilities	same as second, plus: multiaccessing and multiprogramming, time-sharing and remote access, central file systems, automatic resource allocation, relocation and linking, one-level store and virtual memory, seg-mentation and paging, context editors, pro-gramming systems, sharing and protection	

INTERMETRICS INCORPORATED • 701 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

6.3.2 Aerospace Software Characteristics

The programming of the manned spacecraft computer generally parallels development of a commercial program. However, it differs in two important aspects.

a) It can never be checked out "in situ". For example, it was impossible to fly Apollo missions for the purpose of checking out Apollo software. When a mission came, the software had to be checked out and ready to go. This demanded that all programs be tested in a simulated environment; the success of the flight software depended, in no small measure, upon the fidelity of the simulation facility and models. The utter dependence upon simulation is the first distinguishing characteristic of space software.

b) The extreme emphasis on high reliability software, the astronomical cost, not to mention the national prestige that is on the line when a manned space mission is undertaken, demands the flight programs be as close to 100% perfect as possible. Traditionally this has entailed a lengthy, costly, and highly controlled verification process. This requirement for highest quality is the second distiguishing feature.

Accordingly, subsequent sections discuss those computer features which should aid in the production of this type of software.

6.4 Advanced Computer Features

First, it must be re-emphasized that this feature summary is seen from a software point of view. It is recognized that there are other viewpoints and considerations, but the tradeoffs are not included here. The features that are recorded here are motivated by experience in the process of development and qualification of flight software.

In order to establish the motives for this section it is necessary to make assumptions about the operational environment of the Shuttle. First, the use of the Space Shuttle will stretch out over a long period of time,and it will most likely encounter a number of changes of direction and emphasis. Second, the flight qualification process for software is and will continue to be an expensive process and may exceed its associated hardware in total cost over the operational lifespan of the Shuttle. Third, most, if not all, of the programming of the flight computers will most hopefully be accomplished with a higher order programming language primarily because of the necessity of higher reliability, easier maintainability and

and lower cost software. Finally, newer advanced software techniques must be employed to enhance reliability and cut costs.

Accordingly, this section has been included to identify more advanced features of the computer which are not generally available in "off the shelf" aerospace computers that are considered desirable for Shuttle software. They include higher order language machines, stacks, microprogramming and hardware diagnostics. The material presented in this section has been partially derived from an Intermetrics report [2].

6.4.1 Higher Order Language Processing

The flight computer selected for the Shuttle should possess the capability to execute efficiently in a higher order language environment. It must easily accommodate particular characteristics of the higher order language. Ideally, it could be designed for the assumed higher order language (i.e., a higher order language machine).

Computational inefficiencies occur whenever a compiler is required to translate the statement of the problem into machine instructions due to the mismatch between the computer architecture and the HOL architecture. The design of a machine which matches the language will not only eliminate the processing inefficiencies and improve performance over a conventionally structured computer, but it will also reduce memory requirements because a HOL statement is more semantically concise and economic of space. A number of designs have been proposed and implemented. Examples of this include the guidance computer for SAMSO by Cirad, the Burroughs D machine, several APL machines, and the Burroughs 6500, an ALGOL machine. The D machine* is appropriate to the Shuttle type application. This is bound to be extremely important on the Shuttle because of memory restrictions.

6.4.1.1 Advantages of Higher Order Language Processing. The following is a summary of major advantages of using direct execution of a HOL.

a) Cirad has reported [3] that their SPL machine has yielded an overall reduction of 60% in the memory requirements over a traditional single-address architecture for implementing the same set of guidance equations and

---

* Appendix F provides a description of the Burroughs D Machine.

functions. The memory efficiency is reported to be "due to the use of a polish stack with implied addressing, the use of floating point, the number representation used, direct fetch of literals from instructions, built-in array operations and use of one of two byte instructions without word boundary restrictions".

b) Kerner and Gellman [4] have designed a machine which directly executes Fortran statements. Programs written in this language and executed on their machine occupied 75% less memory. This conclusion was reached by comparing the machine code generated by the Fortran compiler for the IBM 7094 with the numbers of words required to represent the instructions for the HOLM. The 4:1 compression of memory space for program storage was the result.

c) Sugimoto [5] has studied the direct execution of the PL/1 language. He has actually implemented the PL/1 reducer and has some experimental results. For typical scientific programs, the length of the object code has been reduced by 25% compared to the object code generated by presently available PL/1 compilers. He also found a speed gain of 28% for arithmetic string operations.

d) Higher order language examples have demonstrated that a traditional machine architecture, viz. the IBM 360, uses at least twice as much memory as a specially designed computer, the Burroughs 6500.

There should be a master plan for computer architecture that recognizes the almost total use of a higher order language and blends the ingredients to produce a harmonious design that executes efficiently. Undoubtedly, the key component would be a stack oriented machine with short instructions. Hardware can be efficiently designed to interpret the HOL operators and execute "sequential execution form" directly through utilization of a push down list (PDL) or stack mechanism. The parsing would be an off-line operation which is performed only once.

It is interesting to observe that although the issue has been raised many times concerning the extra memory used by a higher order language on a conventional machine architecture, it is generally unappreciated that a sizable memory reduction can be accrued through the use of the higher order language computer.

6.4.2 Stacks

In computer terminology a stack is a list of items whose contents may be changed only by a "last-in-first-out" (LIFO) algorithm. A "push down" list is employed in which only the top entry is visible, and the only operations defined are "push" and "pop", which enter and remove entries from the list. The stack has been exploited as a structural element in the design of actual machines, notably by Burroughs [6], and in theoretical studies [5,6,7]. The dynamic behavior of stacks is well suited to the mechanization of recursive procedures involved in several processing activities, such as:

a) the management of "nested" subroutines, or procedures. This commonly occuring program phenomenon involves the tracking of calling and return addresses, the declaration, allocation and protection of variables, and the allocation of physical memory space.

b) The efficient execution of arithmetic statements.

Processing system designs that have made extensive use of stacks have demonstrated additional advantages in the dynamic allocation of memory space, the protection of program and data, the handling of interrupts, and the provision of a dynamic history of the process.

The result of each expression remains on top of the stack. The intermediate results are automatically allocated and deallocated at execution time, rather than statically at compilation or assembly time. The requirement for separate load and store instructions is much diminished, since these operations are implicit in the mechanism of the stack. The contents of the stack are an indication of the history of the process.

The value of a stack mechanism in computer design has been recognized by hardware implementations in a number of computer designs.

6.4.3 Microprogramming

Because it is neither hardware nor software, microprogramming has been termed "firmware". It offers a systematic method to combine the basic elements of a computer at a level lower than the instruction set in order to tailor its functions to the intended usage. The original intent of microprogramming was to introduce a systematic alternative to the usual and somewhat

"ad hoc" procedure of designing digital computers.

The computer is composed of a memory for program and data, a memory address register (MAR), memory buffer register (MBR), operating registers such as index registers, accumulators, program counter, an arithmetic and logical unit, an I/O unit, and an instruction register which drives the control unit. Computer instructions are executed by a series of register transfers, arithmetic or logical operations, and conditional decisions. The details of these sequential operations are implemented by the control unit.

The potential of microprogramming for the future, especially when the control store in which the microprogramming resides is writable as well as readable, lies in its flexibility. Rather than selecting an available hard logic computer, the opportunity exists to select a microprogrammed computer or a soft machine.

6.4.3.1 Advantages of Microprogramming. The advantages of such "softness" are listed below.

a) An instruction set can be optimized to accompany a particular higher order language. It can execute the language directly or perhaps a compiled intermediate language, extracting an even greater savings of required memory. The execution of the language statements will be faster and the amount of operating memory will be less. Kerner and Gellman [4] designed a machine which directly executes Fortran statements. As previously stated, programs executed on this machine occupied 75% less memory than the equivalent compiled program on an IBM 7090 computer. Since microprogramming can be used effectively in executing higher order language sequences, a significant cost saving can be achieved.

b) Both language specifications and implementation techniques may be modified throughout the developmental and operational cycle. The instruction set may be altered and augmented during the life of the Shuttle program to improve the capability of the language or to add wholly new features not originally known to be needed. Only modification to the microprogram and compiler would be required, not hardware redesign or requalification. This would be of immense importance on a program that is expected to have such a long operational period as envisioned for the Shuttle.

c) Specialized spacecraft functions can be absorbed into the microprogram. The tremendous speed advantage of micro-programmed operations because of inherent parallelism and the lack of dependency on slow speed memory make it an ideal candidate to implement highly repetitive spacecraft functions such as data bus servicing. Various Shuttle estimates show this one function taking up to 35% of the expected computer usage; withmicroprogramming this time could be cut significantly.

d) By microprogram changes, the same physical computer can be modified to efficiently accomplish different spacecraft tasks. Thus, if a functionally distributed computer system were planned for the Shuttle, it could be implemented using identical computers with different instruction repertoire. The potential of this approach is intriguing - in the event of failure it would only be necessary to load the micro-program control store on an available computer with the microprogrammed instruction set of the one to be replaced.

e) The argument that microprogramming creates a slower system can be challenged. It is true that the sequencing of micro-memory requires time that is not consumed in a conventional control unit. However, microinstruction look ahead can somewhat alleviate this problem. Even if simple instructions, like load and store, require a few hundred nanoseconds more, the overall execution of functional programs can be much faster. Patzer [8] compares the execution of an often used subroutine in micromemory with conventional programming. His example involved extracting the square root. The improvement was a factor of 9.8 for a 16 bit result and 4.75 for a 32 bit result.

6.4.3.2 Summary. In summary microprogramming could be valuable to the Shuttle software effort. In the last five years it has gained general acceptance and some manufacturers of commercial computers have employed it in both the CPU and I/O controller. Included among aerospace computers which have utilized a form of microprogramming are the Raytheon 251, Burroughs D Machine, Poseidon computer, SIRU computer, and RCA 215. The gain in flexibility is the strongest attribute of micro-programming. The possible performance gain by utilizing micro-routines with the subsequent savings in main memory is also a major consideration. The drastic reduction in the cost of logic, especially high speed integrated memory elements, makes the approach very feasible.

6.4.4 Descriptors

A descriptor is a method of describing the characteristics of an item of data. The data can be a variable, a procedure, a control word, etc. The descriptor contains information as to type, attributes, size, location, etc. Its actual size and format depend on the particular mechanization. In practical terms the use of descriptors enables the computer to aid in achieving more reliable software by assisting in error detection through the following:

a) an automatic identification by the machine, at execution time of the type and cheracteristics of the data (i.e., dynamic data declaration). It includes a direct check on the match between operator and operand (e.g., whether the object of a double precision multiply has indeed been declared as such). Another example is that of array manipulation by indexing, where it is important to ensure that the index does not exceed the array length.

b) A descriptor offers a compact substitute for all operations other than evaluation. In effect it becomes a "pointer" to the actual data.

c) A descriptor can keep track of miscellaneous information, such as:

    1) whether the referenced data is in main memory or out on secondary storage,

    2) whether the data is to be treated as "read-only" by a particular process,

    3) whether the current descriptor is the "main" descriptor, or whether it is a modification or copy, etc.

Practical applications of descriptors have taken advantage of one or more of these properties. The SPL machine of reference [6] keeps the descriptor with the data and places a pointer to the descriptor in the control stack. The descriptor specifies rank and dimensions, and in addition contains initialization data. It is naturally treated as a special declaration instruction to establish a specific value for the data item.

6.4.5 Run Time Diagnostic Aids

Because it is possible that a certain amount of execution and debugging will be accomplished on the actual computer rather than in a 100% simulator environment, it is submitted that a number of run time diagnostic features are appropriate. They could be implemented in hardware as part of a special "test mode" of the computer to be used during ground testing. This would enhance the probable use of the flight computer during software testing phases; it can also be useful for actual hardware in the avionics integration facility.

These diagnostics should be included within the hardware above and beyond the normal maskable run time error conditions, such as: fixed point overflow, floating point overflow and underflow, and addressing exceptions. These error interrupts are usually available for possible programmer recovery. The additional run time diagnostic aids provide capability not necessary in the normal course of execution. They are normally included in the diagnostics package of a simulator. A hardware implementation could include:

a) Trace trapping. It should be possible to designate a few bits in the program status word that will generate interrupts for trace purposes. These bits can specify what instructions or group of instructions should be traced. Possible states of the bits might cause the following:

   1) a trace of all instructions;

   2) a trace of all instructions not in the interrupt mode;

   3) only subroutine call and return and branch instructions to be traced;

   4) no instructions traced.

   This or a similar mechanism offer a great deal of debugging power at low cost.

b) "Coroner capability". A circular buffer or limited length stack is maintained with the address of the last n instructions that have been executed. Another approach just as useful and not requiring as big a buffer, is a list of the last n locations branched either to or from. The detail implementation of the diagnostics is subject to future refinement. A history of the past activity should be maintained for a possible postmortem analysis by the programmer in case of an occurrence of an error in operations. The name "coroner" pertains to the ability to search for the trouble after a run dies.

c) A "derail" or breakpoint location. The concept is to enable the hardware to trap the use of a specific location in memory as either instruction or data. A suggested method is to have several pairs of dedicated locations that may be used to store the address of memory at which it is desired to trap. One would catch instructions by comparing the value in that location with the instruction fetch location. The other performs similarly by a comparison with the addresses used in data fetches. Furthermore, the latter should be able to distinguish between data fetches and data stores, and select one or both to signal the interrupt. A possible refinement would compare the value after the address matched and only interrupt on value disagreement. This selective trapping or "derailment" technique is quite useful in pinpointing the error source when searching out the cause of anomalous behavior.

d) Stack overflow and underflow. In a stack oriented machine it becomes important to catch erroneous manipulations of the stack. A common cause of stack disorganization is the omission of operand fetches so that operators eventually run out of data or the omission of operators causing an excess accumulation of data in the stack. This could be a diagnostic trap or a hardware error condition.

These features can all too easily be dismissed out of hand on the grounds that they are too costly in terms of machine logic and/or time. However, if the manufacturers of the PDP-11 can include some of these, e.g., the trace-trap and stack overflow interrupts, into a computer aimed at an extremely cost conscious and competitive market place, then at least these deserve a fair trial. If the impact of some of these items were given a thorough evaluation, it is felt that their merits may warrant their inclusion.

6.5 General Computer Features

This section includes a description of hardware features generally available within most aerospace computers. Since addressing capabilities and subroutine linkage are of prime importance to software, these features are discussed in detail with a summary of the types which are best from a software viewpoint for the Shuttle. Other features discussed are floating point, unimplemented operators, memory speed hierarchy, and short form instructions.

6.5.1 Addressing

6.5.1.1 Background to Addressing Schemes. The most important characteristic of the architecture of the computer from a software point of view is the addressing schemes. If binary operators (plus, minus, and, or, etc.) are postulated then four additional pieces of information are necessary besides the operator itself. They are:

1) the address of the left hand operand,

2) the address of the right hand operand,

3) the destination address - where to put the results, and

4) the address of the next instruction to be executed.

Although all this information must be available, it is unusual to have it stated expressly. Consequently, it is a waste of space to carry bits in the instruction stream to specify a quantity whose value can be inferred. For example, the address of the next instruction can be left out. Several addressing schemes will now be defined.

a) Three-Address Machine. If the remaining three are specified, the result is what is referred to as a "three-address machine." A number of these have been built; a notable example is the Honeywell 800/1800 series. The three-address machine also tends to be wasteful of space since seldom in practice does a sequence of instructions occur combining two independent quantities and placing the result in a third. Thus, it is advantageous to imply one or more of the remaining addresses.

b) Two-Address Machine. For business computers, "two-address machines" are typical; the IBM 1401 series is a prime example. In this case, the result location coincided with one of the operand locations. Thus, it is possible to add two numbers together, say A and B, and put the result back in A. This is extremely useful for data movement and editing, especially for whole blocks of data characteristic of business data processing.

c) Single Address and "One and A Half" Addressing. Scientific computers initially were "single address machines",particularly in the second generation including the IBM 7090 series and the Control Data 3000 series. A single address machine architecture utilizes an accumulator which is implied as the source of one of the operands as well as the destination location. It is useful in scientific type calculations (e.g., A+B+C+D+E....), and is very efficient when a new quantity is chained to the result of the previous calculations. However, single addressing is less advantageous when the form of the calculations are of a more general tree structure form, (e.g., A*B+C*D+E/F). In these cases, storage of intermediate results into temporaries is necessitated. Thus, multiple accumulators were designed for third generation computers. For purposes of this report these are termed "1.5 address machines." Actually they are two address machines where one of the addresses is specified in small numbers of bits (i.e. 3 or 4). It selects one of 8 or 16 accumulators or special registers as the source of one of the operands as well as the result destination. In theory this offers far greater flexibility in the allowable sequence of calculations at a small cost. Examples of computers using this type of addressing include the IBM 360, the Univac 1108, and the PDP-10.

d) Zero Address Machine. A different approach to addressing has been utilized by several manufactures and termed "zero address" or stack machine. In this concept, operators (i.e., instructions) appear by themselves with no operands specified. The operands associated with the operator are always assumed to be on the top of the stack. Additional instructions are available for loading the stack from memory and restoring from the stack to memory.

A stack is a dynamic realization of special registers of a general register machine, whereas general registers in the "2 address" are static in nature (i.e. their

number is fixed). Unless a global strategy can be defined for general register usage, then program branches are laden with register saves and restores, since no knowledge of their current usage is available. On the other hand, intermediate results in the stack are merely pushed down when a branch to new computations is taken. They are automatically re-available ("pushed up") when the return is made, provided the stack order has been preserved.

The zero address machine provides the most efficient access method for specifying algorithms since very little space is used. Only the operators and the fetch-store addresses need to be given. Computational flexibility is achieved by the logical sequence of the operators and the fetch-stores. It also has the additional advantage that arithmetic expression evaluation and compiler parsing have been developed using a stack effectively. The disadvantage of a stack machine is based on limitations of current memory technology. Hardware stack registers are small in number and as the stack overflows, stack memories have to be simulated by random access primary memory.

From an addressing point of view, all the Shuttle candidate computers are quite similar. Most are general register, 1.5 address machines (exceptions include the Autonetics D216, and the SKC-2000 which are single address machines). They tend to have many useful short forms of instructions, generally the register-to-register format in which both addresses are abbreviated.

6.5.1.2 Direct Addressing. In a direct addressing method, the address field must contain enough bits to define the required address, (i.e. 15 bits for 32K addresses). A common defect found in second generation commercial computers was a limited addressing capability. A common size of the address portion of instructions was 15 bits. This limits direct addressing to 32K words. Index registers, since they were viewed as true index registers and not base registers, seldom allowed more than 15 bits of information and sometimes less. This effectively limited the memory size of the machine. As hardware became available, attempts were made to modify the machine addressing architecture to accommodate large memory sizes, but the mechanisms were invariably complex and tricky. One approach was the bank register. Machine designers were confronted with "grafting-on" an extended addressing capability without sacrificing compatibility with existing computers and programs. The following are examples.

a) Apollo Guidance Computer Addressing. Airborne computer addressing was approached similarly. The Apollo Guidance Computer (AGC) was conceived initially as a 4,000 word computer. Its design initially contained 12 bits for direct addressing, allowing direct access to all of the memory. Through redesign, the memory size grew to 38,000 words and yet the fundamental addressing structure did not change from 12 bits. Even through indexing it was not possible to enlarge the addressing space to greater than 12 bits. The result was that the AGC remained virtually a 4,000 word computer that had 38,000 words of memory physically attached. This feat was only accomplished through a skillful display of juggling and balancing. It required the setting and maintenance of an F-bank register, fixed-bank, an E-bank register, erasable bank, and a B-bank register, a both bank. The process did not culminate until it reached a level picturesquely titled, "Super Bank". The addressability difficulty the AGC had was a serious problem and a keen rival of erasable shortage in nuisance value.

b) IBM 360 Addressing. The IBM 360 recognized the limitations imposed by absolute addressing restrictions and permitted addresses to go up to 24 bits or 16 million bytes of storage. This has proven to be ample, however, the 360 introduced a whole new set of addressability problems. The fundamental reason was a short displacement address field 12 bits out of 32, which limits direct addresses to 4,000 bytes. With no alternate longer form and no absolute or indirect addressing capability the 360 programmer faces addressing difficulties. In many cases, the only solution is with frequent loading and reloading of data addresses into registers. One recent study concluded that the average data address on the 360 took over 5 bytes of instruction length when the otherwise useless register loads of addresses were also counted. This compares to 20 bits (2 1/2 bytes) of raw address in the unaugmented instruction. It is not uncommon for 360 programmers to relate their vexing experiences to addressing problems.

6.5.1.3 Indirect Addressing. Indirect addressing has proved itself invaluable on many varieties of computers, both large and small. It is virtually indispensible in the mini-computer business where memory size is at a premium, and there seems little doubt that it would produce memory savings on flight computers. Moreover, it is a technique with wide-spread applicability. Whenever data is dynamically allocated

and manipulated, it is necessary to access it through pointers, descriptors, or indirect addresses. In addition, when the address field of an instruction is short, indirect addressing performs another function - it broadens the address scope. Almost every modern computer offers some form of indirect addressing. The IBM 360 family is a notable exception to the rule. Much of the thrashing around the 360 does in loading registers could be avoided if it had an indirect addressing capability.

6.5.1.4 Indexing and Base Registers. Indexing involves the selection from an ordered set. It is a keystone to the theory of computing as well as a practical necessity present in one form or another in every computer. Unfortunately, it is difficult to distinguish from base registers in the manner in which they are implemented on a number of computers. This difficulty poses some problems since they are fundamentally quite different concepts and occasionally must be accorded different treatment. A case in point: when indexing, it is advantageous to have the computer automatically multiply the index value by the number of memory quanta (bytes or words) that comprise the type of element that is being indexed, e.g., double precision quantities often take two words. For a word-oriented machine the index should be multiplied by two before adding to the base address. Since the 360 uses byte addressing, the index should be multiplied by four before adding to the base for accessing full words. On the other hand, this automatic index alignment is never appropriate for base registers. The base registers are marked in memory where data begins and as such are absolute addresses. If a clear distinction between these two attributes is made and followed through in the machine architecture, much confusion can be avoided.

6.5.2 Static Versus Dynamic Addressing

The choice of static or dynamic addressing depends on the operating environment and requirements of the software system. Statically addressed computers are usually distinguished by relatively long fields in their instructions to enable direct addressing most of memory. Dynamic addressing computers are characterized by a shorter address field which is used for relative addressing or a displacement off of a base register. The base register serves as a marker in memory of the bench mark point or starting location of currently active variables. The variables may themselves be an array which is indexed to select a particular element. Generalized dynamic addressing requires a multiplicity of base or display registers.

6.5.2.1 Static Addressing Problems. Static addressing machines were the earliest in practice; almost all the second generation computers were statically addressed. Early programming languages i.e., Fortran, were explicitly designed to operate on a static addressing computer. Thus, the memory retention technique for Fortran is static in nature. When the size of a Fortran program and its collected subroutine exceeded the available memory size the programmer was forced to resort to manual overlay techniques. Even Fortran requires dynamic addressing techniques; it occurs in parameter passing or actually in parameter receiving. In order to bind dummy arguments of subroutines and functions to the actual arguments with which they were called, it is necessary to perform address replacement which is equivalent to a form of dynamic addressing. This is not a simple process on some static machines. In a number of cases the compiler implementers had to resort to the technique of a subroutine prologue that went through the whole subroutine and physically replaced any instructions that referred to a dummy argument with the actual address that was received at subroutine call time. This is, of course, a cumbersome process necessitated by a lack of flexibility in the addressing structure. In some statically addressing computers, single item dummy arguments are accessed through indirect addressing. However, these computers have difficulty in coping with indexed arrays that are dummy arguments.

Almost all static address machines have index registers. Although index registers have been utilized as base registers for dynamic addressing purposes, there are some difficulties particularly where true indexing is also required.

6.5.2.2 Dynamic Addressing. As static assignment of variables became less satisfactory due to the operational software environment and multiprogramming operating systems, the first step was the utilization of relocatable loaders. The loader processed a program prior to execution, modifying all addresses to reflect new positions that the program would occupy in memory. It generally required that the compilers and assemblers mark true addresses so that they could be relocated and distinguished from phony addresses which would be left alone. The result was a program that could execute in a different portion of memory than the one the compiler had made static assignments for.

The next step in the evolution was the addition of relocation registers to the computer hardware. A single relocation register could displace an entire program by a constant amount at execution time. Next came the desire for re-entrancy which demands the segregation of program and data. A solution adopted

by the PDP-10 and the Univac 1108, was to offer a pair of relocation registers, one for data and one for programs. Several active processes could share the same program, e.g., a commonly used compiler, and still have their own data area controlled by the data relocation register.

6.5.2.3 Approach to Dynamic Addressing. There exists an underlying assumption that it is possible to establish at compile or assembly time a relationship, i.e., an address of the calling variables. This is generally true for Fortran-like code but false in a more dynamic operating environment. Even in Fortran the argument address must be dynamically created at execution time if the quantity referred to is itself a dummy argument. In a more dynamic environment a normal procedure is to create the values themselves, or the addresses as needed and leave them in a list or on a stack. For this purpose, "load address" type of instructions or "immediate instructions" as referred to on many computers (not the type of immediates used on the 360) are extremely useful for the creation of dynamic addresses. An example of a PDP-10 dynamic calling sequence is given below.

The calling sequence which is used to enter a function (or routine) is

```
HRRZI k, P1
PUSH  S,k          ; push address of 1st parameter
                     onto the stack
HRRZI k, P2
PUSH  S, k         ; push address of 2nd parameter
                     onto the stack
...                ...
PUSH  S, Pn        ; push nth parameter onto the
                     stack
PUSHJ  S, FCN      ; jump to the called function
```

In this case, the cryptic HRRZI stands for Half-word Right-to-Right with Zeroes Immediate. It is the method of doing a load address on the PDP-10.

The other feature that is necessary to support dynamic addressing is a workable two-dimensional addressing scheme. It can take the form of the base registers of the IBM 360.

Indirect addressing is also an important concept; it serves as a conversion point, a transformer to convert dynamic to static or vice versa. It is generally implemented with static addressing, although it can be indexed sometimes. Its static nature is easy to visualize but the manner in which it gives a static environment some dynamic capability is not intuitively obvious. The deferral of addressing, at the heart of indirect addressing, permits an unlimited number of data references through the same indirect memory location. Thus, by changing the value of the indirect cell, multiple references can be modified, a sort of pseudo-dynamic addressing.

6.5.2.4 Shuttle Environment And Addressing Requirements Summary. The operating environment of Shuttle software will require both static and dynamic addressing. Significant portions of the Shuttle computer operating memory will be fixed and dedicated, such as compool data, executive system entries, central routines, etc. However, other parts of the memory will require dynamic addressing, such as dynamically assigned task working memory and phase loaded programs. A multiprogrammed environment will be presumed with reentrant subroutines standard. Accordingly the use of base registers, indexing and indirect addressing are all desirable. A hardware stack is considered extremely desirable for this type of environment and was discussed in Section 6.4.2.

6.5.3 Subroutine Linkage

A second area of importance in analyzing the programming aspects of a computer is the subroutine calling and return procedure. These instructions that occur after the program logic flow are of interest since they are a two-step process; i.e., they call a subroutine by a branch or jump and then leave the old value of the program counter register somewhere for returing. Just where and how this is accomplished is the subject for considerable debate. Several common methods will now be discussed.

6.5.3.1 Return Address Stored in Memory. The most common implementation of this technique proceeds as follows: when a subroutine jump to L is executed, the return address is deposited at location L and execution of the subroutine is commenced at location L+1. Subsequently a return is effected by using location L as a pointer to the return. The advantages of this technique are simplicity and the fact that it does not require any special registers or general registers. The disadvantages of this approach are: first, the program must be in a writable portion of memory. This effectively prohibits the use of read only memory stored subroutines. Second, since data,

a return address, is tied to the program, it is virtually impossible to produce re-entrant code.

In order to eliminate the above drawbacks, an interesting variation of this technique has been proposed for the Control Data Alpha computer. It is called indirect subroutine jump. In this case the return address is not stored in location L but location L has an indirect address or pointer to another location in memory where the return address is to be deposited. Then subroutine exit is accomplished via an instruction called a double indirect jump. It is easy to see that this permits the program to be segmented and stored in read only memory. Location L need only be a pointer to generally writable memory. However, this does not in general enhance the re-entrancy of a subroutine. In the case of the Alpha, four bits of the 32 making up an indirect address are used for index register information. Thus, by a judicious combination of index register settings and absolute displacements it is possible to achieve the re-entrancy conditions, albeit by a somewhat roundabout method.

6.5.3.2 Return Address Saved in a Register. Use of a special register for the return address is another approach to linkage. It is usually automatically saved in the case of job interruption and readily available upon return. Thus, terminal subroutines, i.e., those which do not call other subroutines, are efficiently coded and reentrant. The nesting of subroutines requires the first return address to be saved somewhere while the second subroutine is called. Depending on the methodology to save return addresses, re-entrancy may or may not be achieved.

The disadvantage of this technique is that it needs and utilizes either dedicated special registers or a portion of the available general registers. Usage may be divided between two approaches:

a) One of the general registers is selected as the link register, and its address is specified by the branch instruction. For return purposes the subroutine must be aware of which register is the link register. The prime example of this technique is the IBM 360.

b) A dedicated special register is used to save the return address. Examples of this are the Q register of the Apollo Guidance Computer and the sequence history register of the Honeywell 1800.

6.5.3.3 Return Address Placed in the Stack. This is the natural solution for a stack oriented machine. A stack is not static like special registers but is the dynamic history of program and register usage. Re-entrancy and recursion are then not difficult to accomplish. An example of a computer which uses the stack for return addresses is the PDP-11, a semi-stack machine which uses the stack for subroutine calls and interrupts. It is actually a two-address computer where either or both addresses may be short. The short form allows the register to be addressed or the contents of a register to be used as an address or pointer. It also allows the register to be used as a stack pointer with automatic incrementation and decrementation as appropriate. It can then assume the characteristics of a zero address, one address or two address computer.

The PDP-10 is classified as a 1.5 address, general register computer but also has stack oriented subroutine calling instructions. These are the PUSH DOWN AND JUMP and POP UP AND JUMP instructions. The first is used for stacking return addresses while calling subroutines,and the latter for returning. Actually, the PDP-10 is very interesting since it has a subroutine calling instruction in each of the three classifications of calling types mentioned above. It even offers a fourth called JUMP AND SAVE AC which is combination of 1 and 2. It saves the return address in an accumulator and saves the previous contents of the accumulator in memory.

6.5.3.4 Summary. There seems little doubt that the use of a stack for return addresses is the best choice. The return address stack fits in extremely well with higher order languages. In fact, a normal ALGOL implementation always creates a stack (by software if necessary) if the computer does not have hardware stack instructions. Furthermore, many other language implementations, which are not so dedicated to the stack mechanism, have nevertheless selected a stack for return addresses as the best solution.

6.5.4 Floating Point

The need for floating point data representation and computation for the Shuttle has been presented by many as a desirable feature from a software viewpoint. The MDAC base-line design recommends floating point hardware for the onboard computer. They support it with an analysis estimating over $5 million savings in total software cost using floating point vs. a cost of

$1.5 million for the computer hardware. The assumptions in the analysis may be questioned,but it is concurred that the net savings in software will be greater than the cost for the hardware.

Most of the newly announced flight computers have an option for floating point (i.e., CDC Alpha, SKC 2000, Univac 1832). In the next few years it will become a common feature in the aerospace computer.

The requirement for floating point is motivated by difficulties with fixed point scaling of guidance, navigation, and control programs (about 40% of the software). The dynamic numeric range required in these type calculations as well as accuracy requirements can result in tricky and complex scaling analyses. In order to understand the desire for floating point one must appreciate the software difficulties of implementing complex equation solving in fixed point. A discussion of fixed point problems and the advantages of floating point are presented in Appendix D . In addition, a "top level" analysis of the Apollo Guidance Computer code was performed to determine the impact on memory resulting if floating point hardware was available. The results are also included in Appendix D and reveal an approximate 10% savings overall in memory.

6.5.4.1 Advantages of Floating Point. A summary of the main advantages of floating point hardware are the following.

a) It eliminates the need for some scaling analysis effort.

b) It improves reliability of software since it standardizes the approach and eliminates errors.

c) It does not require the programmer to maintain and test scale factors in addition to parameters.

d) It does not require testing over dynamic range for underflow conditions.

e) It simplifies program maintenance for changes in coding.

f) It aids in diagnostics and recording by eliminating the need to convert.

g) It aids in readability of code; no tricky scaling operations.

h) It requires less memory.

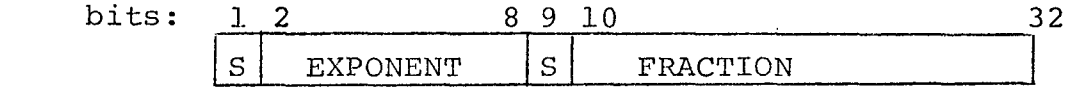
Although floating point may be slightly slower than fixed point in general and will require a cost increase to the computer hardware, it is considered a mandatory feature for the Phase B Shuttle computer.

6.5.4.2 Single Or Double Precision Floating Point. There are two parts of a floating point data representation: the characteristic and the mantissa. The characteristic is the exponent and the mantissa the fractional representation. A number X is represented by

$$X = 2^C M$$

where C is the characteristic and M is the mantissa $-1 \leq M \leq 1$.

Most applicable Shuttle computers have a 16 bit word length. Those with floating point organize 32 bit floating point words with a 7 bit exponent plus sign and 23 bit mantissa with sign as follows:



This gives a maximum range of expression of approximately 10^{37} to 10^{-37} and a precision of approximately 7 decimal places.

An analysis of the floating point word length is provided in Appendix E . It analyzes the precision requirements of the state vector in orbit, approach, and landing. It concludes that 23 bit precision is "adequate", in general, with the following problem areas:

a) the time must be carried in double precision;

b) the Encke integration method should be used to avoid large accumulated roundoff errors in orbital integration of 1-2 orbits. (Encke organizes the state into two parts: a form of double precision.) Encke integration may however be used for independent reasons to increase the speed of computations due to shorter time steps.

c) There is marginal accuracy in rendezvous due to range.

From the results, it is clear that more than seven places of precision would be desirable in certain areas, e.g., 9 or 10.

Secondly, the conclusion of the study reveals that seven bits for exponent and sign is sufficient; the most sensitive area being the Kalman filter.

6.5.5 Unimplemented Instructions

Some form of unimplemented instructions or program operators have been available to the users of many computers. The concept has been to use gaps which have been left in the instruction set, either intentional or not, since their execution will produce an illegal instruction interrupt. It is then possible to trap these instructions, decode their meaning, and branch to a routine that may be user-programmed. This provides a mechanism to create programmed operators which appear as though they were extensions to the computer's instruction set. If the computer is fully microprogrammable, then the need for this feature may be non-existent; but in a conventional computer, it is a highly useful feature. When implementing powerful constructs of a higher order language, it provides an efficient linking mechanism to the basic routines as required. The program operators seem particularly well suited to the task. They are especially beneficial in interpretive execution of instructions.

To make programmed operators efficient, the addresses, if any, that accompany an instruction must be made available by the hardware to the program that simulates the instruction execution. No techniques have been developed that are completely satisfactory. The PDP-10 technique stores the effective address at the trap location and executes the instruction contained in trap location plus one. The Honeywell 1800 leaves two addresses in a particular pair of special registers for subsequent accessing. Again, the stack machine faces less severe difficulties. Operand fetching, including indexing, is pre-accomplished by other instructions, and the values are available on the stack when the unimplemented instruction is executed. As a consequence, operand availability is not a problem. Also implied by this discussion is an efficient trap mechanism which may or may not be the standard interrupt handling hardware. Discounting full microprogramming this feature promises hope for efficient compiler-generated code.

6.5.6 Short Form Instructions

In an effort to save space in the instruction stream, shorter forms of instruction were introduced. The word-oriented second generation computer used exactly one word per instruction. It usually provided machine-wide direct addressing capability. But often in practice, simple instructions with little or no addressing were all that was needed.

Two general approaches were followed to increase efficiency. One was to provide more complex and even compound instructions within the same word structure. However, it becomes increasingly difficult to effectively utilize complicated instructions; e.g., how does one use a subtract logical product instruction? The other path was to provide shorter instructions. It appears to be easier for both hand-coded or compiler-produced code to use more of these instructions, The types of instructions that can be made short include the following.

a) Inter-register operations. It is clear that short address forms will suffice for instructions involving data that resides in general registers, accumulators, or index registers. These are found in almost all new general register computers.

b) Shift instructions. Shift counts are small numbers and indexing is not usually needed; short addresses are then enough.

c) Use of short literals. Experience shows that the vast majority of literals that are used are small integers that can be expressed in only a few bits.

d) Short displacements. Countless examples can be cited from actual programs that require little or no address field for displacement. Three of four bits would be adequate for many practical cases, especially in writing subroutines. In fact, many routines are entered with data addresses in general registers, and no displacements are needed.

6.5.7 Differing Memory Speeds

The central processor should not be designed around a specific memory cycle speed but should accommodate a hierarchy of memories of various speeds and sizes. Not only does this allow the eventual attachment of a mass memory which is slower to save power and weight, but even more importantly allows a small amount of very fast semiconductor memory to be used for repetitive, high frequency calculations. In terms of computers with 1 to 2 microsecond core memories, it should be possible to attach semiconductor memories in the 200 to 300 nanosecond region and have the processor logic keep up for elementary instructions. Again, the need for fast memory diminishes if

microprogramming is present since the microprogrammed control store is presumed to be very fast memory. However, if microprogramming is not present, some amount of fast primary memory can do much to accomplish the same objectives. It can serve as a home for high frequency calculations (e.g., data bus servicing) and for heavily used routines (e.g., executive and/or a run time interpreter). Thereby it improves the overall throughput of the computer.

6.5.8 Standard Computer Features and Characteristics

Typically, several features are used in describing a computer such as word size, memory capacity, speed and instruction set repertoire. Because they are an integral part of the computer's design, most computer surveys tabulate these features as a means of comparing computers. However, after examination of applicable Shuttle computers, many are very similar with respect to these features, with one or two exceptions. Typical features are: 16/32 bit word size, expandable memories to 64-128K, and 2-3 μs add time. Since the objective of this task was directed at identifying features desirable from a software viewpoint, only summary comments are provided on these features.

In general, strictly from the software viewpoint, "the more the better" applies to all. The important aspect of these features is that they should not impose requirements on the software to "work around" hardware deficiencies. For example, inadequate speed can impose requirements on the structure of the software executive system requiring it to avoid job backlog and poor response. Inadequate word size can necessitate the majority of software algorithms and computations to be coded in double precision. Inadequate operating memory, a major problem on Apollo, as previously discussed, can require memory overlays and tricky coding to "fit" the software in the computer.

The speed, word length and memory size should be selected as part of the computer which best meets the requirements of the Shuttle. Clearly, there are several factors and tradeoffs from a total hardware/software and cost viewpoint in selecting the best choice.

6.5.9 Phase B Computer Requirements

Specific computer requirements derived by Phase B contractors are presented in Appendix A and B. From these requirements, a set of general characteristics were identified:

1) Physical characteristics (approximate)

a) weight: 75 pounds

b) power: 300 watts

c) size: 1 ft^3

(militarized at least to MILE5400 Class 2)

2) Processor

a) speed: Both Phase B study reports recommend a computer with approximately 2-3 μs add time. Although overall speed is difficult to measure, an adequate safety margin should be provided.

b) instruction repertoire: Most aerospace computers contain instructions sets which are similar and adequate for Phase B Shuttle applications. There are of course some with unique instructions such as the CDC Alpha which contains hardwired instructions for mathematical functions (i.e., sine, cosine, and coordinate transforms). Such instructions can be extremely useful for Shuttle software.

3) Memory

a) word length: Word lengths of aerospace computers have varied from 9 to 52 bits. Major factors in the choice of word length involve the precision of data and calculations, memory addressing capability, and order code format. Addressing and order codes have been discussed. A word size of 32 bits has been derived by Phase B contractors as adequate. Desirable features include single and double precision words and byte and halfword addressing.

b) capacity: Main memory should be expandable to allow extra memory to be easily accommodated. The use of secondary storage is recommended as discussed in Chapter 4. By loading programs from secondary storage only for the flight phase in which they are used reduces the amount of main memory necessary.

4) Real time clock and interrupt

The computer must contain a programmable real time clock and external interrupts that can be set and reset under program control. The clock is necessary to signal the beginning of a minor cycle as well as manage tasks scheduled on a time basis.

5) Availability

An emphasis on using "off-the-shelf" equipment was a guideline in Phase B. For lack of a definition the following was assumed. The computer must exist at least as a prototype system now and be capable of achieving full production status by the middle to the end of 1972.

A review of applicable aerospace computers indicates that several computers satisfy some of these requirements. Although this list is not meant to be complete, the following computers are identified:

a) IBM 4 Pi AP1
b) CDC Alpha-1
c) Raytheon 251 (or modified 251)
d) GEMIC 32A
e) Hughes HCM 231
f) SKC 2000
g) Autonetics D-216
h) Litton 3070
i) Univac 1832 (power)
j) RCA 215 (weight)
k) Burroughs' D-machine

The scope of this study did not however, include a detailed evaluation of computers; consequently no detailed information is presented in this report.

6.5.10 Selection Criteria

Devising criteria is one of the difficult tasks in selecting a computer from a set of candidate systems. Previous sections of this chpater have identified desirable features from a software viewpoint which can be used as criteria. Certainly, factors other than performance are also of prime importance and can dominate the selection procedure, such as: undesirable features, credibility of the manufacture, experience with the computer in similar applications, off-the-shelf availability, space qualification costs, general technology and total costs. All of these factors are part of the total selection process.

One approach to selection is to weigh the features of candidate systems. However, if the criteria are individual computer features, an evaluation of total system performance is difficult to obtain directly. It is the thesis of the subsequent section that Shuttle software benchmark programs can be devised and should be used to obtain relative measures of candidate computer performance. Although benchmark programs are by no means a solution to the selection problem, they can

provide more useful information on the probable machine performance based on a more realistic model of Shuttle application software. The benchmark development utilizes a higher order language approach.

6.6 Benchmark Programs as an Aid in Computer Selection

6.6.1 Background

Ideally the comparison of two candidate machines could be accomplished after the fact by proceeding to do the job on both, and then determining which one would perform better with respect to cost and execution time. Of course this is not practical. At the other extreme, people try to evaluate the instruction set of a machine by postulating some mix of instruction types and then evaluate the machine's execution time and memory needs based on these instruction types. Memory tends to be the major cost in the hardware of a computer and hence memory size often becomes the measure of system cost. Unfortunately, every machine has a different architecture and a single postulated job mix becomes meaningless. What is really desired is to know how the machine performs when doing useful work.

Often benchmark programs have been devised for comparative testing, but they are seldom representative. They usually consist of a relatively simple set of routines that do some well-defined tasks such as matrix multiply, sort, etc., but these ignore the real characteristics of a job's execution and are inadequate. It is most important to know how the machine executes programs in the application environment. Subroutine calling and exiting, saving of special index registers, linking conventions, and addressing are of interest, but they are important only to the degree that they are utilized in the execution of actual programs.

The approach of using benchmarks is often followed in selecting a computing system by a general purpose commercial computer facility. This is aided by the widespread use of higher order languages. If Cobol or Fortran programs exist that are "representative of the daily workload", they can be bodily removed and compiled on the other machine and relative comparisons can be drawn. The software (i.e., the compiler) as well as the hardware is tested in this fashion: it is only the success of the combinations of both that can produce good results and merits the ranking. It can be argued therefore that fair and reasonable overall conclusions may be obtained.

This approach however, does not seem to be applicable in the case of Shuttle computers. The primary reason is a lack of compilers for Shuttle candidate computers. However, this does not rule out the possibility of a comparison of computers with respect to their performance in a compiler generated environment.

6.6.2 Hand Compiled HOL Benchmarks

One approach to obtaining Shuttle computer benchmarks involves creation of a pseudo-compiler for each machine, doing a hand compilation on representative programs, and then examining the efficiency in terms of memory and time of the resultant code. This method also eliminates one source of discrepancy, the vagaries of the individual compiler writers and their chosen techniques. Since the same people and the same techniques would be used on all of the compilers, it can be argued that the results would be a fair measure of each machine's capabilities.

This approach could be as follows:

a) Shuttle application software (guidance, navigation, checkout, etc.) that seems representative, will be coded in a HOL. Besides these real examples, other coding will be generated that is weighted by the statistics such as have been gathered by Wichmann [9] and Knuth [10] in their surveys of existing source code.

b) The code will be compiled and executed on a commercial machine for which the HOL exists in order to authenticate it.

c) A run time environment will be postulated for each Shuttle candidate computer system with enough detail to define the working environment. For instance, if the machine has a "general register" set, then these registers will be accumulators, those registers will be base registers, and this register for stack pointer.

d) A mechanical translation policy will be developed to transform the "intermediate code" of the HOL into equivalent assembly language statements for each computer. This need be done only for the various HOL constructs that are used in the benchmark programs and not for all the possible coded statements.

e) A manual translation of the actual intermediate language will be performed.

f) Statistics will be obtained from the translated code. Size data can be gathered by direct examination of the resultant code. Speed information can be inferred by counting instructions as they would be executed and by using the manufacturer's supplied data regarding machine instruction times.

g) Summary tables and conclusions will be drawn concerning the experiments. In addition to the data produced, both the good and the weak points of the individual computers uncovered during the translation process, including subjective evaluations concerning the suitability of the various computers, should be analyzed.

6.6.3 Statistical Approach

A second approach of evaluation can be made by extending a method presented by B.A. Wichmann [9]. Briefly, his method consists of defining a representative set of statements of the HOL (in his case Algol) and making a set of time measurements, Tij, for each representative HOL statement i (i=1 to n) on machine j (j=1 to m).

He then models these measurements as:

$$Tij = Fi\ Sj\ Rij \qquad 1 \leq i \leq n \qquad 1 \leq j \leq m$$

where Fi is a measure of statement complexity, Sj is a measure of machine performance, and Rij is a factor related to the machine's relative performance for a particular statement.

The assumption is that the execution time of a statement is somehow directly proportional to the "complexity" of the statement and to the "performance" of the particular machine. The Rij is then a measure of how much the particular Tij measurement varies from the ideal.

After obtaining the Tij measurements, the next step is to use these mn values and to determine the m + n values for the Fi and Sj. This is a valuable approach if the postulated measurements Tij are the only ones obtainable. However, the results are less than satisfying since the relative frequency of dynamic occurrence of the statements of the actual application is not taken into account. An extension of this approach is proposed as a more satisfying view of the problem of determination of statement complexity and machine performance.

Suppose a larger sample of Shuttle software were coded in the HOL. If these programs were executed on a commercial machine, under instrumentation which can observe the relative frequency of dynamic occurrence of each statement type wi, then a more meaningful measure of machine performance (in this case, slowness: Pj) is given by

$$\sum_{i=1}^{n} wi\ Tij = Pj \qquad 1 \leq j \leq m$$

The P-values are analogous to Wichmann's S-values, but are renamed to avoid confusion. These P-values are computed from the measured statement execution times on the j machines as defined by the matrix Tij adjusted by the statement execution frequency estimation for the Shuttle application software.

In an analogous manner the relative measure of the memory utilization can be obtained. Let Mij be the amount of memory needed to represent the HOL statement i, and the machine j. The static distribution of HOL statements can be obtained for the benchmark by counting the HOL constructs in the code. Define σ_i as the static distribution. Then a relative measure of memory efficiency can be obtained by

$$\sum_{i=1}^{n} \sigma_i\ Mij = Aj$$

The Aj values are relative measures of the memory sufficient for each machine.

Since the Pj have been determined, the statement complexities Ci in the Wichmann equation can be written as:

$$Tij = Ci\ Pj\ Qij \qquad 1 \leq i \leq n \qquad 1 \leq j \leq m \qquad (1)$$

where the Qij and Ci are related to the Rij and Fi of the Wichmann equation. This is mn equations in n(m+1) unknowns. To obtain a "best fit", we chose to minimize the variation of the Qij relative to the Ci, therefore define:

$$E = \sum_{ij} (LQij)^2 = \sum_{ij} (LCi + LPj - LTij)^2$$

where the prefix L on a variable indicates the logarithm of that variable. This leads to

$$LCi = \frac{1}{m} \sum_{j} (LTij - LPj)$$

and the Qij may then be computed from (1).

The interpretation to be placed upon the Qij values is that they reflect the inefficiency of machine j executing statement-type i, relative to how that machine executes other statement-types, independent of the statement-complexity and frequency of execution.

The values Qij then, allow for an understanding of the structure of the machine with respect to the HOL. This would allow insight as to the ability of the machine to carry out particular functions not specifically considered in the weighting of the HOL statements.

6.6.4 Summary of the Approach

Since it can be very expensive to fully translate "representative program" into each of the machines on a large enough scale to obtain a "representative" sample of machine code, this second approach of applying statistics of statement execution frequency and statement static frequency as derived in a general purpose computer appears desirable.

With the dynamic frequency of occurrence of HOL "operations" for the particular application, it is possible to obtain a relative measure of execution time for each machine.

Wichmann [1] used over forty different statements to compare each compiler. The statement classes used for comparison should be generated from two different considerations:

1) the different functions of the HOL: scalar operations, array operators, flow control within a program, modularization, input/output; and

2) those features which tend to differ in each machine architecture: integer operations versus floating point operations, the use of literals, both short and long, and of different precisions, etc.

Once these classes of representative operations have been obtained, only their implementation on the various machines need be considered. If a HOL compiler exists, then by actually executing the "translation" of the aerospace computer's imple-

mentation of the HOL statements, a time measurement can be made for each operation; similarly a memory space measurement can be made by examining the instructions. Since a compiler may not exist a manual translation, as in the first method, can be performed.

The dynamic and static frequency of occurrence of these "HOL statements" for the Shuttle benchmark are determined once and can be accomplished via execution on a machine which has a HOL compiler.

This method has the great advantage of being able to vary the postulated HOL statement mixes to determine how the relative merit of the machines changes.

6.6.5 Problems with the Benchmark Approach

While this method can obtain both an "execution" time measurement and a "memory" size measurement, it does not give the "complete picture".

1) The measurements used to obtain the execution time of the HOL statements on a given machine introduce several inaccuracies.

   a) There is an assumption about the method by which the HOL translates into the machine language. This may not be accurate, particularly in the context of more complex statements than the representative HOL statement. Or conversely, perhaps the computer cannot generate as compact code as the assumed translation.

   b) The execution time of the translated HOL statement is very difficult to obtain. If a HOL compiler was used, time must be "measured" on the object machine itself and usually the machine's timing mechanism has very large granularity, if the measurement is possible at all. Otherwise, some external clock would have to be used.

   c) The last point indicates that one method to obtain more accurate measurements would be to embed the desired statement in a DO LOOP for a thousand or million executions. This has the bad side effect of creating a static environment for the given HOL statement. Even though it is being executed one million times, this is not necessarily equivalent to its presence of the statement one million times in a real dynamic environment where

each occurrence would be from another, and different HOL statements. This criticism is equally valid of the manual translation approach.

2) The HOL statements cannot sufficiently take into consideration input/output. The interrelation of asynchronous computations in a multiplexed environment depends highly on the physical hardware characteristics, how they are interconnected, and upon the executive systems.

Still it is recommended that this second approach be taken. The information obtained can, in a relative manner, indicate which machine performs better.

References for Chapter 6

1. Denning, P.J., "Third Generation Computer Systems", ACM Computing Surveys, Vol. 3, No. 4, December 1971, p. 175.

2. Kosmala, Stanten, and Daly, Task Report SA-101, Central Processor Operational Analysis, September 30, 1971.

3. Architectural Study for Advanced Guidance Computers, Part 2, prepared by CIRAD Corporation for USAF SAMSO, TR 71-6, February 5, 1971.

4. Kerner, H., and Gellman, L., "Memory Reduction Through High Level Language Hardware", AIAA Journal, December 1970, pp. 2258-2264.

5. Sugimoto, M., "PL/1 Reducer and Direct Processor", Proc. 4th National Conference, ACM, 1969.

6. Keeler, F.S., et al, Computer Architecture Study, USAF SAMSO, Report TR-240, October 1970.

7. Myamlin, A.N., "Computer with Stack Memory", Information Processing 68, North-Holland Publishing Co., Amsterdam, 1969.

8. Patzer, W.J., and Vandling, G.C., "Aerospace System Implications of Microprogramming", Air and Spaceborne Computers, Technivision Services, Slough, England, April 1970, pp. 87-97.

9. Wichmann, B.A., "A Comparison of ALGOL 60 Execution Speeds", CCU Report No. 3, National Physical Laboratory, Teddington, Middlesex, England.

10. Knuth, D.E., "An Empirical Study of Fortran Programs", Computer Science Department Report No. CS-186, Stanford University, Stanford, California, AD-715-513.

11. Husson, S.S., Microprogramming Principles and Practices, Prentice Hall, Inc., Englewood, N.J., 1970.

12. Burroughs Aerospace Multiprocessor, B-1637D, April 27, 1970.

Appendix A

Phase B North American Rockwell (NAR) Baseline System Summary

1. Introduction

The purpose of this appendix is to summarize the pertinent features of the NAR avionics system. All information presented here was found in the documents listed in the Bibliography, (7. of this Appendix).

2. General System Summary

The NAR baseline avionics system involves a simplex central computer with three standby units for redundancy. All communication with the Shuttle subsystems, such as GNC, will be through a command/respond data bus. The bus will interface with the subsystems by means of standard interface units (SIU).

The executive is based on a fixed sequence, synchronous task structure, which interleaves with the I/O. All input requested by the executive during a minor cycle will be saved for processing in the next minor cycle. A detailed scheduling algorithm has not yet been specified. A summary of the major points of the baseline system is presented below.

3. Hardware Configuration (IBM-Generated)

3.1 General Organization and Description

IBM proposes to have four central computers each having access to all of a shared memory hierarchy. These computers communicate with the other Shuttle subsystems by means of a data bus. Four computers satisfy the FO/FO/FS requirement. See Figure 1 for a diagram of the data management system (DMS) organization.

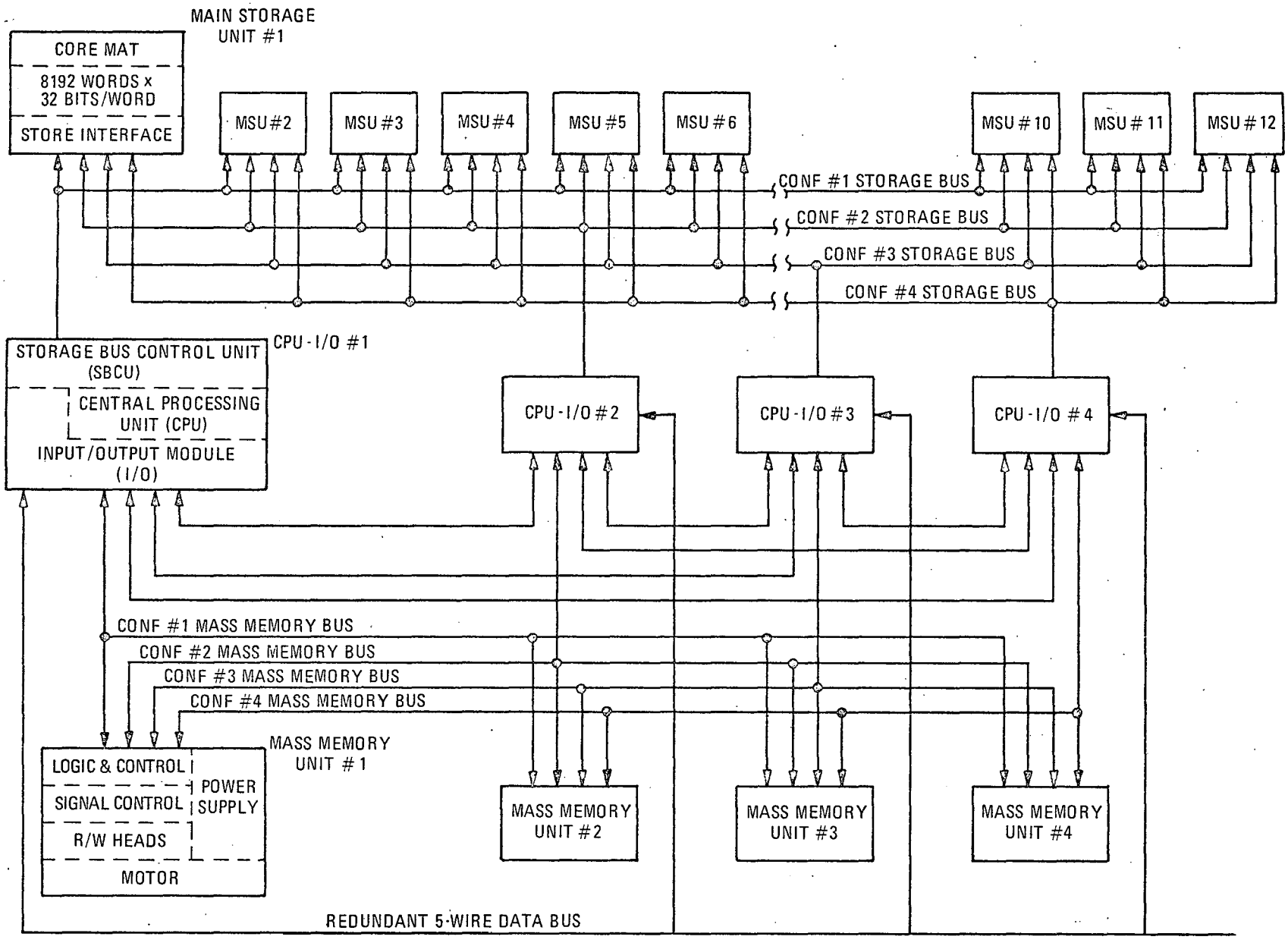


Figure A-1 NAR Baseline Data Management System

3.1.1 Central Computer Characteristics. The proposed computers are IBM 4 Pi-APs each having a single processor with an attached I/O module to provide interfacing and control for computer peripherals*. Table 1 provides a list of important computer characteristics.

Two computers are active during critical mission phases while the other two are inactive spares. The prime computer performs all computation and data transfers. The checking computer also executes the operational program for purposes of error detection.

Shared memory consists of 12 independent modules of 8192, 32-bit words each. Since IBM estimates a maximum of 35K words of storage are needed at any time for the operational program, 5 modules are assigned to each active computer. Thus, 2 modules are kept as inactive spares. All communication between shared memory and the CPUs is through computer storage buses and not the data bus.

During on-orbit phases (except a few minutes surrounding burns for orbit changes) the computer will be operated simplex. The address portion of the computer's instruction format limits the direct addressing capability to 32,768 32-bit words; therefore, a special instruction must be issued to reconfigure the addressing hardware to select a new four module group. Any word of a four-module group can be directly addressed and any module can be configured in any position of the group. An I/O instruction is used for address configuration.

As the flight phases change, new programs are called in from the mass memory unit (MMU) and are overlayed on the old phase routines.

Failure detection and isolation to a faulty computer or memory is accomplished by a combination of three elements: 1) software self-test, 2) software comparison of critical data between prime and checking computers, and 3) built-in test equipment (BITE).

---

* Processing rates in excess of 500,000 equivalent adds per second are expected.

Table A-1 Shuttle Computer Characteristics

Type	General purpose, stored program, parallel, binary
Organization	Fixed point, fractional, two's complement, single address, sequentially executed instructions
Data Flow	32-bit
Storage	Random access; core storage of eleven modules, each having 8192 words 36 bits long, and destructive readout
Storage Cycle Time	1.0 μs
Addressable Unit	16-bit halfword
Instruction Set	77 instructions tailored for applications such as guidance, navigation, and targeting
Instruction and Data Word Lengths	16-bit halfword 32-bit fullword
General Registers	Eight 32-bit hardware registers
Interrupt Facilities	4 externally controlled 11 internally controlled 15 levels of priority program mask control

Typical Execution Times

	Register to Storage (RX)	Register to Register (RR)
Load	2.0 μs	1.2 μs
Store	2.0 μs	
Add	2.0 μs	1.2 μs
Multiply	6.6 μs	6.0 μs
Divide	10.5 μs	10.0 μs
Shift(3 bits)	2.25 μs	
Branch	1.0 μs	1.0 μs

Program Loadable Clocks	Two 6-second program loadable countdown clocks
Input/Output	One 1-MHz serial, channel to the redundant data bus One serial channel to the mass memory unit

In addition to these automatic features the following program-controlled BITE features are used in conjunction with a computer self-test program; 1) a GO-NO GO (watchdog) counter, 2) force bad parity (storage and I/O), and 3) force single or multiple interrupts.

The self-test program checks the functional hardware for proper operation by using a predetermined sequence of instruction and selected bit patterns. Each machine is synchronized at the beginning of a minor cycle and compares the results of calculations for the cycle. A re-try procedure is included in the program to prevent intermittent failures from causing a hardware reconfiguration. Once a hard failure is detected, the self-test program (and BITE hardware) is used to isolate the faulty computer or memory. Checkout and fault isolation of all peripheral devices are under control of the computer subsystem.

3.1.2 Data Bus. The data bus consists of 5 twisted shield pair (TSP) transmission lines. SIUs will perform interfacing functions between the data bus and subsystems using the bus. Two pairs are active at any time, one to transmit the vehicle basic clock and command information and the other to carry the reply. The data bus subsystem will be able to couple I/O data to 300 SIUs at 1-megabit/sec over a maximum length of 600 feet. See Figure 2.

The data bus originates in the bus control unit (BCU), a part of the computer's I/O module. The BCU controls information flow between the computer and attached subsystems on the data bus. The data bus is time-shared by all stations on it under BCU control.

The SIUs will also convert conditioned analog and digital data signals into a standardized digital format and voltage level. These units will have provision to permit selected limit checking for signals received from LRUs. Then they will communicate with the bus as to the acceptability of the data. The SIUs have no data bus access except by BCU command. There is no provision for SIU-to-SIU transfer.

All subsystem information needed by the computer must be sampled at a specific rate. There is no provision for interrupts from subsystems. I/O operations are initiated by the computer, which permits the transfer of blocks of data words and chaining of data blocks with one initialization.

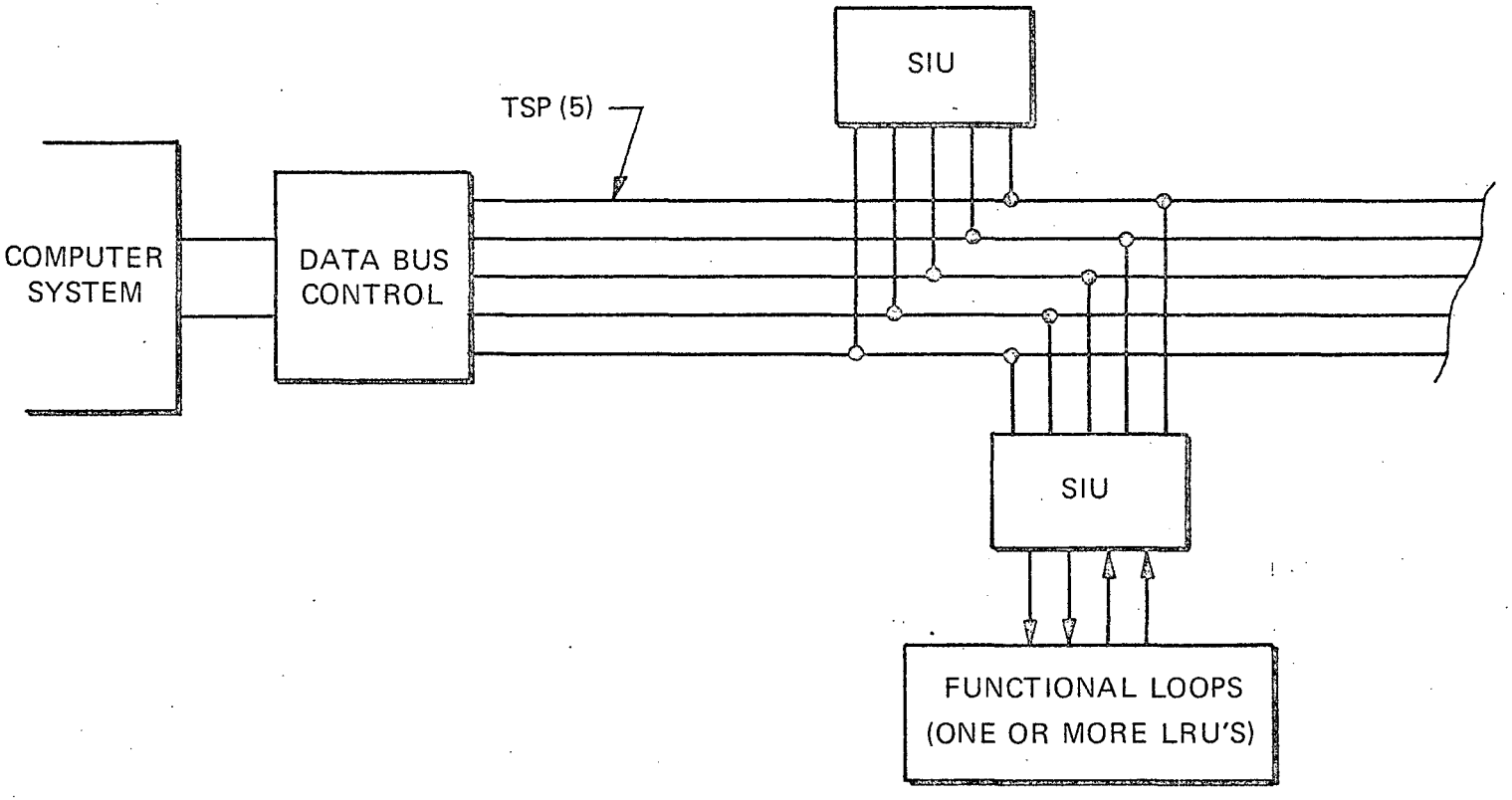


Figure A-2 Interface Block Diagram

3.1.3 Secondary Storage. There will be 3 drums serving as a mass memory unit (MMU). One will be active, and two will be standby satisfying the FO/FO/FS criterion. Each drum can hold 15,000,000 bits ≈ 400,000 words of information. New programs and data will be read from the drum into core as flight phases change. The drum will also be used to record data during the Shuttle flight.

The drums will interface with the computer through a memory bus so as not to tie up the data bus with large information transmissions. The MMU and core storage are the only subsystems interfacing with the computer not on the data bus. The MMU is capable of transferring data in or out at a rate of 3.2 MBPS.

4. Software Requirements

4.1 Executive System Organization

A synchronous executive structure is proposed. All computer porgrams will reside in protected memory. If any program tries to write in protected memory, the computer enters a fail state. All variable data will be in unprotected memory. The flight programs are divided into executive and applications programs.

The following executive functions have been defined.

(a) Computer Synchronization. At every minor cycle the active and checker computers are synchronized. Any failure to meet the synchronization time requirements will result in a fault isolation routine being called.

(b) Program Control. Task dispatching and scheduling is predefined within a mission profile table. Any rescheduling is done prior to the next time slice activity sequence begins.

(c) Interrupt Supervisor. Masks the interrupts and samples the interrupt states on a periodic basis.

(d) System Services. Samples all input control media (control panels and switches) at a predetermined and constant rate.

(e) System Clocks.

(f) Common Subroutines. Including sine, cosine, arctangent, natural log, exponential, square root, inverse square root, vector dot product, vector cross product.

(g) Common Data Blocks. Maintenance of unprotected memory.

(h) Sequencing. Controls predetermined sequence events on a time or event basis.

(i) Telemetry Control. Tables of predetermined data to be sent.

(j) Main Storage Program Loader. Loads programs from drum into core.

(k) Check of Input from MMU. Does "exclusive or" sum of program loaded for error detection.

(l) Mass Memory Management. Controls storage allocation of data to be transferred from core to MMU.

(m) Critical Parameter Error Detection. Compares results of critical parameter calculations prior to outputting the critical parameters via the data bus.

(n) Input Data Limit and Discrete Checks. Checks selected input data against high and/or low limits. Also compares selected input discrete status vs. previously commanded status.

(o) Program Restart. Provides restart of computer program from the point of interruption after detecting and isolating a computer failure.

4.2 Applications Programs

The following are the primary areas for applications programs.

GNC
Display & Controls
Configuration Management
Checkout & Fault Isolation
Subsystem Management

4.2.1 Guidance, Navigation and Control Functions. The following functions, as a minimum, are included in this program.

(a) All program initializations

(b) Targeting

(c) Alignment and calibration of inertial system

(d) Powered and unpowered flight navigation

(e) Attitude reference and control

(f) Lateral and axial load and structural relief

(g) Engine commands

(h) Orbit insertion guidance

(i) Control orbiter engine and reaction control system thrust vector

(j) Perform autonomous navigation and update inertial reference using fix data

(k) Provide star catalog

(l) Calibrate inertial reference using trend data and Kalman filtering techniques

(m) Provide search and acquisition for optics tracking

(n) Perform relative rendezvous navigation and control for rendezvous with cooperative targets, day and night, with direct and multiple orbit techniques

(o) Provide for manual takeover by crew

(p) Maintain orbiter in a station-keeping attitude with respect to a target

(q) Provide vehicle control under all mission conditions and phases including boost, orbit, re-entry, and aerodynamic flight

(r) Provide cruise engine thrust vector and throttle control

(s) Provide cruise navigation and control utilizing airways ground commands

(t) Provide stable steering in all mission phases

(u) Provide for automatic landing or vehicle up to touchdown point

4.2.2 Display and Controls. Display skeletons and programs will be loaded from MMU into main core on decision logic from a mission phase or by operator selection. The program will provide a unique set of 2 dynamic displays, one semi-dynamic, one tutorial and 2 alphanumeric displays for each mission phase.

4.2.3 Checkout and Fault Isolation (COFI). This program allows for automatic switching of failed functional paths and for isolation of indicated failures to the LRU level. It is scheduled when a fault is detected. Thus, because of the synchronous executive structure, selected portions of other programs may have to be de-scheduled to allow sufficient execution time.

There will be 15 diagnostic routines providing for isolation and automatic switching of up to 250 functional paths. After lift-off, thrust, fuel flows, speeds, temperatures, propulsion and vehicle attitudes will be monitored as part of caution and warning logic.

In addition, subsystem parameters will be compared against upper and lower limits with limit changes restricted to one per mission phase. A non-compare will send a warning to the crew, and a diagnostic program will determine the failed functional path. Reconfiguration can then take place.

The COFI functions are:

(a) Obtain test point information,

(b) Compare test point data to specified limits,

(c) Monitor component status using trend analysis,

(d) Perform reasonableness tests,

(e) Utilize diagnostic routines to isolate the cause of failure,

(f) Perform prelaunch checkout prior to descent.

4.2.4 Configuration Management. This program determines proper system configuration as a function of failure indication, mission phases, sequencing and operator selection. This topic has been discussed above as a COFI function.

4.2.5 Subsystem Management. The following subsystems are supported via software in the central computer:

(a) Aero Surfaces

(b) Vehicle Structures

(c) Thermal Protection

(d) Main Propulsion

(e) Orbital Maneuvering

(f) Attitude Control Propulsion

(g) Air Breathing Engines

(h) Cryogenic Tanks

(i) Communications

(j) Electrical Power

(k) Hydraulic Power

(l) Environmental Control Life Support

(m) Flight Crew

(n) Payload

(o) Landing

(p) Docking

5. Size and Speed Estimates

Software memory size estimates are made for operational software for each mission phase and are presented in Table A-2. With a 25% margin the maximum estimated for any phase is

MISSION PHASE	DMC MEMORY REQUIREMENTS (32-BIT WORDS)								
	Guidance	Navigation	Control	Exec.	ID&C	COFI	Subsystem	System Tables	25% Margin	Total
Boost	--	1036	--	2880	3781	3468	2100	7941	5301	26508
Separation	--	1036	3263	2880	3781	3468	2100	7941	6117	30586
Insertion	1604	1036	2373	2880	3781	3468	2100	7941	6296	31479
On-Orbit Powered	1604	1036	2670	2880	3781	3468	2100	7941	6370	31850
Mode 1	3554	947	3444	2880	3781	3468	2100	7941	7029	35144
Mode 2	252	1922	3444	2880	3781	3468	2100	7941	6447	32235
Mode 3	2414	947	3444	2880	3781	3468	2100	7941	6744	33719
Stat. Keep/Dock/ Undock	--	947	752	2880	3781	3468	2100	7941	5967	29836
Reentry	577	1116	3263	2880	3781	3468	2100	7941	6282	31408
Transition	--	1116	3314	2880	3781	3468	2100	7941	6150	30750
Aircraft	1356	1002	2013	2880	3781	3468	2100	7941	6135	30676

Table A-2 DMC Memory Requirements

INTERMETRICS INCORPORATED • 701 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

approximately 36K of 32-bit words assuming a 75/25 ratio of short/long instructions.

Speed is estimated for each phase and included in Table 3. The maximum speed estimated is approximately 274K adds/sec for the landing aircraft mode of operation.

A detailed breakdown of the executive system size and speed estimates is presented in Table 4. An estimated size of approximately 3000 words and a duty cycle of 30K adds/sec results from this analysis.

6. Equipment Interfacing with DMS and Data Requirements

A list of subsystems interfacing with the data bus follows. Included is the number of parameters to be sampled and the sample rate.

(1) Communications; 183 parameters at 2/sec, 2 at 10/sec, 100 at 1/sec.

(2) Structures (Thermal Protection); 276 at 1/sec, 33 at 1/min, 85 at 1/5 sec, 56 at 10/sec.

(3) Orbital Maneuvering; 122 at 2/sec, 24 at 1/sec, 26 at 10/sec.

(4) Main Propulsion; 112 at 2/sec, 22 at 10/sec, 1 at 1/sec, 1 at 12/sec.

(5) Cryogenic Tanks; 150 at 2/sec, 5 at 1/sec, 41 at 1/10 sec.

(6) Environmental Control/Life Support; 108 at 2/sec, 16 at 1/5 min, 7 at 1/10 min, 24 at 1/30 min, 3 at 1/hr, 20 at 1/min, 7 at 1/50 sec, 3 at 1/10 sec.

(7) Hydraulic Power; 70 at 2/sec, 9 at 1/sec, 90 at 1/30 sec.

(8) Airbreathing Engines; 267 at 2/sec, 68 at 10/sec.

(9) Attitude Control Propulsion; 292 at 2/sec, 66/thruster firing.

(10) Power; 979 at 2/sec, 231 at 1/sec, 9 at 1/10 sec.

(11) GNC: 211 at 10/sec.

MISSION PHASE	DMC SPEED REQUIREMENTS (KAPS)							
	Guidance	Navigation	Control	Exec.	ID&C	COFI	Subsystem	Total
Boost	--	6.1	--	29.8	84.8	20.0	4.0	145.
Separation	--	4.1	41.2	29.8	84.8	20.0	4.0	184.
Insertion	5.0	6.1	96.6	29.8	84.8	20.0	4.0	247.
On-Orbit Powered	5.0	6.1	78.8	29.8	84.8	20.0	4.0	229.
Mode 1	1.03	0.82	41.2	29.8	84.8	20.0	4.0	182.
Mode 2	0.53	7.62	41.2	29.8	84.8	20.0	4.0	188.
Mode 3	0.50	0.82	41.2	29.8	84.8	20.0	4.0	181.
Station Keep/ Dock/Undocking	--	0.82	37.6	29.8	84.8	20.0	4.0	177.
Reentry	1.12	5.7	43.7	29.8	84.8	20.0	4.0	189.
Transition	--	7.7	46.1	29.8	84.8	20.0	4.0	193.
Aircraft	14.05	15.77	105.1	29.8	84.8	20.0	4.0	274.

Table A-3 DMC Speed Requirements

Module Iden.	Instructions	Calls/sec	Instr./call	Eq. Adds/sec	Total Memory
Interrupt Supr.	40	50	40	2000	25
I/O Supr.	172	50	172	8600	108
Task Supr.	32	50	22	1100	20
Task Dispatch	119	50	105	5250	74
Command Control	20	50	20	1000	13
Timer Routines	150	1 @ 75 50 @ 30	75/30	1575	94
Math Routines	600	--	--	--	375
Program Loader	625	1	275	275	391
Drum Access M.	500	4	300	1200	313
Data Bus A.M.	820	1 @ 325 50 @ 80	325/80	4325	512
Control A.M.	700	4	250	1000	438
Mass Storage Allocation	500	4	250	1000	313
Executive Ser. Cont.	30	50	30	1500	19
Sequencing	300	10	100	1000	188
TOTALS	4608			29,825	2880

Table A-4 Executive Requirements

INTERMETRICS INCORPORATED • 701 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

7. BIBLIOGRAPHY

1) Space Shuttle Phase B Subsystem Design Data Book, IBM No. 70-D33-0021, February 8, 1971.

2) Space Shuttle Phase B Software Specification (Preliminary), IBM No. 70-D33-0020, December 21, 1970.

3) AP-1 Object Computer Programmers Reference Manual, IBM No. 70-A56-100, December 4, 1970.

Appendix B

McDonnell-Douglas Aircraft Corp. (MDAC)
Phase B Baseline Avionics Systems

1. Introduction

This appendix presents a review of the pertinent features of the MDAC Phase B baseline avionics system with emphasis on the data management computer system. All information presented here was extracted from MDAC Final Report E0395, Space Shuttle Data Avionics Part I and Part II, dated 30 June 1971.

2. General System Architecture

The MDAC baseline avionics system architecture contains quad-redundant central computers which perform most vehicle computations except for main and jet engine control. Each of the central computers is performing the same computation at the same time so that any one of them can do the entire job and completely control the vehicle via all 4 of the data buses. The system control unit (SCU) selects the controlling computer during the time critical mission phases. The crew manually selects a computer during non-time-critical phases and can override the SCU at any time. The SCU receives self-test results from each computer and inter-computer voting data from each input/output control unit (IOCU). Each computer can communicate with each of the 4 vehicle data buses and receives all incoming data, but only one computer is in control at a time.

Each IOCU interfaces a computer to all 4 data buses. It also compares the results of its computers' output with the results of the control computer coming over the data bus. The 4 data buses interface with a common unit only at the IOCUs. Each component is connected to one of the 4 data buses through an area digital interface

unit (DIU). Redundant components are each connected to a separate data bus. All data buses operate at all times, with the data traffic on each bus varying with the mission phase and the quantity of equipment that is turned on. Each data bus is capable of handling up to one megabit per second of data in serial bi-phase Manchester coded format. The maximum required data rate identified to date is less than 200 kilobits per second on any bus.

3. Data Management Computer System

The data management system (DMS) consists of a central computer complex, four redundant data buses, two mass memory tape units, two maintenance recorders, and a number of digital interface units (DIU). It is illustrated in Figure 1. The central computer complex consists of 4 central computers each with a full operating memory of 65K, 32-bit words with an add execution time 2 μsec, 4 input/output control units (IOCU) to control the bus system, and an internally redundant system control unit (SCU).

A summary of the major characteristics of the computer proposed is presented in Table 1. All critical discrete commands are compared bit-by-bit in the input/output control units (IOCU) while they are transmitted on the bus. The responsibility of controlling the data bus belongs to the controlling computer. The only difference between a controlling computer and the others is that its output data is allowed to pass onto the bus through the closed output switch in the IOCU. The output switch is being controlled by the system control unit (SCU).

Each central computer (CC) is directly connected to an input/output bus control unit (IOCU). The IOCU controls four data buses and provides buffering for data transfer between the computer and the data buses. The CC initializes the IOCU via program control. The IOCU then performs the data transfer autonomously.

Upon completion of the input/output cycle, the IOCU notifies the CC and prepares for the next request. Each computer receives all incoming data but only the control computer transmits data to the bus. Each IOCU interfaces a CC with all four data buses. The data bus is capable of handling up to one megabit/second of data in a serial bi-phase modulated (Manchester coded) format using time division multiplexing.

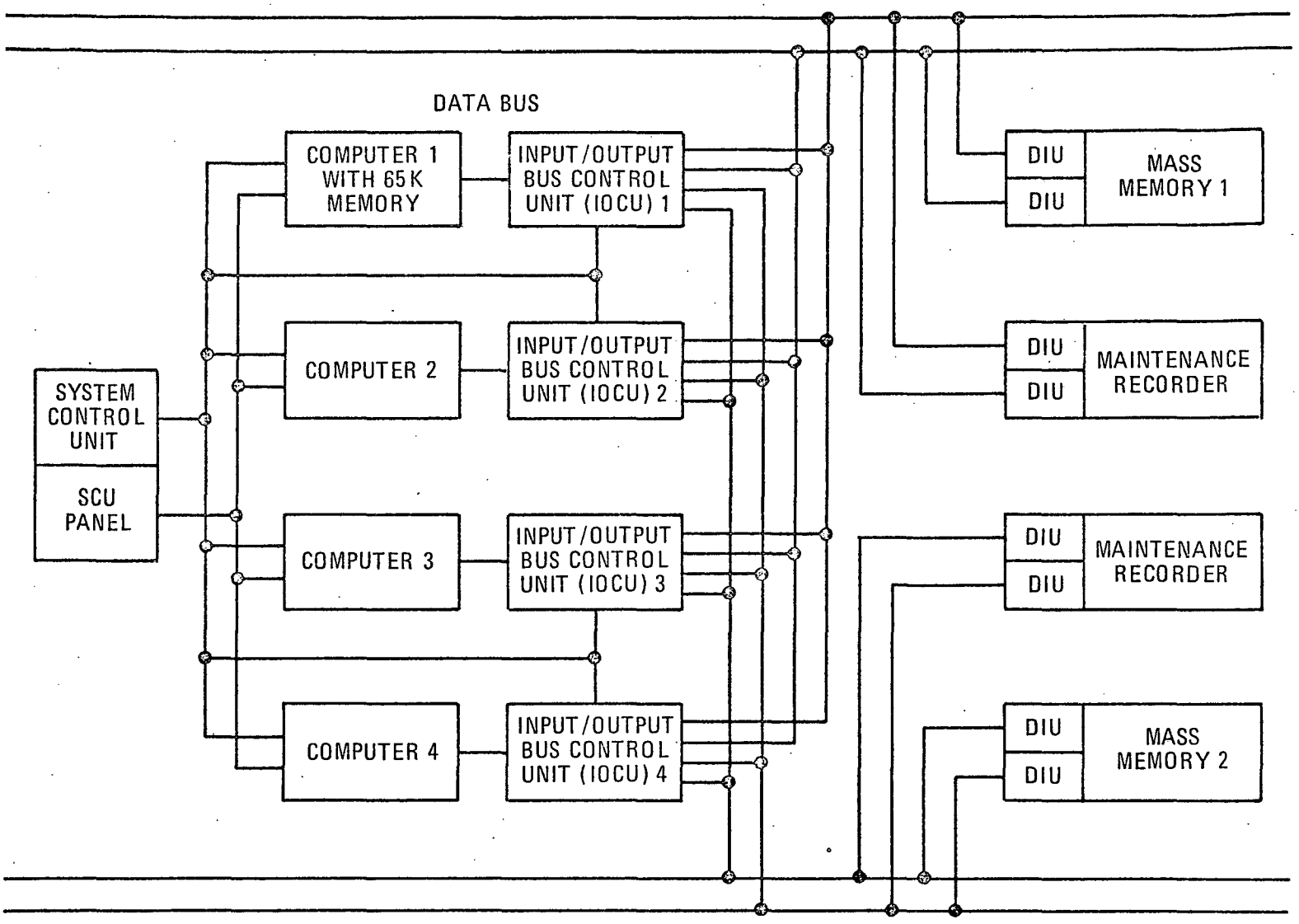


Figure B-1 Central Computer Complex Interconnection and Recorders

1. Processor Speed

   a. 2 μs add time
   b. 1 μs storage cycle time
   c. 4 arithmetic registers and 8 to 16 user registers

2. Internal Storage

   a. 65K word capacity
   b. 32 bits/word with 4 parity bits
   c. Floating point; 24 bit mantissa; 8 bit exponent in floating point precision

3. Instruction Set

   a. 65K direct addressing
   b. Indirect addressing (multi-level)
   c. Full indexing
   d. Large instruction repertoire with half-word processing

4. Other Features

   a. External clock (20 ms) via SCU
   b. Memory protection
   c. Power failure detect
   d. Fault detection

5. Input/Output

   a. Direct memory access channel (500 KHZ max.)
   b. SCU channel
   c. I/O error detection

Table B-1 Summary of Characteristics of the MDAC Phase B Central Computer

The data buses interface with the line replaceable units (LRU) via the digital interface units (DIU). The DIU provides multiplexing and demultiplexing functions, A/D and D/A conversion, excitation and stimuli to certain sensors,and collection of status data. These functions are needed to handle the many different Shuttle subsystems. There may be several LRUs for each subsystem. There are approximately 135 LRUs specified for the orbiter and 155 specified for the booster.

During critical mission phases each of the CCs is performing the same computation at the same time. Therefore, any one CC can perform the job of controlling the Shuttle via all four of the data buses. The SCU selects the controlling computer during time critical mission phases and the flight crew can manually select the control CC during the mission. In the automatic mode the SCU receives inter-computer voting data from the IOCU and combines this with SCU self-test data to determine the control CC. If an error is detected the failed CC is disabled and another CC takes control. The SCU also provides synchronization signals to the CC every 20 ms.

The two mass memory units (MMU) and maintenance recorders (MR) are connected to two data buses via two DIUs. The mass memory units (12×10^7 bits) store the prelaunch checkout and mission flight programs. The flight crew can reload a selected CC main memory with a copy of the flight program (about 40 seconds). Non-mission safety programs can be loaded during flight to circumvent problems of CC main memory overflow. Any CC can be loaded from either of the two mass memories. These recorders store the entire bus traffic including overhead during specified mission phases and store hard-wired command pilot and copilot voice data. Data bus traffic is also stored during any recognized equipment malfunction. The total storage time is about one hour.

4. Software in MDAC Baseline

4.1 General Executive Structure

The executive system is structured to operate on a synchronous basis, with a capability of handling demand events such as system reconfiguration. A synchronized major cycle/minor cycle is used in which the minor cycle

contains the high speed computations while the major cycle contains low execution rate functions. The minor cycle is executed every 20 msec. Within this time period all of the minor cycle computations are done plus one segment of the major cycle. A major cycle is completed every 50 minor cycles.

Each minor cycle is divided into two parts - high priority tasks and one of the fifty available major cycle segments. The same high priority tasks are performed every minor cycle. Usually different major cycle segments are executed every minor cycle. However, one segment may be continued to the next minor cycle. At this time only 25 time slots exist for handling demand processing such as reconfiguration and failure mode analysis. At the end of the minor cycle processing, each computer enters a mode which lasts until the next signal from the SCU. This function is sized at 3000 memory locations and demands 5 milliseconds/second during the ascent phase.

4.2 Data Bus Control

Task-lists are defined for the minor cycle and major cycle segments. Each of the entries in the minor and major cycle task-lists define a complete data transmission. Each task is divided into fields which define the address of the subsystem DIU, the function to be performed, the direction of the data transfer, the number of words to be transmitted, the address of the data, and the specific buses to be used. The initiation of the bus control program causes the minor cycle task list to be performed. As each individual task is completed a counter corresponding to the task number is incremented. By using this counter, the applications programs can determine whether or not the I/O transfer which they need is finished. When the minor cycle transmissions have been completed, a flag is set and the program advances to a list associated with a major cycle segment. The bus control program keeps track of the present major cycle segment number, in case it does not finish and must be executed as a part of the next minor cycle. Upon completion of this task list, the program increments the major cycle segment number and terminates. If errors occur during transmissions, an entry is made in the error status list. This list is used by applications programs to ascertain the status of the I/O transfer. If reconfiguration occurs due to a malfunction, the only changes necessary are modifications to certain entries in the list.

4.3 Execution Functions

The following is a summary description of other areas of the executive system.

a) Computer Self-Test. This program executes once per major cycle and tests the CC's internal logic. A self-test failure causes an immediate reconfiguration to another CC.

b) Mass Memory Control. The mass memory control programs consist of a bootstrap, I/O driver, and a block select and verify routine. The bootstrap acts to load the I/O driver and the block select and verify routine. The I/O driver governs the transfer of data between the CC and the MMU.

c) Mathematical Functions. The following functions are required: sine/cosine, arcsine, arctangent, square root, root-sum, matrix multiply, matrix add, matrix inverse, e^{-x}, natural logarithm. Part of the report mentions that it would be very efficient to implement the mathematical functions in the CC hardware.

4.4 Applications Software

A modular software design approach is proposed in which modules are defined according to their mission utilization. The total software program consists of a central module and a set of mission related modules to provide independence and flexibility. Mission modules are stored on the mass memory as an entity from which the operating memory is loaded prior to entry into a phase.

A detailed function design description for application software is provided in Section X of MDAC Report E0395. Although this information is not included in this report, Table 2 illustrates the modular organization and functional breakdown.

5. Size and Speed Estimates

A summary of the size and speed estimates for application and executive system software is included in Table 3.

The total size estimated for the operating memory is 66,430 locations. This figure is reduced to 49,823 by

MODULE	FUNCTIONAL CAPABILITIES
Central	*Executive (master controller, common task lists, conputer self test routine) *Data bus control *Computational subroutines *Mass memory control *Reconfiguration management *Sensor processing *On-orbit attitude and translation control *Non-mission phase related display and controls *Non-avionics subsystem servicing
Navigation	*Powered flight mode *Coast mode *Autonomous state vector update *Relative motion *Ground aided
Ascent	*Ascent guidance *Ascent-abort guidance *Main engine thrust command *Main engine gimball control *Ascent CRT data display lists
Orbital Phasing	*Orbital phasing ΔV calculations *Execution of on-orbit ΔV maneuvers *Maneuver CRT data display lists
Rendezvous	*Closed loop rendezvous guidance *Docking *Station keeping *Rendezvous CRT data display lists
Reentry	*Reentry guidance *Reentry flight control *Reentry CRT data display lists
Landing	*Terminal area guidance *Transonic/subsonic flight control
Off-Line Utility Modules	
Ascent Targeting	*Calculation of steering aim points, launch time for specific mission input data
Cruise Route Select	*Calculation of flight plan for ferry flights
Retrograde Time Determination	*Calculation of retrograde times for selected landing sites
Ferry Guidance	*Path control for ferry flight operations
Prelaunch Guidance and Navigation Initialization	*IMU Calibration *G&N Initialization *IMU Alignment *Subsystem Sequencing
Main Propulsion Prelaunch Servicing	*Fuel loading *Fuel venting *Fuel topping

Table B-2 Software Module Definition

Table B-3 MDAC Orbiter Software Requirements

MAIN MEMORY ITEMS	PROGRAM SIZE (LOCATIONS)	EXECUTION TIME (MSEC/SEC)
* Executive control	3,000	5.0
* Bus control	3,100	150.0
* Computer self-test	2,000	16.0
* Mass memory control	1,800	-
* Computational subroutines	1,000	-
* Guidance	13,300	13.0
* Navigation	7,400	31.1
* Flight control	4,230	
Main engine gimbal control	(600)	85.0
Main engine thrust control	(500)	2.0
On-orbit attitude	(800)	50.0
On-orbit maneuver	(520)	26.0
Entry	(510)	47.0
Aerodynamic control	(800)	75.0
Overhead operations	(500)	
* Displays and control	12,000	
Display processing	(6,650)	90.0
Instruments and controls	(5,350)	11.0
* In-flight monitoring and reconfiguration	8,000	24.0
* Sensor processing	3,350	50.2
* Communication subsystem servicing	650	
equipment control and checkout	(400)	2.0
telemetry	(250)	1.0
* Vehicle subsystem servicing	6,600	
* Propulsion subsystem	(800)	10.0
* Hydraulic subsystems	(200)	1.0
* Payload subsystem	(2,000)	2.0
* Landing subsystem	(200)	1.1
* ECLS	(2,000)	4.0
* Electrical power	(1,000)	4.0
* Fire protection	(200)	1.0
* Lighting subsystem	(200)	1.0
TOTAL LOCATIONS	66,430	
32 BIT LOCATIONS (.75 TOTAL)	49,823	
MASS MEMORY ITEMS		
* Prelaunch targeting	1,000	
* Cruise route select	1,000	
* Retrograde time determination	1,000	
* Ferry guidance	350	
* Prelaunch vehicle subsystem software	8,000	
TOTAL LOCATIONS	11,350	
23 BIT LOCATIONS (.75 TOTAL	8,513	

( ) Designates subitem of total

multiplying by .75 to account for halfword processing. The ascent mission phase is judged as presenting the most stringent timing requirements. An estimate of 461 msec/sec is made allowing approximately a 100% margin on the timing available for worst case. This estimate is presented in Table 4.

6. Avionics Subsystems and Computer Traffic

Detailed descriptions of avionics subsystems and computer interfaces are included in the MDAC Final Report referenced in 1. An estimate of the maximum bus traffic to and from the computer is in for each major subsystem. Maximum traffic rate for individual subsystems on the bus is summarized in Table 5.

SOFTWARE FUNCTION	COMPUTATIONAL TIME MS/SEC
* Executive	5.0
* Data bus control	150.0
* Guidance	13.0
* Navigation	31.1
* Flight control	87.0
* Display and control	101.0
* Sensor Processing	50.2
* Communication Subsystem Servicing	3.0
* Vehicle subsystem servicing	
* Propulsion	10.0
* Hydraulic	1.0
* ECLS	4.0
* Electrical power	4.0
* Fire protection	1.0
* Lighting	1.0
TOTAL MAXIMUM EXECUTION TIME	461.3 MS/SEC

Table B-4 MDAC Orbiter Computer Timing Estimate

SUBSYSTEM	MAXIMUM RATE (KP_{ps})
G&N Sensors	33
Landing Aid Sensors	0.5
Control Actuators	20
Non-Avionics Functions	15
Electrical Power	
Hydraulic Power	
EC/LS	
Landing Gear/Brakes	
Cash Recorder	
Tankage	
Communications	10
Air Data Sensors	0.4
Flight Control Sensors	5
Crew Controls	5
Crew Displays	20

Table 5. Summary of Maximum Bus Traffic from the MDAC Phase B Avionics Design

Appendix C

Inflight Checkout of Avionics Equipment (Orbiter)

C1.0 Introduction

The following appendix is a synopsis of the Phase B inflight checkout approach for major avionic subsystems of the orbiter. The material is based on information obtained from the McDonnell-Douglas Phase B Avionics Final Report, Volume II. It is presented to indicate the scope of subsystems checkout and its impact on software requirements. Checkout divides into two aspects, ground and inflight. For the purposes of this appendix, inflight checkout occurs when a) the orbiter is "buttoned up" and depending on its own internal resources, free of umbilical cables, carry-on equipment, and other ground systems support; and b) the data management computer is under control of the flight executive using flight software. Also, inflight checkout does not depend on a telemetry link to Mission Control. Thus, inflight checkout is characterized by complete autonomy and 100% reliance upon onboard resources, hardware and software. Within these limits, inflight checkout of orbiter avionics can occur any time, but typically occurs

a) in circular earth orbit prior to return to earth;

b) in a transfer orbit on the way up to a space station rendezvous;

c) prior to any critical mission phase; and

d) on the hangar floor, the ramp, or the booster stack.

The total checkout concept is predicated on vigorous monitoring of subsystems. Heavy emphasis is placed on the use of operational sensors for all phases of the orbiter mission. In instrumenting the various subsystems, several basic factors were examined:

a) meansurement accuracy requirements;

b) calibration requirements;

c) usage of the specific parameters inflight and on the ground; and

d) redundancy requirements.

In the following text, thirteen (13) major orbiter subsystems are described in terms of their checkout requirements. These major subsystems are listed in Table C-1.

C2.0 Subsystem Inflight Checkout Requirements (Orbiter)

C2.1 Guidance and Navigation Checkout (LRUs)

C2.1.1 Inertial Measuring Unit

The onboard checkout of IMU consists of comparison of the respective outputs of the four units. The IMU signals from all four units are transmitted to the data management system via the data bus through separate area DIUs. The IMU signals are expected to be nearly identical from all units. These inputs are then processed for transformation of coordinates and application of scale factors. Subsequently the inputs are compared (rated) within the central processor. By a predetermined selection of weighting process, the valid inputs are passed on for use in the navigation equations. In the event of a deviation of one of the inputs, that input is rejected by the voter and is flagged as such to the crew via the caution and warning lights. This process is followed through the first two failures. After two failures, the voting process is no longer valid. To perform a valid vote, three inputs are required. The third input can be derived from the IMU built-in test equipment. If the built-in test equipment is not able to resolve the ambiguity, then reasonableness criteria will be applied. The test requirement is that four IMUs be operated in an active redundant mode for at least ascent and descent mission phases. Further, deletion of a fault shall result in a switching decision being flagged to the crew.

1. Guidance and Navigation Equipment (IMUs, Horizon Trackers, Star Trackers, and BITE)
2. Flight Control System
3. Communications and Navaids
4. Controls and Displays
5. Hydraulic Systems
6. Landing and Deceleration Gear
7. Environmental Control and Life Support
8. Propulsion Subsystem Group
9. Air Breathing Engines
10. Auxilliary Power Units
11. Electrical Power and Distribution
12. Thermal Protection System
13. Structures and Mechanical Systems

Table C-1 Major Orbiter Subsystems Requiring Inflight Checkout Software

C2.1.2 Horizon Sensor

The onboard checkout of the horizon sensor subsystem poses a different problem than does the IMU. This system has four single-degree of freedom edge trackers and two horizon sensors which establish the local vertical. The malfunction detection for the horizon sensors is handled in pairs, horizon sensor (1,2)(3,4),if these disagree, the sequence will be changed (1,3)(2,4) and so forth. The processing of sensor pairs will solve for the local vertical,and the comparison is made or voted. After failure of two sensors it is not possible to get a disagreement on the local vertical solution; therefore, reasonableness criteria with the platform and built in test equipment will be relied upon to identify the third failure. Horizon sensor inflight fault isolation is required.

C2.1.3 Star Tracker

The star tracker malfunction detection method is based upon comparison of the star tracker outputs. These are three-gimbaled star trackers, two located on one navigation base and one on the other. For calibration purposes, all three trackers will acquire the same star to verify the detector and optics. Subsequently, processed star tracker inputs will be compared or voted. Those trackers located on the common navigation base can be compared directly for gimbal angle readout while the single unit will be compared with its companion IMU. In-flight calibration of the star tracker is the most valid method of determining performance of this LRU. The three star trackers require capability for in-flight calibration and in-flight fault detection.

C2.1.4 Built-in Test Equipment (BITE)

Built-in testing is provided in each LRU of the guidance and navigation subsystem. It provides a discrete failure-present signal when the parameters monitored exceed preset values. The monitor signals are processed within the LRU and a single built-in test is derived. Test and calibration of the built-in test circuit is accomplished during bench test using the ground support equipment test connector. Limited built-in test-circuit testing is provided using stimuli which cause BITE circuitry to detect a simulated out-of-tolerance condition. Additional fault detection is carried out at the system level using voting/comparison and reasonableness tests that are processed within the computer.

C2.2 Flight Control System Checkout

C2.2.1 Actuator Control Electronics

Onboard status and failure reporting of the actuators is reported to the data management system through the DIU via discrete lines. First and second failures in the four channels are detected automatically,and the failed channel shut down. Third failures are detected by use of inline monitoring and digital computer logic. Channel failure detection in the LRU is tested with hardware, which simultaneously reports all parameters to the computer for fault isolation and crew notification. This technique relieves the computer of the burden of high repetition rate comparisons but permits isolating failures to an LRU. The monitoring configuration is implemented with four cross voter comparators. Voting of quad input commands occurs within the digital computer prior to transmission of the commands via the data bus. All four data buses transmit identical commands. A failure in the D/A converter is treated by the monitors exactly as a failure in the channel servo amplifier. The motor assembly for each channel contains a tachometer and a clutched brake. A voter selects one of the four tachometer signals in accordance with a selection rule such as "the least magnitude of the two middle valves". An inline monitor is included in addition to the cross voter comparator as a means of achieving the FO/FO/FS reliability. This means a discrepancy between the measured velocity (derived from the tachometer) and the computed velocity (indicated in an actuator) will trigger a channel failure indication. The combination of cross voter comparators coupled with line monitors and computer algorithms gives a 98% or greater probability of a flyable fail-safe system after third failure.

C2.2.2 Thruster Electronics

The built-in test function for the thruster electronics is based on test circuits packaged integral to the assembly and operated on command from the digital computer. The digital computer also monitors the test results and makes the pass/fail determination. The logic monitors between redundant valve control circuits all operate full time in flight, permitting a continuous test of these circuits. It is important to exercise all of the dual control and reporting circuitry independently and separately during the preflight test to establish the integrity of the interfaces for each thruster. It is also necessary to have an independent time base to check the timing circuits in the thruster electronics. These functions can be accomplished

with greatest safety, efficiency, and speed by the digital computer. Preflight test routines are stored in mass memory. Computer software for BITE control is simple, and consists of a list of test words to be sent to each of the LRUs in the thruster electronics and a list of proper test responses for each test word output. Where timing relationships are significant in the gas valve and spark igniter control circuits, the computer provides time keeping. The tests are devised to provide 95% probability of fault isolation to the correct LRU. Each thruster electronics assembly receives a test word input from the data bus via its DIU. These test words will be of two types. The first type is a normal computer command word (thruster isolate or on/off commands) which is decoded and routed in the normal flight hardware, thereby validating it. The second type is a special preflight test message which is issued by the DIU as a binary coded group of bi-levels. For an LRU of this complexity, five bilevels should be sufficient. Bilevel commands are decoded in a programmable read-only memory in the thruster electronics. The decoded commands operate logic gates and FET switches to accomplish the test commands; these may consist of analog signal injection, logic forcing, and dual channel inhibits. The response to these test stimuli are reported to the digital computer via the normal inflight reporting channels, thus providing positive verification of the integrity of these channels. The computer evaluates the response from all the LRUs involved in the function under test, and then makes a pass/fail/fault isolation decision based on its stored program. It then executes the next test. It is estimated that the entire thruster electronics subsystem can be completely tested in 90 seconds or less.

C2.2.3 Flight Control Sensors

The three axis rate sensors are checked inflight for proper operation by continuous monitoring of proper momentum and the presence of power and excitation voltage levels. The individual sensor status information is reported to the computer in the form of valid discretes. Two axis acceleration sensor inflight failure monitoring is achieved by computer voting of the redundant sensors in each axis. Temperature and pressure sensors (4) are voted by the computer for inflight checkout.

C2.3 Communications and Navaids Checkout

C2.3.1 UHF Equipment

Onboard checkout of the UHF equipment consists of monitoring the BITE module outputs by the data management subsystem. After combining the BITE logic, a discrete go/no-go indication is given to the DMS. A fault indication results in the powering down of the defective unit and activation of a redundant set.

C2.3.2 S Band Equipment

The S-band onboard checkout is accomplished using LRU built-in test equipment in a manner similar to that described for the UHF set. The LRU BITE monitor sends a discrete go/no-go to the data management system as an indication of equipment status. A fault detection results in deactivation of the failed unit and activation of a redundant set.

C2.3.3 Distance Measuring Equipment

The DME is inactive until orbit is achieved. Status assessment of the units is performed using BITE self-test diagrams just prior to deorbital commital. Go/no-go discretes are sent to the DMS. At re-entry, equipment status is checked with BITE prior to ground station lock-on and with a combination of BITE and operation checks after lock-on. A failed unit is deactivated and a good unit switched online until a single good unit remains. Station switching is performed with the single unit to achieve dual station tracking.

C2.3.4 VOR

VOR on-board checkout is performed in a similar way to that described for the DME. BITE discretes from the LRU BITE indicate that the equipment is operational or failed. Tests are performed in orbit just prior to re-entry commital and during re-entry just prior to acquisition of ground station signals. Switching of units is performed in case of a VOR failure.

C2.3.5 ILS

The ILS units are BITE tested during orbit just prior to re-entry,and equipment status indications are sent to the

DMS. During the landing phase a combination of BITE and comparison of output signals from the redundant units is used for checkout and fault determination. Failed units are switched off-line under command of the data management system.

C2.3.6 Radio Altimeter

Inflight checkout of the altimeter is performed with BITE and the data management subsystem prior to re-entry. During the landing phase, the altimeter is continuously checked by BITE and redundant unit output comparisons. Fault detection results in deactivation of the failed unit.

C2.3.7 Intercom

Checkout of the onboard crew station voice links will be accomplished through normal operational use.

C2.4 Controls and Displays Checkout

Both manual and automatic inflight check procedures are planned for the displays and controls. The automatic inflight checks fall into one of two categories: 1) LRU Built In Test Equipment (BITE), or 2) computer voting of redundant control outputs.

The software BITE program for control and displays will be executed at a minimum rate of once every second. A software diagnostic will test the validity of these operations:

1) instruction repertoire operations;

2) signal generator unit arithmetic and logic operations;

3) correct operation of all hardware requestors;

4) C/O and data bus interface; and

5) interrupt processing.

The diagnostic BITE program permits rapid identification and indication of machine abnormalities with a minimum impact on memory requirements. The detection of a fault by the software or hardware BITE will immediately send a malfunction code to the DIU. This failure status of LRUs is sent to the data management subsystem. The computer responds to these reports with appropriate power on/off control and interconnections of the LRUs.

C2.5 Hydraulic Onboard Checkout

The onboard checkout function for the hydraulic subsystem is implemented by the DMS. The data management system data processing is the same for both inflight and ground test. Checkout will be initiated from the crew station with system status verified by normal cockpit readouts. The verification of system functioning will be implemented in checkout software and system status compared to prestored limit values. Interface of sensor outputs are implemented via DMS and will be verified by automatic software check of the parameter limits. Redundant system switching to the standby mode will be automatically verified by switching value position indicators. The built-in tests have been incorporated to satisfy requirements such as system leak detection, flow rates, and filter status.

C2.6 Onboard Checkout Landing and Deceleration

The hydraulic circuits servicing the gears will require the same monitoring functions for leak detection defined for the hydraulic subsystem. Checkout monitoring of the gears requires verification of strut pressure, tire pressure, gear position as well as functions of the hydraulics. The hydraulic power requires switching of power circuits since the actuators are single chamber floating pistons. Failure of the first system to lower the gear requires that the second and third system be switched in. Given that the primary hydraulic subsystem does not function, a second and third system are switched in to an isolated chamber within the actuator. In addition, gear status is displayed to the crew via the DMS. The landing gear provisions include instrumentation of pressures, valve position, gear position, and lock indicators. The anti-skip brake system provides continuous monitoring of open and short circuits and an off-line press to test function for use in flight or ground operations. Built-in test equipment provides the data necessary for an off-line test. In addition, tire pressures and strut pressures are monitored for landing system status. The status, anti-skid system and landing gear position data is displayed to the crew. The software servicing the landing system performs the monitoring and supervision function. The BITE capability for the drag chute checkout includes monitoring of initiator circuits including the indication of a safe condition. In addition, the ability to automatically verify firing circuits will be built in.

C2.7 Environmental Control and Life Support Checkout

The flight crew plays an important part in the inflight checkout of the ECLS subsystem. The required response to many of the ECLS parameter variations is not time critical and therefore action can be taken at the crew's discretion. Caution and warning parameters are acquired via DIUs and limit checks in the central computer. The parameter limits are established based on the safe operating range. Redundant DIU inputs are provided for critical parameters. System monitoring of critical parameters is displayed to the crew on dedicated displays. Selected parameters can be displayed at crew option on the shared CRT display. Those parameters which are displayed on the CRT will have scale factors applied for unit conversion. A shared CRT display will have an attached microviewer which displays checklists and parameter lists. Electronic controllers within the subsystem are equipped with built-in test circuits which are activated by stimuli from the central computer. The subsystem controller will respond with a go/no-go signal output. Examples of the specialized control functions include the cryogenic heat exchanger, ground cooling heat exchanger, and radiator coolant outlet temperature. Instrumentation items such as temperature sensors and pressure transducers,interface with the data bus for signal conditioning and transmission to cockpit displays. Certain ECLS parameters are recorded in real time on the flight recorder and are used for trend data to be analyzed later. This trend data is useful in predicting incipient malfunctions of equipment prior to their occurrence. ECLS data that is recorded includes pump and fan "delta p", fan and pumps "in operation", and radiator coolant outlet temperature. In addition, ECLS parameters that are associated with caution warnings are recorded.

C2.8 Propulsion Subsystem Group Checkout

Inflight checkout is a recording of flight performance to assess malfunctions and/or off nominal operation. Recording of the valve position indications will provide key malfunction detection as well as provide the primary signal for backup system or isolation valve actuation.

The propulsion subsystem checkout provision has included the monitoring provisions for establishing subsystem functional operation and redundancy verification. The philosophy followed on the use of parameters, pressure, and temperature, is to use an analog device which will

1) yield a quantitative reading,

2) provide redundancy which allows direct comparison of more than one transducer output, and

3) be located in the system to check redundancy.

Main propulsion engine controllers presently have the capability to perform an internal checkout. It has been additionally recommended that for fault isolation, it is necessary for the onboard computer, in lieu of ground support equipment, to provide a series of commands which are capable of discretely exercising the various valves and solenoids to check out the main engines. In addition to these commands, the DIU interface with the main engines can be checked out independently.

The propellant distribution system has specific check points on each dual seal joint to detect leakage. These parameters are analog measurements which permit quantitative evaluation of the sealed integrity. The fuel loading instrumentation, i.e., capacitance probes, level sensors, and zero g gauging, have built-in test capability. Consistent with the philosophy previously stated, the operational checkout of propulsion systems will consist of an inert gas flow with valve sequences verified by position indicators. Interface test stimuli for the chamber pressure indicator, ignitor operation, and current and flow meter units make up the BITE in the Propulsion Subsystem Group.

C2.9 Air Breathing Engines Onboard Checkout

T.B.D.

C2.10 Auxilliary Power Units Onboard Checkout

T.B.D.

C2.11 Electrical Power and Distribution Onboard Checkout

The checkout of the various elements of the onboard electrical system would utilize the onboard data management system for data acquisition and display. In the electrical subsystems, the crew participation is required as the overall supervisor. The crew display and controls are used for voltage, current, and cross tie of systems external to internal power transfer. To facilitate these functions, there will be several parameters which will be provided to the caution and warning function. Fuel cell parameters such as fuel pressure, temperature, and voltage will be used for EC and W functions. The onboard capability within the electrical system permits

autonomous operation and checkout of the electrical power subsystems. The monitoring of the power generation, distribution, and fault detection is built into the basic system. The AC generator controller has built-in test equipment for accomplishing the following functions:

1) over and under voltage detection,

2) regulator function,

3) line to neutral fault detection,

4) frequency monitor, and

5) power conditioner.

The operation of the controller is verified under control of the DMS. The internal stimuli which must be simulated include frequency, over and under voltage, phase imbalance, and voltage of the power conditioner. These checks are not performed while the unit is online, rather they will be performed prior to loading the generator. Online checkout of the controller will consist of monitoring the status of the control circuits. Where feasible the internal parameters will be combined to output a logic level indicating the status of controller operation. The output of generator voltage and current are displayed for the crew. The fuel cell status monitoring function is a hybrid function; using transducers for pressure monitoring, temperature and voltage monitoring will be processed by the DMS for display on the caution and warning panel. Functions which are associated with catastrophic failure modes will be hardwired, viz, stack pressure and output voltage. In addition to C and W functions, the fuel cells status is also displayed on dedicated voltage and current meters. Switching of fuel cells will be a crew initiated function. The power control function has several built-in provisions for the purpose of malfunction detection and inflight/ground checkout. The presence of voltage on all power buses is monitored. Commands to the RPC are fed back to the data management subystem so that a running power system configuration is available via CRT, RPC trip commands that have been instrumented to indicate over current sensing or drop out of an RPC. In the ground test a trip stimuli provision has been included. To test remote RPCs the on command is issued; RPC on is confirmed;and a trip signal for test issued. This is an automatic sequence which will be carried out by the DMS. The onboard software has provided for isolation of redundant power system functions. Provisions for diagnosis of subsystems faults have been included in the subsystem software. With the exception of the hardwired control circuits discussed above, all RCPs receive control signals via the data bus regardless of the origin of the command, and all instrumentation is available from the data bus.

C2.12 Thermal Protection System Checkout

Inflight checkout of the TPS consists of monitoring and recording data from the TPS instrumentation. There is no switch or changing of panels while in a flight condition. The data collected during the flight will be analyzed on the ground to assess effectiveness of the TPS to protect the vehicle structure. On condition monitoring of calorimeter values will be made in case a change in re-entry conditions is necessitated because of excessive heating rates.

The sensor data is routed through area DIUs to the data management system for subsequent recording or on condition use. The orbiter instrumentation is similar to the booster except for quantity. There are sixty TPS parameters identified for the operational orbiter. Inflight provisions are not needed for the purging system.

C2.13 Structures and Mechanical Systems

The landing and deceleration subsystem is powered down during orbit but is in-line during horizontal flight. The following checkout is performed:

1) CRT readout, tire pressure, and strut pressure,

2) indicator lights out for gearup and lock positions,

3) visual check of hydraulic pressure at brake valves,

4) pilot initiated test of anti-skid circuits and light indication gives test OK.

Appendix D

Fixed Point Versus Floating Point Arithmetic

1. Introduction

The purpose of this appendix is to review the problems of fixed point scaling and discuss the advantages of floating point for guidance and navigation programming. A top level analysis of the Apollo Guidance Computer fixed point arithmetic and the applicability and benefit of the memory savings through floating point is estimated.

2. Fixed Point Arithmetic

If A is the real world quantity and $\tilde{A}$ is the scaled quantity, then $\tilde{A}$ is represented on a computer in fixed point by a fixed number of bits defined by the word length where

$$-1 < \tilde{A} < 1 \quad .$$

A is related to $\tilde{A}$ by a scale factor P by

$$A = \tilde{A}\, 2^P \quad . \qquad [1]$$

It is convenient in binary computers to take the scale factor as a power of 2. This is useful for multiplication and division since scaling can be performed by shift operations on $\tilde{A}$ It should be noted that if $|A| < 2^P$ then A can be represented on a word length of n bits with a maximum error of 2^{P-n-1}.

In order to examine the scaling procedure consider the equation:

$$A = BC + D/E \quad . \qquad [2]$$

Through analysis of the precision and range requirements of A, B, C, D, E scale factors must be defined. For example,

let (P, q, R, S, t) be the scale factors respectively.

Equation [2] is rewritten in scaled form as

$$(\tilde{A}\ 2^{P}) = (\tilde{B}\ 2^{q})\ (\tilde{C}\ 2^{R}) + [(\tilde{D}\ 2^{S})/(\tilde{E}\ 2^{t})] \quad . \qquad [3]$$

Or clearing scale factors

$$\tilde{A} = \tilde{B}\tilde{C}\ 2^{q+R-P} + \frac{\tilde{D}}{\tilde{E}}\ 2^{S-t-P} \quad . \qquad [4]$$

The programming of this equation can then be accomplished with the scaling performed by the appropriate shifts. The analysis would require insuring that intermediate calculations maintained the necessary precision and held within range of the scale factor. This can be extremely difficult for complex matrix and vector equations. For example, formulating cross product of two scaled vectors such as

$$\overline{R} \times \overline{V} = \overline{H}$$

may require reformulation with unit vectors in order to maintain direction accuracy.

Division in general requires variable adjustment of the denominator and consequently its scale factor prior to division, i.e., $D < E$ for division. This is even more difficult since it demands some knowledge of the minimum value of E, whereas the scale factor only implies the maximum value.

3. Fixed Point Problems

A. The first and foremost fixed point problem is that the scaling is left to the programmer and is not done automatically by the computer hardware. Any time a process is accomplished by programmers it is subject to human failure. A compiler, of course, could help. To review, equation [1] shows the initial step that a programmer must take. He must establish some relationship between the scaled quantities within the computer and its real world counterpart measured in some engineering units (i.e., the scale factor). This factor must be chosen to enable the range of the quantity to be contained within the word length of the computer. Once P is selected, the maximum and minimum values of A which can be represented in the computer are readily determined. A real world computation as shown in equation [2] is replaced by its computer equivalent in [3] and the scale factors can all be cleared as shown by equation [4]. With floating point, equation [2] can be implemented directly, assuming minimum analysis to insure that single precision (number of places) is adequate.

B. The second problem is that fixed point scaling is error prone for the following reasons:

1) The programmer must deal with twice as much information. Each variable has an associated scale factor. Consequently, more chance for coding errors exists.

2) Extra scaling information is confusing, non-intuitive data. In fact, it is often difficult to predict sign correctly.

3) Analysis that is necessary for the selection of scale factors cannot be done by each and every programmer. Yet scaling of intermediate variables, which in many cases is a more severe problem than primary variables, is often never even considered other than by individual programmers.

C. The third key item is that the dynamic range is linear rather than the log scale of machine floating point. Of course, it does not have to be this way. It is possible to do floating point in software even though the machine instructions are not provided on the computer. However, the time used by such a process usually influences the decision against this procedure.

D. Fixed point arithmetic introduces a significance problem. A scaled fixed point number may have several leading zeroes, say k. For a word length of n bits, this fact leaves a maximum of n-k, instead of n, significant digits. Therefore, k information digits can be permanently lost.

E. There are still many other problems related to fixed point usage in aerospace programming. An especially important point is that even though the programming might be accomplished in fixed point, the simulators and debugging have to all be done using raw or unscaled data. When the machine does the scaling, scaled data is available at dump time so that the debugger is looking at meaningful scaled numbers. In the fixed point case, the individual must consult tables of scale factors and must apply it to numbers printed on a listing; and he can only do that if he is sure which data it is. If he does not know what the data is, then he cannot know what scale factor to apply, and the results are even more mysterious. Points like these should not be underestimated.

A summary of the major impact of simulation and testing is:

a) More tests have to be run because many fixed point variables may exceed the limits of their linear region, i.e., overflow. System behavior may change as each one ceases its linear performance. It is difficult to determine how many and which tests should be run; consequently, a "shotgun" approach is often used.

b) The simulators and support-software may recognize only scaled variables and require scaling to be applied on the way in and out of digital simulation runs. This causes many improperly set up runs to be wasted.

c) Debugging still must be done with raw data, viz. computer values. This necessitates manual application of scale factors in order to interpret the results and impedes debugging efforts. More controlled outputs could be converted automatically via EDIT programs, but these EDITs had to be written and kept up to date and modified whenever the scaling changed resulting in more effort in utility software.

d) Floating point is more tolerant of nominal conditions because of greater dynamic range. More latitude puts a higher level of confidence in proper performance.

e) Programs need to be written to simulate the numerical effects of fixed point computations, including overflow limitations. In any case, there must be numerical analysis. It is much simpler when floating point data is under study.

f) Fixed point calculations involve too much complexity at the coding level (often tricky) making it difficult for "eyeball" verification.

4. Analysis of the Apollo Guidance Computer

A top level analysis of the coding in the Apollo Guidance Computer was performed to determine the effect on memory requirements if floating point was available. The approach was to estimate the amount of "interpretive" coding in the AGC since it was predominantly where most scaling was performed. Several limited examples or test cases were programmed using a general floating point capability and comparisons against the fixed point coding were made.

4.1 AGC Code Breakdown

Figure 1 is a breakdown of the code produced for the Apollo Guidance Computer (AGC) in the program Colossus as used on one of the manned Apollo missions. It classifies the programs into one of eight types described on the right of the chart. The total number of words of program in fixed memory needed for each of the types is shown in the bar graph. The values range from about 1200 to almost 13,000 words. Each class is further subdivided to indicate whether the code is interpretive or basic. Basic coding is the coding that was done in the basic machine instructions of the AGC (a very limited instruction repertoire, perhaps twenty in all).

Interpretive means the program was written in the higher level language that was implemented via an interpreter in the AGC. The interpreter language had instructions, such as VECTOR ADD, VECTOR MATRIX MULTIPLY, SINE, VECTOR CROSS PRODUCT, etc. Because of these features it was well suited for the scientific type calculations needed on the AGC. The price paid for this seeming power was in relatively long execution time. Basic coding produced bulkier code but allowed the most in speed gains to be extracted from the AGC.

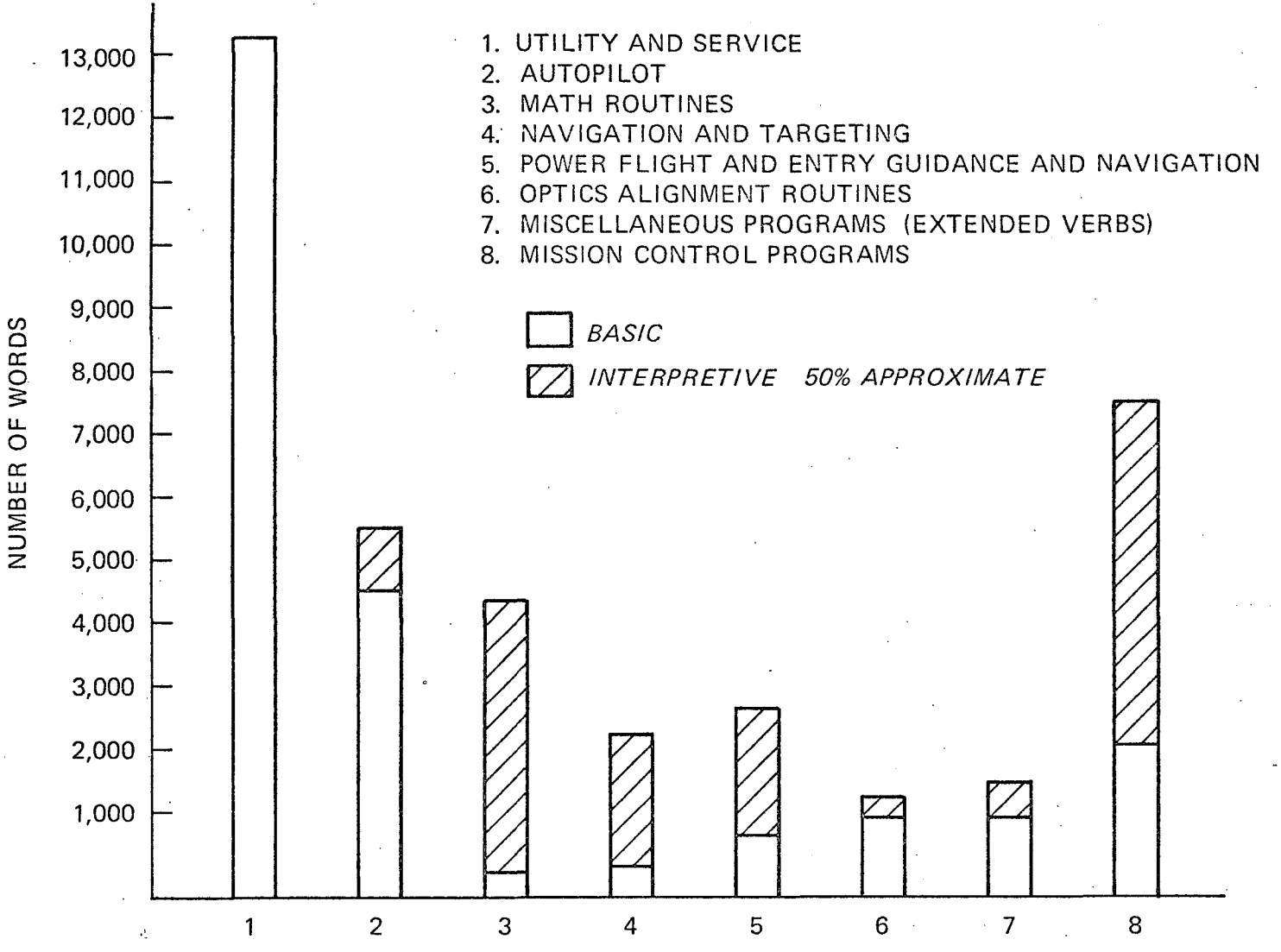


Figure D-1 Distribution of Apollo Guidance Computer Code

The intent of this exercise was to estimate the area of applicability of floating point and the saving that might have been realized by using it. Interpretive code was the primary one on the AGC for floating point, because of the scientific nature of interpretive calculations. For purposes of this report, the areas that use basic AGC code were assumed to have negligible benefit from floating point code. The organization of code of the AGC shows that approximately half the computer was programmed using the interpretive language and half was done in basic.

The conclusion drawn is then that floating point would have some suitability in about half the areas of programming on the AGC. Actually, this is a perhaps somewhat conservative estimate for the following reason. Class 2, the autopilot and maneuver programs, show up as about 80% basic. The reason in this case for the overwhelming preponderance of basic code was not that the nature of the problem was ill suited for interpretive and thus floating point calculations, but rather the time criticality of the problem did not allow the luxury of interpretive coding. The digital autopilots were forced to do scientific type calculations in basic language because the high frequency of the calculations dictated a very restrictive time budget. If a faster computer were postulated so that the time bind were not present, Class 2 probably would have been 70% to 80% interpretive. Be that as it may, the figures show a fairly widespread area of applicability for floating point.

As a side-light, an interesting observation can be made by examination of the classifications of this chart. It is interesting to speculate on where the most time, effort and dollars were spent in programming the AGC. Class 1, the utility and service programs, was almost 13,000 words of basic machine code which was produced by a relatively small group of people and was inexpensive in cost per word. The Class 1 programs were defined and written early in the game and did not change very much but had a heavy volume of users. That is, everyone had to use those system programs during testing periods. Items like the executive, the interpreter, and pinball (the keyboard and display I/O program) became highly reliable at relatively low cost.

At the opposite end of the spectrum were the mission control programs and extended verbs. These were coded by many people and done very late in the schedule. They were also always changing for various Apollo missions. They suffered in that they were only required for special applications without widespread usage by the general AGC user. Thus, they were the most expensive to produce and were also the least reliable. It is significant to observe that these troublesome areas are candidates for heavy floating point utilization. That is, the cost for changes might be improved by floating point.

4.2 Coding Examples - Fixed vs. Floating

Three representative examples of coding done in the AGC in interpretive mode were examined to see what changes could be made if one hypothesized a floating point environment. The assumption was made that nothing else was changed and the interpreter was the same except that it was a floating point version. Then, because of floating point, certain instructions could be either deleted or modified. A measure was made of the savings in time that was realized by counting the instructions that would no longer have to be executed, and of the savings in memory by counting the space occupied by both the instructions and their associated addresses that were eliminated. Addresses were stored in different words in the AGC interpreter, and instructions were packed two per word. For example, look at Case A in Figure 2. This sample was 26 instructions, relatively small but reasonably general and applicable to the scope of this study. Eight instructions were saved when the switchover to floating point was made. This segment of the program occupied 31 words of memory. After modification, it only occupied 20, resulting in a savings of eleven words, which consisted of seven address words and four instruction words. The eight instructions previously mentioned, packed two per word, comprises the four instruction words. A similar analysis revealed other savings in examples B and C. The resulting samples showed that 24 instructions were saved out of 101 which is a 24% total. Thus, if all instructions took equally long to execute, it would be expected that these sample programs would run 24% faster because only 77 instructions need to be executed instead of 101.

The memory saved is more clear cut. The total memory used was reduced from 121 to 97 words of memory, a 20% savings. If these samples are representative (Case A was very favorable, Case B unfavorable, and Case C neutral), in extrapolating these statistics, we could anticipate a 20% saving in memory for that portion of the AGC where the interpreter was used, which was shown to be approximately 50% of the memory of the AGC. All together, it is anticipated that the savings could amount to 10% of memory. In the total AGC case this was about 3600

CASES	INSTRUCTIONS	INSTRUCTIONS SAVED	TOTAL WORDS	WORDS SAVED
A	26	8	31	$7 + \frac{8}{2} = 11$
B	34	4	42	$0 + \frac{4}{2} = 2$
C	41	12	48	$5 + \frac{12}{2} = 11$
TOTALS	101	24	121	24

Instructions saved: $\frac{24}{101} = 24\%$

Memory saved: $\frac{24}{121} = 20\%$

Figure D-2 Code Saved by Elimination of Fixed Point Oriented Instructions

words of memory that might have been saved if there had been floating point. Granted that this is a difficult figure to prove, it is estimated that an overall 10% number is quite realistic.

5. Advantages of Floating Point

A summary of the main advantages of floating point are listed below. They are typically the disadvantages of fixed point discussed previously.

a) Scaling analysis for fixed point is minimized with floating point. One of the key tasks in the design of airborne software is eliminated.

b) Simplifies program development and improves reliability. The programmer does not need to be concerned with detailed accounting of binary point location during program development, and less errors are probable. The program complexity is reduced by removing the tedious scale factor manipulations and adjustments associated with every math operation.

c) Eliminates errors. By avoiding fixed point scaling and the attendant errors associated with it, the program code is less prone to data interface mismatches. A fixed point program is more expensive to debug statically. In addition, a large percentage of the program errors during system integration is removed.

d) Simplifies program change and maintenance. The more complex a program, the more expensive it is to change. Changing a fixed point program that is in operational status requires reinvestigation of the scaling factors and the impact of the proposed change on the scaling. In addition, any change must be checked out exhaustively with a high volume of input data to assure that the program's mathematical integrity has not been impaired.

e) Increases the feasibility of utilizing higher level languages for applications software. The major failing of the majority of higher level languages in aerospace applications has been that the programmer must operate in a psuedo-assembly language mode, keeping track of the contents and the binary point location of each arithmetic register.

Appendix E

Floating-Point Wordlength for Shuttle

[This appendix is the text of Intermetrics Internal Memo #15-72, written by Dr. Neal A. Carlson.]

1. Introduction

The problem addressed here is the floating-point wordlength required in the Shuttle on-board computer for guidance and navigation functions during rendezvous and landing. More specifically, the question is whether a 24-bit mantissa and 8-bit exponent (both including sign) are adequate to meet the computation requirements. In Sections 5 and 6, those computations requiring the greatest precision are identified, and where the prescribed word-length is marginally adequate, alternate computational formulations to ease the requirements are suggested.

2. Numerical Range

Consider a floating-point number to be represented in binary form as follows,

$$N = m\ 2^n \tag{1}$$

where the 24-bit mantissa m and 8-bit exponent n are given by

$$m = \pm\left[\frac{1}{2} + \frac{a_2}{2^2} + \ldots + \frac{a_{23}}{2^{23}}\right], \qquad a_i = 0 \text{ or } 1 \tag{2}$$

$$n = \pm\left[b_0 2^0 + b_1\ 2^1 + \ldots + b_6\ 2^6\right], \quad b_i = 0 \text{ or } 1 \tag{3}$$

We note that the ranges of values for m and n are

$$1/2 \leq |m| \leq 1 - 2^{-24} \approx 1 \qquad \text{or } m = 0 \tag{4}$$

$$0 \leq |n| \leq 2^{7} - 1 = 127 \tag{5}$$

Hence the maximum and minimum (non-zero) magnitudes of N are

$$\begin{aligned} |N| &\leq 1 \times 2^{127} = 1.7 \times 10^{38} \\ |N| &\geq .5 \times 2^{-127} = 0.3 \times 10^{-38} \end{aligned} \tag{6}$$

3. Roundoff Errors

Errors are introduced during computations by rounding each mantissa to 23 numerical bits. The rule for roundoff is generally this: if the 24^{th} numerical bit is 1, increment the 23^{rd} by 1 and drop the remainder; if the 24^{th} bit is 0, retain the 23^{rd} as is and drop the remainder. Hence, if m represents the "true" value (i.e., infinite precision) of the mantissa, the rounded 23-bit mantissa m' is in error by

$$\Delta m = m' - m = 2^{-23} z \tag{7}$$

where z is a random number uniformly distributed between -1/2 and 1/2. Note that

$$z_{max} = .5; \quad \sigma_z = 1/\sqrt{12} = .29 \tag{8}$$

and that

$$2^{-23} = .84 \times 10^{-7} \tag{9}$$

The maximum value and standard deviation of the mantissa roundoff error are thus

$$\Delta m_{max} = .5 \times 2^{-23} = .42 \times 10^{-7} \tag{10}$$

$$\sigma_m = .29 \times 2^{-23} = .24 \times 10^{-7} \tag{11}$$

Likewise, the maximum value and standard deviation of the roundoff error in the floating point number N are

$$\Delta N_{max} = .5 \times 2^{n-23} = .42 \times 10^{-7} \text{ N/m} \quad (12)$$

$$\sigma_N = .29 \times 2^{n-23} = .24 \times 10^{-7} \text{ N/m} \quad (13)$$

where

$$N/m = 2^n = |N| \times (1 \text{ to } 2) \quad (14)$$

Hence, a 23-bit mantissa provides 7 decimal places of precision.

4. Growth of Roundoff Errors

Floating point numbers resulting from a series of numerical operations in general will not be accurate to 23 binary places (or 7 decimal places), since roundoff errors accumulate during such operations due to statistical addition of individual errors in the numbers involved, plus roundoff or truncation of each result.

4.1 Primary Operations

Roundoff errors in the result of a single addition, subtraction, multiplication, etc., exhibit approximately the following maximum values and standard deviations, when the original (primary) numbers are accurate to 23 binary places:

Sum:

$$N_s = N_1 + N_2 = m_s 2^{n_s}$$

$$\Delta N_{s_{max}} = (1 \times 2^{-23}) 2^{n_s} = (2 \Delta m_{max}) 2^{n_s} \quad (15)$$

$$\sigma_{N_s} = (.41 \times 2^{-23}) 2^{n_s} = (\sqrt{2}\, \sigma_m) 2^{n_s} \quad (16)$$

Difference:

$$N_d = N_1 - N_2 = m_d 2^{n_d}$$

$$\Delta N_{d_{max}} = 2^k (1 \times 2^{-23}) 2^{n_d} = 2^k (2\ \Delta m_{max}) 2^{n_d} \quad (17)$$

$$\sigma_{N_d} = 2^k (.41 \times 2^{-23}) 2^{n_d} = 2^k (\sqrt{2}\ \sigma_m) 2^{n_d} \quad (18)$$

where

$$k = \max(n_1, n_2) - n_d \quad (19)$$

= no. bits "lost" during subtraction

Product: $N_p = N_1\ N_2 = m_p\ 2^{n_p}$

$$\Delta N_{p\ max} = (2 \times 2^{-23}) 2^{n_p} = (4\ \Delta m_{max}) 2^{n_p} \quad (20)$$

$$\sigma_{N_p} = (.58 \times 2^{-23}) 2^{n_p} = (2\ \sigma_m) 2^{n_p} \quad (21)$$

Square: $N_{sq} = N_1^{\ 2} = m_{sq}\ 2^{n_{sq}}$

$$\Delta N_{sq\ max} = (1.9 \times 2^{-23}) 2^{n_{sq}} = (3.8\ \Delta m_{max}) 2^{n_{sq}} \quad (22)$$

$$\sigma_{N_{sq}} = (.66 \times 2^{-23}) 2^{n_{sq}} = (2.3\ \sigma_m) 2^{n_{sq}} \quad (23)$$

Square-root: $N_{sr} = N_1^{\ 1/2} = m_{sr}\ 2^{n_{sr}}$

$$\Delta N_{sr\ max} = (.85 \times 2^{-23}) 2^{n_{sr}} = (1.7\ \Delta m_{max}) 2^{n_{sr}} \quad (24)$$

$$\sigma_{N_{sr}} = (.38 \times 2^{-23}) 2^{n_{sr}} = (1.3\ \sigma_m) 2^{n_{sr}} \quad (25)$$

If one or more of the original numbers involved is accurate to fewer than 23 binary places (say, as the result of previous numerical operations), then the maximum and 1σ errors resulting from the operations here will be larger than those shown, but generally by less than a proportional amount.

4.2 Secondary Operations

Now consider the same numerical operations on numbers which are accurate to fewer than 23 binary places, i.e., secondary numbers having maximum and 1σ errors of

$$\begin{aligned} \Delta m_{i_{max}} &\geq \Delta m_{max} \equiv .5 \times 2^{-23} \\ \sigma_{m_i} &\geq \sigma_m \equiv .29 \times 2^{-23} \end{aligned} \tag{26}$$

The resulting errors subsequent to each operation then are increased from the previous values as follows:

Sum:

$$\Delta N_{s_{max}} = \begin{cases} \left(\Delta m + \dfrac{\Delta m_1 + \Delta m_2}{2}\right)_{max} 2^{n_s} & n_1 = n_2 \\ (\Delta m + \Delta m_1)_{max} 2^{n_s} & n_1 > n_2 \end{cases} \tag{27}$$

$$\sigma_{N_s} = \begin{cases} \sqrt{1.5\sigma_m^2 + (\sigma m_1^2 + \sigma m_2^2)/4}\ 2^{n_s} & n_1 = n_2 \\ \sqrt{\sigma_m^2 + \sigma m_1^2}\ 2^{n_s} & n_1 > n_2 \end{cases} \tag{28}$$

Difference:

$$\Delta N_{d_{max}} = \begin{cases} 2^k (\Delta m_1 + \Delta m_2)_{max} 2^{n_d} & n_1 = n_2 \\ (\Delta m + \Delta m_1)_{max} 2^{n_d} & n_1 > n_2 \end{cases} \tag{29}$$

$$\sigma_{N_d} = \begin{cases} 2^k \sqrt{\sigma m_1^2 + \sigma m_2^2}\ 2^{n_d} & n_1 = n_2 \\ \sqrt{\sigma m^2 + \sigma m_1^2}\ 2^{n_d} & n_1 > n_2 \end{cases} \tag{30}$$

Product:
$$\Delta N_{p_{max}} = [\Delta m + 1.5(\Delta m_1 + \Delta m_2)]_{max}\, 2^{n_p} \quad (31)$$

$$\sigma_{N_p} = \sqrt{\sigma_m^2 + 1.5(\sigma m_1^2 + \sigma m_2^2)}\, 2^{n_p} \quad (32)$$

Square:
$$\Delta N_{sq_{max}} = [\Delta m + 2.8\ \Delta m_1]_{max}\, 2^{n_{sq}} \quad (33)$$

$$\sigma_{N_{sq}} = \sqrt{\sigma m^2 + 4.2\ \sigma m_1^2}\, 2^{n_{sq}} \quad (34)$$

Square-root:
$$\Delta N_{sr_{max}} = [\Delta m + .71\ \Delta m_1]_{max}\, 2^{n_{sr}} \quad (35)$$

$$\sigma_{N_{sr}} = \sqrt{1.5\ \sigma_m^2 + .25\ \sigma m_1^2}\, 2^{n_{sr}} \quad (36)$$

Note that the errors after, say, M successive additions or multiplications can be deduced by successive application of the corresponding formulas above.

4.3 Example: Vector Magnitude

The significance of roundoff error propagation and use of the previous formulas can be illustrated by the following example. Suppose we wish to compute the magnitude r of a 3-component vector $\underline{r}$, where the components are of comparable magnitudes, and are accurate to 23 binary places. The following computation steps are performed, with maximum and 1σ roundoff errors as indicated after each step:

Square r_i's:
$$A_i = r_i^2 = m_i\, 2^n$$

(i=1,2,3)
$$\Delta A_{i_{max}} = (3.8\ \Delta m_{max})2^n \quad (37)$$

$$\sigma_{A_i} = (2.3\ \sigma_m)2^n$$

Add A_1, A_2: $B = A_1 + A_2 = m_b\, 2^{n+1}$

$$\Delta B_{max} = (4.8\ \Delta m_{max})2^{n+1} \qquad (38)$$

$$\sigma_B = (2.1\ \sigma_m)2^{n+1}$$

Add B_1, A_3: $C = B_1 + A_3 = m_c\, 2^{n+1}$

$$\Delta C_{max} = (5.8\ \Delta m_{max})2^{n+1} \qquad (39)$$

$$\sigma_c = (2.3\ \sigma_m)2^{n+1}$$

Square-root: $r = \sqrt{C} = m_r\, 2^{n_r}$

$$\Delta r_{max} = (5.1\ \Delta m_{max})2^{n_r} \qquad (40)$$

$$\sigma_r = (1.7\ \sigma_m)2^{n_r}$$

Suppose next that we also have a measured magnitude $\tilde{r}$, accurate (numerically) to 23 binary places, and differing from r by about 1%, or 1 part in 2^7. In computing the difference d, then, 6 binary places of precision are lost, and the following errors in d result:

$$d = r - \tilde{r} = m_d\, 2^{n_d}, \qquad n_d = n_r - 6$$

$$\Delta d_{max} = 2^6(6.1\ \Delta m_{max})2^{n_d} \qquad (41)$$

$$\sigma_d = 2^6(2.0\ \sigma_m)2^{n_d}$$

5. In-Orbit Navigation Computations

Consider the Shuttle to be in a 270 nmi altitude circular orbit about the earth. The orbital radius, velocity, and period are then

$$a = 3710 \text{ nmi} = 22.5 \times 10^6 \text{ ft.}$$

$$V_c = 4.113 \text{ nmi/sec} = 24{,}990 \text{ fps} \tag{42}$$

$$P = 94.45 \text{ min} = 5{,}668 \text{ sec.}$$

With binary exponents, these numbers are

$$a = .67 \times 2^{25} \text{ ft.}$$

$$V_c = .76 \times 2^{15} \text{ fps} \tag{43}$$

$$P = .69 \times 2^{13} \text{ sec.}$$

5.1 Orbital State Vector

Since the orbital radius and velocity magnitude generally are computed as the magnitudes of vectors,

$$r = |\underline{r}| = \sqrt{\underline{r} \cdot \underline{r}}$$

$$v = |\underline{v}| = \sqrt{\underline{v} \cdot \underline{v}} \tag{44}$$

eq. (40) can be used to obtain the maximum and 1σ roundoff errors in these quantities (presuming the original components to be accurate to 23 binary places):

$$\Delta r_{max} = (2.14 \times 10^{-7})(3.36 \times 10^7) = 7.2 \text{ ft}$$

$$\sigma_r = (.41 \times 10^{-7})(3.36 \times 10^7) = 1.4 \text{ ft} \tag{45}$$

$$\Delta v_{max} = (2.14 \times 10^{-7})(3.28 \times 10^4) = .0070 \text{ fps}$$

$$\sigma_v = (.41 \times 10^{-7})(3.28 \times 10^4) = .0013 \text{ fps} \tag{46}$$

A typical position vector might be

$$\underline{r} = \begin{bmatrix} 18.1 \times 10^6 \\ -10.0 \times 10^6 \\ 9.0 \times 10^6 \end{bmatrix} \text{ft} = \begin{bmatrix} .54 \times 2^{25} \\ -.60 \times 2^{24} \\ .54 \times 2^{24} \end{bmatrix} \text{ft} \tag{47}$$

The maximum and 1σ roundoff errors in this vector are obtained from eqs. (12) and (13), and are

$$\Delta\underline{r}_{max} = \begin{bmatrix} 2.0 \\ 1.0 \\ 1.0 \end{bmatrix} \text{ft}, \quad |\Delta\underline{r}_{max}| = 2.4 \text{ ft} \tag{48}$$

$$\underline{\sigma}_r = \begin{bmatrix} 1.15 \\ .58 \\ .58 \end{bmatrix} \text{ft}, \quad |\Delta\underline{r}|_{rms} = 1.4 \text{ ft} \tag{49}$$

Likewise, a typical velocity vector might be

$$\underline{v} = \begin{bmatrix} -17{,}600 \\ 17{,}000 \\ 5{,}000 \end{bmatrix} \text{fps} = \begin{bmatrix} -.54 \times 2^{15} \\ .52 \times 2^{15} \\ .61 \times 2^{13} \end{bmatrix} \text{fps} \tag{50}$$

in which case the maximum and 1σ roundoff errors are

$$\Delta\underline{v}_{max} = \begin{bmatrix} .002 \\ .002 \\ .0005 \end{bmatrix} \text{fps}, \quad |\Delta\underline{v}_{max}| = .0029 \text{ fps} \tag{51}$$

$$\underline{\sigma}_v = \begin{bmatrix} .0012 \\ .0012 \\ .0003 \end{bmatrix} \text{fps}, \quad |\Delta\underline{v}|_{rms} = .0017 \text{ fps} \tag{52}$$

Note that the velocity roundoff errors would propagate into position errors comparable in magnitude to the position roundoff errors within about 1000 sec (i.e., about 1/6 orbit or 1 rad of angular travel).

The maximum and 1σ roundoff errors in flight time after various periods in orbit are also obtained using eqs. (12) and (13):

t	t (sec)	Δt_{max} (sec)	σ_t (sec)
90 min.	$.66 \times 2^{13}$	.0005	.0003
24 hr.	$.66 \times 2^{17}$	.0078	.0045
30 da.	$.60 \times 2^{22}$	.25	.15
6 mo.	$.90 \times 2^{24}$	1.0	.58

(53)

Note that the quantization errors in flight time are about 1 msec after 2 orbital revolutions, and about .01 sec after 2 days in orbit (about 30 revolutions). Time quantization errors can be related to downrange position errors Δs using the orbital velocity V_c, with the following results:

t	Δs_{max} (ft)	σ_s (ft)
90 min.	12.5	7.5
24 hr.	195	113
30 da.	6,250	3,750
6 mo.	25,000	14,500

(54)

The roundoff errors summarized by eqs. (45), (46), (48), (49), (51), and (52) indicate that the spacecraft position and velocity data can be retained by 23-bit mantissas with adequate precision for orbital navigation purposes. Orbital navigation uncertainties in the spacecraft state are typically on the order of 100's or 1000's of feet in position and 1/100's or 1/10's of feet/sec in velocity, or two to three orders of magnitude worse than the available precision. However, it appears that a 23-bit mantissa does not provide adequate precision in flight time for more than one or two orbits. To maintain time with 1 msec precision for 30 days in orbit would require

at least 31 numerical bits. (Maintaining time in double precision would be more than adequate.)

5.2 Orbital Integration

That 23-bit precision is sufficient for the state vector at one point in time does not guarantee that it is adequate for projecting the state forward in time by numerical integration. At each integration step t_k, increments $\delta\underline{r}_k$, $\delta\underline{v}_k$ in position and velocity are computed by an appropriate integration formula, and added to the position and velocity at t_{k-1} to obtain the new values:

$$\begin{aligned} \underline{r}(t_k) &= \underline{r}(t_{k-1}) + \delta\underline{r}_k \\ \underline{v}(t_k) &= \underline{v}(t_{k-1}) + \delta\underline{v}_k \end{aligned} \tag{55}$$

Hence, at every integration step, further roundoff error in each component of the state is introduced by the addition of a relatively small number to a large rounded number.

The accumulated effects of these errors on the subsequent orbital state can be determined by use of the state transition matrix and superposition of linear errors. First, consider $\underline{r}(t)$, $\underline{v}(t)$ to be the exact orbit that would result from infinite precision. The actual, finite-precision orbit $\underline{r}'(t)$, $\underline{v}'(t)$ is then in error by

$$\begin{aligned} \Delta\underline{r}(t) &= \underline{r}'(t) - \underline{r}(t) \\ \Delta\underline{v}(t) &= \underline{v}'(t) - \underline{v}(t) \end{aligned} \tag{56}$$

These errors can be expressed in terms of the roundoff errors $\Delta\underline{r}_k$, $\Delta\underline{v}_k$ introduced at each time step as

$$\begin{bmatrix} \Delta\underline{r}(t) \\ \Delta\underline{v}(t) \end{bmatrix} = \sum_{k=0}^{n} \Phi(t,t_k) \begin{bmatrix} \Delta\underline{r}_k \\ \Delta\underline{v}_k \end{bmatrix}, \quad t \geq t_n \tag{57}$$

In terms of 3x3 partition matrices of Φ,

$$\Phi(t,t') \equiv \begin{bmatrix} A(t,t') & B(t,t') \\ C(t,t') & D(t,t') \end{bmatrix} \tag{58}$$

the accumulated orbital position error at t_n is

$$\Delta\underline{r}(t_n) = \sum_{k=0}^{n} [A(t_n,t_k)\ \Delta\underline{r}_k + B(t_n,t_k)\ \Delta\underline{v}_k] \tag{59}$$

Since the various roundoff errors are uncorrelated, the covariance of the accumulated position errors is

$$P_r(t_n) = \sum_{k=0}^{n} [A_{nk}\ P_{rk}\ A_{nk}^T + B_{nk}\ P_{vk}\ B_{nk}^T] \tag{60}$$

where

$$P_r \equiv E[\Delta\underline{r}\ \Delta\underline{r}^T], \quad P_v \equiv E[\Delta\underline{v}\ \Delta\underline{v}^T] \tag{61}$$

Further, since at each t_k the roundoff errors in different components of $\underline{r}$ and $\underline{v}$ are uncorrelated, P_{rk} and P_{vk} are diagonal matrices:

$$P_{rk} = \begin{bmatrix} \sigma_{r_1}^2 & 0 & 0 \\ 0 & \sigma_{r_2}^2 & 0 \\ 0 & 0 & \sigma_{r_3}^2 \end{bmatrix}, \quad P_{vk} = \begin{bmatrix} \sigma_{v_1}^2 & 0 & 0 \\ 0 & \sigma_{v_2}^2 & 0 \\ 0 & 0 & \sigma_{v_3}^2 \end{bmatrix} \tag{62}$$

If orbital calculations are performed in coordinates such that r_1, r_2, r_3 are generally of comparable magnitude, and likewise v_1, v_2, v_3, then the three variances will generally be equal.

$$P_{rk} = \sigma_r^2 \, I, \qquad P_{vk} = \sigma_v^2 \, I \tag{63}$$

where the 1σ errors in rounding $\Delta\underline{r}_k$, $\Delta\underline{v}_k$ during addition are

$$\sigma_r = (\sigma_m) 2^{n_{ri}} = (.29 \times 2^{-23}) 2^{24} = .58 \text{ ft} \tag{64}$$

$$\sigma_v = (\sigma_m) 2^{n_{vi}} = (.29 \times 2^{-23}) 2^{14} = .00057 \text{ fps} \tag{65}$$

Thus eq. (60) can be simplified to

$$P_r(t_n) = \sigma_r^2 \sum_{k=0}^{n} A_{nk} A_{nk}^T + \sigma_v^2 \sum_{k=0}^{n} B_{nk} B_{nk}^T \tag{66}$$

The mean-squared position error at t_n is given by the trace of $P_r(t_n)$, or

$$\overline{|\Delta\underline{r}_n|^2} = \text{tr}\,[P_r(t_n)]$$

$$= \sigma_r^2 \sum_{k=0}^{n} \text{tr}\,(A_{nk} A_{nk}^T) + \sigma_v^2 \sum_{k=0}^{n} \text{tr}\,(B_{nk} B_{nk}^T) \tag{67}$$

Further, the trace of a matrix times its transpose is simply the sum of the squares of all its elements:

$$\text{tr}(MM^T) = \sum_{i=1}^{3} \sum_{j=1}^{3} m_{ij}^2 \tag{68}$$

Note also that the summations in (67) can be approximated by integrals if the time steps Δt_k are equal and relatively small:

$$\frac{1}{\Delta t} \sum_{k=0}^{n} f(t_n, t_k) \Delta t \approx \frac{1}{\Delta t} \int_{t_0}^{t_n} f(t_n, t) dt \tag{69}$$

Hence we can write eq. (67) as

$$\overline{|\Delta \underline{r}_n|^2} = \sigma_r^2 A + \sigma_v^2 B \tag{70}$$

where

$$A \approx \frac{1}{\Delta t} \int_{t_0}^{t_n} \text{tr } [A(t_n,t)A(t_n,t)^T]dt \tag{71}$$

$$B \approx \frac{1}{\Delta t} \int_{t_0}^{t_n} \text{tr } [B(t_n,t)B(t_n,t)^T]dt$$

These integrals can in fact be evaluated analytically for a near-circular orbit. The circular state transition matrix in rotating, local-vertical coordinates is rather simple in form. The matrices A and B are

$$A(t_n,t) = \begin{bmatrix} 2-C & 0 & 0 \\ 2S-3\tau & 1 & 0 \\ 0 & 0 & C \end{bmatrix} \tag{72}$$

$$B(t_n,t) = \begin{bmatrix} S & 2(1-C) & 0 \\ -2(1-C) & 4S-3\tau & 0 \\ 0 & 0 & S \end{bmatrix} \frac{1}{w} \tag{73}$$

where

$$\begin{aligned} w &= \sqrt{\mu/a^3} = V_c/a = 1.109 \text{ mrad/sec} \\ \tau &= w(t_n-t) \\ S &= \sin \tau \\ C &= \cos \tau \end{aligned} \tag{74}$$

Thus, the traces are

$$\mathrm{tr}(AA^T) = 3 + 2(1-C)^2 + (2S-3\tau)^2 \tag{75}$$

$$\mathrm{tr}(BB^T) = [2S^2 + 8(1-C)^2 + (4S-3\tau)^2]\ \frac{1}{w^2} \tag{76}$$

and the integrals from $t' = t_0 \to t_n$ or $\tau' = 0 \to \tau$ are readily obtained. The results are

$$R \equiv w\Delta t A = 3\tau^3 + 4\tau(2+3C) - S(16+C) \tag{77}$$

$$V \equiv w^3 \Delta t\ B = 3\tau^3 + 3\tau(7+8C) - 5S(8+C) \tag{78}$$

where

$$\Delta t = \frac{1}{n}(t_n - t_0) = \frac{1}{n}\ \frac{\tau}{w} \tag{79}$$

The non-dimensional coefficients R and V have the following values at quarter-orbit points:

τ	$R(\tau)$	$V(\tau)$
$\pi/2$	8.16	4.60
π	80.5	83.7
$3\pi/2$	368	453
2π	870	1026

Noting that $\sigma_v/w = .51$ ft, we can now write the rms position error magnitude at any time t as

$$|\Delta \underline{r}(t)|_{rms} = \frac{1}{\sqrt{w\Delta t}}\ [.34\ R(\tau) + .26V(\tau)]^{1/2} \text{ft} \tag{80}$$

Thus the rms error at a given point in time clearly grows as $1/\sqrt{\Delta t}$ or $\sqrt{n}$, where n is the number of integration steps. For a given integration step size Δt, the rms error grows with total integration time t as $t^{3/2}$. The following rms errors result at quarter-orbit points, for integration step sizes of 10,100, and 1000 sec:

τ (rad)	RMS Position Error $\overline{\lvert \Delta \underline{r} \rvert}$ (ft)		
	Δt=10 sec	Δt=100 sec	Δt=1000 sec
π/2	18.8	5.9	1.9
π	66.3	21.1	6.6
3π/2	148	46.6	14.8
2π	224	70.9	22.4

(Note: These numbers should be checked by numerical test cases before any further work is based on them.)

Hence, for orbital navigation with state-vector updates once or twice per orbit, a 10 to 100 sec time step appears marginally adequate with 23-bit precision. (Increasing the precision would reduce the indicated errors by 1/2 for every additional bit.) To extrapolate the state vector ahead by several orbits would introduce excessive errors unless very large time steps on the order of 1000 sec or more were utilized.

Large (effective) time steps of 1000 sec or more are possible if a modified form of the equations of motion is utilized. For example, Encke's method calculates the state as an analytic conic state vector $\underline{r}_{con}$ plus a deviation $\underline{d}$ relative to that. The equations of motion for $\underline{d}$ are integrated, with the additional roundoff error at each time step proportional now to d rather than r, where d<<r. The major roundoff error in the full state vector is introduced only at rectification points, where a new conic is calculated from the old conic plus deviation,

$$\underline{r}'_{con} = \underline{r}_{con} + \underline{d}$$
$$\underline{v}'_{con} = \underline{v}_{con} + \dot{\underline{d}} \qquad (81)$$

If rectification occurs, say, three times per orbit, the accumulation of roundoff error corresponds to about a 2000-sec effective time step, even if the actual time step for integrating $\underline{d}$ is on the order of 10's of seconds.

5.3 Rendezvous Navigation

During the latter phases of rendezvous, the Shuttle (pursuer) state relative to the Station (target) is of primary interest. If the relative state is computed by differencing the full pursuer and target states, e.g.,

$$\underline{\rho} = \underline{r}_p - \underline{r}_T \tag{82}$$

then the maximum and 1σ roundoff errors in $\underline{\rho}$ are 2 and $\sqrt{2}$ times those in $\underline{r}_p$ and $\underline{r}_T$, or

$$\begin{aligned} |\Delta\underline{\rho}_{max}| &\simeq 4 \text{ ft} \\ |\Delta\underline{\rho}|_{rms} &\simeq 2 \text{ ft} \end{aligned} \tag{83}$$

The errors in the computed range to the target are comparable in magnitude. (These errors are of the same order of magnitude as the errors in a range-measuring device for relative navigation.)

If both the target and pursuer states are integrated independently, the accumulated roundoff errors in each state (previous table) can become excessive for relative navigation purposes. However, if Encke integration schemes for each orbit are utilized, with 1000-sec or greater rectification intervals, the relative state can be computed with sufficient accuracy ($\sigma_\rho \leq 2.7$ ft) for 1/4-orbit or shorter intervals between relative navigation periods.

If for some reason the relative state accuracy is not sufficient when computed by differencing the pursuer and target states, precision can be greatly increased by integrating the relative state directly. Rather than computing the full target and pursuer state vectors, we compute the full target (or pursuer) state and the relative pursuer state vectors directly. The equation of motion for the relative pursuer position $\underline{\rho}$ is not difficult to derive or program. The accumulated roundoff errors in $\underline{\rho}$ are then reduced over the previously cited errors by ρ/r, where $\rho/r = .03$ at 100 nmi range, for example.

5.4 Numerical Range

A question independent of 23-bit precision is whether the 7-bit exponent provides adequate numerical range. The maximum and minimum floating point numbers allowed, by eq. (6), are

$$0.3 \times 10^{-38} < |N| < 1.7 \times 10^{38} \tag{84}$$

One of the largest numbers encountered in orbital calculations is

$$r^3 = .114 \times 10^{23} \text{ ft}^3 < 10^{38} \tag{85}$$

Note that r^6 would exceed the upper limit, creating a floating point overflow.

Often the most critical numbers for overflow or underflow are in the error covariance matrix. Consider the diagonal elements for position errors, velocity errors, and IMU misalignment errors. For example, to avoid overflow we must have

$$\sigma_r < 10^{19} \text{ ft} = 2 \times 10^{15} \text{ mi} = 2 \times 10^{7} \text{ au} \tag{86}$$

We clearly do not expect positional uncertainties on the order 20 million au! To avoid underflow, we must have

$$\sigma_r > 10^{-19} \text{ ft} = .3 \times 10^{-9} \text{ Å} \tag{87}$$

$$\sigma_\theta > 10^{-19} \text{ rad} \tag{88}$$

both of which exceed expected accuracies by many orders of magnitude. Underflow is more likely to occur in off-diagonal (cross-correlation) elements of the covariance matrix. However, if a number so small as to cause underflow is replaced by zero, no difficulties should result.

6. Approach and Landing Navigation

Shuttle navigational accuracy requirements on altitude and altitude rate during landing may be as stringent as

$$\sigma_h = 1m = 3 \text{ ft}$$
$$\sigma_{\dot{h}} = 5 \text{ cm/sec} = .15 \text{ ft/sec} \tag{89}$$

If this requirement is imposed, results of the previous section indicate that 23-bit precision is marginally adequate for position vectors referenced to the center of the earth. For example, computing altitude above the runway as the difference between radii from the center of the earth would result in a 1σ roundoff error of

$$\sigma_h = 2.0 \text{ ft} \tag{90}$$

Since virtually all navigation measurements (e.g., ranges to ground transponders) are taken with respect to installations on the earth's surface, within a few hundred miles of the Shuttle vehicle, it is computationally undesirable and seems unnecessary to reference all such positions to the earth's center. For example, it is possible to define a transponder location near the final runway to be the origin of coordinates for trajectory computation purposes. If all other transponder locations, runway location, etc., are given as differences in altitude, latitude, and longitude from this reference location, negligible roundoff errors will result from 23-bit precision. (The roundoff errors in earth-centered state variables are in effect ascribed to uncertainty in position of the earth's center relative to the reference transponder location, which by definition has zero error.)

In this case, where altitude is computed relative to the reference transponder (or, say, the reference ellipsoid), the altitude of the Shuttle above the runway at landing is

$$\delta h = h_s - h_r \tag{91}$$

If both are, say, about 200 feet above or below the reference transponder, the maximum and 1σ relative altitude errors are

$$\Delta h_{max} = 2^{-15} \text{ ft} = 3 \times 10^{-5} \text{ ft}$$

$$\sigma_h = .41 \times 2^{-15} \text{ ft} = 1.2 \times 10^{-5} \text{ ft} \tag{92}$$

Likewise, if the range to a ground transponder is 100 nm

$$\rho = 100 \text{ nmi} = .57 \times 2^{20} \text{ ft}$$

the roundoff errors in the computed range $|\underline{\rho}|$ will be, from eq. (40),

$$\Delta\rho_{max} = .32 \text{ ft}$$

$$\sigma_\rho = .06 \text{ ft} \tag{93}$$

This precision appears quite adequate for approach and landing navigation.

7. Summary

The 23-bit mantissa appears to provide adequate precision for all orbital, approach, and landing computations. One exception is flight time (from launch), which should be maintained in double precision for flights exceeding 3 hrs.

The accumulation of roundoff error during numerical integration becomes excessive after 1 or 2 orbits if the full state equations are integrated, necessitating a short (10-100 sec) time step, and if no navigation updates are incorporated. Integrating the Encke equations effectively increases the time step to the rectification interval, insofar as it affects roundoff error, and greatly reduces the error growth rate.

For rendezvous navigation relative to the target, computing the relative state by differencing the target and pursuer states yields marginal accuracy with 23-bit precision. Accuracy can be greatly improved, if so desired, by computing the relative state directly, i.e., by integrating the relative equations of motion.

For approach and landing navigation, computing the vehicle state in earth-center-referenced coordinates yields marginal accuracy for position and velocity relative to the runway. Again, utilizing a nearby point (e.g., one ground transponder) as coordinate reference greatly improves numerical accuracy to beyond that required.

The numerical range provided by the 7-bit exponent should be quite adequate to accommodate the largest and smallest numbers of interest. Isolated exceptions (e.g., r^6) may occur, but should be rare and readily reprogrammed to avoid problems. Underflows may occur in off-diagonal elements of the error covariance matrix, but can be replaced by zero in such cases with no problem.

Appendix F

General Description of Burroughs D-Machine

[This appendix was prepared from notes by Dr. James S. Miller.]

1.0 Architectural Highlights

The Burrough's D-machine is an unusually modular and flexible architectural design, which is capable of successful and economical application to a wide variety of problem areas. In its basic multiprocessor configuration, it consists of three major building blocks: interpreters, switch interlock, and memories. The interpreter is a microprogrammed processor and is used to perform both arithmetic/logical computation and I/O device control. The switch interlock is the communication network which links interpreters, operating memory, and I/O devices. A two-interpreter layout is shown in Figure 1.

1.1 The Interpreter

The D-machine interpreter is constructed from five functional parts: memory control unit (MCU), control unit (CU), logic unit (LU), microprogram memory (MPM), and nanomemory (NM). (See Figure 2.) The word-length of the interpreter depends only upon the logic unit, which is modular in 8-bit blocks, from 16 bits to 64 bits. The use of microprogramming enables the control logic to be quite regular in structure, resulting in economy of manufacturing. Additionally, different microprograms may be used with the same hardware to implement different instruction sets for different applications. Furthermore, if read-write rather than read-only MPM is attached, the system can reload its MPM dynamically to run programs written in different machine languages at different times.

To save storage, the microprogram structure of the interpreter has been divided into two logical sections: micro and nano. The control of functional operations within the interpreter is dictated by the contents of a location in nanomemory. Each of the 56 bits corresponds to a control line for the elements of the LU, CU, and MCU. A given nanoword is selected under control of a microword which specifies the nanoword's address in nanomemory. As a result, nanowords may be referred to by many microwords; hence, the bit saving.

1.2 The Switch Interlock

This communication network connects interpreters, operating memories, and devices. The basic configuration is designed for

four interpreters, eight memories, and eight devices. Because of the lesser logical complexity of the operations in the SWI per clock cycle compared with the interpreter, the SWI may be run at a higher rate than the interpreter. This permits introduction of a degree of serialization which reduces the number of required parallel paths, diminishing complexity. Consequently, the building block of the SWI is a single bit width. These may be combined to the degree necessary for high throughput.

2.0 State of Development

Burroughs is producing two versions of interpreter-based systems: commercial and military. The commercial version is being used for disk controllers and for other applications not yet announced. The military configuration is funded by the Avionics Lab at Wright-Patterson AFB and consists of a five-interpreter system which will be ready for delivery about 1 April 1972, although it may be retained at Burroughs for a period of time while facilities are prepared at WPAFB.

Microprograms have been written which enable the machine to run D825 programs and B300 programs. A demonstration of these and of dynamic switching between them is in current operation in Paoli. The full operating system for multiprocessor configurations is expected to be operational in March.

The proposed flight packaging has weight, power, and volume of 141 lb., 569 watts, and 1.8 cubic feet, respectively. This includes five 32 bit interpreters, 64K of 32 bit words of memory (Honeywell plated wire at 1 μsec), power supplies, and switch interlock.

3.0 Software

The flexibility of the microprogrammed structure of the D-machine is especially beneficial to the Shuttle program. Rather than being faced with selection of the best of the available hard-logic computers, NASA has an opportunity to utilize a soft machine. The advantage of such "softness" is three-fold.

First, an instruction set of particular power for the Shuttle application can be developed and implemented by microprogramming as described in Chapter 6. This language can be as primitive as common machine languages, or can be of higher order, such as APL or HAL. The choice of an intermediate language, such as partially compiled HAL, could allow a substantial saving in the amount of operating memory required, because of the enhanced conciseness of such a language. (The weight of the configuration described above included 80 lb. of the 141 lb. total for the plated wire memory; reduction of required capacity has an obvious payoff.)

A second advantage of the microprogrammed processor from a software viewpoint is that the instruction set may be altered, tuned, or augmented during the life of the Shuttle program to improve the capability of the language or to add wholly new features not originally desired. Only modification of the micro-program and compiler would be required, not hardware redesign or requalification.

A third advantage is that the instruction set of the computer could be dynamically switched between specialized languages for these applications if it were necessary, or an additional interpreter could be provided with a second instruction repertoire. The potentialities of this approach are vast, e.g., one interpreter could be dedicated to operating-system functions and be provided with a tailored instruction set for that purpose. In the event of failure, it would only be necessary to load the OS machine image into one of the surviving units.

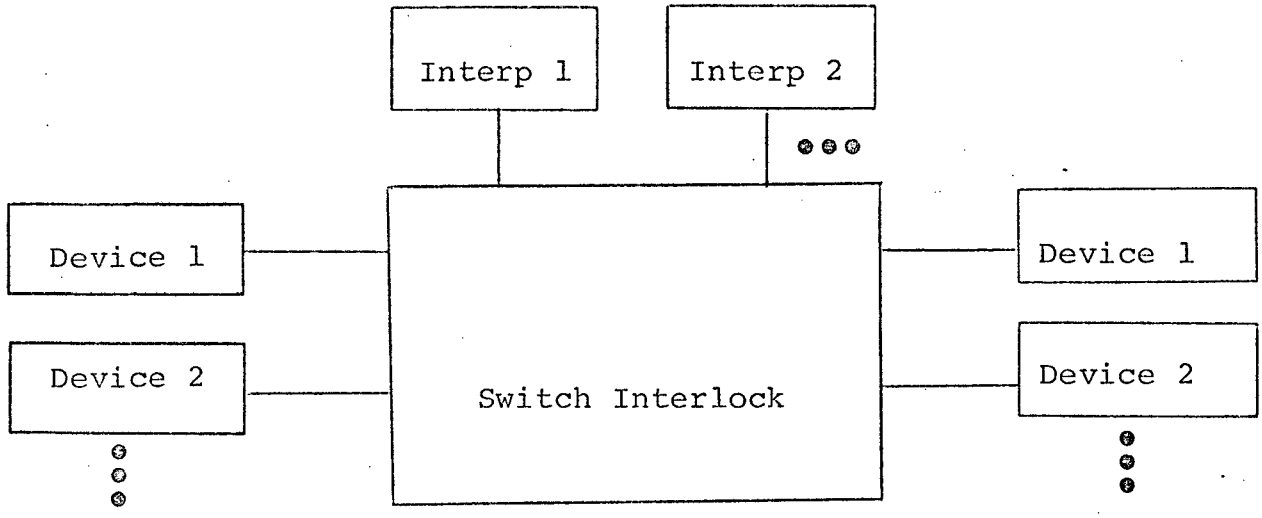


Figure 1: A D-Machine Configuration

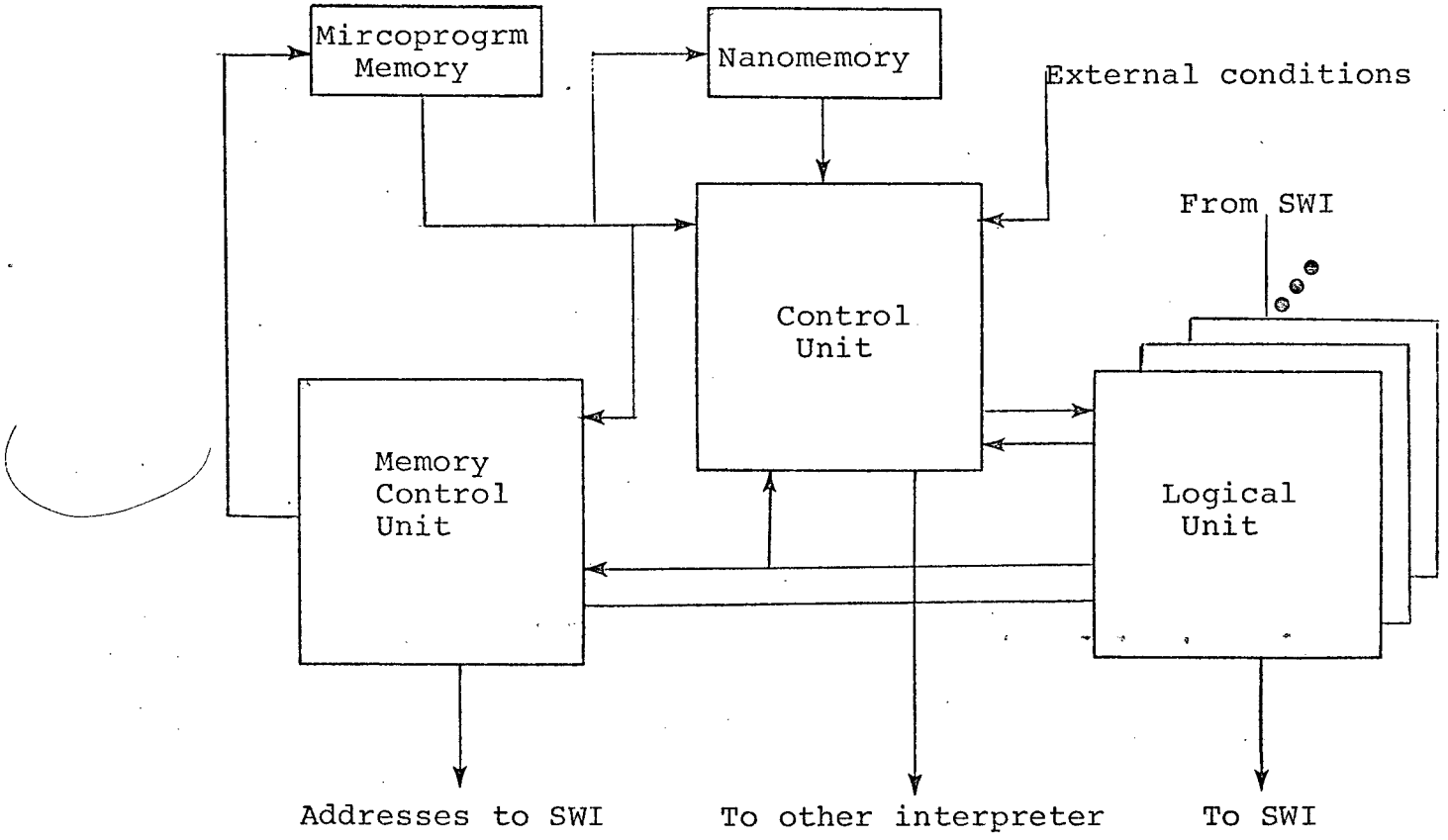


Figure 2: Interpreter Block Diagram