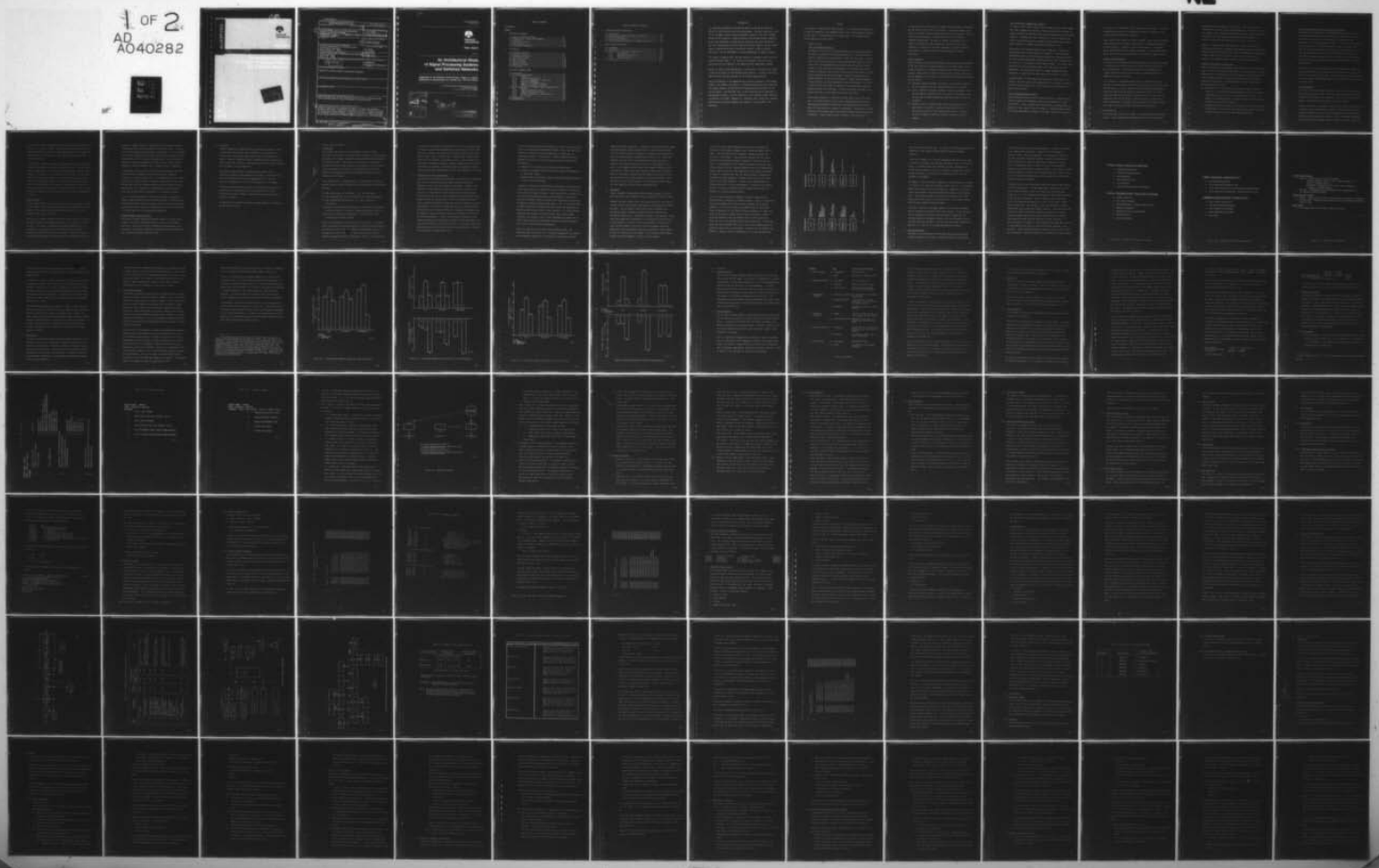
AD-A040 282 ROCKWELL INTERNATIONAL NEWPORT BEACH CALIF COLLINS G--ETC F/G 9/2
AN ARCHITECTURAL STUDY OF SIGNAL PROCESSING SYSTEMS AND SWITCHE--ETC(U)
MAR 77 DCA100-76-C-0070
UNCLASSIFIED NL

1 OF 2
AD
A040282



AD A040282

Rockwell International

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution

Volume 1

D D C
JUN 7 1977

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER

2. GOVT ACCESSION NO. (9)

3. RECIPIENT'S CATALOG NUMBER

(6) 4. TITLE (and Subtitle)
AN ARCHITECTURAL STUDY OF SIGNAL PROCESSING SYSTEMS AND SWITCHED NETWORKS.
Final Report, Volume 1, Appendix L, Volume 3
Appendices, Volume 2

5. TYPE OF REPORT & PERIOD COVERED
Final Report.
16 Aug 76 — 15 Mar 77.

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)

(15) 8. CONTRACT OR GRANT NUMBER(s)
DCA100-76-C-0070

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Rockwell International, Collins Radio Group
4311 Jamboree Boulevard
Newport Beach, CA 92663

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
PE 33126K
Task 15306C

11. CONTROLLING OFFICE NAME AND ADDRESS
Defense Communications Engineering Center/R830
1860 Wiehle Avenue
Reston, VA 22090

(11) 12. REPORT DATE
15 Mar 77

13. NUMBER OF PAGES

14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)
(12) 152p.

15. SECURITY CLASS. (of this report)
UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)
Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
Modular Architecture, Modular Design Methodology, Switch Simulation, High Order Language Analysis, Signal Processing, Switching Systems

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)
A modular design process as applied to signal processing and switching systems is described. Architecture analysis, high order language evaluation and comparison, system simulation, and hardware/software tradeoffs were made in the context of modular design of signal processing and switching systems. The principal motivation for modular system design is lower life cycle cost.

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

392 669

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

523-1001818-001821
15 March 1977

Rockwell International

final report

Volume 1

An Architectural Study of Signal Processing Systems and Switched Networks

Submitted to the Defense Communication Agency in partial fullfillment of requirements for contract No. 100-76-C-0070

(3) Collins Government Telecommunications Division
(4) Rockwell International
(2) Newport Beach, California 92663

ACCESSION for
NTIS White Section ✓
DDC Buff Section
UNANNOUNCED
JUSTIFICATION for 1473
BY
DISTRIBUTION/AVAILABILITY CODES
Dist. AVAIL. and/or SPECIAL
A

Printed in the United States of America

S/C 392669

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

Table of Contents

Introduction

Summary

1. Architecture Analysis

1.1 Generic System Definition..... 1-1
1.2 Motivation for Modular Architecture..... 1-2
1.3 Modularity..... 1-4
1.4 Application Effects..... 1-7
1.5 Architectures..... 1-12
1.6 Architecture Selection..... 1-13

2. Simulation

2.1 Simulator Selection..... 2-1
2.2 Model Description..... 2-4
2.3 Parameter Description..... 2-7
2.4 Statistical Outputs..... 2-18
2.5 User Interface..... 2-21
2.6 Model Evolution..... 2-29
2.7 Design Evolution..... 2-32
2.8 Simulation Summary..... 2-35
2.9 Deliverables..... 2-46

3. High Order Language Study

3.1 Introduction..... 3-1
3.2 HOL Design Requirements..... 3-1
3.2.1 General..... 3-2
3.2.2 Preliminary Assertions..... 3-2
3.2.3 Choice of Languages for Comparison..... 3-6
3.2.4 Overview of Bridging..... 3-9
3.2.5 Support Proliferation Problem..... 3-10
3.2.6 Software Development Practices..... 3-12
3.3 Use of Existing HOLs in Modular System Supplementation... 3-17
3.3.1 General..... 3-17
3.3.2 System Modularity & HOLs..... 3-17
3.3.3 System Architectures & HOLs..... 3-21
3.3.4 Software Architecture Considerations..... 3-27
3.3.5 HOL Criteria..... 3-32
3.4 Conclusions..... 3-34
3.5 Recommendations..... 3-34

Table of Contents (Continued)

4. Hardware/Software Alternatives

4.1 Introduction..... 4-1
4.2 Implication of Hardware/Software Tradeoffs..... 4-1
4.3 Specific Tradeoffs..... 4-3
4.4 Other Design Tradeoffs..... 4-7
4.5 Software/Software Tradeoffs in Message Switching..... 4-9

5. System Design Methodology

5.1 Background..... 5-1
5.2 Definition..... 5-1
5.3 Description of the Design Methodology..... 5-4
5.3.1 Introduction..... 5-4
5.3.2 Requirements Definition..... 5-6
5.3.3 Functional Analysis..... 5-10
5.3.4 Design Synthesis..... 5-16
5.4 Conclusion..... 5-17

INTRODUCTION

This study was undertaken to evaluate the modular system design process as applied to signal processing and switching systems. The goals identified in the initial statement of work included architecture analysis, high order language evaluation and comparison, systems simulation, and hardware/software tradeoffs. As the study progressed, prerequisite and associated tasks were identified so that the following report not only addresses the above subjects, but the related topics of module descriptions, architecture selection criteria and their application, and the development of a design methodology for modular systems.

This report is organized much like the contract's "Statement of Work" with four sections matching Tasks I - IV. Results are presented at the end of each section and are also combined in the Summary section immediately following.

Section 5, Modular Design Methodology, is codification of the experience gained in forming the results of the preceeding four sections. It seeks to apply this experience with analytic processes to the iterative design process.

Much of the material in the appendices was originally presented in the October, November, and December 1976 reports. Conclusions presented at those times have not changed; however, further work has allowed updating and expansion of the earlier results. (An additional topic, array processing, was an outgrowth of the modularity studies. A brief discussion of this alternate signal processing architecture is included - Appendix B). Additional work in new areas, primarily simulation and high order language (HOL) studies is also included in the appendices.

SUMMARY

This summary is organized into three sections: (1) A task-oriented description of work corresponding to the Statement of Work, and including natural extensions of that work. (2) Recommendations for related study. (3) A summary of the various appendices to this report.

1. SUMMARY BY TASK

Task 1 Architecture Selection

The analysis of the goals of system modularity led to the establishment of the bases of comparison for architecture selection. Principal among the potential benefits of a modular system is lower life cycle cost. The cost over a 15 year cycle originates in several quantifiable areas which group fairly well into development and acquisition cost, reliability, and performance. These areas were first presented in the October 1976 progress report and resulted in the choice of the parallel bus architecture for both signal processing and switching systems - though the analyses were performed separately for these two applications. Despite the correspondence of these results, application dependency cannot be overlooked. This point is made clear when the graphical summaries of the architecture selection process (Section 1) are reviewed. A single fixed system constraint (for a specific designer's target system) could alter the results.

The principal argument for modularity as noted above is lower life cycle cost. The several aspects of the military design/specification/procurement/deployment/maintenance/modification/ and support process are considerably different than their commercial counterparts. Accordingly it is not suprising that different results with regard to system design criteria will be obtained. (There criteria reflect a planned 15 year life cycle.)

A separate result of the analysis of signal processing system architectures was the possible role of the array processor. Because the highest speed modules in a signal processing system are usually at the system's input, an array processor employed as a matched filter or similar element is frequently encountered. The result of interest (see Appendix B for a summary description) was that although this architecture was not as supportive of modularity as the other candidates, its reserve capacity could result in a total solution to specialized tasks (in signal processing).

Task 2 Simulation

System simulation has verified that modular system structures and elements of the type required for signal processing and switching systems can be formally described and subsequently employed in an iterative fashion to improve a system design. Examples of this work are included in Section 2 and the Appendices to this report. The following points summarize the work of this section:

(1) Interface simplicity between the physical (or functional) module and its bus translates into ease, and hence accuracy, of simulation.

(2) Use of a generally available simulator is highly recommended. Availability supports a potential user's acceptance of these design aids, as their use is highly iterative and hence convenience of access is necessary.

(3) Related to item 2, convenient user access to any desired level of model detail is desirable. Future work may be done for the specialized application areas covered in this study. This could include for example, modeling of functions internal to critical modules.

Task 3 High Order Language (HOL) Analysis

In response to the specific goals of the statement of work, several common high order languages were analyzed for their suitability to the tasks of signal processing and switching system algorithm development, and executive software. A separate topic which surfaced as a result of analysis and review of the long term plans of several military agencies was the more general question of language acceptance and support by D.o.D.

The results of the language comparisons are summarized in tabular form; selected topics are further discussed in Appendix M. These annotated descriptions enable a prospective system designer (or stipulating agency) to select the most appropriate language among three final candidates (COL, PL-1, an SPL/1) as a function of their specific needs. For current signal processing and switching systems design, SPL/1 was found to be most appropriate. The more general topics of language implementation and acceptance, and especially long term support are discussed in Section 3.

A separate result of this study was the recognition of the need for HOL design of real time systems to be supported by (i.e. contain) the means for transfer to machine level code. This escape is at once necessary and in need of strict control.

Task 4 Hardware/Software Tradeoffs

Hardware/software tradeoffs are far more easily made in a modular system. Interface complexities frequently impede such considerations in other architectures. General guidance and identification of corresponding cost elements in these trades is given in Section 4. Hardware/software tradeoffs during development of the parallel bus model (Section 2) resulted in

the addition of a distributed control capability to this model. The use of hardware/software trades in a system design context is contained in the design methodology description, Section 5.

Other system design tradeoffs are often important. These include tradeoffs among implementation techniques (identified here as hardware/hardware and software/software). Examples of these include central vs. distributed memory, and alternate control schemes, respectively. These related system design decisions are discussed generally in the switching system context.

Modular Design Methodology

The results of the architecture selection process and the experience gained in simulation provide the guidelines for establishing a modular system design methodology. These design techniques are complemented by the software design steps at the various stages of system development in a suggested procedure in Section 5.

To be effective, a formalized method should not constrain the system designer with requirements other than the operational and cost goals of the defining agency. It should, however, provide recognizable benchmarks for monitoring progress and for providing intermediate results in a form usuable by support functions (e.g., maintainability). To this end, the methodology described in Section 5 stresses analysis and tabulation of system requirements, and suggests specific outputs of the design process for review at project Preliminary and Critical Design Reviews.

II. RECOMMENDATIONS

The current work, as defined in Tasks 1-4, has been mainly of an analytic nature. It has included system requirement analysis, architecture and

language comparative analyses, and simulation. This has led to the development of a modular systems design methodology. Early application of the results of this study has indicated several areas in which further work is necessary to bridge the gap between design and implementation, i.e., methods to support the user employing the results of this study.

Support of a design system takes many forms, ranging from the formal methodology definitions, to supporting documentation for data organization, system libraries, and training aids. Three recommendations for development of these items as well as two items requiring further investigation relative to HOL design of modular systems are briefly described below.

Develop a formal module library system

Many signal processing and switching system modules have been described and used as a part of the simulations performed in the current study. Each of these modules was identified by the system designer, and was defined to the level of detail necessary to support the objectives of the candidate systems and simulations. The establishment of a formal library would include the following items:

(1) Development of an indexing and cross-reference system. Candidate indices are: (a) software implemented modules based upon a standard instruction set (e.g., PDP-11); (b) categorization by input (e.g., analog); (c) designs that are microprocessor based; and (d) preferred groupings by level of field usage, etc.

(2) Development and test of a basic set of signal processing and switching system modules. As a core set these would include control and timing functions required for each of these application classes.

Design Methodology Verification (System Design Example)

Demonstration and enhancement of the methodology described in Section 5 is best supported through a trial development. The methodology may impose constraints to creativity or efficiency that are undetected until encountered in an actual development program. To this end, the application of these processes to a modular VLF receive system is proposed. Much applications and requirements research has been completed for this system and, based upon this fact and Collins prior system experience, it is reasonable to assume that application of the modular design methodology would yield results unaffected by extraneous influences such as uncertain systems design.

The outcome of this test case would be a refined methodology wherein the input and output tables, design review aids, and simulation routines would be tested and improved as indicated. An example design (through completion of the preliminary design phases) would also be available as a teaching aid for future users.

Model Enhancements

Use of the GPSS-based architecture models to date has indicated two areas where improvements may be made. These are, user interface simplifications (designer to GPSS reports), and expansion of the set of system functions modeled. Development of these features is described below.

User/GPSS interfaces occur whenever a file must be created or modified, and of course, in the interpretation of results. File building is time-consuming and somewhat error prone in that parameters may be unintentionally changed and such changes may be undetected until a subsequent simulation. This (system building) procedure would be greatly simplified through the

use of FORTRAN inputs allowing explicit changes to only those parameters of interest. In a similar fashion, FORTRAN report generators can be used to format the simulation results in a very readable fashion, e.g., module and parameter names can be explicity called rather than the current "row/column" organization.

System functions modeled at present are common facility (bus) transactions and module input/output events over any specified period. Functions which could be added include: (a) functions implemented within a module (usually software); (b) queues; (c) bus hierarchies; and (d) peak (i.e., one time) statistics as compared to averages. (The last item can be accomplished within the current model, however, it requires greater user familiarity with GPSS than necessary for the remainder of the design aid process.) The alternatives available through use of a dual-bus CPU (e.g. PDP-11/70) would be explicitly available to the system designer as a result of enhancements (a) and (c).

System Software

Two HOL design-related topics can be further explored in support of modular systems design. These are process concurrency as a methodological consideration and the provision of high level design aids such as "built-in functions" and "machine-like code".

Concurrency is important whenever the system designer wishes to utilize CPU time made available by the hardware execution of any machine or algorithm function. The problem is complicated when code transportability is a prerequisite. Many systems utilize asynchronous and/or concurrent processes and require control of these processes. Several HOL constructions have been developed for interfacing and controlling these concurrent

processes. However, these HOL constructions do not provide the methodologies which lead to efficient design of concurrent systems. In particular, when two processes are executing concurrently, it is to be expected that one will complete before the other and time will be wasted until the other completes. This is especially aggravating if the first process to complete is software implementation and the second is special purpose hardware (a classical example of this is magnetic tape I/O functions).

A methodology is desired which permits the design of modular concurrent software without the burden of excessive flag testing, completion interrupts, or wasted processing time. The methodology should not require excessive care by the programmer to achieve software which can be concurrent to other processes and provide useful work to the processor during wait periods. It is expected that a methodology for concurrency will impact the HOL provisions for construction of concurrent processes.

Built-in functions and machine-like codes are examples of methods for compiler level stipulation of time-critical functions (or functions where some other dictate of convenience requires optimally efficient execution). These functions again limit code transportability, and their documentation is a proper topic for an improved design methodology.

Hardware-Oriented Operating Systems

The design of hardware-implemented operating systems is currently being discussed for special purpose applications, such as real time signal processing. Based upon the needs of communications applications signal processing, a study should undertake to stipulate the features desired in such a specialized hardware operating system.

III. Appendices

Primarily because this study required the generation of analytic "tools", a large part of the results cited in Sections 1-5 are supported by analyses and/or descriptions whose details would overly encumber the report. Therefore, whenever possible, intermediate results, analyses of related topics and, of course, reference material such as program listings were included in a set of appendices.

Work which has been reported in monthly progress reports, and is particularly germaine to the purposes of this study, can be found in the areas of architecture selection and language analysis. These are described in Appendices D, G, and H (architecture) and, M (language).

A separate study which arose as a part of the module definition process was performed on the applicability of array processors for signal processing in a modular context. Results were fairly promising and are reported in Appendix B.

The remainder of the appendices contain background material of interest to narrower groups of users.

1. Generic System Definition

1.1 Introduction

The principle goal of Task 1 of the Statement of Work is system definition. The result of this major part of the study is an architecture description and its model, supported by representative signal processing and switching modules. This result is based upon prior architecture selection which was in turn supported by development of sets of analysis criteria covering cost, reliability, and performance--with various elements within these categories being separately weighted for signal processing and switching.

Before undertaking this extended period of classification and analysis several topics of a generic nature (to real time systems) were considered. These were:

(1) What are the bases for modularity? i.e., "why go modular"?

(2) What functional level is most appropriate for module development?

(3) What distinguishes architectures, i.e., what is generically significant?

(4) How can one apply objective and analytical techniques to architecture selection, normally an intuitive or determined-by-default process.

(5) What are the salient and distinguishing requirements of signal processing and switching systems?

Topics 2-5 are addressed in the remainder of this section. The question of the desireability of adopting modularity is properly raised because of the realization that if one time production and deployment costs are a developer's sole cost criteria, the simplicities of specialization commonly outweigh the benefits of modularity. That is, a system designer

is usually not accountable for the long term cost of a system but rather is responsible for minimum development and (occasionally) minimum production costs. However, the balance of this study is based upon the attainment of lower 15 year life cycle costs. Architectures to attain these goals necessarily feature a high degree of order - thus hardware that is functionally and physically modular, software systems that coincide with forthcoming DOD standards, and design methods that support both design review and maintenance documentation are the basic elements of this study.

1.2 Motiviation for Modular Architecture

The objective of any system design endeavor is to transform an established user need into a functioning system whose performance attributes adequately satisfy the users operational requirement. However, a user requirement, once established, generally persists and evolves over a period of time such that functional systems fulfilling the need must periodically be modified and usually expanded and made functionally more complex to match the evolving need. During this system evolutionary process, advancing technological means are applied to enhance performance effectiveness - a term which includes such factors as increased performance per cost unit, increased reliability, reduced size, expanded range of automation, and more efficient and accurate technical fulfillment of the user need addressed. In fact, the evolution of technology itself creates expanding user needs and we currently find that development cycle time for new systems is such that by the time a system finally enters the operational phase its technology is obsolete and new demands on the user are stimulating a new development cycle.

Clearly, the system designer must recognize and accept these evolutionary forces by structuring and implementing systems in such a way that their evolution in response to need and technology is economically and efficiently achieved. The concept of a "modular system architecture" addresses the reality of system evolution and implicity provides the mechanisms for:

a) adaptibility to evolving use requirements and technology;

b) gradual or incremental upgrade without unacceptable development cost or operational trauma;

c) standardization of functional elements without unacceptable penalties on total system effectiveness.

"Modularity" implies that system functions, processes or algorithms exist as discrete components or modules which can be organized or reorganized into a variety of system structures. Each system structure composed of a collection of interdependent modules must be capable of performing a distinctly different collective function while preserving the functional and physical identity of each discrete module. This in turn implies that a common interconnection scheme exists which permits the aggregation of modules without destroying their unique identities. The interconnection scheme must be selected to permit the widest range of total system functions to be achieved - that is, the interconnection scheme must not itself impose constraints on potentially achievable system performance over the set of possible module aggregations.

There is a reluctance by industry to build modular systems. The production cost, size, power, etc., of modular equipment will be greater than non-modular equipment due to the penalty of designing structured

modules with common interfaces. In addition, the responsibilities assumed with modularity may imply a greater need for customer design than most users are actually willing to undertake - considering such possible outcomes as reduced warranty, incomplete specifications, etc. Unless the complete life cycle cost is considered which includes acquisition, operation, and logistic support the true cost of the equipment is not seen. It was suggested at the "Mini/ Micro Computer Conference" held in San Francisco, October 1976, that a method of changing this reluctance would be to purchase a warranty with the equipment causing the seller to reflect the true cost of the equipment at the time of purchase. While this may not be the correct solution, it illustrates the problem which both the industrial and the military communities have.

1.3 Modularity

Modular equipment implementation has been studied, proposed, and tried many times before and literature is full of examples of the earlier work.

Standard Electronic Modules (SEM), Quick Easy Design (QED), and Register Transfer Modules (RTM) are three good examples of the attempts at modular equipment. The RTM group of modules were actually implemented and marketed by Digital Equipment Corporation but have not gained the acceptance of popular use. These modules compete in the commercial market place where added cost of equipment is a large portion of the total cost of the system due to the short life of the equipment. Because of the longer and more complex life cycle of military equipment, additional acquisition costs for modularity-supportive equipment are small compared to total life cycle cost of the system. This fact should cause modular systems to be more acceptable in the military environment.

The earlier studies have progressed with electronic technology from modularity at the standard component level through the standard minicomputer. With the advent of micro-computers, imbedded processors are practical in new designs. These processors implement far more complex functions than previously possible in modules of a practical physical size. As a consequence of this increased performance capability, there exists the need to define a standard method of control and communication between modules which will insure compatibility and preserve performance. It should be recognized that with this type of system, a module may be considered as either hardware or software and the two will most likely co-exist in the same equipment. (The most likely implementation of a hardware module will be via microprogramming.) Therefore, the interface and control problem is different than that identified in earlier all-hardware or all-software module systems.

The Q.E.D. paper referenced in Appendix N provided a good basis for defining levels of modularity in hardware systems. Figure 1.3-1 shows the relationship between levels found in modular hardware and in modular software. As in the Q.E.D. paper, an assumption made in defining the levels of modularity for this study is that the elements of a lower level can be used to define a higher level but never the reverse. Hardware design has progressed from the component level to the building block level, and in the last few years entered the function level. The top two levels are major equipment levels, and a form of unit standardization has existed for years when it was practical. (Avionics and data terminals are examples.) Software design has progressed at all levels and major work

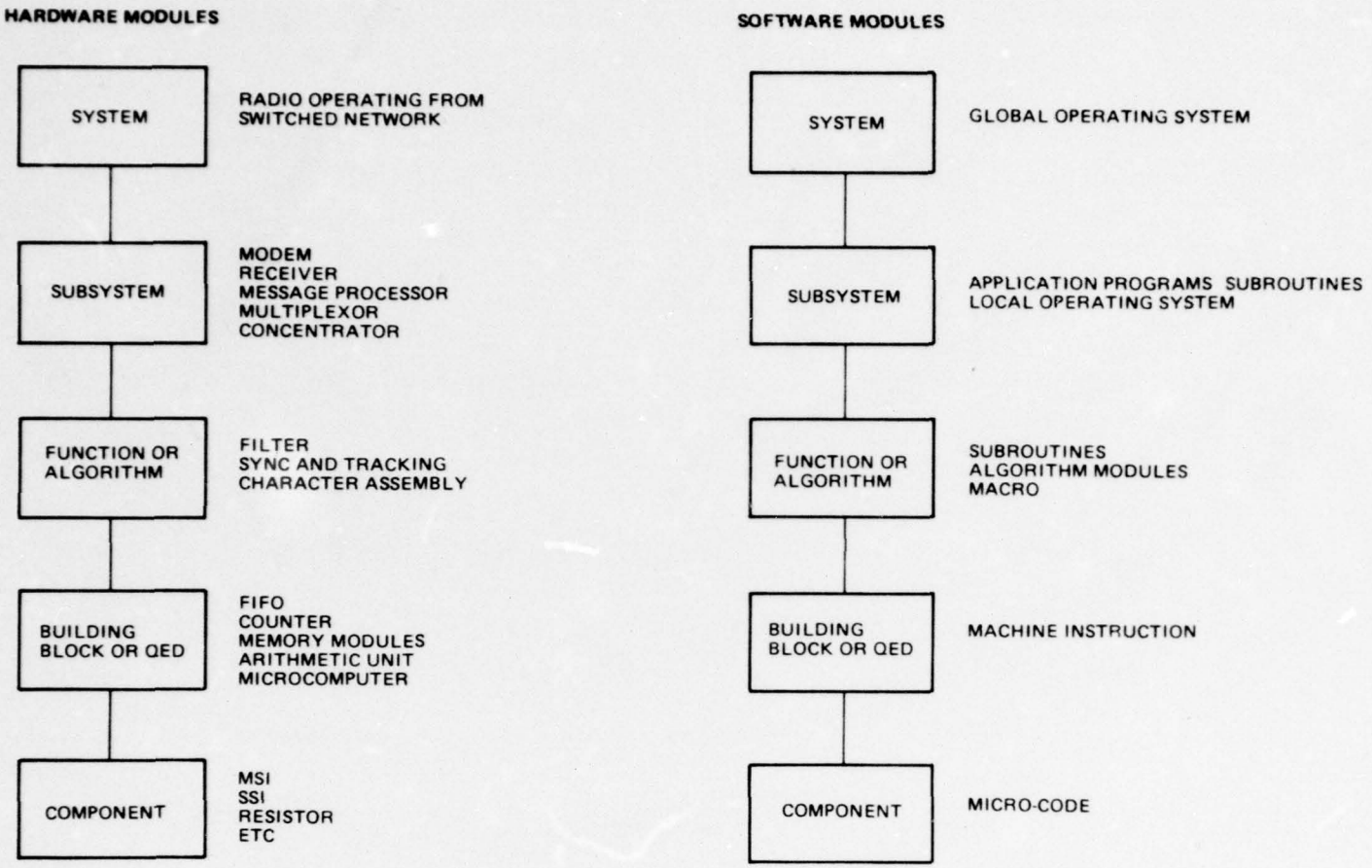


Figure 1.3-1. Relationship Between Hardware and Software Modularity.

12177-2

has focused on the function level. As a means of effectively managing and scheduling a large software system, modular software has become a necessity.

"High Level Languages" and "Top Down Programming" have been used as tools or methods for designing large software development tasks in an efficient manner. It should be noted that although "Top Down" programming will achieve modular software, the use of a High Order Language (HOL) does not necessarily provide it, unless used in conjunction with a "Top Down" approach to programming.

The method of defining a module implemented in either hardware or software is in terms of its functional algorithm, control, and I/O. This statement appears to be true at any level of modularity. Appendix C is a description of the parameters required when defining a module at the functional level. Some of the parameters listed are not available at the initial module partitioning phase of the design process. As the overall design proceeds these parameters become available.

The control structure the designer chooses to tie the modules together greatly influences the success of the overall design. At the functional level of modularity, the control must be capable of supporting the processing power of the modules selected i.e. allow distributed control, yet allow the system designer a "utility" type service when simplicity is possible, as is the case in low speed deterministic processes.

1.4 Application Effects

Selection of an architecture for both signal processing and switching requires an analysis of the type of functions included in each system.

These functions tend to sort into two categories: 1) Signal Processing; and 2) Communication Processing. Figure 1.4-1 lists the types of functions found in these two categories. The separation of switching functions and signal processing functions is vague since operations of either category can be found in either type of equipment or system. Still, the differences shown in Figure 1.4-2 for the two categories suggest using different types of modules for them. This concept fits well with the algorithm or function-level partition selected. Modules at this level may include imbedded processors, each uniquely suited to solve the specific algorithm or function.

The specification of an architecture includes the method of data transfer within the architecture. A second requirement is determination of the type of module execution control to be used within an architecture. Figure 1.4-3 is a table categorizing the general control methods available to the designer of equipment. The last category, called mixed mode, points to a categorization problem, i.e., the list of control methods could be quite long. The task then, is to limit the number of control methods used, while preserving essential features. It has been found that centralized control of the task allocation routines within a signal processing system simplifies the overall control and can be achieved, since processing times are fixed intervals in signal processing. A switching system is different, in that message arrival rates are not fixed so that decentralized control of the task allocation routines is more desirable. These differences point to a need for flexible control within the architecture. Another consideration for flexibility is that although

TYPICAL SIGNAL PROCESSING FUNCTIONS

- BASEBAND CONVERSION
- CARRIER SYNCHRONIZATION
- BANDSPREADING
- BIT SYNCHRONIZATION
- BIT DETECTION
- POST DETECTION SIGNAL ENHANCEMENT

TYPICAL COMMUNICATIONS PROCESSING FUNCTIONS

- SYMBOL DETECTION
- KEY DEMULTIPLEXING
- WORD PROCESSING – CODING, FRAMING, EDAC
- MESSAGE EXPANSION
- MESSAGE DELIVERY AND PROTOCOL
- RECORD KEEPING
- EDIT/ROUTE/QUEUE

12877-1

Figure 1.4-1. Comparison of Processing Functions.

SIGNAL PROCESSOR CHARACTERISTICS

- HIGH SPEED COMPUTATION
- DATA INDEPENDENT COMPUTATIONS
- FIXED TIME EXECUTION DUE TO NON-BRANCHING OPERATIONS
- HIGHLY REPETITIVE OPERATIONS – SCALING, MULTIPLY, SUM

COMMUNICATION PROCESSOR CHARACTERISTIC

- HIGH VOLUME THROUGHPUT
- FEW OPERATIONS PER TRANSFER
- FEW ARITHMETIC OPERATIONS
- MANY BRANCHING FUNCTIONS
- DATA BASE

12877-2

Figure 1.4-2. Comparison of Processor Requirements.

CENTRALIZED CONTROL

- DRIVEN – INTERLOCKED OR NON-INTERLOCKED
  - CENTRAL TIMEBASE DRIVES A TASK TABLE RESIDENT IN A CENTRAL CONTROL MODULE
  - DISTRIBUTED TIMEBASE DRIVES A TASK TABLE RESIDENT IN A CENTRAL CONTROL MODULE
- POLLING – INDEPENDENT SERVICE REQUESTS ARE INTERROGATED BY THE CENTRAL CONTROL MODULE

DECENTRALIZED CONTROL

- DRIVEN – MODULES MAY START OTHER MODULES BY MEANS OF INTERRUPTS
- STATUS FLAGS – MODULES PASS CONTROL BASED UPON COMMAND WORDS OR STATUS FLAGS.

MIXED MODE

- ANY COMBINATION OF THE ABOVE TYPES OF CONTROL

12877-20

Figure 1.4-3. General Control Methods.

information transfer takes place through the common facility, certain interface modules should be allowed access to I/O ports independent of the common facility.

For example, in signal processing, the amount of information that needs to be passed from module to module tends to decrease as more processing is completed. Thus a radio's "IF" data is sampled at a high rate, filtered, and only the results of this pre-processing operation are passed to the next module for further processing. The output of the last module might be characters whose output rate is much slower than the original sample rate of the "IF" signal.

Another application dependency is physical in nature. Signal processors tend to be located in a smaller physical area than switching systems; the latter may contain terminals which are geographically isolated. This study focused on the non-distributed operation; however, switching systems occupying space equivalent to a medium sized building were considered. In a typical switching system, this system could comprise small message switches, concentrators, etc., which may be tied by wire-line to other geographic locations.

1.5 Architectures

The three architectures selected represent the generic families of Serial (TDM/FDM), Matrix (Serial, Space Switch, Separate Control Channel), and Parallel Bus (Ansychronous, Interlocked). Appendix A contains a description of these architectures. A major portion of the description defines the method of control within the architecture. The control structure of the architecture determines to a great extent the performance characteristics of the architecture.

For any architecture or modular design technique to be accepted and used, a certain amount of flexibility must be allowed within it. This closely parallels the concept of the escape mechanism required in a HOL to allow lower level coding. The "escape" in the architecture must allow some freedom in module communication, control, and the level of module implementation, to be acceptable to a wide variety of users.

1.6 Architecture Selection

The process of selecting an architecture for further analysis and study pointed to a need for selection criteria. Appendix D lists the criteria developed for application to a variety of systems. These criteria can be used as a checklist against architectures being considered for other application areas to insure that all meaningful attributes have been considered. (Other applications, e.g. control systems, would naturally associate a different weighting than those shown in Appendix D for signal processing and switching.) The list was generated by applying previous experience and by applying new knowledge gained from a survey of current technical literature.

Further practical considerations of problem sizing established the need to determine the class of problem in which the architecture will be used. This eliminates distortions due to constraints placed on the limits of performance of any one attribute. A typical example would be a system with data transfers in the gigahertz range which could not be passed through any of the three architectures; they are all unsuitable for this class of problem. Appendix E and F describe example systems of the class we are considering. The class of signal processing problem we have considered is of the encrypted data communication category. The switching system is an example of a small military message switch.

Complete descriptions of the selection process are contained in Appendix G and H, for signal processing and switching systems, respectively.

Figures 1.6-1 through 1.6-4 are graphs summarizing the results of the comparison process. The results were analyzed from two viewpoints. The Architecture Comparison Summary Figures 1.6-1 and 1.6-3 were achieved by summing the score by attribute (1st, 2nd, 3rd) and weighing the attributes by relative importance.[1] The latter weighing is always application dependent. The results shown on this chart did not clearly reflect a general advantage to be obtained by using any of the three architectures.

A second way to look at the comparison process is to sum the best/worst choices by attribute (see Figures 1.6-2 and 1.6-4). These charts are more meaningful because they more clearly reflect the advantages/disadvantages of using a given architecture. The parallel bus architecture described in Appendix A-1 was separately selected for future modular systems design for both signal processing and switching systems.

---

[1]The architecture selection process, briefly described, consisted of separate specialists developing attributes in 3 areas: cost, reliability, and performance. The architectures were compared on a "by attribute" basis, and scored best, median, worst. The specialists further indicatd the most and least important attribute in each area; in the final summations those items were respectively weighted doubly (most important) and disregarded (least). The weighting of best-median-worst for each attribute was converted to a numercial 4-2-1 for the signal processing raw summaries, however "best", "worst" was used directly for te comparisons shown in Figures 1.6-2 and 1.6-4. Appendices G and H describe the comparison process in greater detail for signal procesing and switching systems respectively.

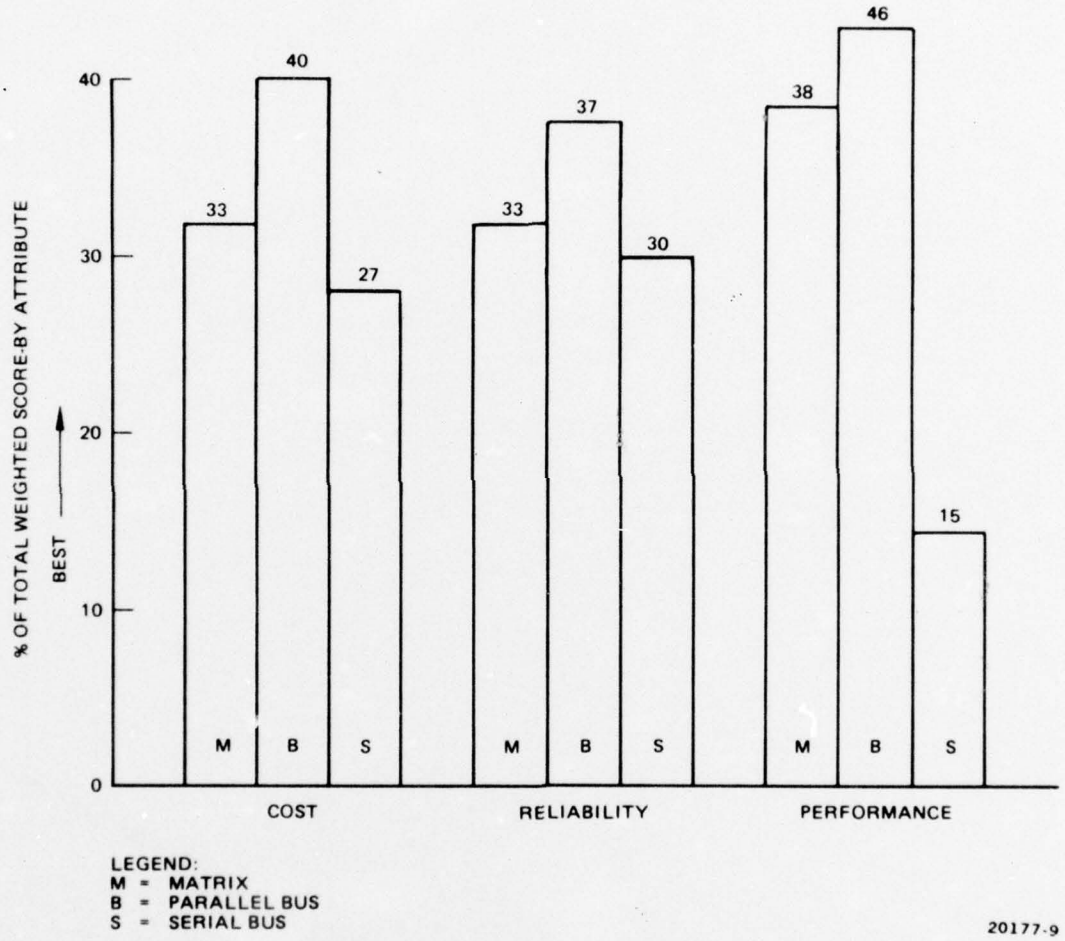


Figure 1.6-1. Architecture Comparison Summary for Signal Processing.

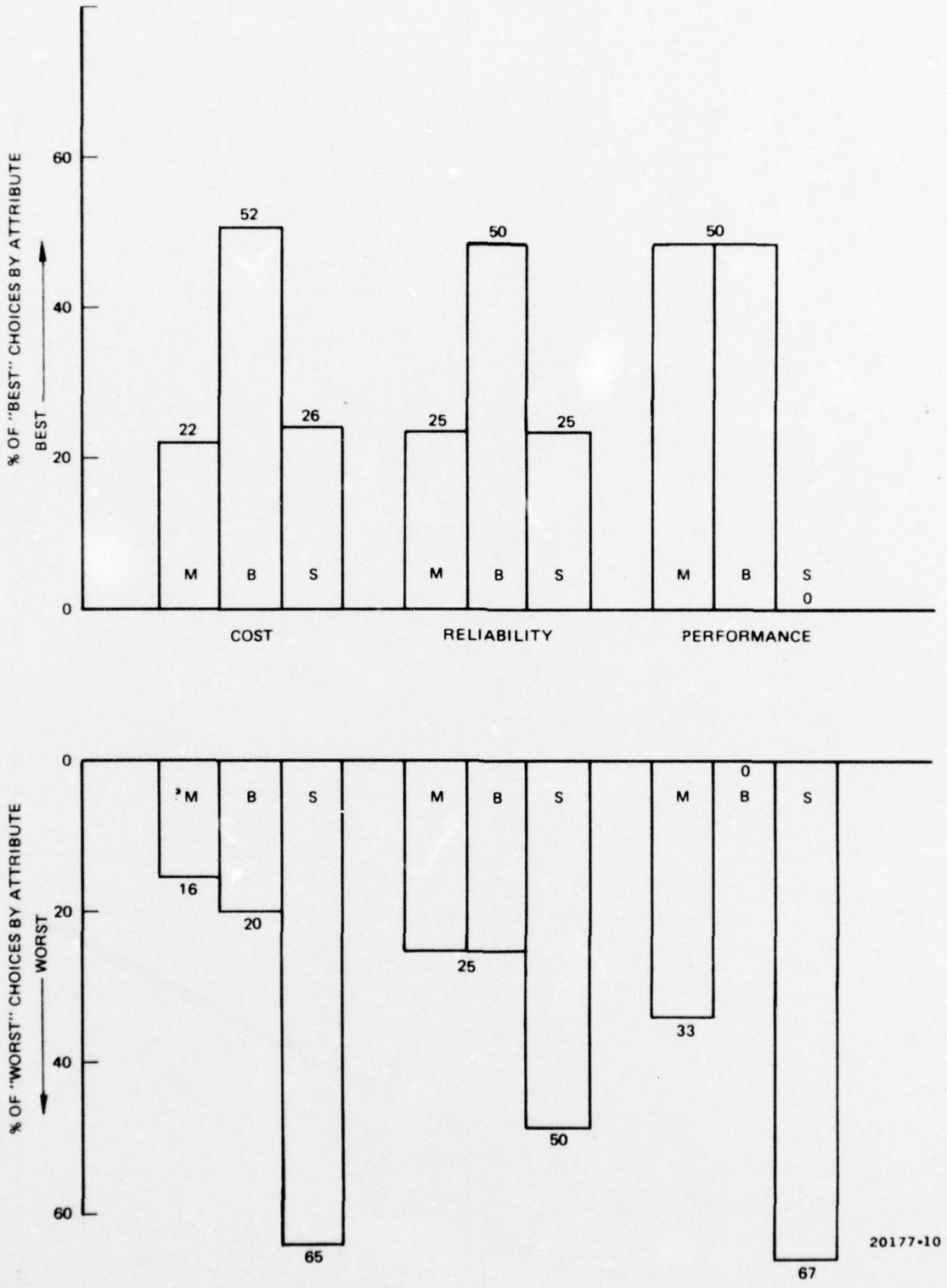


Figure 1.6-2. Best/Worst Comparison by Attribute for Signal Processing.

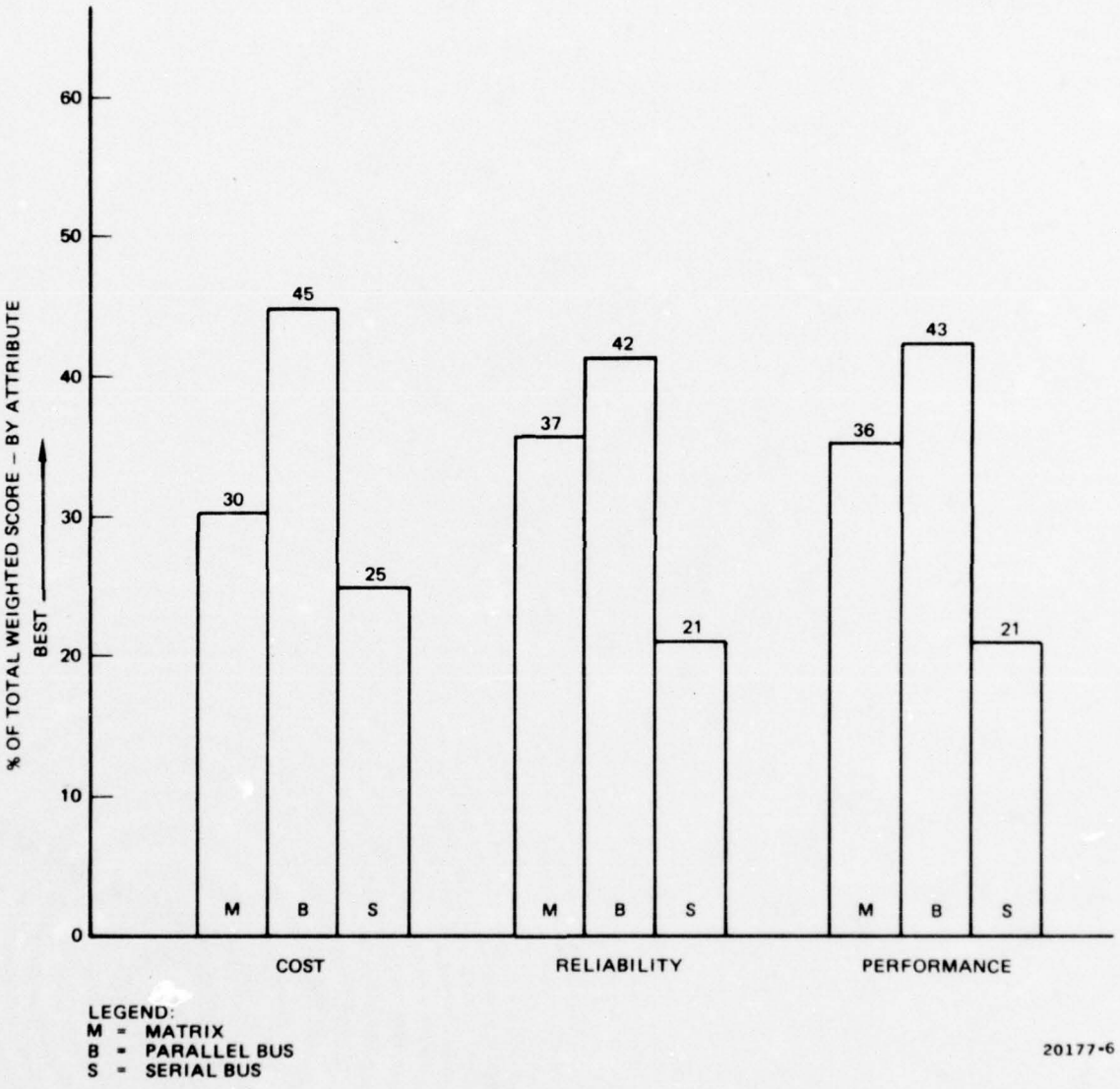


Figure 1.6-3. Architecture Comparison Summary for Switching System.

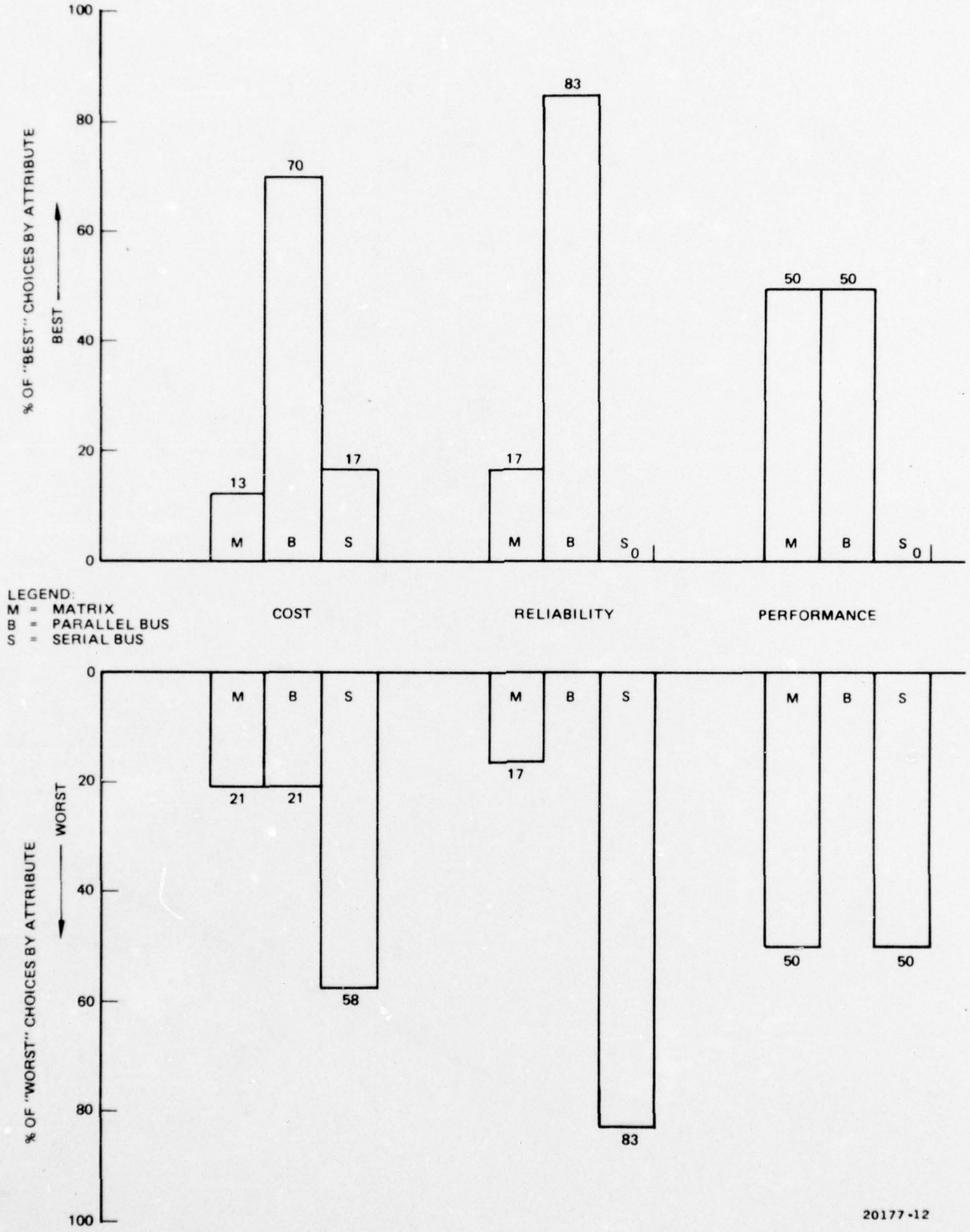


Figure 1.6-4. Best/Worst Comparison by Attribute for Switching Systems.

2.0 SIMULATION

2.1 Selection of GPSS

The IBM General Purpose Simulation System (GPSS) was used as the simulation system for this study. GPSS provides a language which is concise and adaptable to a wide range of problems. GPSS is well documented thus making the learning and use of it very accommodating. It provides standard reports for all identified facilities, queues, storage, etc. It also provides instructions to create specific formats and house specialized reports if the standard printouts are not sufficient. "HELP" instructions to access designer coded FORTRAN routines or machine coded routines are also available if more unique operations are required beyond the GPSS instruction set.

2.1 GPSS Description

The IBM General Purpose Simulation System (GPSS) provides a flexible tool for the analysis of systems of a complex logical and procedure oriented nature. It provides an effective means of testing, evaluating, and weighing alternatives of a proposed system. The models, however, are not the precise analog of the actual systems, but rather a symbolic representation of the system.

GPSS is a generalized language which incorporates a series of entities which are described by GPSS statements to represent the features of the system to be modeled. These entities are divided into six categories. Basic, equipment, computational, statistical, reference, and chain. Refer to table 2.1-1 for the types of entities in each category.

Category	Type	Switching System Examples
1. Basic entities	1. Transactions	Message traffic
	2. Block	Operations on message traffic
2. Equipment entities	3. Facilities	CPU, disc, tape, etc.
	4. Storages	Memory, secondary storage
	5. Logic Switches	Events that block traffic (circuit down or on hold)
3. Computational entities	6. Arithmetic Variables	Any computation of arithmetic combinations
	7. Boolean Variables	Single decision on many attributes; i.e., circuit down and type of message.
	8. Functions	Computation of rate of message traffic.
4. Statistical entities	9. Queues	Traffic to output circuit, to disc storage, or tape storage.
	10. Distribution Tables	Additional statistics, i.e., message type distribution by time.
5. Reference entities	11. Savevalues	General use for recording data at one time and referred to at another.
	12. Matrices	Circuit type, status, etc. by line (line tables)
6. Chain entities	13. User Chains	Message retrievals
	14. Groups	Accounting of multi-routed messages.

Table 2.1-1 Entities

The most commonly used entities are the "transaction" and "block" entities, both classified as "basic" entities in the GPSS system (see Category 1, Table 2.2-1). A model must as a minimum be represented by blocks and transactions, the latter must be generated to stimulate the system. Other entities are used as required to represent the system features. Once the system is described in terms of GPSS blocks, transactions which represent traffic or data are created by the program, and are moved through the specified blocks, executing the actions associated with each block.

A clock is maintained by GPSS that records the current simulation clock time. With this clock, GPSS may execute simulated events in the correct time sequence. The clock times are maintained in integer values, however, they are not incremented in the conventional manner, but are updated to the value at which the next event is to occur. For example, if the current clock time is 2, and the next event is to take place at 7, the clock time is incremented by five units. The time units do not represent any specific unit of time. It may represent any unit of time needed to obtain modeling precision. Once the time unit has been defined to be a specific unit of time, all time requirements in the model must be expressed in the same terms.

GPSS also maintains a number of standard statistics of the model being run depending on the entities being used. These statistics are available at the option of the user to be automatically printed-out at the end of a run, selectively printed out, or reformatted (within the GPSS format statement capabilities).

A more detailed description may be obtained from the IBM General Purpose Simulation System V Users Manual (SH20-0851-1).

2.1.2 System Used

GPSS V is resident at the Western Computer Service Center, Rockwell International in Downey, California. Five IBM/370 Model 168 computers are installed there with various equipments that are shared and/or switchable from one model 168 to another.

Input to the center is made via batching on a DATA100 (card/printer) terminal located at the Collins Radio Newport Computer Center in Newport Beach, California, or via TSO terminals located throughout Collins.

2.2 Model Description

The architecture selected for modeling was the parallel bus architecture, as described in Appendix A-1.

The model is a representation of the parallel bus structure. The primary aspects of the bus operation; next master, bus master, and priority were modeled while the detailed logic flow (see bus acquisition sequence, Appendix A-1, Figure A.1-5) of the bus activity was not represented. Modeling the detailed logic of the bus activity was determined to be unnecessary because (1) the parallel bus architecture closely resembles the operation of known equipment (PDP-11 Unibus) and (2) the study of the bus utilization and congestion due to data transfer and communications was found to be unnecessarily delayed when a detailed logic representation of the bus was used.

The model can accommodate up to 60 modules controlled by various methods as described in Section 2.3.1. Each module is assigned a priority which

represents position on the bus which in turn determines which module is to be granted control of the bus. However, in order to be granted control of the bus (bus master), the module must have already attained next master status. Priority position thus provides for attaining next master status rather than bus master. Once next master status is attained it is not relinquished to the higher priority. Next master status proceeds to bus master status as soon as it is relinquished. This model allows only a single word transfer during each bus mastership whereupon the module relinquishes control and again bids for bus control by bidding for next mastership. With this sequence, a module does not monopolize the bus with any data transfer, nor does a high priority module continuously receive bus control. The highest priority module is, however, guaranteed control every other cycle time, if desired.

The bus operates on a master-slave relationship in that for each control signal issued by the master, a response must be received from the slave (destination module). In this model, the slave always answers, whether it is busy or not. The model maintains a queue for each called module, thus maintaining statistics on its activities. A separate queue is also maintained for output in the event that separate control is issued to a module for output of data. A call made to a module that is currently busy will thus be held in queue until the current process is complete. The process includes such activities as input and output of data on the bus (module to module), if common memory is available the intra-algorithm use of memory by the module, and the estimated module execution time. These processes are done serially and statistics are recorded on each.

This model is a general purpose model in that it is driven by parameter specification of each individual module. These parameters are described in Section 2.3.

The statistics accumulated during the simulation are presented at the end of the simulation run. The printout is in a standard GPSS format. The report types are described in section 2.4.

Verification of the model was accomplished in two levels. The first level, was the verification of results as each option (parameter setting) was added. These results (printouts) were compared to the expected results of the option. The next level of verification was accomplished by selecting several specific sets of modules and comparing the results of the simulation with hand calculated results. For the deterministic case of signal processing these comparisons were exact.

The Switching System bus utilization was estimated by computing an estimation of the total characters tranferred over the bus. This was done by using the mean values for total messages handled, characters per message, number of output messages per input message, and the time required per transfer. As an example of these steps, the busy hour is tabulated below. (PLRS, Position Reporting Location System, defines a separate class of fixed-length, perishable, messages for which statistics were kept separately.)

Total Messages	20,000 PLRS	+ 2500 non-PLRS
Mean characters/message	X 80	X 320
Characters in	1,600,000	800,000
output message/input	2.8	3.0

	1,600,000	800,000		
	X 3.8	X 4.0		
Total characters/hour	6,080,000	+ 3,200,000	=	9,280,000
Total characters/second			=	2577.7
2577 ch/sec X 5.5 usec/ch X 100%			=	1.42%

In the simulation of the busy hour characteristics, the total I/O Data Transfer was listed as 1.61%, which verifies the above calculation.

2.3 Parameter Description

The model of the parallel bus architecture will accomodate 60 separate modules and operates on each module according to its individual parameter specification. Each of the modules may be defined by up to 21 different parameters. Fourteen parameters are optional (have known default to zero conditions) and seven parameters must always be specified. These are identified (*) in the parameter list in Table 2.3-1. As shown in Table 2.3-1, there are currently 21 parameters that define a module; 10 byte size parameters, 6 halfword size parameters, and 5 full word size parameters. These three sizes were defined according to information length requirements of each parameter.

2.3.1 Byte Parameters

1. Module Activation: This parameter defines whether the module is to be included in the simulation run. It can either be a "1" for active or a "0" for inactive. Thus, modules may be defined* and activated as required prior to a simulation.

---

* A module summary, Table 2.8-3, lists a representative set of signal processing modules.

TABLE 2.3-1A. BYTE Parameters.

MATRIX NAME – BPARM

ROWS – MODULES 1 THRU 60

COLUMNS

1	MODULE ACTIVATION	1 = ACTIVATE
2	PRIORITY 0 TO 127 (HIGHEST)	
3	TYPE OF CONTROL	0 = DECENTRALIZE 1 = CENTRALIZE 2 = MODULE INTERLOCK (REFER: COL. 8) 3 = MODULE CALL (NOTE: THIS MODULE NO. IS DEFINED IN CALLING MODULE PARAMETERS)
4	TYPE OF MEMORY USE	0 = DISTRIBUTED MEMORY 1 = COMMON MEMORY INPUT 2 = COMMON MEMORY OUTPUT 3 = COMMON MEMORY BOTH
5	INPUT/OUTPUT TIME DISTRIBUTER MODIFIER	
6	MODULE TIME DISTRIBUTION MODIFIER	
7	I/O DATA DISTRIBUTION MODIFIER FOR MODIFIERS ABOVE	0 = NO MODIFIERS 1 = EXPONENTIAL 2 = NORMAL, STD DEV = 1.0 3 = HYPERBOLIC 4 = ERLANG M = 2 5 = ERLANG M = 3 6 = ERLANG M = 5 7 = ERLANG M = 10
8	MODULE INTERLOCK NO.	
9	MODULE NO. OUTPUT DIRECTED TO (SEE CONTROL TYPE 3)	
10	MAXIMUM QUEUE LENGTH	

12877-17

TABLE 2.3-1B. HALFWORD Parameters.

MATRIX NAME – HPARM
ROWS – MODULES 1 THRU 60
COLUMNS

1	NO. OF INPUT WORDS
2	INPUT READ TIME, EACH INTEGER = 50 NS
3	NO. OF OUTPUT WORDS
4	OUTPUT WRITE TIME, EACH INTEGER = 50 NS
5	NO. OF INTERNAL READS FROM COMMON MEMORY
6	NO. OF INTERNAL WRITES FROM COMMON MEMORY

12877-18

TABLE 2.3-1C. FULLWORD Parameters.

MATRIX NAME – FPARM
ROWS – MODULES 1 THRU 60
COLUMNS (NOTE: EACH INTEGER = 50 NS, e.g., 5000 = 250 μS)

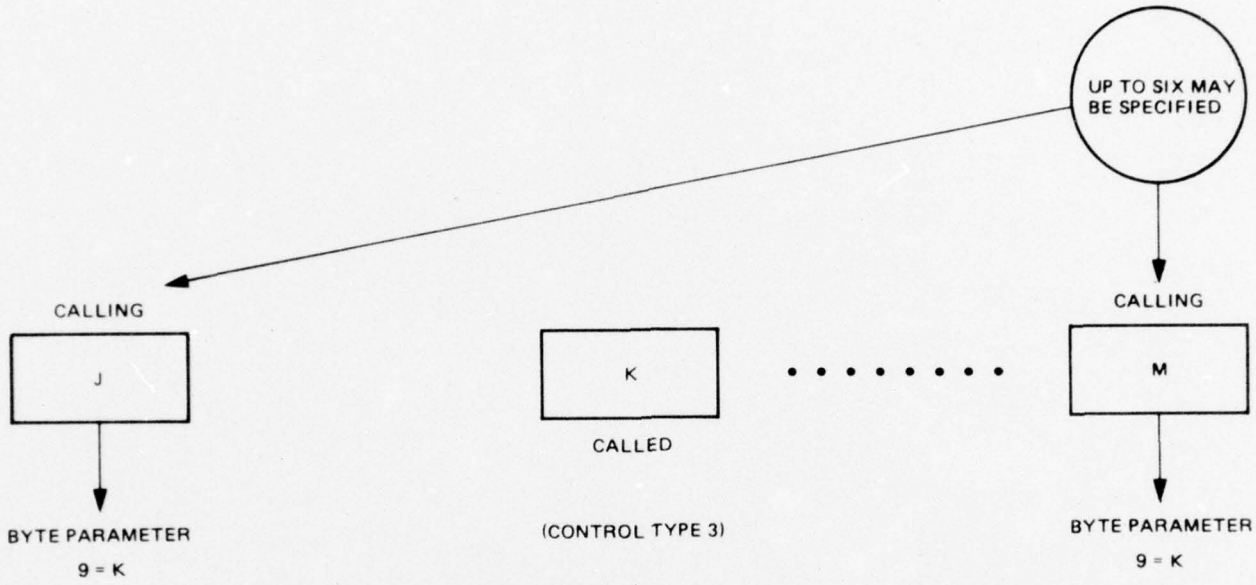
1	MODULES START CYCLE TIME
2	MODULES OUTPUT INTERVAL
3	MODULES PROCESSING TIME
4	START TIME OFFSET
5	OUTPUT TIME OFFSET

12877-19

2. Priority: This parameter defines the module priority; that is its position on the bus which determines the priority order of obtaining next master status. "0" to "127" may be specified, where the highest number defines the highest priority.

3. Type of Control: Currently, four types of control may be specified in this model. Since this is a module parameter, a mix of controls may be specified.

   - o "0" = decentralized control: In this type of control, the module assumes the responsibility of scheduling itself. No bus activity results from this type of control.
   - o "1" = centralized control: A built-in control module in the model schedules the modules according to each module's time parameters (see fullword parameter). It simulates the generation of an interrupt vector on the bus and records its bus activity.
   - o "2" = Module Interlock: This type of control is provided by another module, at the completion of its function (at output). Thus, module "a" is said to be interlocked with "b" when the completions of b's function is required to initiate "a". That module number is specified in byte parameter 8. More than one module may be interlocked in a pipeline fashion. Interlock modules may be used to simulate software modules that comprise a primary module, such as a CPU module.
   - o "3" = Module Call: This option provides the capability for a module to receive control from another module, similar to control type 2 above. The called module control type parameter is set to a 3, and its module number is set in the calling module parameters (see byte parameter 9 and refer to Fig. 2.3-1). Up to six



TABULATION OF SUMMATION OUTPUT FROM J TO K:

IF J'S OUTPUT IS GREATER THAN K'S INPUT TIME, J'S OUTPUT IS USED.
IF K'S INPUT IS GREATER THAT IS USED.

TABULATION OF SUMMATION OUTPUT FROM M TO K:

IF M'S OUTPUT IS GREATER THAN K'S INPUT TIME, M'S OUTPUT IS USED.
IF K'S INPUT IS GREATER THAT IS USED.

20377-8

Figure 2.3-1. Module Call Option.

calling modules may be specified. The model tabulates all the transfer times of the calling modules output data to the called module. It selects the greater of the two access times (calling versus called) for data output.

4. Type of Memory Use: The model currently has a built-in common memory which has a single parameter (specified as a system level parameter) which specifies the memory access time. Usage of the common memory by a module is tabulated using the memory access time. However, if the module access time specified is greater, that time will be used. Each module may specify one of four types of common memory use as follows:
   - o "0" = distributed memory; common memory not used.
   - o "1" = common memory input; data is input from common memory.
   - o "2" = common memory output; data is output to common memory.
   - o "3" = common memory both; data is both input from and output to common memory.
5. Input/Output Time Distribution Modifier: This parameter specifies a continuous numerical function that is used as a modifier to the specified module start cycle and output interval (see fullword parameter description) to provide a variable start/output cycle. Currently, seven functions are available (see Table 2.3-1).
6. Module Time Distribution Modifier: This parameter specification is similar to (5) above except that it is used as a modifier to the specified module processing time (see fullword parameter description).
7. I/O Data Distribution Modifier: This parameter specification is similar to 5 above except that it is used as a modifier to the specified module input word and output word count (see halfword parameter description).

8. Module Interlock Number: This parameter is set in conjunction with control type equal to 2 (refer to control type parameter 3 description above). The module number of that module which provides the control is specified.

9. Module Number, Output Directed To: The module number to which output is directed to (called module) is specified in this parameter. The model tabulates the output time (the greater of the two I/O times) under the called module statistics. Control to the module may also be provided if the control type parameter of the called module is set to 3 (see type of control above).

10. Maximum Queue Length: Specification of this parameter will terminate the simulation abnormally when and if the conditon is met. Queue statistics for that module will be printed followed by the normal report format (statistics will be tabulated up to point of termination). Normally in real time systems such as signal processing, any module in the stream must complete its processing before being called again, otherwise it will not catch up again. The signal processing modules are all initially set to 2.

2.3.2 Halfword Parameters

1. Number of Input Words: This parameter specifies the number of words to be input under this module's control. If a modifier has been specified (byte parameter 7), this parameter indicates the mean value of the number of words. These words are input on each Input Start cycle.

2. Input Read Time: This parameter specifies the input time for each word input by the module. Each integer specified represents 50 X 10^{-9} seconds. This time plus the bus time (system parameter) is the

actual bus access time. If common memory input is specified, the input read time is compared with the common memory access time (system parameter), and the largest of the two plus the bus time is the actual bus access time used, i.e., Access Time = I/O time + bus cycle time, where I/O time = the larger I/O time of the source and destination modules.

3. Number of Output Words: This parameter specifies the number of words to be output under this module's control. If a modifier has been specified (byte parameter 7), this parameter represents the mean values of the number of words to be output. These words are output at each output interval or each time the module is provided control via the interlock or call method of control.

4. Output Write Time: This parameter specifies the output time for each word output by the module. Each integer specified represents 50 X 10^{-9} seconds. This time plus the bus cycle time (system parameter) is the actual bus access time. If common memory output is specified, the output write time is compared with the common memory access time (system parameter), and the largest of the two plus the bus cycle time is used as the actual bus access time for each word.

5,6. Number of internal common reads, writes from common memory: These parameters specify the number of reads and writes to common memory used internally by the module for such things as indicators, flags, temporary storage, etc. The actual bus access time per word is determined identically to the read (or write) time above.

2.3.3 Fullword Parameters

1. Module Start Cycle Time: This parameter specifies the rate at which the module is provided control. Each integer represents 50 x 10^{-9} seconds. As an example, if the module start cycle time is 625 x 10^{-6} seconds, the parameter is set to 12500. If a modifier is specified (see byte parameter 5), the time specified must represent a mean time. The first interrupt vector is sent at time = 0.
2. Module Output Interval: This parameter specifies the rate at which the module is to output data. This interval must equal to or be a multiple of the module start cycle. Each integer also represents 50 x 10^{-9} seconds. If a modifier is specified (see byte parameter 5), the time specified must represent a mean time. At startup time, the model does not send the interrupt vector at time = 0, but begins the output interval at time = 0 plus output interval.
3. Module Processing Time: This parameter specifies the processing time required by the module each time it is started (refer to fullword parameter 1). Each integer also represents 50 x 10^{-9} seconds. If a modifier is specified (refer to byte parameter 6), the time specified must represent a mean time.
4. Start Time Offset: This parameter defines a value which the start cycle time be offset from real time. As an example, if the offset is 5 and the start cycle time is 100, the first interrupt vector will be generated by the model at time 0 plus 5. The succeeding interrupts will then be generated at time 0 plus 105, time 0 plus 205, etc.
5. Output Time Offset: The parameter defines a value in which the output interval is offset from real time. Using the same example above for the output interval and remembering that output interrupt does not

begin at time = 0 (refer to fullword parameter 2 above), the first interrupt vector will be generated at time 0 plus 5 plus 100.

2.3.4 System Parameters

Several parameters are specified at the system level. These include the bus speed, the common memory access time, and the length of simulation time.

1. Bus Cycle Time: This system parameter represents the speed of the bus. The bus speed is currently specified at 4 MHz. This represents 250 x 10^{-9} seconds, maximum bus access. At 50 x 10^{-9} seconds per unit, the integer 5 is currently set as the bus cycle time. This value is added to the I/O rates specified in each module when performing word transfers on the bus.
2. Common Memory Access Time: This system parameter represents the speed of a common memory unit on the bus. Currently a 1 x 10^{-6} second unit is used, or 20 units. If common memory is specified, this value will be used as the I/O time if it is greater than the specified module I/O times.
3. Length of Simulation Run: This parameter specifies the length of the simulation run and is represented by two parameter values. The first TPERD specifies a certain period of time, i.e., for a 100 ms period, TPERD is 2000000. The second parameter TIME specifies the number of periods to repeat. Thus, if set to 5, the total run time equals to 500 ms.

2.4 Statistical Printouts

The simulation printouts provided are categorized as (1) Configuration Parameters and (2) Performance Characteristics. The former lists, by parameter type, all the parameter values described in section 2.3 for each of the modules defined. The active indicator (set to 1) describes the modules activated in the simulation. The performance characteristics provide the tabulation and statistics as a result of the simulation. Each of the characteristics are described in the following sections.

2.4.1 Input/Output Average Times (IAVG)

This report is a summary of the I/O times of each of the modules activated. It provides the average times for input data (halfword parameter 1), output data (halfword parameter 3), internal reads (halfword parameter 4) and internal writes (halfword parameter 5). The measurement of time is from the beginning of each I/O block transmission to its completion. It thus includes bus time as well as any wait times due to higher priorities. This measurement is accumulated throughout the simulation run. At the end of the run, averages are taken of each type of I/O block transmission by dividing the total I/O time accumulation by the total number of block transmissions.

Also included in this report is the average wait time for nextmaster and module time difference. The average wait time for nextmaster is a measurement from the time the module first tries to obtain next mastership to the time it finally obtains it. These times are accumulated each time next mastership is required which is done at each word transferred according to the system definition. The average is then determined at the end of the simulation run.

The time difference is determined at the end of the simulation run by summing the average I/O times and the specified module time (fullword parameter 3), and subtracting this sum from the start cycle time (fullword parameter 1).

These times are all represented in units of 50×10^{-9} second.

2.4.2 Interlock Modules (LOCKS)

This report provides a summary of these modules that were interlocked for the simulation run. It provides the chain of modules (numbers) beginning with the start module, and will report up to seven interlocked modules. For example, if the system has been defined to interlock module 3 by module 2, and module 4 by module 3; the report will show module 2 as a start module interlocking (passing control) to module 3 which in turn interlocks (passes control) to module 4. The report will also show module 3 as a start module interlocked to module 4.

The total I/O times and module times are accumulated for all the interlocked modules. This information (expressed in units of 50×10^{-9} seconds) when summed provides the total time required by these modules; and when compared with the total elapsed time of the run, it provides a ratio of those modules real time requirement. The I/O and module times are reported for all active modules.

2.4.3 Call Modules (CALLS)

This report provides a list of the modules (calling modules) sending data to a particular module (called module). Up to six calling modules are tabulated. I/O totals are reported by calling modules and by the called module. Total module processing time is by called module only. The

called module I/O and processing time totals are reported for all active modules.

2.4.4 Bus Time (TIM1)

This report provides a summary of the actual total time of bus utilization by module. The bus time utilization for each module is shown as a total unit time (each unit equals 50 x 10^{-9} sec) and as a percentage of the total bus time. The percentage is shown in hundredth of a percent. For example, if under the Percent column a value of 112 is shown, it would read as 112 hundredths of a percent or 1.12 percent. The last two rows (row 61 and 62) are the totals of the control module (when centralized control is specified) and the system totals (sums of all modules unit time and percent of bus time) respectively. If centralized control is not specified in any of the modules, row 61 would not be printed or would be equal to zero.

2.4.5 Common Memory

This report provides the percent of total bus time (of all transmissions to/from common memory) by the modules that utilize the common memory. Along with the total percent utilization of time are percent utilization of time by I/O data transfers and by internal read/writes of the modules. Analysis of these statistics is a part of the centralized vs. distributed memory trade study.

2.4.6 Wait Statistics

This report is a standard GPSS report on using the Queue/depart entity of GPSS. Queues 1-60 are the wait statistics for module starts (refer full word parameter 1) and 61 to 120 are the wait statistics for module outputs (refer fullword parameter 2) for modules 1-60 respectively.

When using the maximum queue length parameter feature (refer to byte parameter 10), an individual report is generated when the maximum queue length specified is exceeded. The standard reports are then generated consisting of statistics up to the abnormal termination.

2.5 User Interface

The following sections are instructions on (1) how to access the 370 system to run the GPSS model and (2) how to activate and/or redefine or add modules in the model.

2.5.1 System Access

This description is based on an IBM 370 168 main frame with OS/VS2 TSO System and GPSS Version V as well as the delivered simulation program (written in GPSS) in its files. The latter may be resident in only the user (TSO) file. The file names and procedures referenced in the following description are those cataloged at Computing Service, Rockwell International, Western Region.

2.5.1.1 Cataloged procedure and JCL requirements

The GPSS Version V program modules reside in a private, cataloged data set named ASYS.GPSS at Computing Services. This data set must be referenced in a JOBLIB or STEPLIB statement to access the GPSS programs. The program module named DAGOIV should be entered on the EXEC statement. For other JCL requirements, refer to the IBM General Purpose Simulation System V-OS (GPSS V-OS) Operations Manual.

To minimize the number of control cards required to use GPSS, the following cataloged procedure is stored in the ASYS.ASPROCS (AGPSS) library. This cataloged procedure allows execution of a GPSS job that requires neither the update nor the jobtape feature.

```
//AGPSS        PRØC
//G EXEC       PGM=DAGØ1V,PARM=B,REGIØN=12ØK
//STEPLIB          DD DSN=ASYS.GPSS,DISP=SHR
//DØUTPUT          DD SYSØUT=A
//DINTERØ          DD UNIT=SYSDA,SPACE=(TRK,(1Ø,1Ø))
//DSYMTAB          DD UNIT=SYSDA,SPACE=(TRK,(1Ø,1Ø))
//DREPTGEN         DD UNIT=SYSDA,SPACE=(TRK,(1Ø,1Ø))
//DINTWØRK         DD UNIT=(SYSDA,SEP=(DINTERØ)),SPACE=(TRK,(1Ø,1Ø))
//DINPUT1          DD DDNAME=SYSIN
//   PEND
```

To invoke this cataloged procedure, the programmer should set up his deck as follows:

```
//  EXEC       AGPSS
//SYSIN     DD    *
     (GPSS Model Statements)
```

The JCL cards used to invoke the GPSS model at Collins Newport are as follows:

```
//$NLA1504 JOB 'WA/809     B5Ø1675771*52Ø873138                XXX0001
//    TIME=(4,Ø1),REGION=24ØK,MSGLEVEL=(2,1)
//*MAIN ORG=RM1Ø8,ACHOLD=NO
//*FORMAT AC,DDNAME=DOUTPUT,PRINT=YES
/*FORMAT PR,DDNAME=SYSMSG,FORMS=1412/1PT
/*FORMAT PR,DDNAME=DOUTPUT,FORMS=1412/1PT
//      EXEC AGPSS
//SYSIN  DD  *
 REALLOCATE COM,1ØØØØØ
 UNLIST ABS
 SIMULATE 4
```

This procedure was filed in the user's library. The JCL cards should be modified as required for the system used, especially the JOB card and the MAIN card.

Note that three GPSS control statements are included. The remaining GPSS model program statements are sectioned into three files:

a. File "FMODEL" contains the model statements.

b. FILE "XXXXX": This file is the parameter set file and is different according to the configuration being modeled. It consists of only "INITIAL" statements.

c. File "FREPORT" contains the report formatting statements of the model and the "END" statement.

The job is submitted via TSO* as follows:

SUBMIT (TJCL,FMODEL,XXXXX,FREPORT)

2.5.2 Module activiation

The parameter file, referred to as file "XXXXX" above, is constructed according to the system configuration to be modeled. A base parameter file consisting of 60 sets of INITIAL statements (one for each parameter identified in section 2.3) may be duplicated and used to construct the modules to represent the system configuration to be modeled. Currently there are two base parameter files one for signal processing and the second for switching. Each of these base files contain a set of standard (predefined) modules which may be activated by initializing the appropriate parameters. Thus the system designer may use predefined modules, may elect to modify them to suit similar requirements, or may define a new set of modules.

*Refer to IBM doc. GC28-0646-1 OS/VS2 TSO Command Language Ref.

2.5.2.1 Base file duplication

The two base files are named as follows:

a. Signal Processing: XXXXX = SPMODULE

b. Switching: XXXXX = SWMODULE

The duplication command via TSO is (as an example)

COPY SPMODULE.CNTL NEWNAME.CNTL

NEWNAME is the name to be assigned to the file that is to be created (
to eight alphameric characters are allows). This newly created file i
the parameter file used to construct and activate the modules of the
system configuration to be modeled.

2.5.2.2 Initializing the parameters

As mentioned earlier, the base parameter file has INITIAL statements f
each of the 21 parameters defined in section 2.3; a set for each of th
allowable modules. A non-initialized set in the parameter base file i
shown in Table 2.5-1.

The INITIAL statements are GPSS statements which initialize matrices.
47 identifies the row number which is (for this model) equivalent to t
module number. The symbol following the row number (e.g., ACT, PRI, C
identifies the column number which represents a parameter type as show
Table 2.5-2.

In Table 2.5-1, the number to the right of the symbol (following the
comma) is the value of the parameter to be initialized to.

Table 2.5-1 Example of a Non-Initiated Parameter Set

```
*                                                                  00011370
*     MODULE 47 PARAMETERS                                         00011380
*                                                                  00011390
*                                                                  00011400
      INITIAL     MR$BPARM(47,ACT),0                               00011410
      INITIAL     MR$BPARM(47,PRI),0                               00011420
      INITIAL     MR$BPARM(47,CONT),0                              00011430
      INITIAL     MR$BPARM(47,MEM),0                               00011440
*1                                                                 00011441
      INITIAL     MR$BPARM(47,OMDNR),0                             00011442
      INITIAL     MR$BPARM(47,QMAX),2                              00011443
      INITIAL     MR$BPARM(47,FMIOT),0                             00011450
      INITIAL     MR$BPARM(47,FMMPT),0                             00011460
      INITIAL     MR$BPARM(47,FMINW),0                             00011470
      INITIAL     MR$BPARM(47,INTRL),0                             00011480
      INITIAL     MH$HPARM(47,WDSIN),0                             00011490
      INITIAL     MH$HPARM(47,RTIME),0                             00011500
      INITIAL     MH$HPARM(47,WDOUT),0                             00011510
      INITIAL     MH$HPARM(47,WTIME),0                             00011520
      INITIAL     MH$HPARM(47,CORWD),0                             00011530
      INITIAL     MH$HPARM(47,COWWD),0                             00011540
      INITIAL     MX$FPARM(47,IINTR),0                             00011550
      INITIAL     MX$FPARM(47,OINTR),0                             00011560
      INITIAL     MX$FPARM(47,SOFST),0                             00011570
      INITIAL     MX$FPARM(47,OOFST),0                             00011580
      INITIAL     MX$FPARM(47,MOTIM),0                             00011590
```

Table 2.5-2 Parameter Symbols

```
*
*     BYTE COLUMN SYMBOLS
*
 ACT    SYN          1                    ACTIVATE COLUMN
 PRI    SYN          2                    PRIORITY
 CONT   SYN          3                   CONTROL TYPE COLUMN
 MEM    SYN          4                    MEMORY TYPE COLUMN
 FMIOT  SYN          5                FUNCTION MODIFIER FOR START CYCLE AND
 FMMPT  SYN          6                 FUNCTION MODIFIER FOR MODULE PROCESS
 FMINW  SYN          7                  FUNCTION MODIFIER FOR I/O WORDS
 INTRL  SYN          8                  INTERLOCK MODULE NUMBER
 OMDNR  SYN          9                  MODULE NR. * OUTPUT TO
 QMAX   SYN          10                  MAXMUM QUEUE LENGTH
*
*    HALFWORD COLUMN SYMBOLS
*
 WDSIN  SYN          1                    NR WORDS INPUT
 RTIME  SYN          2                    READ TIME
 WDOUT  SYN          3                    NR WORDS OUTPUT
 WTIME  SYN          4                    WRITE TIME
 CORWD  SYN          5                    COMMON READ COUNT
 COWWD  SYN          6                    COMMON WRITE COUNT
*
*    FULLWORD COLUMN SYMBOLS
*
 IINTR  SYN          1                    INPUT INTERUPT PERIOD
 OINTR  SYN          2                    OUTPUT INTERUPT PERIOD
 MOTIM  SYN          3                    MODULE PROCESS TIME
 SOFST  SYN          4                     START CYCLE TIME OFFSET
 OOFST  SYN          5                OUTPUT TIME OFFSET
*
```

The numbers on the right side of the INITIAL statements are sequence numbers assigned to the statement. This number identifies the statement which is to be edited (parameter value assignment). The TSO instruction (in the EDIT mode*) is as follows:

C SEQ NR /CURRENT DATA/NEW DATA

for example:

C 11410 /0/1 This command changes the 0 to a 1, which thus activates module 47. The entire module parameter set is initialized in this manner. It is recommended that a comment be included after the module number to identify that module. Comments are identified by an asterisk in the first column. For example:

C 11390 /*/* MATCH FILTER 1600 CPS

After all module parameter definitions have been made, the new file is saved (using the SAVE command) and the simulation request may be made as described in section 2.5.1.

In using a predefined module, the same method of initialization as described above is used. The difference being that certain parameters have already been defined. Any of the parameters may be set to the value as defined in the system configuration. As a minimum, the active (ACT) parameter must be set.

An example of a predefined module is shown in Table 2.5-3.

*Refer to IBM doc. GC28-0646-1 OS/VS2 TSO Command Language Ref.

Table 2.5-3 Example of a Pre-defined Module

```
*                                                                    00004030
*       MODULE 15  PARAMETERS                                        00004040
*        CORRELATOR SM100 .5 CO WINDOW 8                             00004050
*                                                                    00004060
*                                                                    00004070
         INITIAL     MB$BPARM(15.ACT).0                             00004080
         INITIAL     MB$BPARM(15.PRI).122                           00004090
         INITIAL     MB$BPARM(15.CONT).1                            00004100
         INITIAL     MB$BPARM(15.MEM).0                             00004110
*1                                                                   00004120
         INITIAL     MB$BPARM(15.OMDNR).17                          00004130
         INITIAL     MB$BPARM(15.QMAX).5                            00004140
         INITIAL     MB$BPARM(15.FMIOT).0                           00004150
         INITIAL     MB$BPARM(15.FMMPT).0                           00004160
         INITIAL     MB$BPARM(15.FMINW).0                           00004170
         INITIAL     MB$BPARM(15.INTRL).0                           00004180
         INITIAL     MH$HPARM(15.WDSIN).0                           00004190
         INITIAL     MH$HPARM(15.RTIME).5                           00004200
         INITIAL     MH$HPARM(15.WDOUT).16                          00004210
         INITIAL     MH$HPARM(15.WTIME).5                           00004220
         INITIAL     MH$HPARM(15.CORWD).0                           00004230
         INITIAL     MH$HPARM(15.COWWD).1                           00004240
         INITIAL     MX$FPARM(15.IINTR).200000                      00004250
         INITIAL     MX$FPARM(15.OINTR).10000000                    00004260
         INITIAL     MX$FPARM(15.SOFST).0                           00004270
         INITIAL     MX$FPARM(15.OOFST).0                           00004280
         INITIAL     MX$FPARM(15.MOTIM).0                           00004290
*                                                                    00004300
```

As a minimum, statement #4080 (ACTIVE parameter) would be set to a 1. If this module is defined to call another module, then statement #4130 should be set to the module number to be called. Any of the preset parameters may be changed to any value at the discretion of the system designer.

2.5.3 System Parameter Initialization

The system parameters identified in section 2.3.4 are initialized in the same manner as described for the module parameters described in section 2.5.2. These system parameters are contained in the same files as the module parameter files. They are currently preset to the type of system - Signal Processing or Switching. For example, the Signal Processing system parameters are set as follows:

```
INITIAL     XF$TPERD,2000000          100 MS PERIOD                    00000030
INITIAL     XH$TIME,7           700 MS RUN                             00000040
INITIAL     XH$CCWRT,5          SET WRITE TIME TO 250 NS               00000050
INITIAL     XH$CMEMT,20         SET COMMON MEMORY TIME                 00000060
```

2.6 Simulation Model Evolution

The simulation model was initially written to model the "strawman" signal processing system in the parallel bus architecture. The results showed that the bus was less than 1 percent utilized. This realization led to redesign of the strawman signal processing system including an increase in processing load. To accommodate the rate change, the design of a general simulation model to accommodate various parameters was commenced. These parameters initially included the following:

1. Start Cycle Time
2. Output Interval
3. Priority
4. Number of Data Words Input

5. Input I/O Time
6. Number of Data Words Output
7. Output I/O Time.

The above parameters represented all the necessary parameters of the strawman signal processing system. However with only these parameters the model and simulation matched only a system configuration as defined by the strawman system, i.e. centralized control and distributed memory.

To provide greater flexibility, four more parameters were added. These were:

1. Type of Control; Centralized or Decentralized.
2. Type of Memory Use; Distributed or Common
3. Number of Internal Reads and Writes (for use by the module for storage, status, flags, etc.).
4. Module activation

The latter parameter was added for convenience when using a common library of preset modules. These library modules may be activated in the model run by initializing only one parameter. At the same time, the module count was expanded to 60 modules available to the general purpose simulation program. In actual reality, the number of active modules would probably be much less.

Next, it was noted that a measurement of module execution time relative to start cycle times would be a useful tool for software/hardware tradeoffs; and that in certain conditions, modules may be required to start or output at a time interval offset from real time, i.e., propagation delay. Three more parameters were thus added:

1. Module processing time
2. Start time offset
3. Output time offset

The model was further enhanced to accommodate switching systems. It was recognized that switching system modules do not operate on a fixed time but rather on a variable time based on message traffic input and routing. Three parameters were added to provide a time distribution modifier and data distribution modifier:

1. Input/Output Time distribution modifier
2. Module time distribution modifier
3. I/O distribution modifier

Seven functional distributions are currently available (refer to table 2.3-1).

Finally, to expand the module interconnection configuration, the module call feature and interlock feature were added. At the same time the queue max limit was added to halt the simulation if the module was falling behind by a predetermined amount. The last three parameters added are:

1. Module interlock
2. Module call
3. Max queue length

The inclusion of these parameters resulted in the requirement for additional statistical reports. Inclusion of all these optional features does not imply that the generalized module program fits all the needs to

design and model all system configurations. As time evolves, more ideas and configuration connectivity are brought out along with more statistics and reports.

2.7 Design Evolution

Section 5 of this report presents a design methodology for building a modular system. Several nodes in the design flow call for design verification using a simulation model. This section is intended to further the readers understanding of how to apply the system simulation model described above to support design verification, at the equipment partitioning point in the methodology flow. This model was built to simulate congestion of the communication path between modules. As a fall-out of this model, some simulation of control flow and module processing congestion is possible. Functions which do not yet exist in the model yet are desirable and possible in the architecture are primarily concerned with control flow. Due to the diversity of techniques available all cases have not been covered. For example, if one is using distributed interrupts feeding a centralized control, statistics on the control queue can not be measured. Also modeling of task suspension and priority within this control queue are not possible with this model. Earlier paragraphs in this section have described the simulator variables and how to input or change the parameters. There are four catagories of parameters available to the designer:

1. Information arrival rates
2. Processing times
3. Module to module data flow
4. Process control

Data control, which is the control of the movement of data over the bus from one module to the next, is built in the simulator.

The method for specifying information arrival is relatively straight forward in the model. Data is specified in terms of arrival rate at the module interface (with a distribution modifier about a mean if required) and in terms of module bus access or delivery time. The examples in Appendix L associated with signal processing used the fixed arrival rate with several runs illustrating the effect on bus utilization as a function of information rate. The switching example specifies data arrival with a distribution modifier about a mean. This distribution is modeled using the variables associated with the start input and start output control functions.

The method for specifying processing time is also straight forward. This time may also be specified as a mean with a distribution. The processing time specification can be used to simulate either the time to complete a hardware computation cycle within a module or the time to complete a software execution. In the hardware case this is often not important due to double buffering and the relatively fast execution times possible. It should be noted that information for specifying a new module can be gained from a simulation run by setting the processing time to zero. The time then accumulated in the "Time Difference" output column specifies the time available for module computation if no buffering is provided in the module. For the case when a processing time has been specified, a value less than zero in the "Time Difference" column indicates a problem in the system.

The parameters which direct data flow are used to simulate the case when I/O times are a function of bus time plus the receiving modules input time. Since the bus in this architecture is asynchronous and interlocked, the time to transfer a word of data is calculated by summing the time to achieve bus mastership + the bus transfer time (fixed) + the time required to acknowledge the transfer. The last function is determined by either the Masters output time or the Slave input write time, whichever is greater. Statistics are listed which accumulated this I/O times on a given module. When many outputs are directed to one input, the sum of all transfer times are accumulated on the receiving modules input and listed in the column "Calling Mod I/O Totals".

The process control function is concerned with the direction of execution and I/O of algorithm modules within the equipment. Appendix A.1 specifies a control technique which is highly desireable in a signal processing system. This control method assumes fixed rate execution times typical for signal processing for each task. By using this technique, the control is simple and flexible. Adding or deleting a module in this architecture is achieved by changing the task table in the central control module. It was found during the modeling of switching systems that this control method does not match the statistical nature of the switch. To use a centralized control scheme in a switch requires a polling scheme to be incorporated.

Figure 1.4 is a listing of the various control methods by general catagory found in systems. Many systems mix control structures between catagories which complicated the modeling process of the control flow.

It was found that the simulation model gathered many meaningful statistics for the signal processing system but is somewhat limited in the types of control which can be handled. The examples shown in Appendix L and which are described in section 2.8 of this report illustrate the use of the model to measure parameters of interest within the equipment.

2.8 Summary of simulation runs

Six different but similar signal processing models were simulated. These models are all derivatives of the strawman signal processing system (Figure 2.8-1) described in Appendix E. The results of each are summarized in Table 2.8-1. In the switching model (Figures 2.8-2 and 2.8-3), two simulations were made; at peak hour load and peak second load. Their loads were based on a typical switching system as described in Appendix F. The results of each are summarized in Table 2.8-2.

A walk through of one of the signal processing models (RUN 1) will be discussed. The differences of the other signal processing models from the model discussed are summarized in Table 2.8-1.

Before a simulation of a system can be made, the system designer must design the system and prepare a model to represent that system. In this example, signal processing modules from a signal processing module library are available to the system designer. Table 2.8-3 lists the types of modules available in the Signal Processing library. Also refer to signal processing module file listing in Appendix L. The task is to build a signal processing link that will detect and error correct an encrypted,

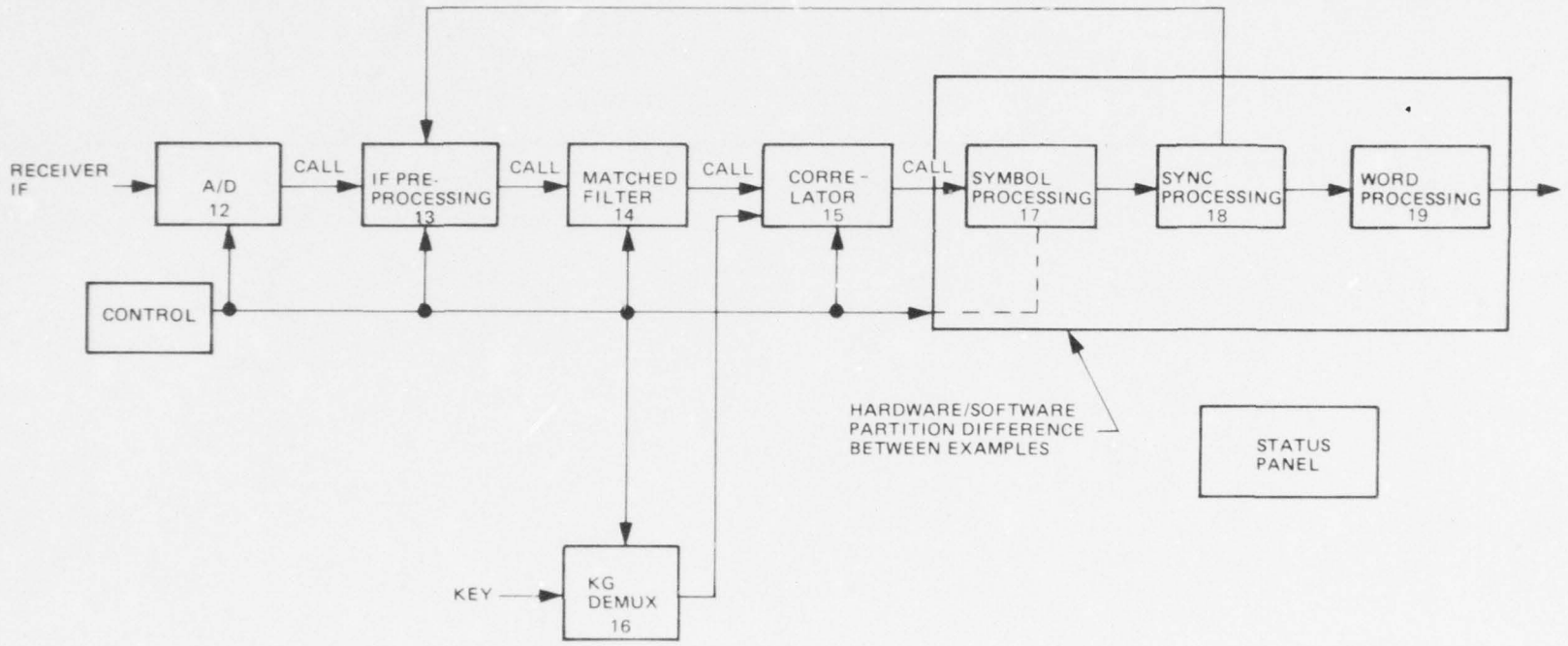


Figure 2.8-1. Signal Processing Model.

Table 2.8-1. Summary of Signal Processing Simulations

Run No.	Difference From Run #1	% Bus Utilization	% Bus Utilization Due to:		Remarks
			Central Control	Status Panel	
1	See example.	3.58	1.06	1.82	Bus use for control vs. data is 1:2.5
2	Transmission modulation Rate = 800 Baud	4.32	1.24	2.00	Bus Utilization increased by less than 1 percent with the increase of the baud rate by a factor of eight.
3	Transmission Modulation Rate = 1600 Baud	5.16	1.44	2.20	
4	Transmission Modulation Rate = 1600 Baud. Sync and Word Modules Interlocked	5.16	1.44	2.20	No affects in moving a slow module from software to hardware.
5	Transmission Modulation Rate = 1600 Baud Correlation Coding = M-ary Sync and Word Modules are not included (insignificant bus time usage) Correlation Module I/O = 250 ns	10.70	1.44	2.20	Correlation coding change from CO to M-ary resulted in twice the bus use. Other results showed data loss.
6	Same as 5 except Correlation Module I/O = 1 us	18.85	1.44	2.20	Slowing the access rate to accommodate double buffering solves loss of data but increases bus time.

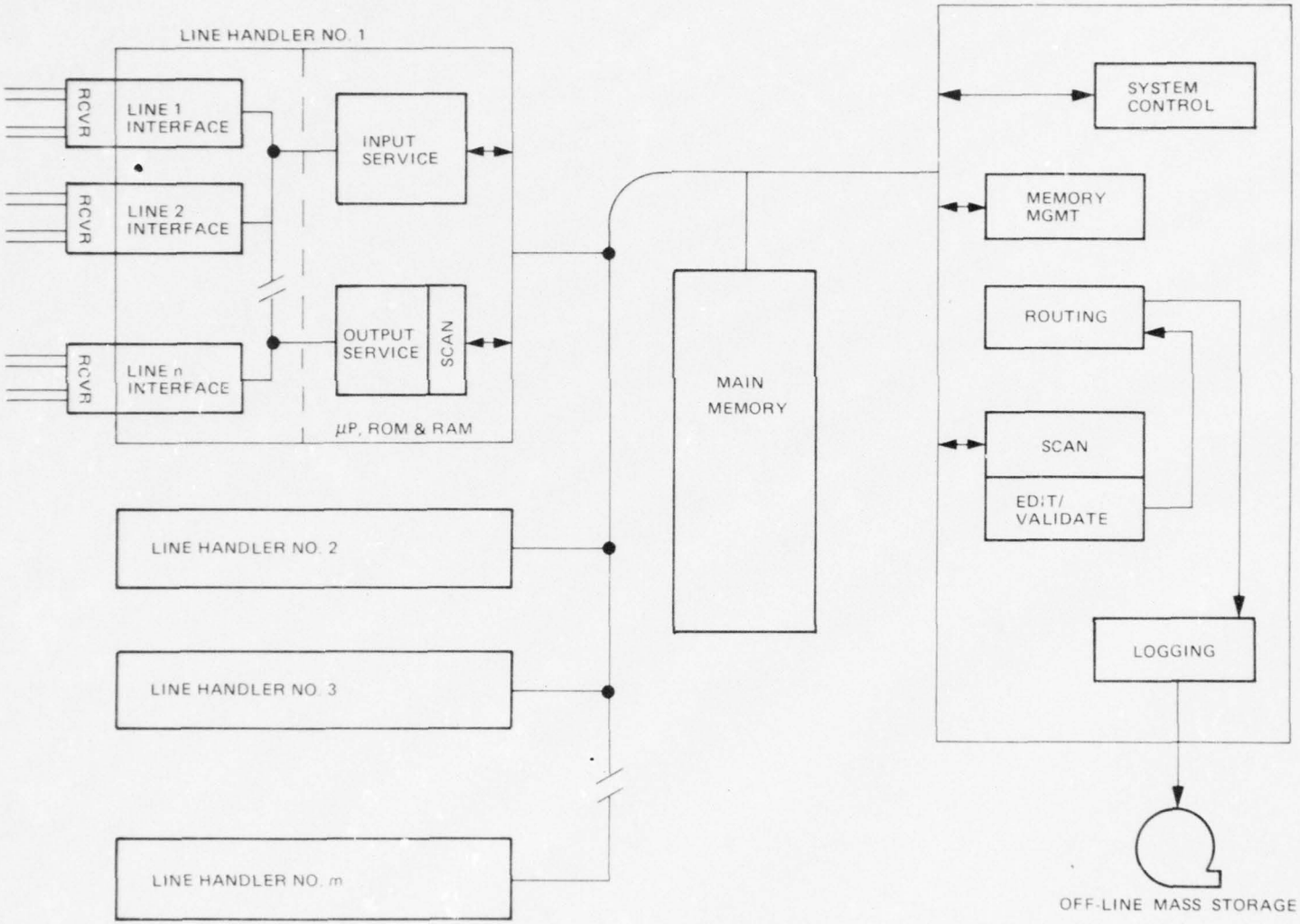


Figure 2.8-2. "Strawman" Message Center

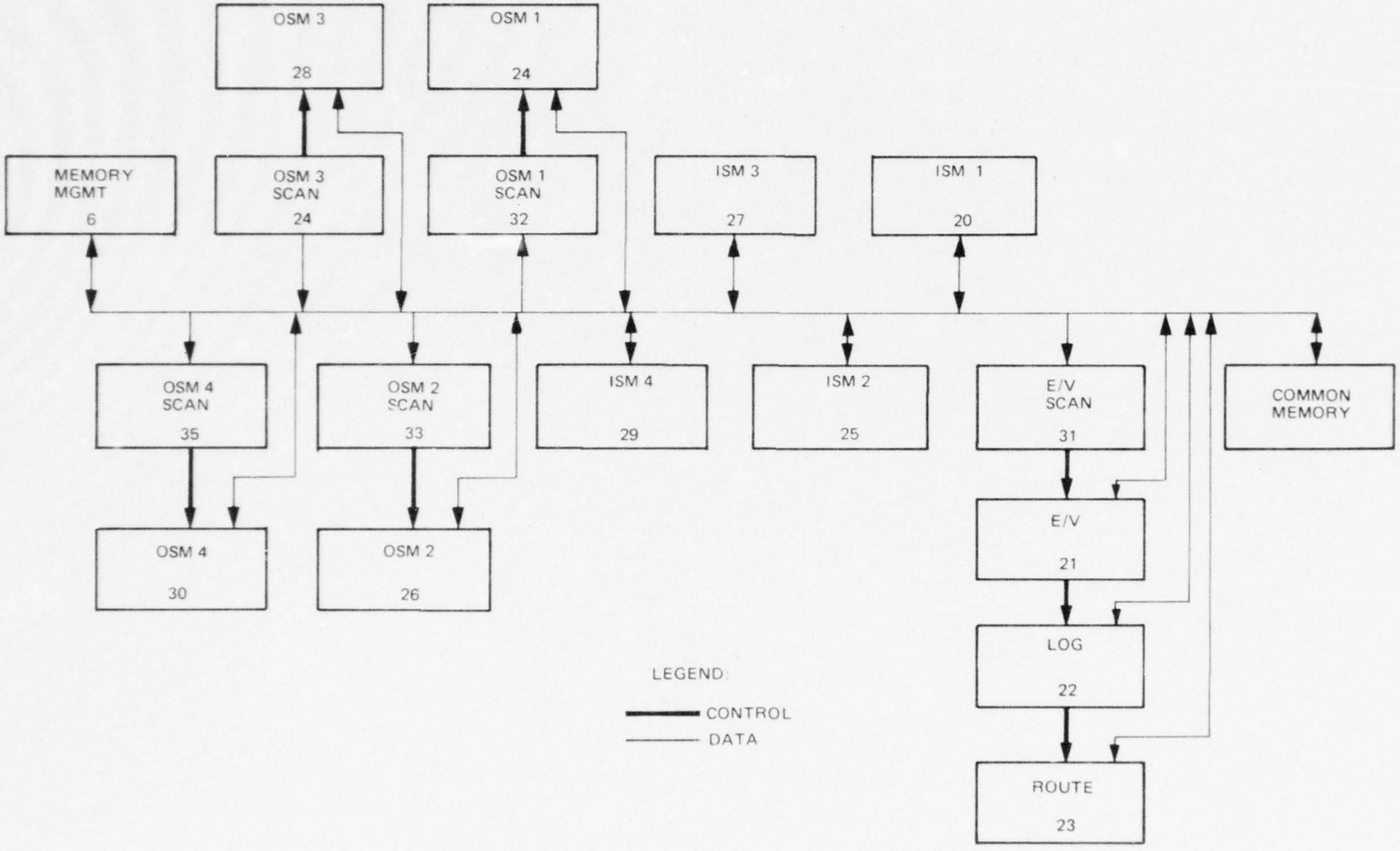


Figure 2.8-3. Switching System Model: Decentralized Control, Common Memory

20177-18

Table 2.8-2 Summary of Switching Simulations

Simulation Model	Percent of Bus Utilization			Percent of CPU* Utilization
	Total	Polling	Data	
Busy Hour	7.89	6.27	1.61	24.6
Busy Second	13.02	6.36	6.66	29.5

Busy Second Data = 4.5 X Busy Hour Data

*Percent of CPU Utilization = ($\sum$CPU I/O Times + Execution Times)/ Total Time)

CPU Modules - Memory Management, Edit/Validation Scanner, Edit/ Validation, Logging, Routing.

Note: The times listed in matrix "LOCKS" are the total of all the modules that are listed as being interlocked to that module. The total for the CPU (except Memory Management) is totaled under Edit/Validation Scanner.

Figure 2.8-3. Signal Processing Modules for Systems Simulation.

MODULE TYPE	FUNCTION
IF PRE-PROCESSING MODULE	SAMPLES DATA AT A FIXED RATE, FILTERS AND OUTPUTS 4 SAMPLES PER BAUD RATE. THERE ARE THREE MODULES DEFINED FOR BAUD RATES OF 100, 800, AND 1600.
MATCHED FILTER	RECEIVES 4 SAMPLES PER BAUD INTERVAL AND OUTPUTS TWO CHIPS AT THE BAUD RATE. THERE ARE THREE MODULES DEFINED FOR BAUD RATES OF 100, 800, AND 1600.
CORRELATOR	COMBINES CHIPS WITH KEY AND SUMS OVER THE INTEGRATION PERIOD. THERE ARE FIVE MODULES DEFINED WITH VARIABLE INTEGRATION PERIODS, BAUD RATES, CODING TYPE AND WINDOW SIZE.
KG DEMULTIPLEX	PROVIDES SUMBOL KEY AT THE BAUD RATE. THERE ARE THREE MODULES DEFINED WITH VARIABLE BAUD RATES AND CODING TYPE.
SYMBOL PROCESSING	RECEIVES DATA FROM THE CORRELATOR AND OUTPUTS AT THE SYMBOL RATE. THERE ARE FIVE MODULES DEFINED WITH VARIABLE INPUT RATE, AND WINDOW SIZE.
SYNC PROCESSING	RECEIVES DATA FROM SYMBOL PROCESSING AND PROVIDES OUTPUT AT THE END OF THE SYNC INTEGRATION PERIOD. THERE ARE THREE MODULES DEFINED, ONE FOR EACH OF THE BAUD RATES 100, 800, AND 1600.
WORD PROCESSING	RECEIVES DATA FROM SYMBOL PROCESSING, FORMS CHARACTERS AND OUTPUTS AT THE CHARACTER RATE. THERE ARE THREE DEFINED WITH VARIABLE PROCESSING RATES.

bandspread, antipodal, phase-continuous FSK signal received by a radio receiver. The specification of the communication mode is as follows:

data transmission modulation rate = 100 baud

correlation coding = CO

data rate = .5 sec

synchronization window = 8

(These parameters provide key word references for a library search in this example. The reader need not relate them to specific signal processing implications).

Based on the communication requirement, the designer selects from the library those modules that meet the specification. The designer then determines how to interconnect (communicate between) these modules so as to provide a solution to the system requirement. The result is a model from which a simulation may be made. In this example, the model determined is shown in Figure 2.8-1.

The modules drawn from the library were modules 12, 13, 14, 15, 16, 17, 18, and 19. The next step taken was to copy the main library file (see Section 2.5.2.1). Each of these modules are initialized to the active state (see Section 2.5.2.2 on how to initialize parameters). Modules 12 through 17 control parameters are initialized to 1 (centralized control). Since modules 17 through 19 are interlocked (i.e., software modules in a CPU module), modules 18 control is received from module 17, and module's 19 control from module 18. The control parameter for module 18 and 19 are thus set to 2; and correspondingly, the interlock parameter for module 18

is set to 17, and the interlock parameter for module 19 is set to 18. Any words sent between these interlock modules are set to 0 (refer input words and output words parameter).

As identified by the lines marked call (see Figures 2.8.1), the module output call parameters (byte parameter 9) are initialized to the module as shown by the direction of the call lines. Since centralized control is specified, the specification of the output call parameter does not cause control to be passed, but provides statistics to be accumulated on I/O transfer to the called module.

Priorities are set according to the position of the module on the bus. The closer the module is to control, the higher the priority. As seen from Figure 2.8-1, the model shows module 12 as the highest priority and module 19 as the lowest. The priority parameters are initialized accordingly.

To simulate the status panel, the command memory function is used. Therefore, the internal write parameter (halfword parameter 6) of each module is set to a one.

Once these parameters have been set and the file saved, the simulation run may be submitted (see Section 2.5.1.1).

An example of parameter settings are shown in Table 2.8-4.

The results of the simulation (refer to performance statistics of run #1 in Appendix L) of this model, showed that the total bus utilization was 3.58 percent, of which 1.06 percent was used by the centralized control module, and 1.82 percent was used by all the modules in updating the

Table 2.8-4 Example of Module 18 Parameter Set

```
*                                                                    00004250
*       MODULE 18  PARAMETERS                                        00004260
*        SYNC PROCESSING SM .5 CO                                    00004270
*                                                                    00004280
*                                                                    00004290
         INITIAL     MB$BPARM(18.ACT).1                             00004300
         INITIAL     MB$BPARM(18.PRI).120                           00004310
         INITIAL     MB$BPARM(18.CONT).2                            00004320
         INITIAL     MB$BPARM(18.MEM).0                             00004330
*1                                                                   00004331
         INITIAL     MB$BPARM(18.OMDNR).13                          00004332
         INITIAL     MB$BPARM(18.QMAX).2                            00004333
         INITIAL     MB$BPARM(18.FMIOT).0                           00004340
         INITIAL     MB$BPARM(18.FMMPT).0                           00004350
         INITIAL     MB$BPARM(18.FMINW).0                           00004360
         INITIAL     MB$BPARM(18.INTRL).17                          00004370
         INITIAL     MH$HPARM(18.WDSIN).0                           00004380
         INITIAL     MH$HPARM(18.RTIME).0                           00004390
         INITIAL     MH$HPARM(18.WDOUT).1                           00004400
         INITIAL     MH$HPARM(18.WTIME).20                          00004410
         INITIAL     MH$HPARM(18.CORWD).0                           00004420
         INITIAL     MH$HPARM(18.COWWD).1                           00004430
         INITIAL     MX$FPARM(18.IINTR).10000000                    00004440
         INITIAL     MX$FPARM(18.OINTR).10000000                    00004450
         INITIAL     MX$FPARM(18.SOFST).0                           00004460
         INITIAL     MX$FPARM(18.OOFST).0                           00004470
         INITIAL     MX$FPARM(18.MOTIM).2000                        00004480
*                                                                    00004490
```

status panel. The symbol processing module, the sync module, and the word processing module (modules 17, 18, and 19 respectively) are accessed so infrequently, that their times are insignificant compared to the other modules. The ratio of bus time used for control versus data is 1:2.5. This suggests that data transfer is very tightly linked to control. In this example the total bus time use is low, but if bus use became a problem, grouping modules to common interrupts would improve the ratio.

Runs two and three reflect the additional bus bandwidth needed when the baud rate is increased. It can be seen that increasing the baud rate by a factor of eight only increased the use by one percent. This change is small because of the small percentage of bandwidth used by the modules that changed.

Run four illustrates the effect of moving a selected module from software to hardware. The choice of module to be moved would be on bus use statistics or processing requirements. As can be seen in the table, no apparent change to bus use can be seen by moving this function from software to hardware, indicating that bus use is not a tradeoff factor in this proposed change.

The last two runs, five and six, illustrated an example which points to a need for buffering on a module. Run five shows a -5230 time units in the "Time Difference" column of the run. This difference occurs with a fast access correlation module. Run six slows the access to the correlator and assumes double memory buffering. This solves the data loss problem but increased the bus utilization from 10.7 percent to 18.8 percent. A check of the "Time Difference" column indicates that I/O is complete between output start cycles.

Table 2.8-5 lists the modules (numbers) selected from the Signal Processing Library for each of the six models. The simulation report of each (including the listing of the signal processing module library) may be found in Appendix L.

In the case of the switching system simulation, listed in Appendix L, it can be seen that the utilization of the bus by the distributed control is weakly dependent on the amount of data being transferred through the switch. Referring to Table 2.8-2, it can be seen that for the increase from busy hour to busy second, (busy second equals 4.5 times the total characters received during the busy hour) the bus utilization increases from 7.9 percent to 13 percent. This is almost all as a result of increased data words, with internal data transfers increasing 0.1 percent. While it was originally thought that a dual port memory was necessary, to eliminate contention between the CPU and the Line Handlers requesting memory access; it is seen from the simulation results that there is no foreseeable contention problem.

2.9 Deliverables

2.9.1 Listing of Program

This listing consists of GPSS statements of the general simulation program (name = FMODEL) which was developed to represent the operation of the parallel bus architecture and to operate according to a set of parameters for each module activated in the model. A second listing consists of GPSS format statements on the construction of the simulation run report.

2.9.2 Flowcharts

The flowcharts are the GPSS block and data path diagram of the general simulation program FMODEL.

Table 2.8-5. Index of Signal Processing Models

Run Number	Identification	Modules Used from Library ($NLA1600)
1	$NLA1601	12 through 19
2	$NLA1602	36,37,38,40,44,45,46,47
3	$NLA1603	1 through 8
4	$NLA1604	1 through 8
5	$NLA1605	24 through 29
6	$NLA1606	24 through 29

2.9.3 Listing of Module Library

Two listings are provided. These listings are the base file used to specify modules for (1) signal processing modules, and (2) switching modules.

2.9.4 Files (Magnetic Type) of Program and Module Library

Source files of the general simulation program, reports format, and module libraries will be provided on magnetic tape.

3.0 TASK III - LANGUAGE STUDY

3.1 Introduction

This task considers the mutual interactions between the design characteristics of High Order Languages (HOL's) and requirements for their efficient and effective application to the area of modular design of digital signal processing systems and message switching systems.

Section 3.2 is an overview of various technical and procedural considerations involved in setting the design objectives for a suitable HOL and its support software. Several difficulties are identified and a long-range approach toward their solution is discussed.

Section 3.3 presents the current technologies and techniques used in implementing modular systems for signal processing and message switching applications. Three common system architectures and their impact on HOL design are discussed, followed by a comparison of three existing HOL's.

Section 3.4 gives conclusions and recommendations.

Section 3.5 suggests areas where further study of this topic may be productive.

3.2 Overview, HOL Design Requirements

HOL design requirements must be defined for the signal processing and message switching applications area. A start can be made by comparing existing HOL's

- to each other,
- to the needs of the application area,
- to other factors which increase its acceptability and usefulness.

3.2.1 General

The first step in comparing the many available and proposed HOL's is to discover and list the features and capabilities of each HOL.

The second step is to compare the lists against each other. However, such comparisons cannot mean much unless there is also a set of criteria which allow the analyst to assign favorable or unfavorable weights to each difference found. The question arises, how does one formulate such criteria?

One way to proceed is simply to make some preliminary assertions about what is initially considered to be important, and hope to refine the assertions into criteria which will be acceptable to those who are interested in the same applications area.

3.2.2 Preliminary Assertions

A. Applications Area

The applications area is that of modular systems for signal processing and message switching.

A1. System architectures containing multiply-connected processors and peripherals must be accommodated.

A2. Processors within the given architecture may be similar to one another or may be very different.

A3. Multi-processing is assumed.

A4. Event-driven multi-tasking operations must be supported with due regard to priorities and contentions.

A5. A wide variety of data types must be processed and distributed from place to place. They include boolean, logical, integer, real, complex and character, together with constructs containing

these types. In addition the controlled transfer of machin
from source to destination must be supported. Bit manipula
and precision control are required.

B. Bridging conventions, hardware/software.

Before the software designer can instruct elements of the system
perform the tasks, he must have a defined interface to access th
tasks.

Consider the time-honored precedent for the assignment of a proc
a specialized hardware module, namely, floating point hardware.
hardware designer provides the instructions which are symbolical
known as FADD, FSUB, FMUL and FDIV for the use of the programmer
(Note that the programmer often does not care whether these inst
tions invoke hardware or trapped subroutines, provided that the
interface and functions are identical.)

Another important part of the above package is FDCH, the floatin
point divide-check, which yields status rather than causing
processing.

Similar interface control instruction examples can be found in t
usual I/O conventions. These instructions

- initiate actions,
- provide status/error information, and
- terminate actions.

In the applications area under consideration, the system designe
supply interfaces analogous to the above. The software designer
vitally interested in the design of such interfaces for:

- ease of use
- generality/similarity to previous cases
- all possible status checks, especially error conditions

The interface can take various forms:

- instructions (implemented in hardware or microcode)
- traps
- protocols.

As far as the software designer is concerned, the preferred metho description is to define the interfaces as if they communicated w software rather than the actual hardware.

Some preliminary assertions about bridging conventions are:

B1. Interfaces to all actions to be directed by software must be available to the programmer.

B2. Invocation of the interfaces must be possible at the symboli level.

B3. Sufficient status information to allow efficient control of flow of processes must be available. Prior decisions must b made regarding whether error indications are simply indicate whether they force actions to occur.

B4. Interface definitions must include timing information.

B5. The bridging process includes the development of an Operatir System which performs most or all of the accesses to the abo interfaces. The Operating System itself should be written i chosen HOL to the greatest possible extent.

The Operating System normally provides software interfaces to Applications software modules, which communicate to it by means such as request packets. Such packets must be supported by the chosen HOL.

C. Compiler, HOL, and support

The choice and implementation of a suitable HOL should be in the context of widely accepted current principles. The study of candidate HOL's is one source of such information; the general literature is another.

C1. The overall intent of the DoD efforts to specify a common HOL for imbedded processors is directly applicable to this applications area. An interpretation is given in section 3.2.3.

C2. Block structuring must be possible. This not only allows for the use of top-down design methods, but provides for the future use of formal program verification techniques.

C3. Compile-time error analysis should be as helpful as possible. The strong type-checking features of several recent HOL compilers is a good example.

C4. Escape mechanisms from the HOL language rules must be formalized, and instances of their use must be highlighted in the program listing.

C5. Generated code for cases within the defined scope of the HOL must be demonstrably efficient enough to discourage the use of escapes. As a minimum, an optional printout of generated code must be available for the programmer's use in validating that efficient code has been produced. The most convenient format is to have the code resulting from HOL statements immediately follow

the statement(s) that generated it, preferably in an intermediate-level symbolic form rather than binary or hex.

C6. The programmer must have control over the form of the generated code. Examples include reentrant code, open and closed subroutines, etc.

C7. Ancillary software such as linking loaders must prevent or at least flag inadvertent escapes from language rules when linking separately compiled modules.

C8. Support software should be designed for efficient use by a sizeable programming team. Examples:

- The ability to include a closely controlled copy of a lengthy data declaration into source text prior to compilation.
- Clearcut software-assisted configuration control
- Optional cross-reference table generation

C9. Execution-time error checking facilities should be addressed in the design of the overall system. Little more can be said here since the details are necessarily too dependent on a particular system architecture and software Operating System.

A partial approach to the problem is to provide the means of running module checks on a host which supports a system emulator.

C10. All printed output should be designed to serve directly as a part of the total program documentation package.

3.2.3 Choice of languages for comparison

A number of candidate HOL's were examined with respect to the foregoing assertions. Large areas of overlap were found that satisfied many of the

preliminary assertions. Comparisons here tended to produce "distinctions without a difference" in the essential power of expression. Some HOL's were dropped simply because they satisfied few assertions or violated others, and were poor prospects for reworking.

What emerged was the gap between required features and a "composite- best" of the available HOL's. It was concluded that without generating a detailed ranking of all candidates, the main attributes of interest could be highlighted by selecting only three HOL's: PL/I, COL and SPL/I. (See Appendix M for a more detailed description of the selection process.)

Comparison of these HOL's can be expected to:

- Demonstrate the direction of needed revisions or additions to existing HOL's and their supporting software
- help refine the preliminary assertions into more generally useful criteria.

An interpretation of assertion C1 is appropriate at this point because of its impact on the way that software should be implemented:

- DoD Directive 5000.29, "The Management of Computer Resources in Major Defense Systems" applies.
- Life-cycle costs of software for imbedded processors shall be minimized. Life cycles are 15 to 20 years.
- Long term reliability and maintainability are important life-cycle cost factors. So is flexibility in the event of future changes required to respond to currently unknown future threats.

- A baseline of well-tested "correct" software modules should and will become part of the DoD inventory, with resultant decreases in software acquisition costs and improvements in reliability. (To paraphrase an observation made by N. Wirth, designer of the Pascal language: the probability that a given piece of software is unreliable is precisely the probability that its incorrect parts are executed.)
- The way to accomplish life-cycle goals is to use an approved HOL together with appropriate documentation.
- Escapes from this HOL, especially ingenious ones, jeopardize the goals.
- All software elements are included; it is not acceptable to claim that certain items such as the Operating System are excluded from life-cycle maintainability and flexibility requirements.

Thus, assertion C1 puts a heavy weight on what is probably the acid test for a HOL, namely that an acceptable Operating System can be written in it.

Each of the three languages has many of the required capabilities. Each requires escapes to machine-oriented code. It is difficult to conceive of a relatively stable HOL which would not require such escapes in this application area.

Therefore the integrity of the escape mechanisms is a major item of concern if assertion C1 holds.

The primary reasons these particular languages were chosen here is:

PL/I - Wide implementation and use.

SPL/I - Implemented, authorized for use. (DoD Instruction 5000.31)

COL - Best treatment of machine-oriented code.

In addition, each of them has a variety of algorithmic and data handling capabilities, as shown in Appendix M.

It should be noted here that a trend toward implementing Operating Systems entirely in special-purpose hardware is likely (Not ROM or microcode). Such standard hardware modules will impact single-processor architectures and change the role of HOL's in this case. It is not clear how multiple-processor Operating Systems will be affected.

3.2.4 Overview of "Bridging"

The bridging procedures deal with S/H (software/hardware) interfaces and with S/S (software-software) interfaces. The steps include:

a. The chosen architecture is analyzed. Processes to be controlled by software are identified and defined.

b. The S/H interfaces are identified and characterized. The binary formats of new instructions or protocols are defined. It is very desirable that a S/H interface should look identical to a S/S interface. In many cases the most cost effective solution is to force this constraint on the hardware design.

c. Hardware/software tradeoffs are studied, with a probable return to step a.

d. The means of implementation of each S/H interface is chosen. The choices normally include micro-code, assembly-language routines or linkages, and compiler options/modifications.

e. Symbolic references to S/H interfaces are devised and documented.

f. Implementation of new HOL compiler options and modifications, if any, can now begin. Example: a built-in function which produces appropriate machine code upon the recognition of some new (HOL) symbolic reference.

g. A design plan for the Operating System is generated. Factors in the design include:

- use of available systems or modules
- distributed or symmetric versus centralized control
- use and control of escapes from the HOL
- definitions of S/S interfaces
- test and acceptance plans.

At this point, work can be progressing on three software fronts: the Operating System, the Applications programs, and the HOL compiler.

3.2.5 The Support Software Proliferation Problem

In government applications, it is often a contractual requirement to deliver the HOL compiler and other support software along with the operational software. In addition, formal documentation of all software is usually required. Reference: DoD Directive 500.29, section V.E.

This brings us to an unpleasant contradiction which is inherent in the preliminary assertions:

- On one hand, we wish to standardize on a HOL and build a well-tested baseline of standard software modules for the use of system designers in this applications area. We hope to minimize the huge direct and indirect costs associated with constant reinventions and/or modifications of equivalent software functions.

- On the other hand, we are talking about compounding the problem by supplying a new version of a HOL compiler for each new complex architecture, or even worse, for each new version of a system.

When we consider this in terms of 20 year life cycles and the number of systems (and their versions) which will be using imbedded processors, we can appreciate the urgency of finding a workable compromise.

What is needed is an overall approach which will localize and limit the modifications for each case. The following points can be made:

(a) The current DoD efforts to standardize instruction sets is an encouraging development. For example, the joint Navy/Army team which studied the choice of a "standard instruction set architecture" selected the PDP-11 set. (This does not mean that each system has to contain a PDP-11; any other CPU capable of supporting the instruction set may be used).

(b) Given a reasonably small number of different instruction sets, the approach taken by the designers of the language COL is very interesting.

Briefly, COL addresses the question of machine oriented code in the following way:

- The compiler generates all object code; no intermediate level assembler code is permitted.
- The reserved words "CODE XXX" and "END" bracket the machine oriented source code statements. The "XXX" modifier specifies the target instruction set.
- The programmer has symbolic access to all instructions, including those for register manipulations, interrupt processing and absolute address assignment.

- Reasonably high level syntax is provided, for example:

  X := A + B ; IF OVERFLO_ THEN GO TO ERROR;

- A macro facility is provided
- And most importantly, the compiler can maintain most language rules (such as type checking) within the CODE....END bracket.

(c) If we extend this idea to a distributed architecture and call for the HOL to support architecture-oriented-code (AOC) statements, we will have succeeded in localizing and limiting the modification area while retaining most of the benefits of using a HOL.

Any escape method, including this one, gives the programmer the power to defeat the intent of the HOL. However, in this case the compiler has an opportunity to issue warnings and, in general, make potential sources of error highly visible in the program listing.

With careful design of the AOC, it may be possible to cover whole classes of architectures with one version of the AOC. This would help contain the HOL compiler proliferation problem.

This long-range approach would require agreements between vendors and customers if it is to be of value. The alternative is to learn to cope with a growing collection of mixtures of compilers, assemblers and other support software over the long term.

3.2.6 Software Development Practices

(a) The HOL/AOC approach described above is only a tool. It can be used to produce good software only if certain programming conventions and disciplines are enforced.

In this applications area, "good" software

- is modular,
- is well structured and efficient,
- has program listings and associated printouts which are readable and easy to understand, and is well documented,
- is potentially portable to another system at a reasonable cost, and
- has a discipline of programmer accountability which is maintained over the life of the system.

In theory, the HOL segments of a software system are inherently quite portable, subject to the usual warnings about differences in bits of precision, etc. However, see (b) below.

The AOC segments are what alarm project leaders about as much as the mixture of Assembly and HOL code. To most project leaders the thought of a large HOL program, liberally sprinkled with AOC segments at the whim of individual programmers, is intolerable. In addition to inviting excessive checkout expense, it makes the portability problem harder.

Solutions lie along the following lines:

- Limit the accountability for AOC segments to one lead programmer.
- Confine the appearance of AOC segments to one region, say, up front.
- If an AOC segment cannot be put in this region, do everything possible to advertise its existence.

Further restrictions may be desirable, such as using the AOC just to produce blocks of machine code which can be copied in-line when the name is invoked. This would produce an effect which is equivalent to built-in compiler functions.

Systems are obviously more portable when the AOC modules can be replaced on a one-for-one basis with no changes to the HOL modules.

Also, as will be discussed in section 3.3, these module replacements could take the forms:

- software, different instruction set
- micro-code
- hardware.

(b) HOL portability has its problems in this application area. The underlying assumption is that the transported HOL software will execute its functions in the same sequence and with the same relative timing of function completions. This assumption may lead to gross inefficiencies. For example, suppose a lengthy computational task "X", which was done by software in the first implementation, is assigned to a hardware module in the new implementation. The unrevised HOL calling software will wait for X's completion before executing the next function. But say the motive for reassigning the task to hardware was precisely to free the HOL for other tasks while task X is in process. In this case we have a redesign effort:

- rework the HOL function reference so as to initiate parallel task X
- make sure the HOL recognizes X's completion and responses correctly

- implement the tasks that execute in parallel with X
- recompile, retest, redocument, etc.

We could argue that task X should have been designed as a parallel software task in the first place to allow for this type of modular conversion. Realistically, however, the extra overhead involved in treating all possible candidate tasks (multiply? divide?) in this way could not be supported.

In addition, task X may not really need to complete quickly, and a slow but inexpensive piece of hardware can be used. In this case, the redesign effort described above would no doubt be mandatory.

(c) Figure 3.2-1 shows typical software paths and the tools used during a software development cycle. The functions Editor through Emulator are usually carried out on a large host machine. The executable software ready for hardware/software integration is usually loaded by some special means from media such as magnetic tape, paper tape, or diskettes.

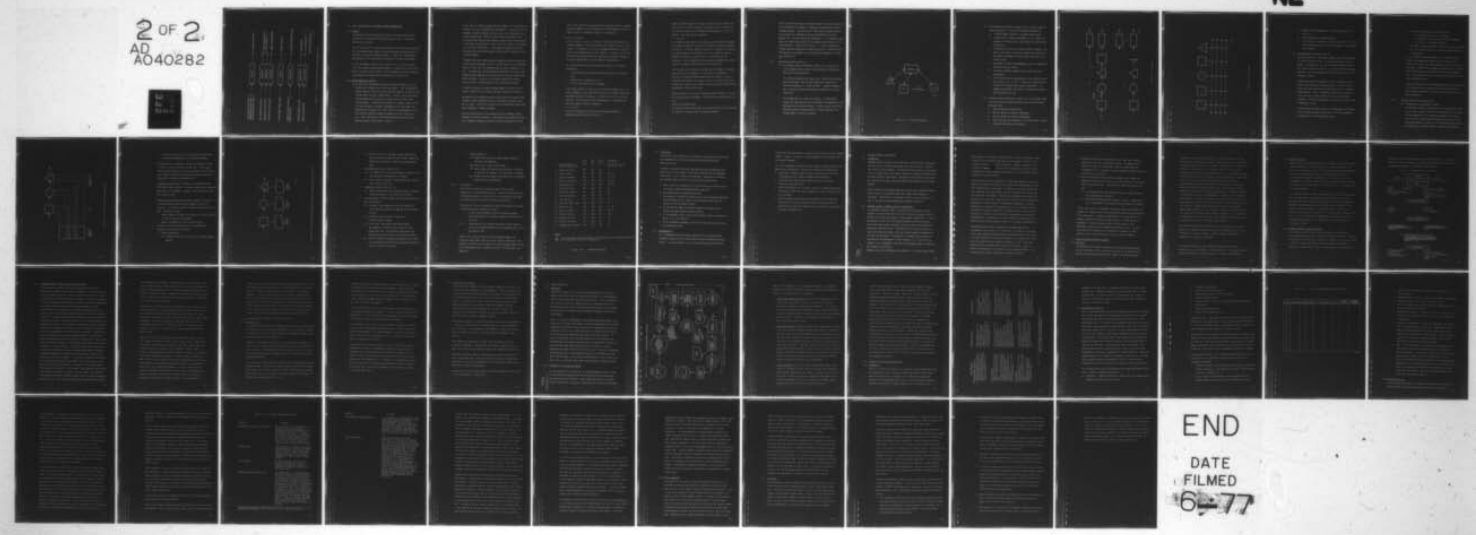
The accessibility, completeness and ease of use of the software tools heavily influences both cost and schedule factors in a development cycle.

Transportability of the tools themselves is a consideration in their design. It is common to write compilers, assemblers, micro-code generators, linkers and emulators in Fortran because of its portability (not its suitability). File handling, editors and libraries are usually provided as standard features of the particular host system.

AD-A040 282 ROCKWELL INTERNATIONAL NEWPORT BEACH CALIF COLLINS G--ETC F/G 9/2
AN ARCHITECTURAL STUDY OF SIGNAL PROCESSING SYSTEMS AND SWITCHE--ETC(U)
MAR 77 DCA100-76-C-0070

UNCLASSIFIED NL

2 OF 2
AD
A040282



END

DATE
FILMED
6-77

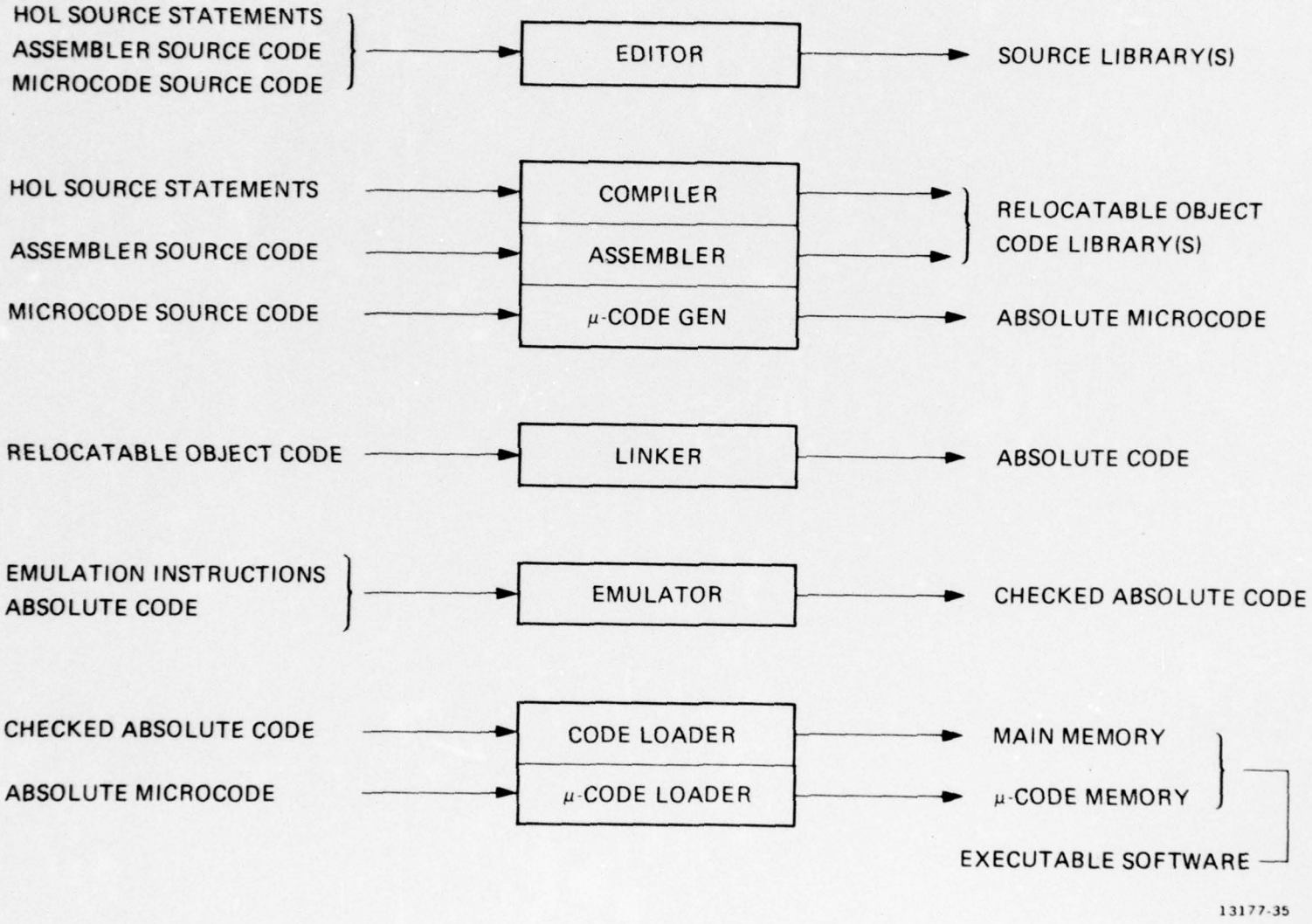


Figure 3.2-1. Typical Software Paths in the Development Cycle.

3.3 Use of Existing HOL's in Modular System Implementation

3.3.1 General

The preceding section discussed the overall characteristics that are desirable in a future HOL for this applications area. A long-range approach was outlined.

In this section we will proceed from the point of view that for the short term and probably much longer, existing techniques and technologies will be used in the design of modular systems. Mixtures of implementation technologies will be required for the sake of efficiency and economy.

In this environment, existing HOL's such as PL/I, SPL/I and COL have a clear role without extensive modifications. A comparison of these languages is provided in the context of the requirements of three representative system architectures and the assertions of section 3.2.2.

3.3.2 System Modularity and HOL's

(a) A significant requirement and result of modular systems is that implementation freedom exists within each module. This implementation freedom enables a module to be significantly modified or replaced with limited impact on other system modules. Further, various implementation techniques may be applied to various modules with the same limited impacts. Therefore, the designer of a modular system is free to select hardware or software implementation for each module and among various techniques (TTL, MOS, fixed wired, programmable, Microcode, Machine Language, Assembly Language, and one or more HOL's, etc.) Thus, the design of each module requires choices of implementation based on that module's function.

In the case of software systems and their modules, it is practical to consider four basic implementation technologies. These are HOL, Macro Language, Assembly Language, and Micro-code. Each technology offers unique documentation, flexibility and efficiency characteristics. It is expected that sophisticated software systems would contain modules of each type of construction. Thus, a 100% HOL system is as unlikely as a 100% micro-code system. A requirement of modularity is that these variously constructed modules can be properly connected to form a complete system.

It appears that each software module construction can be accomplished using a single technology. That is, a given module can be constructed using all micro-code, all assembly code, all macro language, or all HOL. The module may interface modules constructed from another technology but there appears to be little need for multiple technologies within a module. Further, it will greatly simplify the module building effort if technologies are not mixed within a module.

It remains necessary to connect between modules of differing technology. (Recall 3.2.4 (b) above regarding interface definitions.)

An example of this modularity would contain mostly HOL modules which connect to other modules by "built in functions" and routine calls, per (b) below. The built-in function may be constructed using either macro language or assembly language.

Built-in functions may in turn connect to macro language, assembly language or micro-code modules. Subroutines may be constructed from HOL or assembly languages and may also connect to modules of micro-

code. Thus, basically HOL systems may be supported by macro language, assembly language, or micro-code modules without compromising the HOL modules since all connections appear to be procedures.

(b) Built-in Functions

Built-in functions (BIF's) are compiler operations which enable HOL software segments to directly interface non-standard hardware or subroutines. In general, BIF's are not language dependent but are products of the implementation of each compiler. Therefore, any HOL can efficiently perform most functions performed by assembly language by relying on appropriate BIF of the compiler implementation.

It is worth pointing out the hierarchical nature of the software technologies:

- micro-code provides building blocks to be accessed by machine code,
- machine code's produced by a BIF,
- The BIF is invoked by a HOL reference.

A BIF usually appears in the HOL as an ordinary procedure call. Indeed, procedures in one compilation may be BIF of another compilation by changing compiler directives. A familiar use of BIF is the sine function. This function is often implemented as a subroutine call which may appear in the HOL program as:

Y : = SIN (X) ;

As such, the compiler will utilize its standard treatment of subroutines and SIN need not be a BIF.

However, standard subroutine linkages are usually quite general and inefficient so a simpler linkage is desirable for the SIN function. This simpler linkage can be designed and designated as a BIF to the compiler. Now, when the HOL statement:

```
Y  := SIN (X) ;
```

is encountered, the compiler will generate the BIF code which happens to be special linkage to a specially designed (in assembly language) SIN routine. If further speed is desired, it is possible to have the BIF produce the entire SIN routine as in-line software so that no linkage is necessary (in practice this is seldom done for SIN functions). The HOL program remains unchanged throughout these transitions of subroutine design and efficiency trade-offs.

BIF's can also be utilized to interface special hardware or computer instructions (e.g., INTERRUPT). For example, if a hardware instruction is available which performs the SIN function, say using microcode, the compiler can recognize SIN as a BIF and produce that instruction. Thus, the HOL software remains unchanged through further improvement of the SIN efficiency.

Another example of a BIF is an "interrupt return" function to be used in a multiple-stack environment. The call to the function would look like

```
INTRTN (STACK.DESCRIPTOR);
```

The resulting machine code loads the stack descriptor and then invokes the machine's interrupt return switching mechanism.

Built-in functions are best considered as macro instructions which can be interpreted by the compiler to generate the appropriate assembly language sequence. The definition of these macros determine the BIF and provide a HOL interface to various implementations of other modules (e.g., in this example the SIN module). It is important that these macro definitions be carefully organized and controlled by the system designer. (See Section 3.2.6(a).) It is also important to provide convenient methods for coding these macros. The Machine-like Code feature of COL appears to be an excellent candidate for a language to define BIF.

3.3.3 System Architectures and HOL's.

(a) The simplest system architecture, Figure 3.3-1, consists of a single computer which contains various algorithms required for the specific processing task.

To provide more parallelism, speed, etc., some of the algorithms may be "extracted" from the main program and implemented in fixed-wire hardware, or "algorithm boxes". Assembly language or BIF's are used to connect the processing software to the algorithm units.

I/O hardware may also require HOL support. As requirements change, the algorithm boxes may be expanded and themselves become processors with alterable software. Clearly, the HOL must allow the modification and implementation of these processors with minimum impact on existing software.

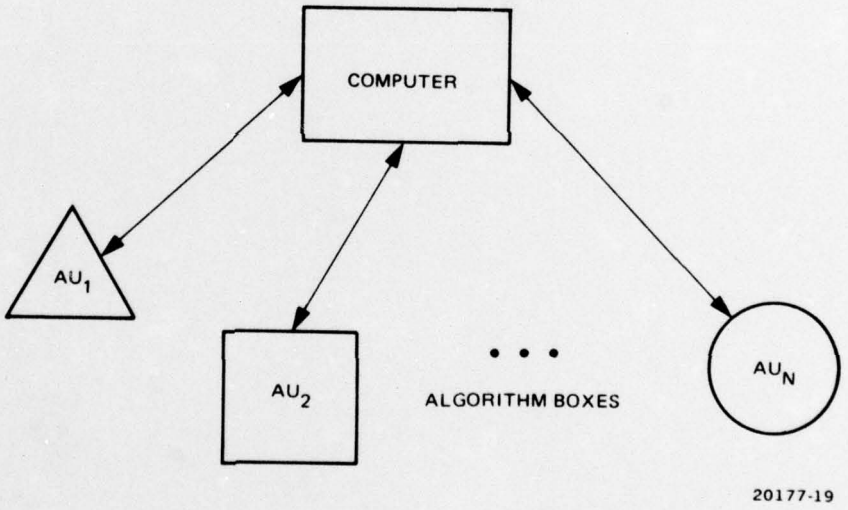


Figure 3.3-1. Simple Architecture.

(b) The pipelined architecture (figure 3.3-2A) is usually used for signal processing. It has the following characteristics:

1. A limited number (usually 1) of higher speed inputs;
2. Incoming data is sequence dependent; i.e., if input b to processor P follows input a, then output P(b) must follow output P(a);
3. The pipeline may fan out to several parallel outputs. Perhaps in this case it is best to consider the structure as one pipeline feeding into several other processors, as in Figure 3.3-2B;
4. The system is usually non-homogeneous; that is, composed of processors that are unlike;
5. The various hardware elements usually have fixed task assignments;
6. Fault tolerance may be unnecessary on a per pipeline basis. Each process (P, Q, R, etc.) is made up of combinations of hardware and software which execute concurrently. I/O between the processes and control of concurrent execution may be managed by the HOL.

(c) The Matrix/Array Architecture (Figure 3.3-3), on the other hand, is usually used for message processing, and has the following characteristics:

1. Many low-to-medium speed inputs;
2. Various inputs are timewise independent;
3. Various outputs are timewise independent;
4. Total load may vary drastically, in the short term. Often a long-term increasing trend persists;

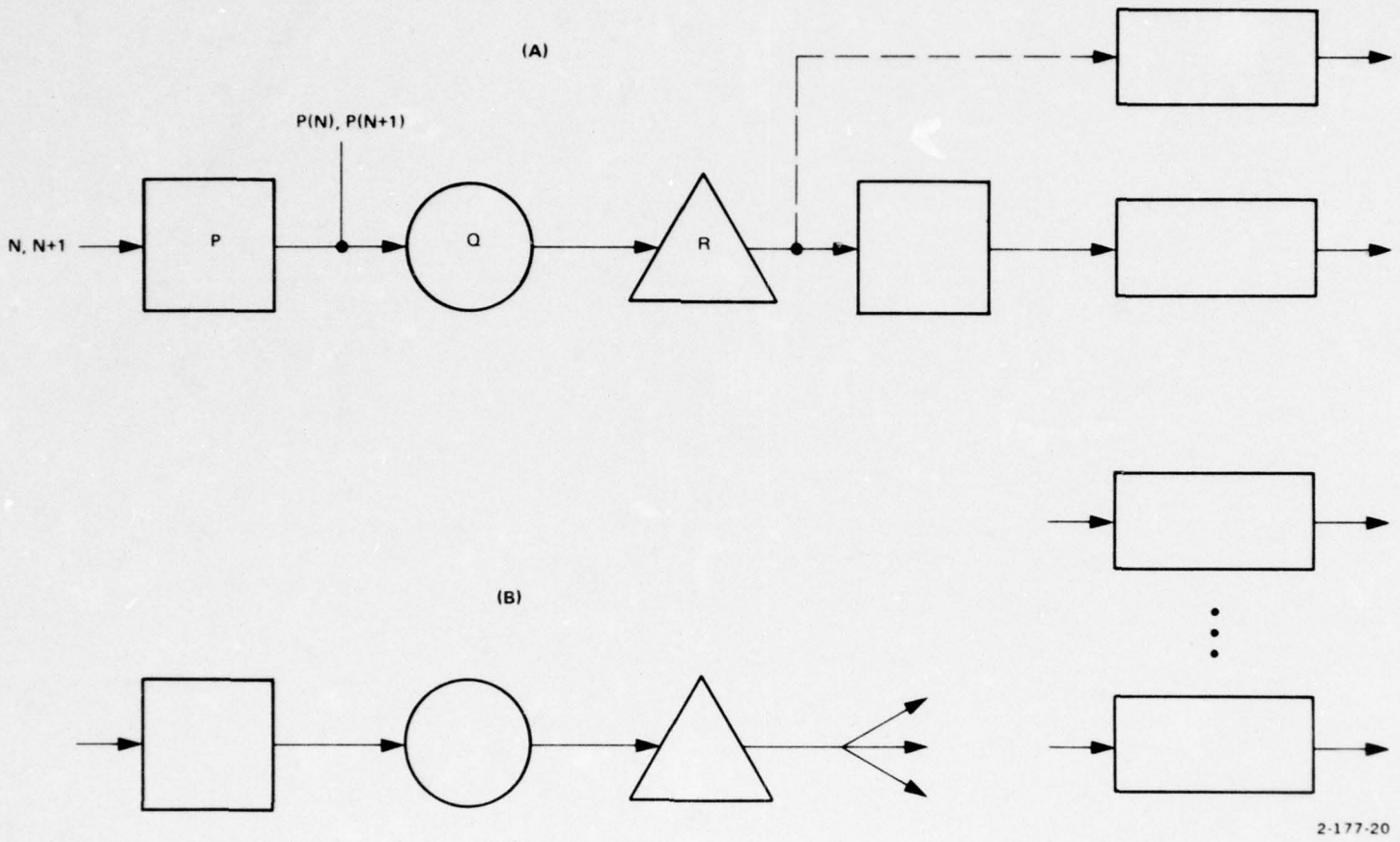


Figure 3.3-2. Pipeline.

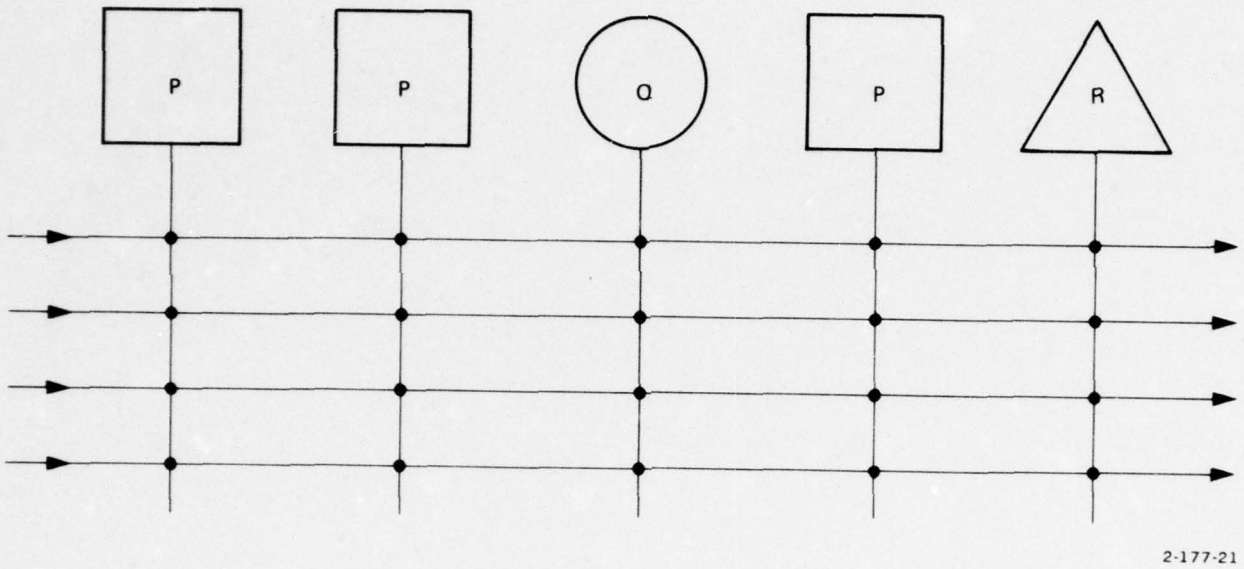


Figure 3.3-3. Matrix/Array

5. System is often homogeneous (i.e. similar processors) or nearly so;
6. Task assignments are usually dynamic;
7. Fault tolerance is often on a per processor basis. Here, as in the pipeline system, the HOL must support I/O and concurrent execution.

(d) Non-Homogeneous Architecture and HOL

System architecture 1 (the simple system) presents few problems to the HOL except the control of concurrent processes and I/O. However, architectures 2 and 3 are more complex and so present more complex problems to the HOL. The most significant of these problems is the non-homogeneous environment presented by a non-homogeneous system.

A homogeneous system is composed of similar processors having similar scope and similar interconnections. The HOL of such a system can treat each processor identically and in a straightforward manner. Often, a homogeneous system can be treated as little more than a simple system with some task assignment mechanism. Little more requirement is placed on the HOL of such a homogeneous system.

A non-homogeneous system presents a non-homogeneous environment to the software processes and control. This non-homogeneous environment can arise from several causes:

- by incorporating processors that are unlike:
  - by original design (e.g., a pipeline),
  - by evolution (e.g., retrofit with newer hardware),
  - by failure of a redundant processor.
- by design allowing dissimilar scope; that is, not all memory (or I/O or other resource) is available to/addressable by all processors.
- by dissimilar interconnections, e.g., dedicated slaves.

Further, non-homogeneous systems are more likely to have fixed-wired (or built-in) task assignment, which may require HOL interfaces. Another task assignment alternative is to allocate tasks (either manually or automatically) at compile/assemble time. This requires the HOL or its compiler to provide for the task assignment process.

Compiler utilization for non-homogeneous systems must ensure that correct machine code is generated for the correct processor. Data representations must be compatible with the processor(s) in which they reside.

3.3.4 Software Architecture Considerations

(a) Contiguous versus fragmented software.

By contiguous software, we mean that the processing description for an entire system is described in one program, and that somehow the components of this program are mapped, as in figure 3.3-4, into the various hardware components. The goal of the long range approach of section 3.2 is to provide the mechanisms for achieving a contiguous design, in which we can achieve.

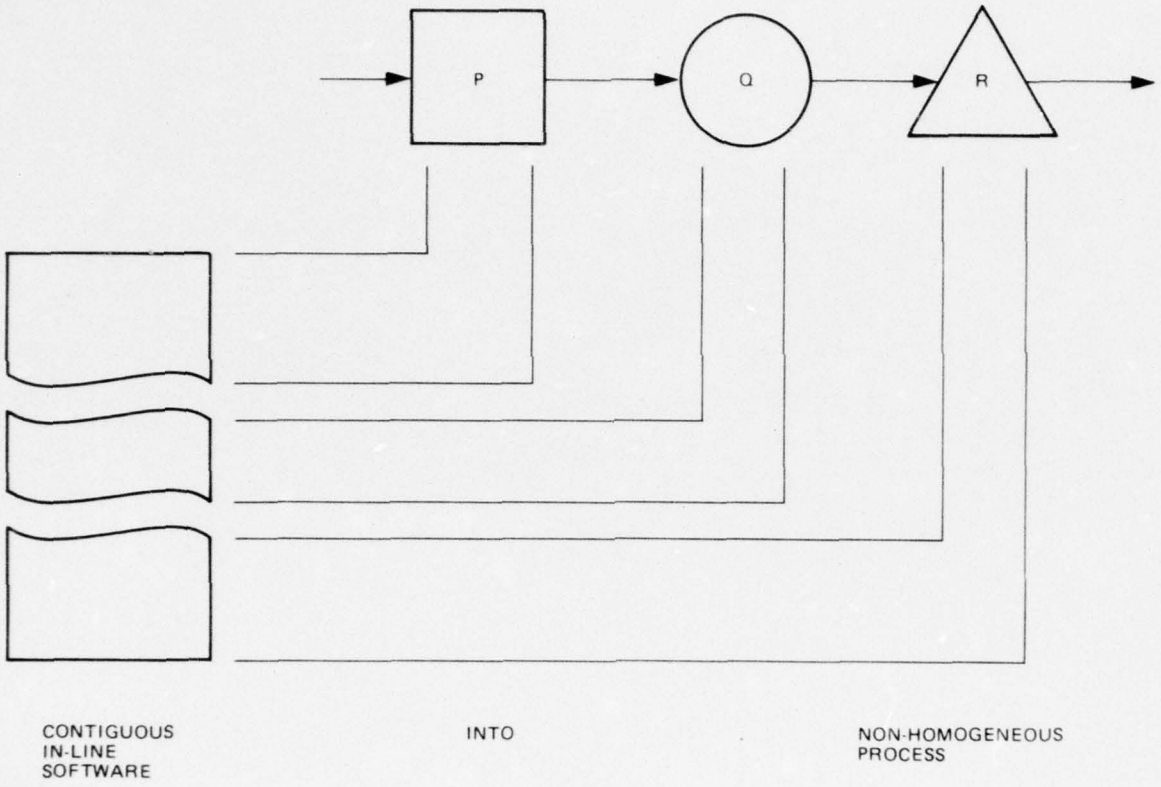


Figure 3.3-4. Program to Hardware System Mapping.

- A unified description of the total processing algorithm
- A convenient documentation for simulation purposes.

In the near term, "fragmented" software as depicted in figure 3.3-5 will no doubt continue to be the rule. Here we have disjoint program modules, one for each processor in the system. Here, software modules do not communicate directly, but only through the I/O connections between units.

Fragmented systems require I/O queues for communication and buffering of tasks and data. The implied queue control should be kept simple. For example, a hardware first-in-first-out (FIFO) stack could be used.

Another Operating System design area of concern is the set of interfaces needed for Applications programs to access I/O functions. This interface must be simple to avoid costly overhead. Techniques include:

- Memory-managed I/O, where I/O accesses are simply treated as if they were memory store/fetches.
- Special subroutine calls with minimum overhead

b. Influence of hardware architecture on HOL requirements.

A further set of assertions are made:

o Simple architecture:

D1: Procedure calls and BIF's handle the linkages between modules.

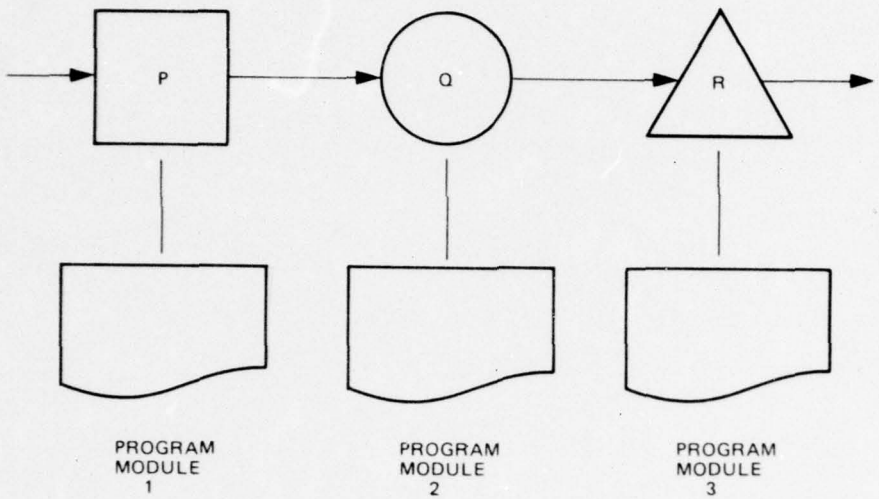


Figure 3.3-5. Modular Software Fragments Dictated by Hardware System Architecture.

D2: Parallelism in this case would require that the HOL have provisions for parallel task control, namely the mechanisms for starting, stopping and synchronizing tasks.

- Non-homogeneous system, multiple CPU's:

  D3: If a common HOL is to be used, multiple compilers or at least separate code generators are needed.

  D4: A more likely situation is that multiple HOL's would be used on the basis of cost.

- Homogeneous system, multiple CPU's:

  D5: Resource sharing and Load sharing imply that reentrant code be generated to allow sharing of routines.

- Task assignment, matching hardware system characteristics to task requirements:

  D6: At compile time, automatically by the compiler, by dissecting the program into fragments that correspond to tasks.

  D7: At compile time, manually in the case of multiple-compiler systems.

  D8: At run time: the compiler would tag software task-modules to identify which resource(s) could execute the task. The Operating System would then dynamically assign the task to an available resource.

  D9: Both the automatic and dynamic approaches imply a means for specifying task requirements and machine characteristics in the HOL source code.

- o Memory managed I/O:

  D10: Requires the ability to equate symbolic names to absolute fixed addresses.

- o Applications area, special requirements:

  D11: Signal processing requires that arithmetic facilities be available for integer, real and complex arithmetic.

  D12: Message switching requires the capability for character handling and bit manipulation.

3.3.5 HOL Criteria

We are now in a position to establish some criteria for HOL comparisons for this applications area. They can be derived from the Preliminary Assertions of section 3.2.2, as influenced by the above considerations based on specific hardware architectures.

The objective is to allow meaningful comparisons whichever of the two following (opposing) viewpoints is taken:

I. We want the most powerful and well-designed language; efficient compilers and software support facilities are sure to follow.

or II. Almost any HOL can be caused to perform a given task; we want one with demonstrated efficient implementations in this applications area.

Figure 3.3-6 shows the criteria that have been developed, the assertions that support them, and a very "binary" comparison of the presence or absence of this capability in each language, where YES and MLC indicate merely that a capability exists. For more detail see Appendix M.

		PL/I	COL	SPL/I	Assertions
1.	HOL mechanisms for parallel task control	YES	MLC	YES	A1, A2, A3, A4, B3, D2, D6, D7, D8
2.	Integer arithmetic	YES	YES	YES	A5
3.	Real arithmetic	YES	YES	YES	A5, D11
4.	Complex arithmetic	YES	NO	YES	A5, D11
5.	Character handling	YES	MLC	YES	A5, D12
6.	Bit manipulation	YES	YES	YES	A5, D12
7.	Precision control	YES	YES	YES	A5
8.	Controlled escapes	NO	YES	NO	C1
9.	Block structuring	YES	YES	YES	C2
10.	Type checking	NO	YES	YES	C3, C7
11.	Efficient object code	YES	TBD	YES	C5
12.	Reentrant code	YES	YES	YES	C6, D5
13.	Support software	YES	TBD	YES	C8, C9
14.	Absolute address	YES	MLC	NO	D10
15.	Machine char spec	NO	MLC	NO	D9
16.	Built-in functions	NO	MLC	NO	D1
17.	Transportable compiler	YES	TBD	YES	C1

NOTES:

MLC: Can be done using COL's machine-like code together with other language features.
TBD: To be determined. COL is not implemented.

Figure 3.3-6. Language Comparisons

3.4 Conclusions

Existing HOL's were studied in the context of on-going DoD initiatives and the requirements of the signal processing and message switching applications area.

A number of assertions were made regarding necessary and desirable HOL capabilities in this context. From these assertions and the supporting discussions, a set of criteria was developed. Comparisons of existing HOL's against these criteria led to the following conclusions:

a. None of the HOL's studied met all of the criteria; all would require an escape to some Architecture-Oriented Code (AOC).
b. Such escapes must be carefully controlled.
c. The approach to controlling escapes should closely resemble that taken by the designers of COL, namely, the processing of AOC under the auspices of the HOL compiler itself.
d. By limiting the number of different computer instruction sets, a viable long-range approach becomes possible.
e. For the purposes of this study, only three HOL's need be considered in detail: PL/I, COL and SPL/I.
f. Of the implemented languages, SPL/I most closely fits the criteria for this application area.

3.5 Recommendations

It is recommended that DCA consider requiring COL to include complex arithmetic capabilities and a more explicit method of handling character strings. If this were done, it is our opinion that COL would meet the

criteria for this applications area more closely than any of the languages studied. Further, it would be a strong candidate for the proposed set of DoD standard languages.

It is also recommended that DCA consider a number of topics where further study would be productive in the area of Architecture-Oriented Code:

a. Define which elements of AOC are most likely to show a one-to-one correspondence over a range of hardware implementations.

b. Define the macro capabilities needed in the AOC for greatest flexibility and ease of use.

c. Define the complete role of the HOL compiler in standard handling of different AOC sets, with special emphasis on checks, diagnostics and error messages.

d. Provide examples showing how the implementation of the foregoing definitions can lead to a methodical and reliable technique for converting an existing AOC module to a new one which is directed at a different instruction set.

4. Hardware/Software Alternatives

4.1 Introduction

Hardware/software tradeoffs were identified as a part of the architecture selection process, and the system model described in Section 2 provides user control of these options. The purpose of this section is to describe (first in general terms) the items of interest in hardware/software trades and their potential implication. Subsequently these trades are described for switching and signal processing systems, and summarized for reference purposes.

Other tradeoffs are discussed under the classification "hardware/hardware" wherein trades of significance other than simple speed changes are considered. Application of the tradeoff process is also described as a part of the overall design methodology presented in Section 5.

4.2 Hardware Software Tradeoffs and Their Implications

In a message switching system, all functions are realizable in either hardware or software once sufficient basic electronic hardware exists to perform minimum required functions (input/output, storage, etc.). This is due primarily to the fact that message switches perform event-driven, rather than clock-driven, functions, and have almost no requirements for high-speed, high-volume serial operations (such as multiplications in a signal processing environment) that may dictate hardware implementations for speed considerations. Since nearly all message switch processing functions can be implemented in either hardware (i.e., fixed logic) or software (i.e., programmable logic with stored-program memory), tradeoffs can be made on a cost basis.

Hardware has certain advantages over software: it is almost always faster

than software in performing a given function, it uses no additional system resources (such as memory) other than its own integral circuitry, and it is simpler to use from a control standpoint. However, software is more flexible and adaptable. Relative costs of implementing a given function in hardware or software depend on the complexity of the function and its frequency of use in the system.

Functions that are best implemented in hardware are generally simple and compact, well-defined in terms of both process and inputs/outputs, and are very frequency used in the system. As functions become more complex and time consuming, less well-defined, and less frequently used, they become more economical in software. Flexibility and adaptability can easily be interpreted in cost terms: every change in process involves redesign and remanufacture if that process is implemented in hardware; it involves redesign but (in general) no remanufacture if it is a software process. Performance of multiple functions in a module requires a multiplicity of designs and a multiplicity of electronic configurations for a hardware implementation; it requires only a single electronic configuration and a multiplicity of stored programs in memory for a software implementation.

Decreasing hardware costs are being found in memory devices and programmable logic as well as in fixed logic, so no general conclusions can be drawn regarding hardware/software tradeoffs because of this trend. Of greater importance to the systems engineer is the question of modification and update of both hardware and software. Costs in this area exceed those of procurement over a machine's life cycle. Again, no specific recommendation may be made via-a-vis hardware/software. The following points must be considered to support a decision:

(1) Hardware modification and maintenance costs, when labor intensive methods are used, will continue to increase. If automatic test features are built-in and they pass tests for completeness and programmability (i.e. adaptible to future equipment changes) costs may be expected to drop 20-50%.

(2) Software modification and maintenance expenses, when changes are compatible with the basic machine, are a direct function of the user's familiarity with the code. Two areas will result in excessive user cost:

(a) The programming task was subcontracted and the detailed code is poorly understood by the user.

(b) A non-standard (uncontrolled) compiler is used. (Recognition of D.O.D. requirements in the future should mitigate this problem.)

Finally, the decision to implement a particular function in hardware or software will depend on the decisions regarding other functions performed in the same module. A hardware implementation of a given function is normally designed to perform only that function. If that function is implemented in software instead, there is often capacity in the programmable logic to perform additional functions as well. In these cases, additional functions can be performed in software at little additional cost, almost certainly at less cost than in hardware.

4.3 Specific Hardware/Software Tradeoffs

4.3.1 Switching

The interface units defined previously for each candidate architecture (MIC for the Matrix, MIU for the Serial Bus, BEU for the Parallel Bus) perform several functions which are the subject of hardware/software

tradeoffs in designing a Message Switch. All the interface units may perform address translation on traffic leaving the modules. MIUs and BEUs may perform address decode on bus traffic to determine if the traffic is intended for their modules (it is assumed that an MIC will not see incoming traffic addressed to another module).* If the modules are widely separated physically, or if the Message Switch environment is electrically noisy, it may be necessary to perform parity checks or even error detection and correction (EDAC) functions on all traffic. Fixed logic devices exist today to perform simple (e.g., single character) parity checks and short (up to 8 bit code) polynomial error correcting functions. For these cases, hardware implementations are cheaper as well as faster than software. Hardware implementations may also be forced by processing speed requirements due to high traffic volume. For more complex parity or EDAC functions, software implementations may be more cost effective, barring speed requirements as noted above. The parity and EDAC functions envisioned for the Message Switch outlined in Appendix F will be performed on input/output lines in the Line and Trunk Handler Units, and should be of sufficient simplicity for hardware implementation.

Two other functions subject to hardware/software tradeoffs are addition/deletion of character start and stop bits on asynchronous lines, and code translation (e.g., ASCII/Baudot). Both are normally implemented in a combination of hardware and firmware.

* A major consideration in these hardware/software tradeoffs is the complexity (i.e., word length) of the addresses to be processed; the more complex the addresses, the more likely is a software implementation.

4.3.2 Signal Processing

Referring to the selected architecture, the modeled configurations allow either centralized or distributed control. Centralized control most typically will be used in real-time-driven signal processing systems and in similar systems characterized by the need to perform a fixed (non-branching) sequence of operations in a relatively short period. Alternatively, distributed control, wherein each module bids for service as a function of its input data (or of flag testing), is most appropriate for message switches.

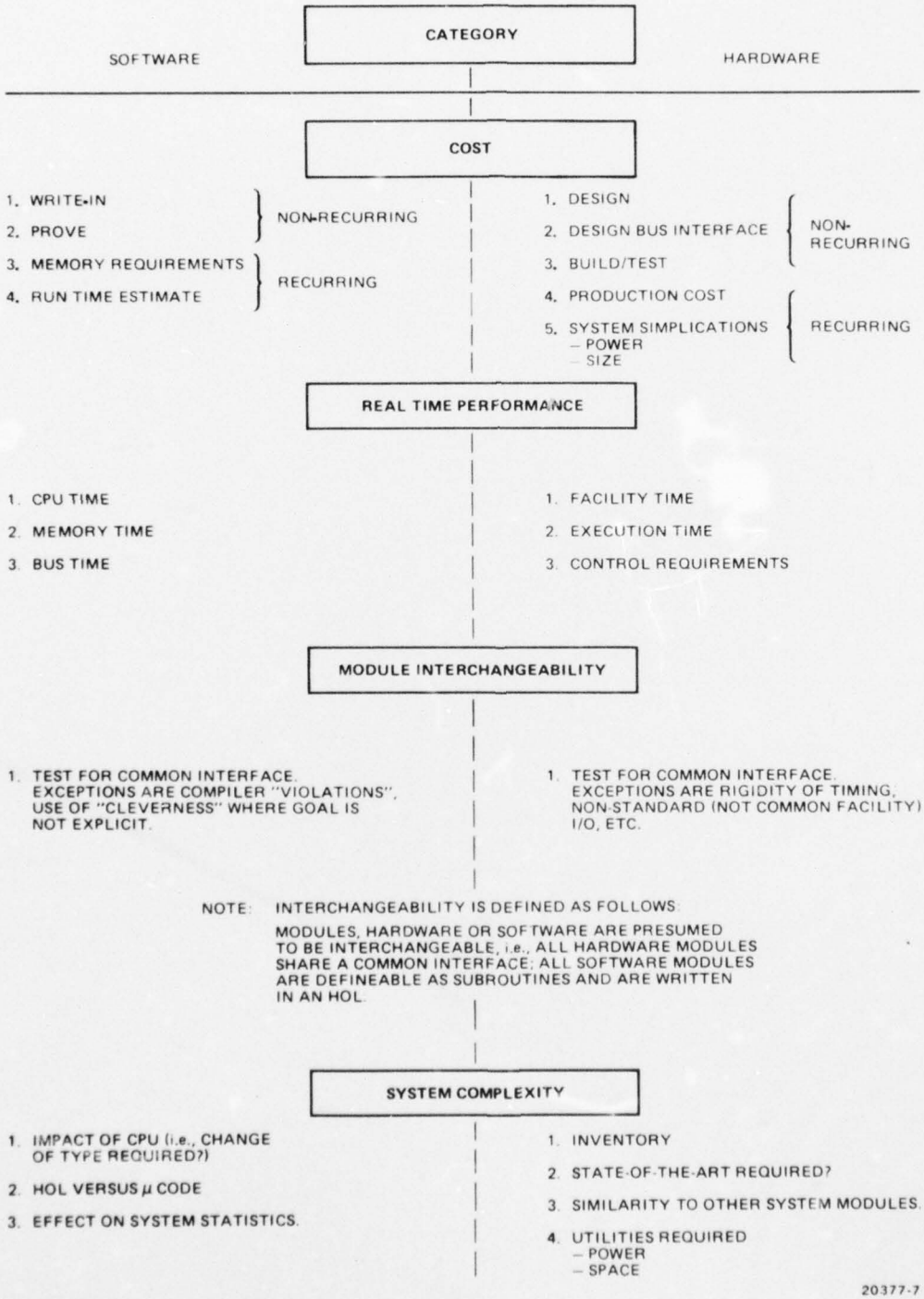
It is this difference of control requeirements that led to the provision for alternate methods in the parallel bus architecture. In terms of physical realization, the centralized control can be either hardware or software--the basic determiner being system speed or allowable timing jitter on function initiation. Distributed control allows for the possibility of greater flexibility, and in certain smaller systems, lower cost. It also anticipates the implementation of many modules in micro-processor based hardware, i.e., systems which are easily adapted to distributed control.

4.3.3 Hardware/Software Comparison Summary:

The preceeding material deals with the tradeoffs and alternatives associated with development of the modular parallel bus system. Decisions concerning the desirability of hardware vs. software for a specific module in specific system applications will be based upon the designer's evaluation of these and many other related criteria. The methodology in

Section 5 is intended to organize and quantify this effort. In addition, Table 4.3-1 below is organized to show corresponding decision types in the evaluation of hardware vs. software for module design.

Table 4.3-1. Hardware/Software Drawing Types.



4.4 Hardware/Hardware Tradeoffs and Their Implications:

Hardware/hardware tradeoffs have been made in three major areas in designing the Message Switch outlined in Appendix F: Centralized versus Distributed Memory; Bus Access Control; and Process Control. All these tradeoffs involve alternative design philosophies, not merely substitution of different hardware devices which are generally functionally equivalent. The tradeoff between centralized and distributed memory involves considerations of bus contention in movement of message blocks, of bus access control, and of the total memory required for in-transit storage of messages. In a decentralized memory configuration, each Line Handler or Trunk Handler contains in-transit storage for its incoming messages. This configuration requires only one transit on the central bus for a message between lines on separate Line Handlers, and no transits on the central bus for a message between lines on the same Line Handler. Contention for use of the central bus is greatly reduced from that in a centralized memory configuration, where two transits on the central bus (one from the input Line Handler to memory, one from memory to the output Line Handler) are required for each message. Distributed memory requires enabling of individual Line Handlers to access the memory within other Line Handlers for message movement; this interferes with the parallel processing of messages in all the Line Handlers. In a centralized memory configuration, all Line Handlers may access common memory, but not each other's local memory, and the parallel processing is unimpeded. Distributed memory allows dynamic allocation of a Line Handler's local memory to the various lines terminated in that Line Handler, but this scheme will invariably require more total memory (impacting system size, cost, power consumption, reliability, etc.) than a centralized memory configuration where storage

can be dynamically allocated to inputs from all the switch ports. It was felt initially that bus contention would not be so serious a problem as to require distributed memory, and so a centralized memory configuration was chosen for the target system. Subsequent simulations of the target system in this program have shown low bus utilization and contention, justifying the choice of centralized memory.

The question of centralized versus distributed bus access control is still being investigated. While distributed bus access control might require less complex hardware and software than centralized control, it is not clear which implementation results in less bus contention or higher system throughput. The switch simulations to date have shown such low bus utilization that a choice is difficult to make. The bus access control method is clearly not a limiting factor on system performance for the target system.

The question of centralized versus distributed process control is also still being investigated. In this context, process control refers to the coordination, assignment and initiation of individual process tasks such as code translation, logging, and table searches. Process control is normally a function associated with the functional modules of a system and not with the architectural hardware. (Technical control, on the other hand, refers to those functions involved with maintaining the integrity of the architecture's transmission paths, and is normally associated with some part of the architectural hardware.) To date, simulations have assumed distributed process control; the CPU and the Line Handler Units operate in parallel, each driven by external events (e.g., incoming messages from subscribers or other switches) or by communications in a

portion of main memory which is periodically accessed by all units. It is felt that distributed process control is more appropriate to the general problem of message switching, because of its flexibility and growth potential. However, it may be that for a very small, lightly loaded switch, centralized process control could be implemented in a small CPU and result in a less costly configuration. Since a realistic simulation of the centralized process control configuration would require simulation of the internal processes of individual modules as well as the overall switch architecture, such investigations have not been done to date.

4.5 Software/Software Tradeoffs in Message Switching

4.5.1 Language Level

In developing the software for a message switching system tradeoffs can be made between the use of High Order Languages (HOL) and assembly languages. To analyze the tradeoffs, several fundamental differences between HOL and assembly languages must be understood.

Generally, a programmer can develop, test and debug the same number of lines of code in a given time irrespective of the level of those lines of code. Because a HOL characteristically will require less lines of code per program than assembly, a manpower savings is realized by using HOL in software development.

A HOL requires a greater object program size than a corresponding assembly language program. As the cost of memory continues to decline the program, size becomes less critical. However, in small military message switches, where space is still at a premium, it may not be feasible to increase the memory size to accommodate use of a HOL.

Another consequence of larger program size is longer execution ties caused by additional instructions that must be executed. In most cases this additional time is insignificant in the overall performance of the system. One case where this is not true is in the processing of high-speed data lines. Assembly language code may be required in this instance to meet the speed of service requirements.

A primary advantage of HOL use is the transportability of programs from one computer architecture to another. In small military message switching systems there is little need for this transportability since most systems are tailored to match a particular computer architecture.

One advantage of HOL's that is often overlooked is the ease of documenting. Military message switching systems generally require detailed and complete documentation of all system software. Because of this requirement, it may be more cost effective to use a HOL that would reduce the amount of documentation required.

In conclusion, the system software will contain a mixture of assembly language and HOL code. The breakpoint between their usage can only be determined from careful analysis of the particular message switch. In general, those sections of software that receive the most use should be implemented with assembly code. For instance, the input/output routines and major sections of the operating system should be coded in assembly. On the other hand, less critical software, such as editing and routing, can be coded with a HOL.

4.5.2 Microcoding to extend HOL

Just as trade-offs between a HOL and assembly language can be made, so can trade-offs between assembly and microprogramming. Since microprogramming is used to define the assembly language instructions, new instructions may be implemented by changes to the control store where the microprograms reside. In general, it is rather difficult and time consuming to make these modifications on the currently available computer mainframes. However, the message switching application may lend itself to the development of an extended instruction set that would greatly reduce the execution time of repetitive sequences of code.

Input/output processing is one area that would lend itself to improvement to receive a character from a telecommunication line requires upwards of 100 instructions. By performing all the functions of these 100 instructions in one new instruction, a significant speed improvement could be realized.

Table look-up or maintenance is another area that could profit from instruction extentions. Combining the instructions needed to perform a table search into one new instruction would offer reduced execution time.

Carrying this concept farther, a new instruction could be developed for each HOL statement type. This would allow the processor to execute the HOL directly. Such an implementation would be considerably faster than a conventional software implementation.

In conclusion, microprogramming offers a powerful extention to a HOL, but is cost effectiveness is questionable.

5.0 DESIGN METHODOLOGY

5.1 Background

The previous sections of this report have developed the criteria for a modular architecture which has attractive attributes for the implementation of modular data and signal processing systems. It is the purpose of this section to describe a design discipline or methodology which can facilitate both the initial and evolutionary development of such systems by assuring that cost effective and technically adequate system designs are produced.

There is an implicit methodology in any system development endeavor that yields results; however, the degree to which the methodology is explicitly recognized and defined in order to quantify and communicate the evolving design decisions can be a significant factor in determining the degree to which the resultant system meets the needs of the intended user. An explicit methodology defines the directed flow of time sequential and interdependent development steps and insures by quantitative and qualitative evaluation that system objectives will be achieved. A design methodology may then be defined as a codification of processes, techniques and approaches into a time sequential structure of interdependent steps which, when followed, assure efficient transformation of well defined user needs into an effectively functioning and operationally adequate system.

5.2 Definition of a Design Methodology

For the purpose of this discussion, the design methodology shall include all necessary steps between the first identification of an operational user need and the verification of total system performance as a prerequisite to production. Figure 5.2-1 shows the total design process broken

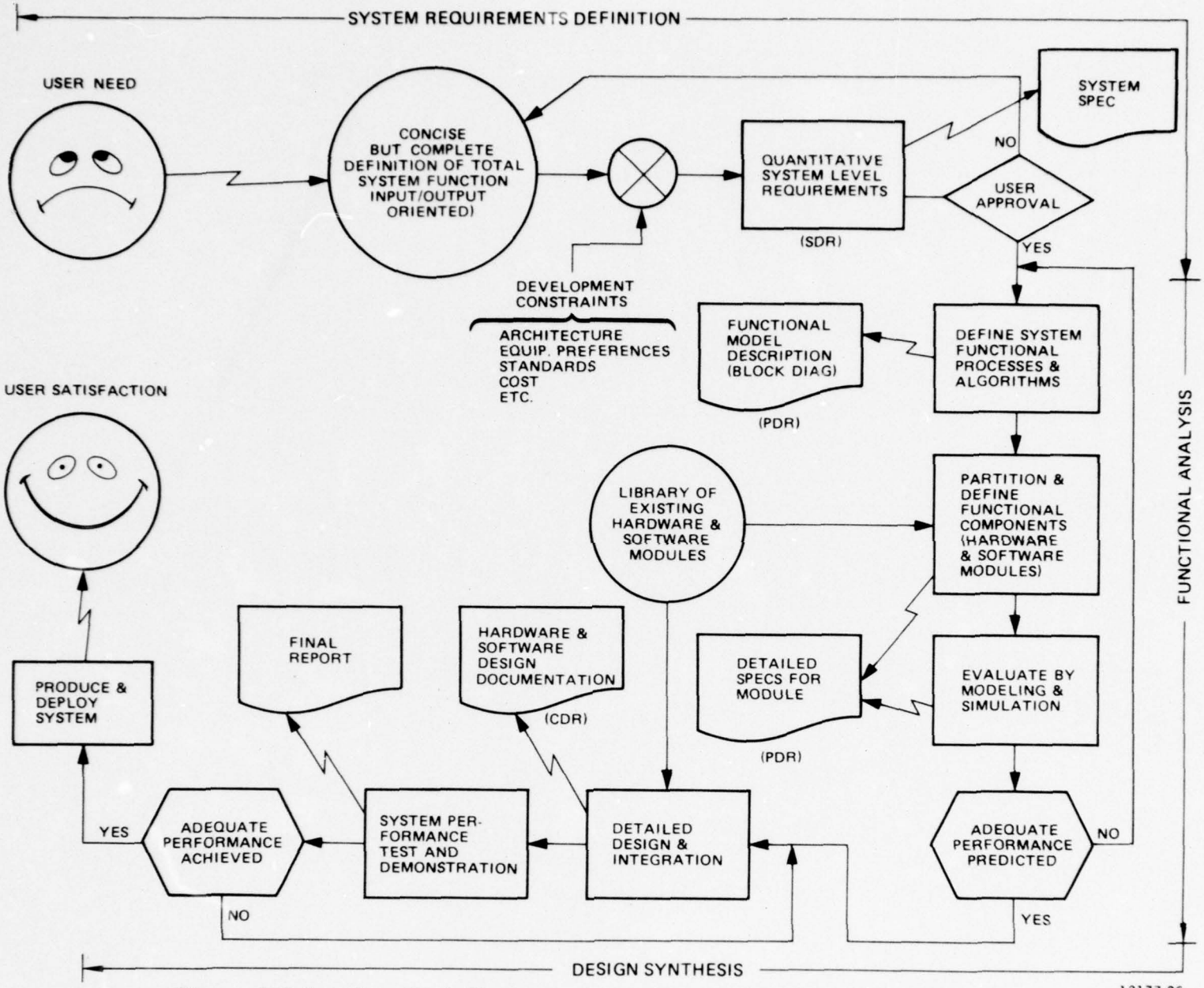


Figure 5.2-1. Essential Elements of a System Design Methodology.

down into time sequential and interdependent elements. Each element belongs to one of the three major phases of the design methodology, the results of each phase being well documented.

1. System requirements definition-during which the operational need is analyzed and developed into clear and, to the maximum extent, quantitative statements of the system requirements. This phase, principally performed by the designated system engineering function includes the identification of desired constraints upon the subsequent design activity. The principal documentation of this phase is a definitive system specification.

2. Functional analysis-during which system level requirements are broken down into functional processes and algorithms required to transform the input data into the desired output data. Once necessary functions are identified, these are allocated to hardware and software modules (both real and postulated) based upon the results of trade studies on alternative approaches. Individual and collective function models are constructed and computer-aided simulations run to validate the design approach and assist in trade-off studies. The principal documentation of this phase is detailed specifications of hardware and software modules consistent with system design requirements.

3. Design synthesis-during which detailed hardware and software logic design is completed and design verification tests are performed at module through integrated system levels. Satisfactory completion of this phase initiates production and operational deployment and the principal documentation consists of hardware and software design documentation and a final test and engineering report.

The entire design process may be viewed as the development of progressively more mature models of the object operational system finally constructed. The final "hardware" system may, in fact, consist of special hardware-implemented processing modules, general purpose processing modules (computers) and program modules (software) which control the computers. All expressions or representations of the system design prior to the realization of verified operational system are then "models" of that system. The final and most accurate model is the set of detailed design drawings from which the system can be produced. The design methodology must then assure that in the process of constructing successively more complete and accurate models all system requirements are properly weigned and incorporated. This must include cost, properly apportioned to the life cycle elements. Factors such as operability, maintainability, reliability, services life, and projected application and performance evolution, which influence technical design, performance and cost, must be not be omitted from early design models or the objectives related to these issues cannot be achieved. The design methodology must explicitly recognize the progressive model nature of the design process and establish the discipline within which all system requirements factors are adequately considered.

5.3 Description of the Design Methodology

5.3.1 Introduction

Table 5.3.1 relates each phase to its objectives, the pertinent models and the documentation form. Rather than repeat the content of that figure, the following discussion supplements it by commenting on particular features of the methodology, particularly as related to modular systems. Note that the evolution of models proceeds from a relatively simple

PHASE/OBJECTIVES	DESCRIPTION OF MODELS	DOCUMENTATION FORM
System Requirements Definition		
o Analyze and understand user need. o Identify and establish applicability of implementation constraints. o Catalogue and quantify system inputs, outputs and transformation functions. o Achieve user concurrence through iteration as required	o Verbal statement of user need supported by diagrams. o Verbal expression of system level requirements including implementation constraints. o (If needed) System level computer model to verify input/output relationships especially relative to interfacing systems.	o Operational requirement definition document (primarily user-originated). o System requirements specification (Generated by system design function and approved by user). o Separate report but results must be factored into system requirements specification.
Functional Analyses		
o Identify and define internal system functions, processes and algorithms needed to accomplish system requirements. o Partition system (allocate functions to hardware and software modules) to adequately satisfy all requirements. o Demonstrate adequate system performance through simulation (necessity proportional to system complexity). o Achieve preliminary design approval by responsible agency.	o Functional block diagram showing input to output flow. o Computer or analytic models of individual functions or aggregations to validate functional definition. o System block diagram showing allocation of functions to hardware and software modules. o System simulation model sufficiently complete and accurate to predict adequate system performance after detailed design.	o Functional flow diagram and written definitions of individual processes or algorithms (technical design notes or working papers). o System interface specifications and module interface and function specifications. o Preliminary Design Review (PDR) data package including above documentation and supporting computer simulations.
Design Synthesis		
o Detailed design of system hardware and software modules o Integration of hardware and software modules to achieve overall system function. o Formal verification that system design meets all requirements. o Approval of the final design by the responsible agency.	o Circuit schematics, logic diagrams, flow charts, program source and object listings. o More advanced computer models of actual designs or prototype hardware and software system. o Complete or representative portions of prototype system subjected to tests.	o Design documentation package - all specs and drawings required for production. o Integration test plans, procedures and data supported supplemented by computer simulations. o System performance test plan, procedure and data. o Final engineering design and test report.

Table 5.3-1. The Relationships of Design Methodology Phases, Objectives, Models and Documentation.

statement of the user need to a prototype or partial prototype of the system itself. Modularity, especially in large systems such as message switches having a multiplicity of the various module types, makes it feasible to test limited subsets of the total system (with appropriate cautions) to verify the design.

5.3.2 Requirements Definition

Early in the methodology during system requirements definition, statements of user need and system level requirements constitute the most accurate and complete description of the system possible at that point in the development cycle. It is particularly important at this point to maximize the accuracy and completeness of the requirements definition and to document these unambigiously in the form of a System Requirements Specification, for this statement of requirements subsequently drives the entire design process. For the same reason, the intended user must review and approve the system specification to provide initial assurance that his need will be satisfied. Tables and charts should be employed in the system specification to a maximum practical extent to concisely state requirements and minimize susceptibility to verbal ambiguity. High level computer models of the system may be employed at this time to a validate gross system performance requirements in terms of input and output characteristics and to establish specifications for interfacing systems.

The following topics must be considered during system requirements definition to achieve a complete specification:

A. Interfaces: a complete catalogue of system inputs and outputs must be prepared and should include for each:

1. I/O types and quantities
2. Signalling form and data rates
3. Communication Protocols (e.g. SDLC, ADCCP)
4. Message formats and codes
5. Message statistics (arrival rate and distribution and length distribution)
6. Electrical characteristics
7. Noise or error characteristics

A tabulation such as the example given in Table 5.3-2 is particularly effective for this purpose and can be later used during partitioning of lines to line handler modules. Other tables should be prepared to relate message and noise statistics to specific inputs.

B. *System Function: the nature of the transformation(s) performed by the system on data flowing through it should be defined to the extent that the maturity of the system concept permits. Where specific or quantitative information is not available, the need to develop it should be documented.* Operational control requirements should also be defined.

C. Development Constraints: As a minimum, the following kinds of constraints should be considered for specification as "givens" to subsequent development:

1. System architecture, such as modularity, parallel bus, pipelined, parallel processing. Each architecture requires a description as contained in Appendix A-1.
2. Role of special purpose hardware versus program-controlled general purpose processors and specific hardware to be used.

Table 5.3-2. Example of Input/Output Interface Catalogue

Line No.	Data Rate	Code	Protocol	Synch/Asynch	Modulation	Levels	Effective Data Rate	Line Handler Assignment
1								
2								
3								
4								
•								
•								
•								
•								
•								
•								
•								
•								
•								
•								
•								
•								
•								
N								

20377-6

3. Interface and interconnect standards to be applied (e.g. MIL-STD-188 series).

4. Requirements to use specific high order programming languages and operating systems. (DoD Instr. 5000.29 and 5000.31)

5. Specific processing algorithms, hardware and software modules to be used from existing libraries.

6. Maintenance philosophy - a figure of merit can be developed as a function of the equivalence of the levels of modularization and of maintenance.

7. Schedule for development - Schedule for development and integration may impact the technical approach. Compromises in performance, future growth potential, cost and other factors may be required in order to meet an urgent schedule. Modularity permits use of the off-the-shelf hardware and software with interface adaptation to the system interconnect scheme.

8. Cost - System costs should be allocated to system modules in the same manner as technical performance factors to establish controlled cost goals. All trade-offs leading to firm module design definition must consider cost as a primary factor. The life cycle cost concept permits the evaluation of future operation and support as well as current acquisition costs. Specific goals for future and current costs permit design tradeoffs which properly weigh these cost factors. DoD Life Cycle Costing Guides LCC-1, LCC-2 and LCC-3 should be applied.

5.3.3 Functional Analysis

Once the system requirements are completely and accurately specified, the functional analysis process is pursued to achieve firm definition of

system modules as a prerequisite to detailed design. As a first step, identify all the functional processes and algorithms required to address the system data input, transformation and output requirements. Also identify the operational control requirements and their impact on data processing. When the analysis of functional requirements is well in hand, the relative roles of program controlled processors and special hardware will begin to be evident but several alternatives will likely exist. It will be necessary to make preliminary estimates of the timing and memory load on processors and the question of how to implement specific algorithms and processes will be strongly influenced by the relative efficiency of execution in hardware vs. software. For example, execution of the NBS data encryption algorithm places a significant time burden on a centralized CPU can be allocated to a hardware module wich is accessed upon completion of each data block encryption.

A list of functions or functional flow diagrams should be prepared which shows how each system output is created by a transformation of system inputs. In signal processing systems, as defined earlier in this report, one input generally yields one output after a fixed sequence of internal processing. In switching systems, however, the internal processing of input data may vary depending upon several factors including the nature of the data content, its source, its destination, its classification, time periods during the day, and operational control exercised by the system operator. Each possible processing sequence must be identified as a function of controlling parameters. This kind of analysis leads to a definition of the system control structure. When combined with input data statistics, it also permits a determination of relative utilization of

functions (loading). The models developed in Section 1 and the Simulation described in Section 2 together form the method to be used for loading analyses.

A near optimum allocation of system functions to specific hardware and software modules (partitioning) is possible by application of the functional flow analysis together with preestablished performance criteria including maximum interconnect bus loading, maximum processor time loading, maximum memory utilization, cost targets, etc. An important attribute of the modular approach is that previously designed modules (hardware and software) can be selected or adapted to the extent that their use yields cost and development time benefits without acceptable compromise of other performance factors. Where necessary, new modules are designed (sometimes combinations of existing modules) and added to the module library.

System management or technical control functions must also be identified and partitioned in a way consistent with operation and control concepts. Status monitoring, adaptive configuration switching, diagnostic testing, fault isolation, and operational statistics and history keeping are included in this category of functions. The effects that these functions have on system loading must be included when applying performance criteria during system partitioning.

As an example, Table 5.3-3 presents the criteria for allocating input and output lines to line handler modules.

Table 5.3-2 provides a sample structure for preliminary Line/Line Handler partitioning. The I/O lines should be listed, perhaps in descending order

Table 5.3-3. Line Handler Assignment Criteria

FUNCTION	CRITERIA
Maximum Data Rate per Line Handler	As the number of lines increases on a given Line Handler, the maximum allowable data rate decreases (to allow for internal addressing to separate buffers for each line). For example, a Line Handler with a capacity of one 16 kb/s* I/O line with code translation will handle only about four 2400 b/s lines, not six; it will handle only about six 1200 b/s lines, not thirteen.
Message Code	Code translation is normally performed with a mixture of software and firmware. Grouping together lines with the same code (e.g. ASCII, Baudot, Moore) will minimize the number of Line Handlers containing the same software/firmware package.
Line Protocol	Similarly, protocol functions are normally performed with a mixture of software and firmware. Grouping together lines with the same protocol will lower costs.
Synchronous/Asynchronous Lines	For convenience, it is usually best to separate synchronous from asynchronous I/O lines. Since the line interfacing of synchronous and asynchronous lines (and the handling of start and stop bits on the latter) is normally a firmware/hardware function which is preprogrammed (by setting various control signals) on a line-by-line basis, and since there is no difference in the circuit board space required by synchronous and asynchronous receiver/transmitters, it is not absolutely necessary to separate these two types of lines. In practice, the separation of synchronous and asynchronous lines often takes place when lines having different codes are separated (in 2 above).

*Current microprocessor (8080A, 6800) designs will handle two 16 kb/s I/O lines without code translation, or one 16 kb/s I/O line with code translation.

FUNCTION	CRITERIA
Line Electrical Characteristics	It is convenient to group together lines having the same signal modulations (e.g. conditioned diphase, balanced or unbalanced NRZ), and those having the same impedances and peak-to-peak signal voltages. Once again, the interface functions are generally implemented in hardware on a line-by-line basis, thus, it is not necessary to group lines this way.
Load Equalization	Within the context of the above guidelines, one should try to equalize the loads on the various Line Handlers. The "Load" is a combination of effective data rate (data rate multiplied by usage factor), protocol overhead, and code translation overhead, and is at least partially a subjective value. Commonality of hardware among the Line Handlers is desirable for minimizing costs of design, acquisition, and maintenance. In a system where Line Handlers are to be designed, load equalization will allow implementation of common hardware designs at the slowest (and presumably least expensive) level. In a system implemented using pre-existent Line Handlers of common design, load equalization will offer the greatest protection against module overflow under random peak loading conditions.

of data rate, and the seven columns of data on each entered. I/O line number 1 can arbitrarily be assigned to Line Handler number 1. The data on I/O line number 2 should be compared to that on I/O line number 1 to determine, on the basis of the above factors, whether I/O line number 2 can be assigned to the same Line Handler as for I/O line number 1. As each successive set of data on an I/O line is reached, it should be compared to all those above it to determine if that line can be assigned to a previously-numbered Line Handler, or if a new Line Handler number must be established. When all I/O lines have been assigned to Line Handlers, load equalization can take place either by moving line from one Line Handler to another or by consolidating two or more Line Handlers. For example, if there are two Baudot, two Moore, and twelve ASCII lines in a system, the Baudot and Moore Line Handlers could be consolidated inspite of their differing codes, provided the total load was not too great. Successive partitioning of I/O lines over Line Handlers may be done in the same way, using data established in simulation or by test bed experience.

Computer-aided simulations find maximum utility during the process of partitioning. A system model is constructed based upon the initial module set selected. There may in fact exist several module sets or several partitionings between hardware and software implementations which require comparative evaluation. Computer simulation of the system functions and its data inputs and outputs permits a prediction of system performance directly and without extensive manual analysis which may not be tractable or even possible and, in any event, is error prone for a system of any size. The General Purpose System Simulator (GPSS) is an effective tool for this purpose and was used successfully during the Collins study. The model requires accurate definition of module input, output and timing

parameters; verification is aided by trial simulations using simplified system input and timing conditions, the results of which also serve as "benchmarks" for evaluating subsequent and more complex "actual" runs. (The simplified switching system results presented in Section 2 were developed as model verification aids.) By plugging alternative module sets and input conditions into the system model, quantitative system performance prediction data are obtained and are used to select the final partitioning of system functions into a set of hardware and software modules. The requirements of the System Specification and their logical derivatives are the criteria applied to determine whether system performance, as predicted by simulations, is adequate.

In constructing simulation models, it is very desirable to progress incrementally by modeling portions of the system individually using simplified input and output parameters. This not only makes the process of debugging more efficient but permits detailed performance of subsets of system functions to be evaluated under controlled conditions. Ultimately, it is desirable to simulate the entire system. This may be accomplished by linking together independent simulated subsets or by constructing a gross system model for which only worst case (or other specific case) input, output and module operating conditions are treated. In the total system model, system management or non-processing functions must be included to evaluate their impact on total performance.

The objective of the functional analysis phase of the design methodology is to firmly define the detailed requirements for design of modular system elements whose integration as a totally operating system is confidently predicted to meet all system specification requirements. The design

requirements for each module are documented as module specifications, both hardware and software. These specs stress inputs, outputs, control and timing, and define internal module functions through the use of block diagrams, flow charts, state transition diagrams, etc. as may be appropriate. The standardization of hardware module interfaces to a common interconnect facility (e.g., a parallel bus) simplifies module specs and permits a sharper focus on internal functions and reduces the time for their preparation. The module specifications, supported by simulation results, provide a well documented basis for Preliminary Design Review (PDR). They are updated as detailed design proceeds and eventually reside in the module library as the definitive module description document. The need for discipline in preparing and updating module specifications is dictated by their potential use in other compatible modular systems. Exploitation of modularity benefits can only be assured if future users have confidence in the accuracy and completeness of module library data.

5.3.4 Design Synthesis

This report will not dwell on the design synthesis process but it is important to point out that benefits of modularity accrue during this phase as well. Not all detail designs are completed simultaneously. This is true not only because of varying complexity but also because of the practical limits of peak manpower loading. In systems of even moderate size, design of all hardware and software modules in parallel would require an extremely high peak manning in relation to the program average. An engineering organization cannot function realistically under these conditions and the design load must be spread out over time to be acceptable. Modularity, with a common interconnect scheme, permits an incre-

mental checkout and integration of module designs as they are completed. Inputs to a module under test can be simulated by the control processor when the actual source module is not available. Therefore, meaningful system development progress is not adversely affected by staggered design completions, either planned or caused by technical problems.

As actual hardware and software design and integration progress, the results of incremental test can be compared with corresponding simulations performed during the functional analyses phase or with new simulation runs using the same system model. Discrepancies between simulations and actual hardware/software performance can be reconciled either by design changes or updates to the system model. It is desirable to continuously upgrade the computer based system model in order to maintain a simulation resource which can be used throughout the system life cycle to evaluate changes and upgrades prior to commitment to final design. The necessity and expense of system simulator maintenance must be evaluated against the size, complexity and operational criticality of a given system, however.

5.4 Conclusion

The advantages of a modular system have been expressed in terms of reduction of cost and time to implement systems and their evolutionary successors. Modularity increases the potential for use of common hardware and software modules between systems within a generic class and minimizes the impact of evolutionary changes to a given system. In order to exploit these attributes, system design or design changes must be conducted within a design methodology or dicipline which stresses the use of available

modules and which encourages the development of standard or widely acceptable functional modules and interconnect schemes, simulation systems and high order programming languages, rather than unique designs.

During the system definition phase, the responsible system engineering agency may impose constraints on system development by specifying a system architecture which defines such features as a parallel bus interconnect scheme and hardware/software modularity. These constraints should be applied with caution, but can be justified by the longer term benefits which accure from a more standardization design approach. Constraints may extend to the definition of a specific bus implementation, high order language, the use of a standard simulation system, and the requirement to use existing module designs for particular functions. Such constraints at this stage should be judiciously applied, however, to avoid premature foreclosure on valid design alternatives; but, the ability to specify these constraints helps to assure the proliferation of modularity for long term benefit.

Significant advantages of modularity are realized during system functional analysis and design synthesis. Time and cost to firmly define two system designs, to predict performance and to synthesize the system and achieve that performance, can be significantly reduced by a number of modularity effects:

1. Once developed, a large library of well documented hardware and software module designs provides a resource from which previously proven designs can be selected to a maximum extend or, if necessary, can be modified to implement new system functions. Costs for new and modified designs are thus minimized.

2. A predisposition to use existing module designs where possible helps to formulate initial system partitioning. It is likely that the basis for modular partitions of previous system designs, of the same generic class from which the library of modules has been created, will have some validity in a current system design.

3. Assuming that a standard system simulation system has been adopted and test simulation models for each library module are maintained (this is highly recommended), the time to achieve debugged running simulations of new system configurations will be reduced.

4. Even when new modules are required by a new functional requirement or the need to apply advanced technology, existing similar designs provide a basis of approach which helps to reduce design cost and risk.

5. A library of standard module designs and the use of a standard simulation system provide the opportunity to employ computer-aided automated system design techniques when such techniques become sufficiently mature to be applied at the system level. Note the analogy to the standard cell approach to LSI design which has become highly automated. For a bus-oriented modular system, both power and signal interconnect buses are standardized.

6. System modularity, especially when proliferated over many systems and their evolutionary successors, greatly simplifies the logistics support effort and its associated costs.

7. Many modules will be designed with programmable features so that their function can be adapted to a greater range of system applications.

The trend to modularity is evident every where in the electronics industry from LSI standard cells, to IC packaging techniques, to microprocessor functional elements, to consumer electronic equipment. The essential feature of modularity which benefits signal and communication systems as discussed in this report, however, is a standard electrical and logical signal interconnect scheme - a bus - which achieves flexibility and adaptability at the function level in system applications.