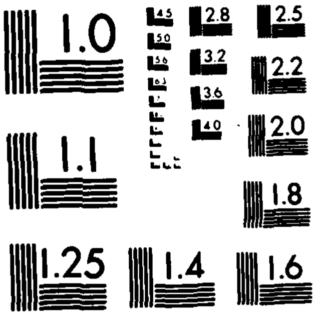
AD-A137 686 DISTRIBUTED PROCESSING TOOLS DEFINITION VOLUME 1 1/3
HARDWARE AND SOFTWARE TE..(U) GENERAL DYNAMICS FORT
WORTH TX DATA SYSTEMS DIV H C CONN ET AL. JUN 83
UNCLASSIFIED RADC-TR-83-107-VOL-1 F30602-81-C-0142 F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

12

RADC-TR-83-107, Vol I (of three)
Final Technical Report
June 1983

DISTRIBUTED PROCESSING TOOLS DEFINITION Hardware and Software Technologies for Tightly-Coupled Distributed Systems

General Dynamics Corporation

Herbert C. Conn, Jr.; David L. Kellogg and Dr. David J. Rodjak

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-83-107, Volume I (of three) has been reviewed and is approved for publication.

APPROVED: Lawrence M Lombardo

LAWRENCE M. LOMBARDO
Project Engineer

APPROVED:

JOHN J. MARCINIAK, Colonel, USAF
Chief, Command and Control Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COEE) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document requires that it be returned.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER RADC-TR-83-107, Vol I (of three)	2. GOVT ACCESSION NO. AD-A137 686	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) DISTRIBUTED PROCESSING TOOLS DEFINITION — Hardware and Software Technologies for Tightly-Coupled Distributed Systems		5. TYPE OF REPORT & PERIOD COVERED Final Technical Report Jun 81 - Jan 83
		6. PERFORMING ORG. REPORT NUMBER N/A
7. AUTHOR(s) Herbert C. Conn, Jr. David L. Kellogg David J. Rodjak		8. CONTRACT OR GRANT NUMBER(s) F30602-81-C-0142
9. PERFORMING ORGANIZATION NAME AND ADDRESS General Dynamics Corporation Data Systems Division P O Box 748, Fort Worth TX 76101		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 62702F 55811829
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (COEE) Griffiss AFB NY 13441		12. REPORT DATE June 1983
		13. NUMBER OF PAGES 208
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Same		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

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)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Lawrence M. Lombardo (COEE)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Computer Embedded Distributed Processing Systems
Computer Software Technologies
Computer Hardware Technologies
Computer Object Oriented Modularization
Computer System Life Cycle Phase Support

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

The objective of this three-phase effort is to (1) Identify the hardware/software technology pertinent to the implementation of tightly-coupled embedded distributed systems for DoD applications, (2) Establish an integrated approach regarding the total life-cycle software development period with correlation as to the applicability of existing/near-term software engineering methodology, techniques and tools to each life-cycle phase, and (3) Define the functional design requirements pertinent to the far-term development of needed software engineering methodology, techniques and tools. A product of this effort is the recommended design of a system support environment encompassing the integrated implementation of candidate software engineering tools.

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF FIGURES

LIST OF FIGURES

LIST OF TABLES

1.0 TECHNICAL REPORT SUMMARY

General Dynamics Data Systems Division is under contract to Rome Air Development Center to conduct a study entitled Distributed Processing Tools Definition. The objectives of this study are to investigate the requirements for software lifecycle support of embedded distributed processing systems and then to specify tools and techniques pertinent to each lifecycle phase. This study is divided into three phases as illustrated in Figure 1-1. Phase I of the study has been completed, and its results are described in this Technical Report.

Two generic classifications of military systems are addressed in this study: weapon systems (including armament, aeronautical, and missile and space configurations) and communication systems (including command/control/communication and mission/force management type configurations). This classification is based upon characteristics inherent to each group (ref. Appendix A) and permits specification of requirements for software tools and techniques for a larger class of generic military systems.

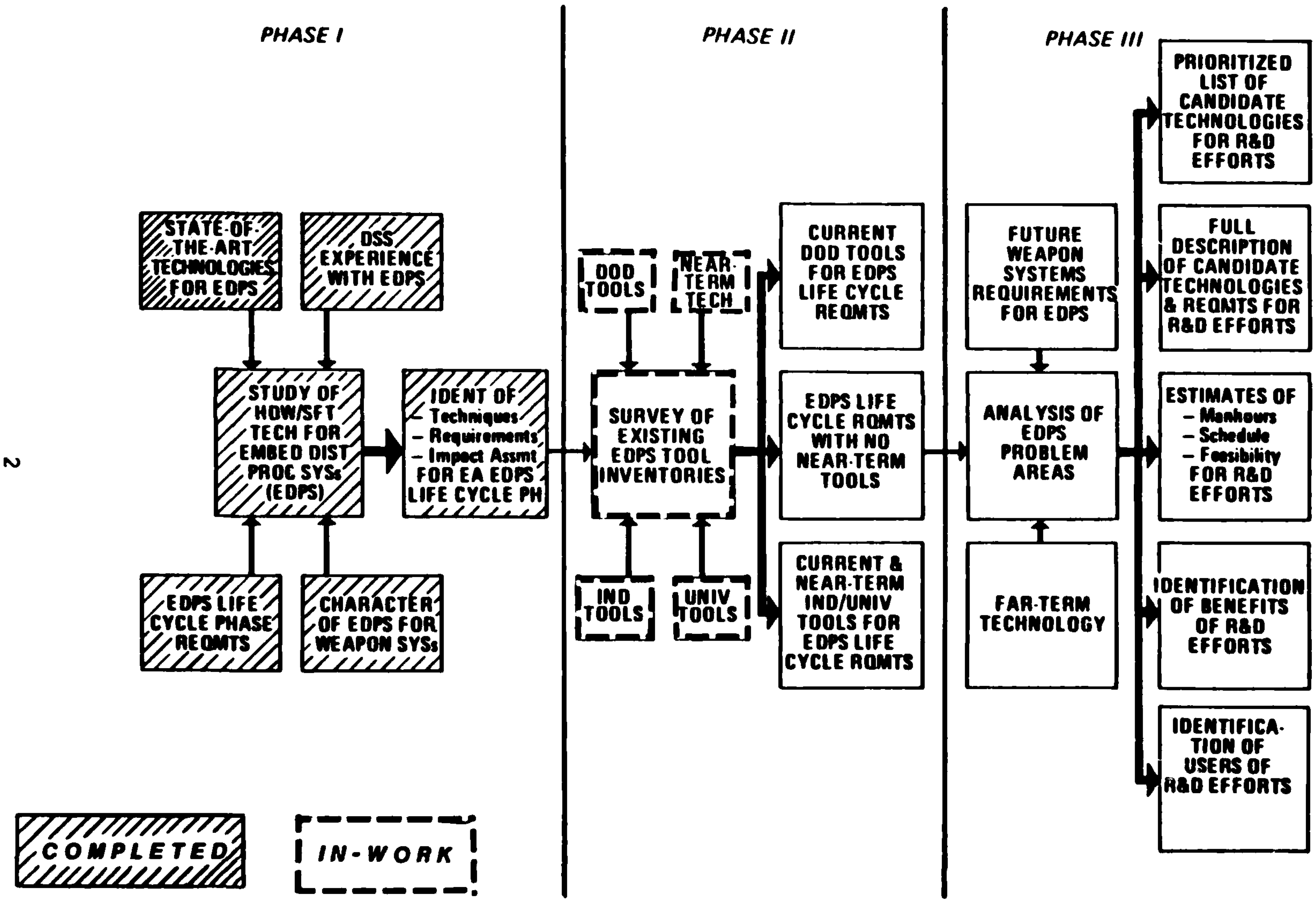

PHASE I
PHASE II
PHASE III
STATE-OF-THE-ART TECHNOLOGIES FOR EDPS
DSS EXPERIENCE WITH EDPS
STUDY OF HOW/SFT TECH FOR EMBED DIST PROC SYSs (EDPS)
EDPS LIFE CYCLE PHASE REQMTS
CHARACTER OF EDPS FOR WEAPON SYSs
IDENT OF – Techniques – Requirements – Impact Assmt FOR EA EDPS LIFE CYCLE PH
DOD TOOLS
NEAR-TERM TECH
SURVEY OF EXISTING EDPS TOOL INVENTORIES
IND TOOLS
UNIV TOOLS
CURRENT DOD TOOLS FOR EDPS LIFE CYCLE REQMTS
EDPS LIFE CYCLE RQMTS WITH NO NEAR-TERM TOOLS
CURRENT & NEAR-TERM IND/UNIV TOOLS FOR EDPS LIFE CYCLE RQMTS
FUTURE WEAPON SYSTEMS REQUIREMENTS FOR EDPS
ANALYSIS OF EDPS PROBLEM AREAS
FAR-TERM TECHNOLOGY
PRIORITIZED LIST OF CANDIDATE TECHNOLOGIES FOR R&D EFFORTS
FULL DESCRIPTION OF CANDIDATE TECHNOLOGIES & REQMTS FOR R&D EFFORTS
ESTIMATES OF – Manhours – Schedule – Feasibility FOR R&D EFFORTS
IDENTIFICATION OF BENEFITS OF R&D EFFORTS
IDENTIFICATION OF USERS OF R&D EFFORTS
COMPLETED
IN-WORK

Figure 1-1 Overview of the Distributed Processing Tools Definition Study

The principal technical conclusion of the Phase I study is the requirement that software lifecycle phase support tools for embedded distributed processing systems must view the hardware and software from a total systems perspective. This perspective is accomplished by using the Concept of Object-Oriented Modularization (ref. Paragraph 6.0). Particular manifestations of the Concept of Object-Oriented Modularization applied to embedded distributed processing systems yield the following additional conclusions:

(1) Software tools must be small and integrated with larger environments; for example, the Ada Programming Support Environment or an Integrated Software Support Environment as proposed in the DOD Candidate R&D Thrusts for the Software Technology Initiative. In addition, these tools must be available at the beginning of the software development program (ref. Paragraphs 5.2.1, 5.2.2, and 5.2.7).

(2) Static analysis techniques may not be useful because of the probabilistic nature of distributed systems. This conclusion is corroborated by recent work of Dr. W. E. Howden (ref. Paragraphs 7.1.4 and 7.3.1).

(3) Rapid prototyping and impact analysis tools are critically needed (ref. Paragraphs 7.1.3 and 7.3.3).

(4) The real-time rendezvous and package set capabilities of the Ada language need further analysis in a distributed processing environment (ref. Paragraph 7.1.2 and Appendix B).

(5) Simulation of distributed hardware interconnections and networking must be done before the system is built because performance is critically dependent on these features (ref. Paragraphs 5.1.2 and 5.1.3).

In addition, the anticipated effects of static analysis, branch testing, and impact analysis techniques upon the lifecycle phases of embedded distributed processing systems are evaluated (rεf. Paragraph 7.3). Finally, global timing and methods of distributed data base management have been identified as topics worthy of future research efforts.

Software lifecycle phase support tools are also required for communication systems (i.e., command/control/communication and mission/force management configurations). In such applications distributed processing has increased the importance of Object-Oriented Modularization (ref. Paragraph 6.0). Specific manifestations of such modularization are evident in the following conclusions concerning communication systems:

(1) Data distribution can vary throughout different levels of a multi-layered network. Object-Oriented Modularizations require alternative layering be examined (ref. Paragraphs 6.1 and 7.2.2).

(2) The performance behavior of multi-layered networks with distributed intelligence must be established before such networks are implemented (ref. Paragraphs 7.1 and 7.1.2).

(3) The capability to simulate a multi-layered network must become part of the Integrated Software Support Environment. (ref. Paragraphs 5.2.7 and 7.1.3).

Decision making concerning embedded distributed processing systems as well as communications systems requires knowledge of data movement and computational efficiency. Enough information must be known within specific real-time constraints to reach appropriate decisions. Subsequent functionality is a product of such decisions. Distributed processing functions are performed upon data bases residing within interconnected architectures. Growth in these data bases can force changes in their tightly-coupled architectures to accommodate increased information flow. The bottom line is functionality for both weapon systems and communication systems. The systems themselves can be viewed as interconnected hardware components but their functionality is a result of software. Using analytical techniques, such software can be validated and optimized. Without analytical techniques, the multiple processor operating system will allocate its resources. Under such an approach validation and optimization issues would not be addressed before the start of a weapon or communication system's operational software. Higher reliability at reduced costs and shorter development schedules would not be produced. Instead the emergence of a "fix it later" attitude would ensue. When real-time performance of both hardware and software is not considered at the outset, they will force subsequent consideration when problems inevitably occur. Proper analytical techniques anticipate and eliminate whole groups of validation and optimization problems. They also establish what performance can reasonably be expected from specific multiple processor configurations. Taken as a whole, these techniques become part of a support environment. The functionality of such a support environment can best be described as a configuration testbed. Although the issues addressed by an embedded system are diff-erent from a communications system, a single configuration testbed can accommodate both. In conclusion the need for testing command/control/communication systems and embedded distributed systems can be addressed by a single testbed built upon a programming support environment, e.g., the Ada Programming Support Environment. Such an effort is worthy of future research initiatives.

Phase II of the Distributed Processing Tools Definition Study will research industrial, university and Department of Defense tools that satisfy the requirements established in Phase I. Finally, Phase III will identify and describe candidate research and development efforts to solve any Phase I requirements not supported by a Phase II tool or technique.

2.0 SCOPE AND PURPOSE OF THIS DOCUMENT

Projected military requirements for improvements in performance, reliability, and field maintainability of weapon and communication systems, indicate an increased utilization of embedded distributed processing technology. Software tools and techniques need to be identified and developed that support the already existing and future embedded distributed processing hardware. Consequently, General Dynamics Data Systems Division and Rome Air Development Center have contracted to conduct a study of software tools and techniques pertinent to embedded distributed processing systems (EDPS). This effort is entitled Distributed Processing Tools Definition (DPTD) study, and it is being conducted in three phases, namely:

Phase I -	Study of Hardware and Software Technologies
Phase II -	Survey of Existing Tools and Techniques
Phase III -	Analysis of Problem Areas and Recommendation of Candidates for Research and Development Efforts.

The objectives of the DPTD study are to investigate the requirements for software lifecycle support of embedded distributed processing systems and then to identify and specify tools and techniques pertinent to each lifecycle phase. The availability of specifications for software tools and techniques to support embedded distributed processing systems will enable the tools to be developed prior to the start of the weapon or communication system operational software. This prior availability will result in higher reliability of the operational software at reduced costs and shorter development schedules.

The purpose of this document is to present the results of Phase I of the DPTD study in accordance with the requirements of paragraph 4.1.1 of the Statement of Work. The results of Phases II and III of the DPTD study will be concatenated to this Phase 1 report. The benefits of this approach are a smooth transition of the description of the results of the three phases for the reader and the opportunity to update previous sections as newer technologies are identified and deeper insights are gained into their impacts upon embedded distributed processing systems.

3.0 INTRODUCTION AND BACKGROUND

The gap between literature and practices is a chasm when viewed from the standpoint of distributed systems. While researchers advocate symbolic execution or proofs of correctness or even automated verification systems, currently operating distributed systems require hundreds of thousands of lines of computer code to be maintained. Consequently, the overriding commitment of the present is to maintain what has already been implemented. In most instances these systems have been developed without the benefit of automated verification. Static analysis has been limited to diagnostics produced by a typical compiler. Consistency checking and documentation about the definition, reference, and communication of data within a program has not been addressed. As processing is distributed over several nodes, dynamics become more important. Analysis of a program can be performed at the statement-level or through the examination of global execution. Most currently operating distributed systems have not been analyzed from a dynamic standpoint. These current systems accommodate a wide variety of hardware architectures and an even wider variety of software algorithms and data structures. The maintenance of these systems has grown too complicated for any one person to comprehend. However, their complexity continues to grow and their applications continue to expand. New techniques to cope with their increased complexity are beginning to emerge. Foremost among these techniques is the process of object-oriented modularization.

When implementing object-oriented modularization techniques individuals do not need to understand all modules within a distributed system. Although the idea is simple, its realization is often difficult. Obviously someone must know how all the modules of a system fit together. Furthermore, the operations in one module should not rely upon operations within another module unless very carefully controlled. These operationally independent modules characterize the object-oriented approach and distinguish it from the more conventional modularization techniques. The individual who performs the object-oriented modularization does not have to know the operational details in each module. Those details are decided by individuals who implement the specific modules.

Several factors contribute to the difficulty encountered during modularization. First, the modules in software are produced completely independent of the modules in hardware. The subsequent operational mix is sometimes mismatched. Hardware modularization takes place on a process basis. Each process is a module of concurrency with programs being constructed from one or more processes. If several processes are involved, communication between them becomes increasingly important. Such considerations originated within multiprogramming environments where each job could be viewed as a process. From multiprogramming the present methodology for implementing concurrency has evolved. As new hardware instruction sets are envisioned, they are implemented in terms of concurrency architectures. An example would be the MIL-STD-1750A Instruction Set Architecture implemented by RCA and TRACOR for Wright-Patterson Air Force Base. The CPU is subdivided into two pairs of General Processor Units (GPUs) and operates on an 8-bit slice architecture. The microcode is partitioned between the two GPUs and has no duplication. These partitions are accessed as needed and operate as concurrency modules. The instruction set of MIL-STD-1750A is accommodated through access to 16 different partitions or concurrency modules. To optimize a process requires knowledge of its access sequence. Of course such an approach emphasizes the importance of hardware modularization. Once the operating system has been written the emphasis on hardware modularization is downplayed. The importance of software and its associated algorithms and data structures is elevated. Object-oriented modularization is applied producing software with operationally independent modules. When these modules run concurrently, they can be viewed as concurrency modules within their respective hardware environments. Each such module exhibits its own efficiency with respect to access sequences on the hardware level. In most instances those access sequences are closely associated with the operating system. To bridge that gap between object-oriented modularization and hardware modularization requires a precise software tool. In the case of the MIL-STD-1750A implementation that tool was a derivative of SPICE which was developed at Carnegie-Mellon University. It is a message-based system containing two kinds of entities: messages (a collection of types and data) and ports (queues of messages). It does not emulate the object-oriented modularization but specializes in concurrency modules. Consequently, the implementation of the MIL-STD-1750A Instruction Set Architecture by RCA and TRACOR illustrates how modules in hardware are developed independently of modules in software.

Another difficulty encountered during modularization originates with scheduling. Production schedules for distributed systems are usually controlled by external factors. Such systems are usually components within larger systems; i.e., they are embedded. The schedules imposed by the larger systems take precedence over the embedded distributed processing systems. Despite this scheduling precedence, the need for object-oriented modularization persists.

Also, unless the interface between software modules is completely established, the subsequent software effort may be unclear.

Lack of clarity extracts a price later in the maintenance of the software system. Of course, clarity has its price too. More effort must be expended during the requirements study and preliminary design phase of the software development lifecycle. Such additional effort is a one time cost and it reduces the maintenance effort which is an ongoing cost. Consequently, unclear interfaces between software modules perpetuate a maintenance effort. The more complicated those interfaces, the larger the effort. Much can be avoided through an object-oriented modularization when software systems are initiated.

As distributed systems become more complex, a consensus concerning modularization has formed. It applies the principle of "information hiding" to distributed software procedures and data structures. Its basic idea is to group related procedures and their data structures together. The subsequent groupings comprise modules which can be viewed as either software or hardware; i.e., object-oriented modules or concurrency modules. Assuming the hardware stance is taken, related procedures refer to operational state transitions. Since computers are state-driven devices, their hardware transitions transform their internal operating states. Concurrency modules transform internal operating states in parallel and independently. Since such modules operate concurrently, the access problem among data abstractions is highlighted. Extra precautions must be taken to manage accesses to the same data by multiple concurrency modules. The tools to regulate access are in their infancy. In effect, the ability to produce concurrency modules has outstripped the present capability of simulation tools. Complicating the situation are software procedures which transform operating states by way of algorithms. Since these algorithms are usually expressed in higher order language statements, the operating transformations are more obscure. The subsequent tools available for access control are based upon logic and are divorced from considerations of hardware. Their analysis is static while the operational situation is concurrently dynamic. The problem has now been assigned the subsequent implementation of a higher order language, e.g., Ada. Whether the language can solve the problem has not been addressed because other issues must be resolved first. Until that language is implemented, the problem has simply been deferred. Despite the delay the access problem between concurrency modules remains significant.

Several programming languages are attempting to support "information hiding" modularization techniques. The latest such language is Ada which has been syntactically specified by the Department of Defense. Its implementation is underway but no validated compiler has yet been produced. Consequently, observations concerning its algorithms and data structures are constrained to its present syntactic definition. To make any observation assumes a future compiler will meet the syntactic specifications and will survive validation. Obviously the current definition of Ada is far removed from the actual state transitions within the concurrency modules of a computer. In instances of concurrency, Ada relies upon the rendezvous technique.

Under that technique, the fastest concurrent operation is completed and waits upon the slowest concurrent operation to complete. Consequently, all concurrent operations complete before processing is resumed. The programming location at which these concurrent operations await completion is called the rendezvous point. Actual wait time within a specific rendezvous depends upon the specific mix of concurrent operations. From the standpoint of an Ada compiler, these wait times are beyond the scope of object-oriented modularization. Such an observation is invalid from the standpoint of hardware concurrency modules. Unless such times are carefully delineated, the rendezvous becomes non-deterministic. In this document the non-deterministic concept will refer to a process whose outcome depends upon the choices and transitions made by the system components. This is in contrast to a deterministic process whose outcome depends only on the current system state. Such non-determinism is not acceptable within most military applications. The performance limitations of embedded distributed processing systems in most military applications must be known before implementation. Otherwise limits could be exceeded under catastrophic circumstances; e.g., battlefield conditions. Since Ada syntactic constructs accommodate non-deterministic situations, a problem is created for subsequent Ada Programming Support Environments (APSEs). The expectation that such APSEs will solve all instances of concurrency modularization is unrealistic. The state transitions available within hardware of the present and immediate future should be accessible from higher order languages like Ada. Otherwise the state transitions will be manipulated at the machine language level.

Obviously Very-Large-Scale-Integration (VLSI) devices capable of concurrent operations are impacting the marketplace. As more of these devices meet or exceed military specifications, the capability for embedded distributed processing systems increases. Beyond the present VLSI devices are the Very-High-Speed-Integrated-Circuit (VHSIC) technologies. Such technologies have as their objective the insertion of speed into defense systems. To achieve this objective, certain barriers must be overcome and subsystems must be built to demonstrate improved capability. Consequently, a VHSIC program has been created to address these barriers on a tri-service basis under the Under Secretary of Defense for Research and Engineering. In addition to the Air Force, Army, and Navy, the program involves the industrial and scientific communities. It has been subdivided into four phases: Phases 0, I, II, and III. While Phases 0, I, and II must operate consecutively, Phase III can operate concurrently. Phase 0 is the Study Phase which defines the work and generates a detailed approach. Phase I is subdivided into two efforts: one to implement electronic brassboards within three years and another to extend Integrated Circuit technology into submicron dimensions. Phase II is also subdivided into two parallel efforts: one provides subsystem demonstrations of the brassboards produced under Phase I and the other continues the submicron work begun under Phase I. Phase III addresses near-term efforts in key technologies which impact the total program. This particular phase is intended to encourage participation by universities and small businesses in very specific problems, e.g., advanced architecture and design concepts. The capability for modularization within VHSIC devices exceeds the current VLSI marketplace. The impact of VHSIC technologies on software languages is not completely understood. Several VHSIC Phase III contracts are addressing that issue. Results have not yet been generated. The non-deterministic nature of the present Ada rendezvous techniques is a crucial issue in VHSIC technologies. The reason rests with the capability of VHSIC hardware environments to provide ever-increasing concurrency modularization.

Furthermore, software produced by "information hiding" must reside in these new VHSIC environments. The near-term and far-term capability for concurrent operation will be enhanced significantly.

Appropriately defined object-oriented modules and concurrency modules proceed independently through their individual lifecycle phases. The reconciliation of these software lifecycle phases to hardware lifecycle phases is of paramount importance to "information hiding" modules. VLSI technologies are putting larger numbers of electronically active devices in hardware modules. As these numbers increase, the operational verification of each electronically active device becomes more difficult. The test phase of such high density VLSI hardware now depends upon statistical sampling techniques. When such hardware is released to the marketplace, its subsequent operation is non-deterministic or probabilistic. The Boolean logic accommodated by such hardware has seldom been completely tested before delivery to the customers. The future deliveries of VHSIC devices will only exacerbate the situation. The problem of marginally functional hardware modules has become the hidden problem of embedded distributed processing systems. Many compromises must be made before such equipment becomes functional. In most instances hardware deficiencies are subsumed by the software process itself. This creates a double bind for the software modularization effort. As software modules are designed and implemented, they may not run because of previously undetected hardware errors. If a software module does not run under such circumstances, is the problem attributable to hardware? Alternatively, the problem could lie with software. Furthermore, as a worst case, the problem could lie with both hardware and software. The end result of this situation is that the software development lifecycle phases are becoming the test phase for hardware modularization. Any discussion of software lifecycle phases must address the hardware testing issue. To ignore such an issue only compounds the problem faced by software modularization and development. An inventory of software tools must be accumulated to test the actual operation of hardware modules. Such a toolset serves the purpose of hardware quality assurance. Its objective would be the elimination of hardware problems within the subsequent software lifecycle phases. If the objective is achieved, software development costs should be reduced. Whether the reduction is temporary or permanent has not been established.

Assuming the quality of hardware modules can be assured, the subsequent software lifecycle phases need construction tools for separate software modules. Each tool should be associated with one or more lifecycle phases. Taken collectively all tools would cover the lifecycle spectrum: requirements tools, design tools, coding tools, testing tools, documentation tools, and maintenance tools. A clever design for such a toolkit would use "information hiding" concepts. Such concepts accommodate the trend toward increasing hardware modularization. Shrinking hardware modules present shrinking targets to the software development toolkits. These hardware modules are often overlooked by developmental toolkits. The present situation is summarized by the following observation. Tools for software lifecycle phases in distributed systems should operate within the distributed systems themselves. To accomplish such operation requires lifecycle tools to become smaller and more specialized. Such a trend runs counter to the current tool marketplace.

Currently available tools address a general systems orientation and combine several lifecycle phases together. The immediate result of such an orientation is to place tools in large uniprocessor configurations. The computer talent in such computer settings seldom appreciates the real-time problems of the distributed system. Ada suffers from the same orientation. Although capable of targeting small hardware modules, it must reside in an excessively large computer environment. In this large computer environment the modularization in the uniprocessor configuration itself is seldom evident. Unless the computer talent exerts extraordinary effort to discover how the large uniprocessor operates, the software tools it produces is not likely to exhibit such understanding.

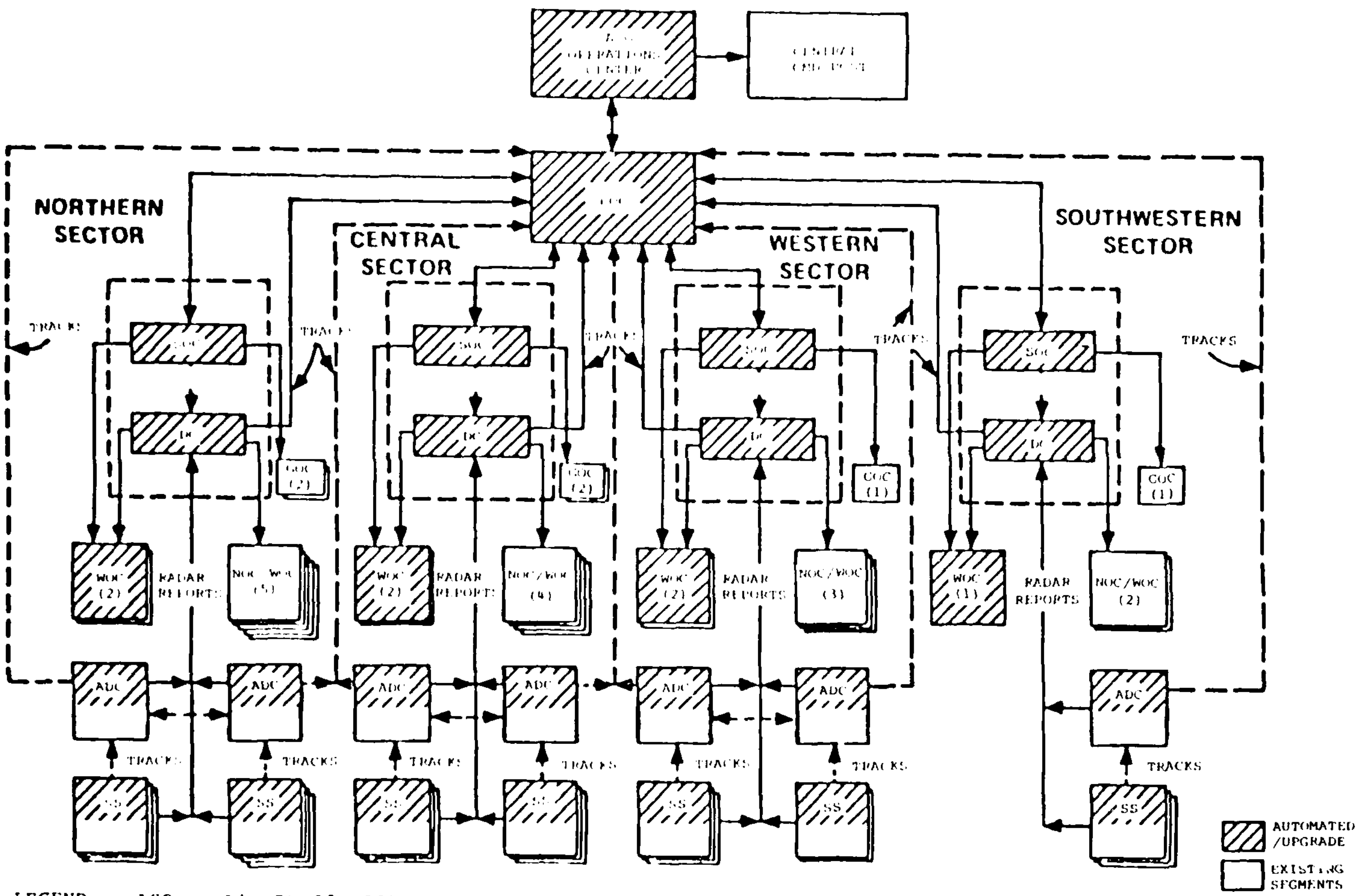
Smaller and more specialized lifecycle tools can be combined into larger sets to accommodate a general systems orientation. The combination capability is provided by the process of software modularization. Current tools can be converted to these smaller and more specialized formats through additional analysis. Their components with respect to lifecycle phases have not been analyzed in sufficient detail. Upon completion of such an analysis, the resulting tools would be smaller and more specialized with respect to hardware modules. The optimal environment for these new toolkits is the emerging Ada Programming Support Environment (APSE). The intent of Ada has never been to reside in a distributed system but eventually it must. The current proponents of Ada subsets exhibit a distributed system orientation. As currently defined, Ada allows no subsetting. However, such constraints apply to the Dept. of Defense and the U.S. defense industry. The policy of U.S. allies is not as clearly delineated, e.g., Japan and France. To complicate matters Russia is even translating the Ada syntax. If an unauthorized subset of Ada succeeds quickly and impacts the marketplace, the continued insistence against subsetting may itself be called into question. In effect, the complete syntactic definition of Ada requires a very large uniprocessor to accommodate it. Subsets of the syntactic definition can easily be accommodated within the distributed environment itself; e.g., the Telesoft configuration. Systems are definitely becoming more distributed which means the argument for a distributed Ada is growing. Since Ada can be made sufficiently modular, it can eventually fit into a distributed system. Sufficiently modularized toolkits can also fit into distributed systems. Consequently, both Ada and a toolkit of highly specialized tools become part of an integrated software support environment. Obviously how well Ada accommodates itself to underlying hardware state transitions is of paramount importance. The future plans of the Department of Defense software efforts assume the availability of this integrated software support environment.

Subsequent sections will emphasize the importance of such an environment.

4.0 CHARACTERISTICS OF DISTRIBUTED PROCESSING SYSTEMS

The past several years have seen an increasing interest in the application of distributed computing systems, because a distributed processing system is one in which the computing functions are dispersed among several physical computing elements. These computing elements may be colocated or geographically separated. The distributed computing systems take many forms covering a diverse range of system architectures. In fact, the very term distributed processing may invoke radically different images of technology and problem solutions depending upon the user. To some, a distributed processing system is a collection of multiple computers or processing elements working closely together in the solution of a single problem. An example might be an Air Defense/Command Management system which is comprised of many data processing subsystems linked together by shared memory, communication lines/networks, or common buses. Each data processing center processes a subset of air/ground situation transactions and updates a portion of a common data base to develop a dynamic composite-air-situation picture against which force management can be exercised. Users of such systems are concerned with issues of hardware and software design, reliability, operating/executive systems, and how to optimally decompose programs and data bases. Figures 4-1 through 4-6 illustrate this type of modern distributed processing system in a generalized manner. Figure 4-1 shows the overall Air Defense Ground Environment (ADGE) System with Figures 4-2 through 4-6 showing the generic data processing elements in increasing detail. Since distribution of control is a key characteristic element of distributed processing systems, Figure 4-3 was included to highlight this characteristic.

To other users, a distributed system is a set of intelligent terminals located at the point of use to give local organizational elements more responsive computer support. These terminals perform most of the computing functions for the local group. When necessary the terminals communicate with remote host computers and each other for enhanced support. An example might be the remote Data Entry Display Stations (DEDS) shown in Figure 4-2 which could be located at weather control centers. To yet another set of users (e.g., aircraft pilots), distributed processing systems may mean a very tightly-coupled distributed system used for navigation, weapon delivery, or control of an aircraft.
Figure 4-7 illustrates this type of distributed system by showing the architecture of data processing systems



```
LEGEND:  ASO  - Air Staff Office            DC  - Direction Center          ADC  - Alternate DC
         COC  - Combat Operation Center     WOC - Wing Operation Center     SS   - Surveillance Station
         SOC  - Sector Operation Center     NOC - Nike Operation Center     ----   Mode II Data Flow
         GOC  - Group Operation Center
```

CENTRALIZED TRACKING – CENTRALIZED WEAPON CONTROL

Figure 4-1 General ADGE System

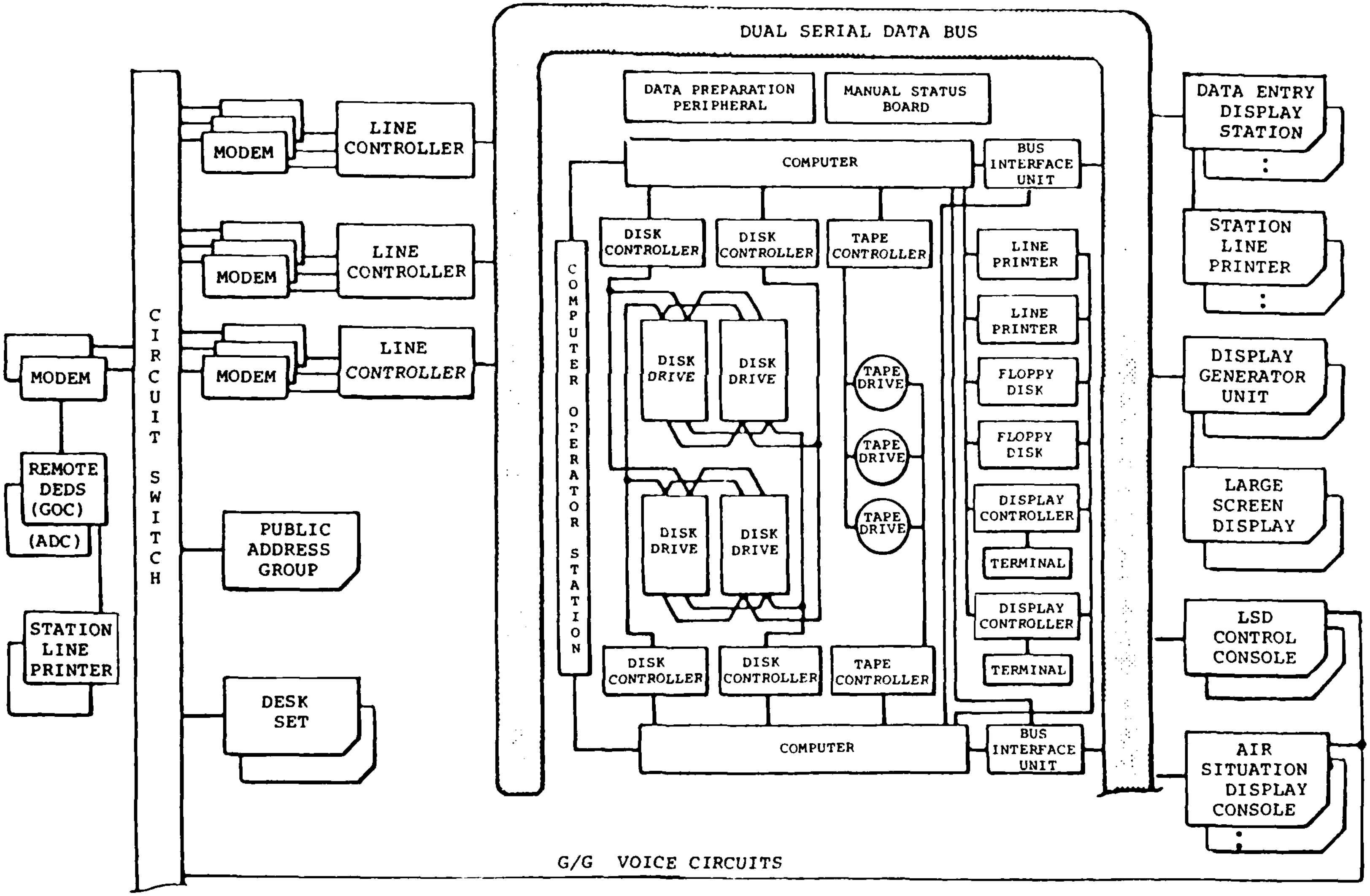


Figure 4-2 Generic Data Processing System for the Operation/Direction Center(s)

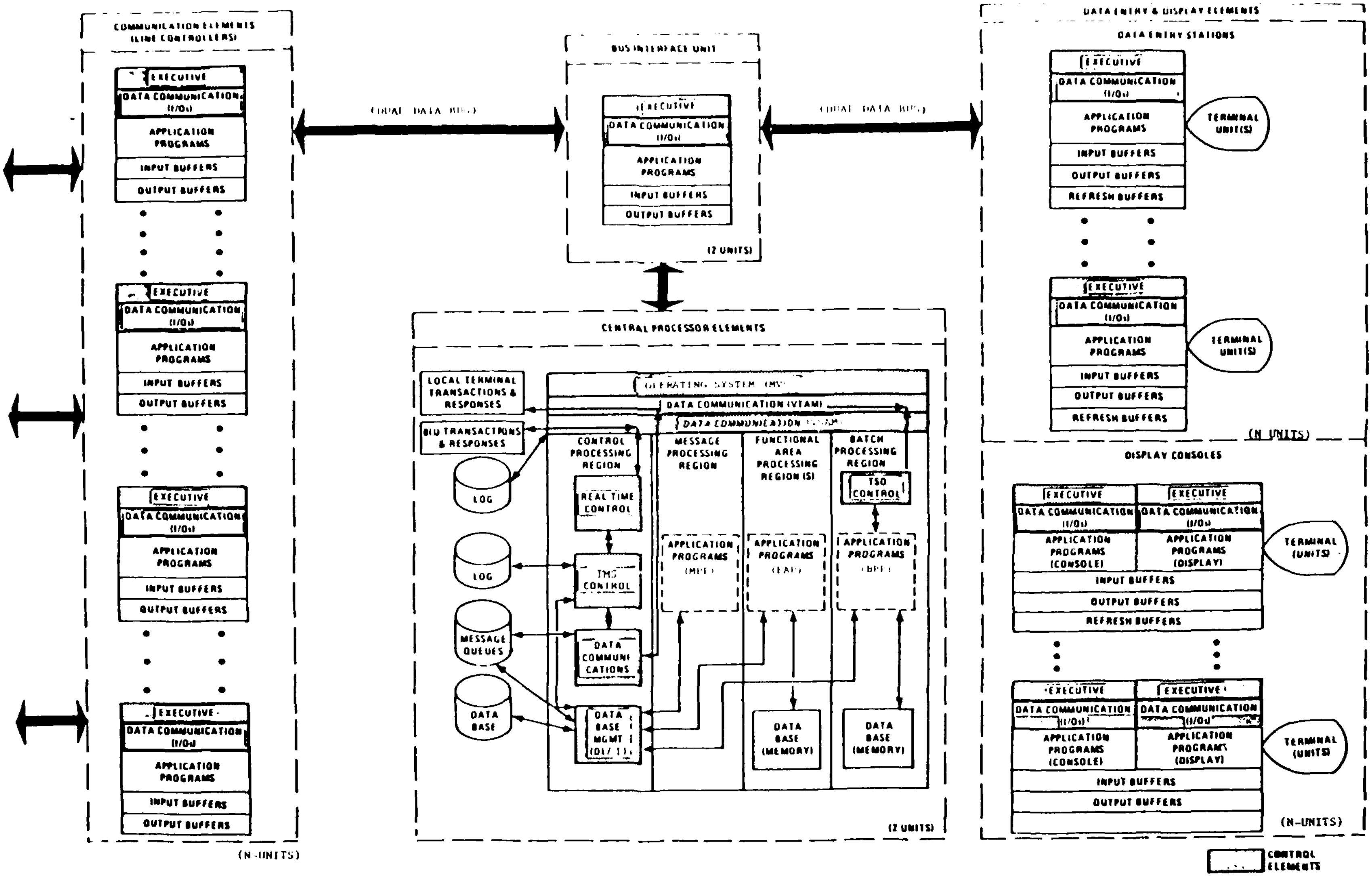

COMMUNICATION ELEMENTS (LINE CONTROLLERS)
EXECUTIVE
DATA COMMUNICATION (I/Os)
APPLICATION PROGRAMS
INPUT BUFFERS
OUTPUT BUFFERS
(N-UNITS)
(DUAL DATA BUS)
BUS INTERFACE UNIT
(2 UNITS)
CENTRAL PROCESSOR ELEMENTS
LOCAL TERMINAL TRANSACTIONS & RESPONSES
BIU TRANSACTIONS & RESPONSES
OPERATING SYSTEM (MV)
DATA COMMUNICATION (VTAM)
DATA COMMUNICATION
CONTROL PROCESSING REGION
MESSAGE PROCESSING REGION
FUNCTIONAL AREA PROCESSING REGION (S)
BATCH PROCESSING REGION
TSO CONTROL
REAL TIME CONTROL
TMS CONTROL
DATA COMMUNI CATIONS
DATA BASE MGMT (DL/1)
APPLICATION PROGRAMS (MPP)
APPLICATION PROGRAMS (FAP)
APPLICATION PROGRAMS (BPP)
DATA BASE (MEMORY)
LOG
MESSAGE QUEUES
DATA BASE
DATA ENTRY & DISPLAY ELEMENTS
DATA ENTRY STATIONS
REFRESH BUFFERS
TERMINAL UNIT(S)
(N UNITS)
DISPLAY CONSOLES
APPLICATION PROGRAMS (CONSOLE)
APPLICATION PROGRAMS (DISPLAY)
TERMINAL (UNITS)
CONTROL ELEMENTS

Figure 4-3 Distribution of Processing Control for the Operation/Direction Center(s)

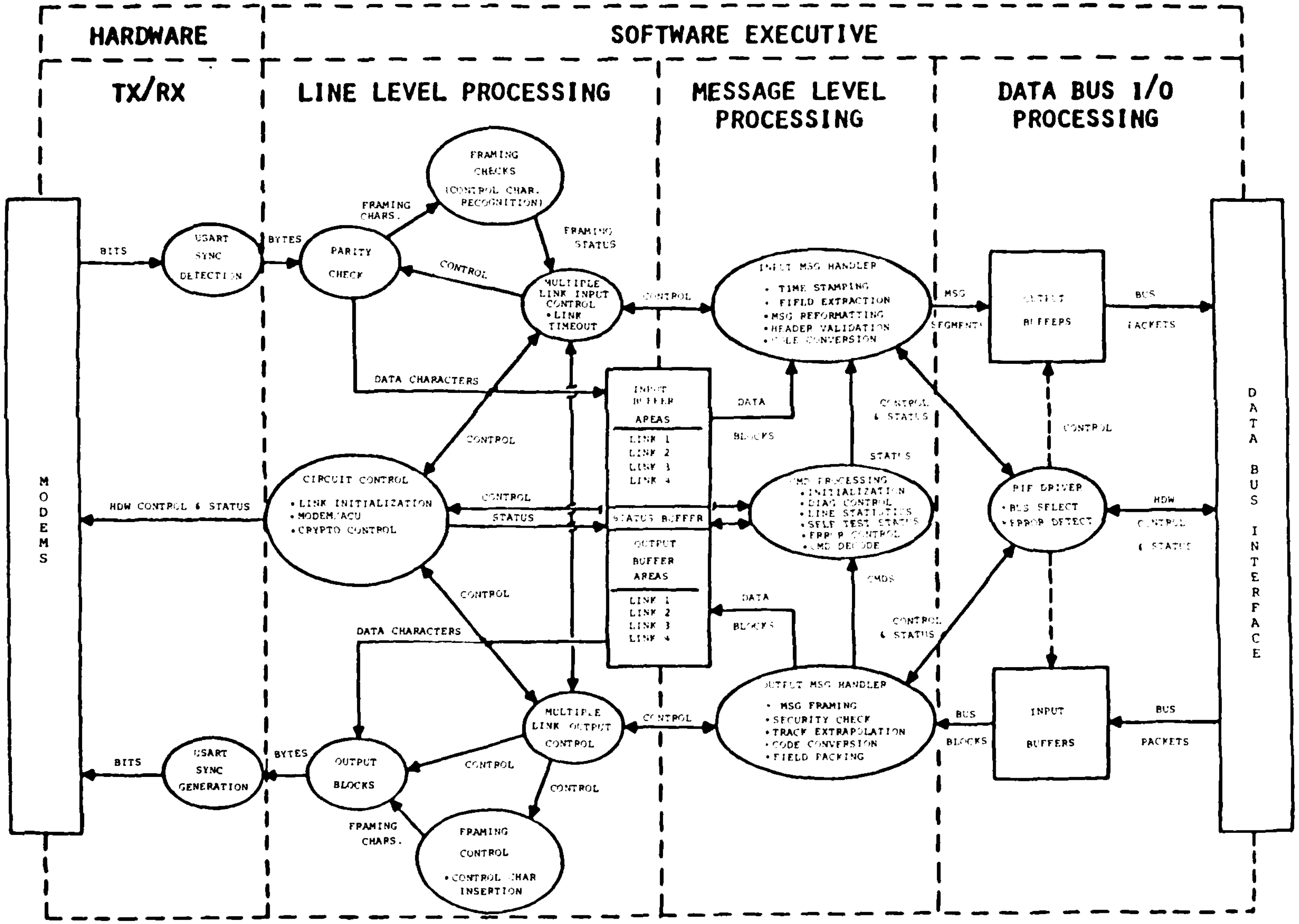


Figure 4-4   Generalized Line Controller Distributed Proцessing

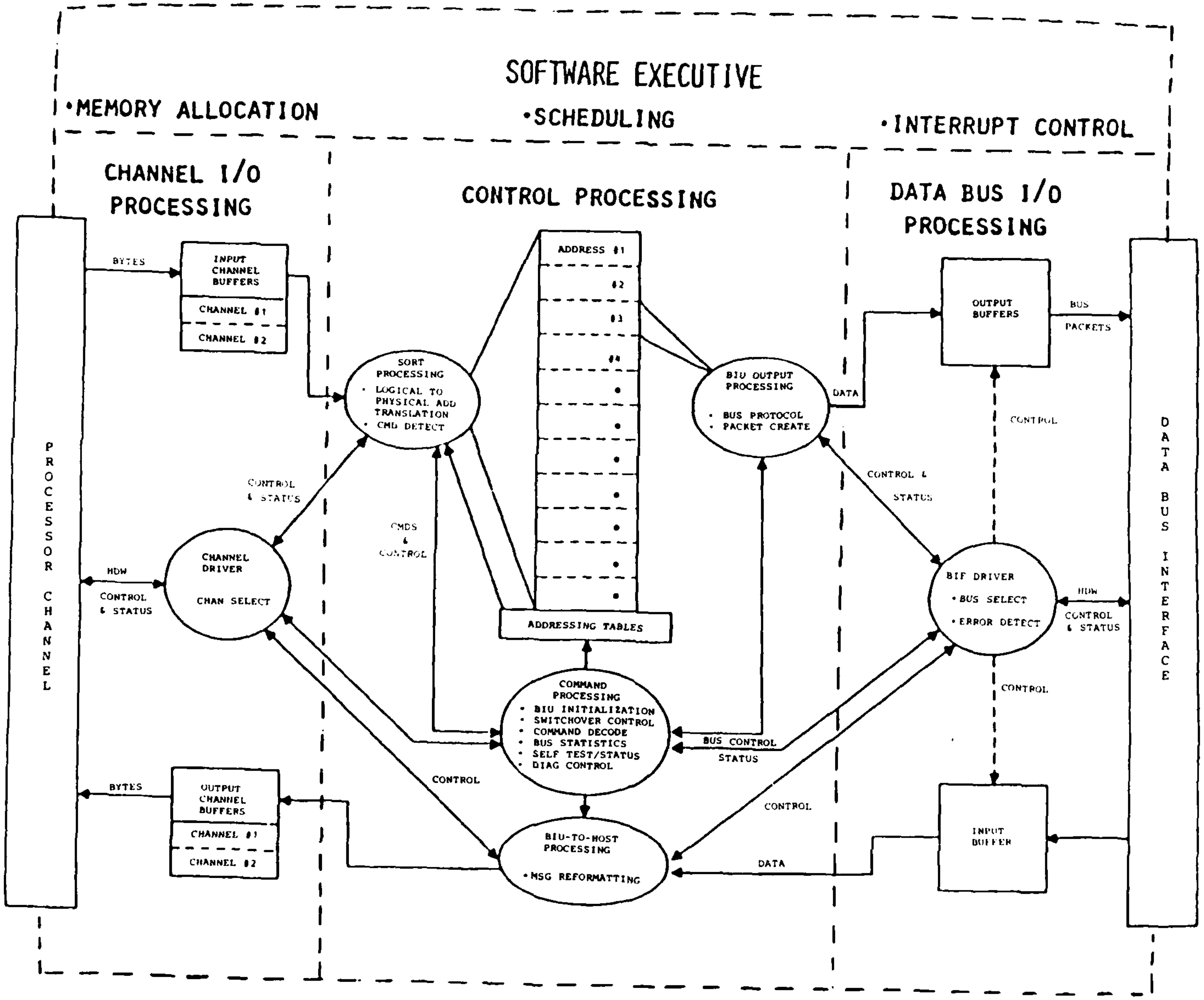


Figure 4-5 Generalized Bus Interface Unit Distributed Processing

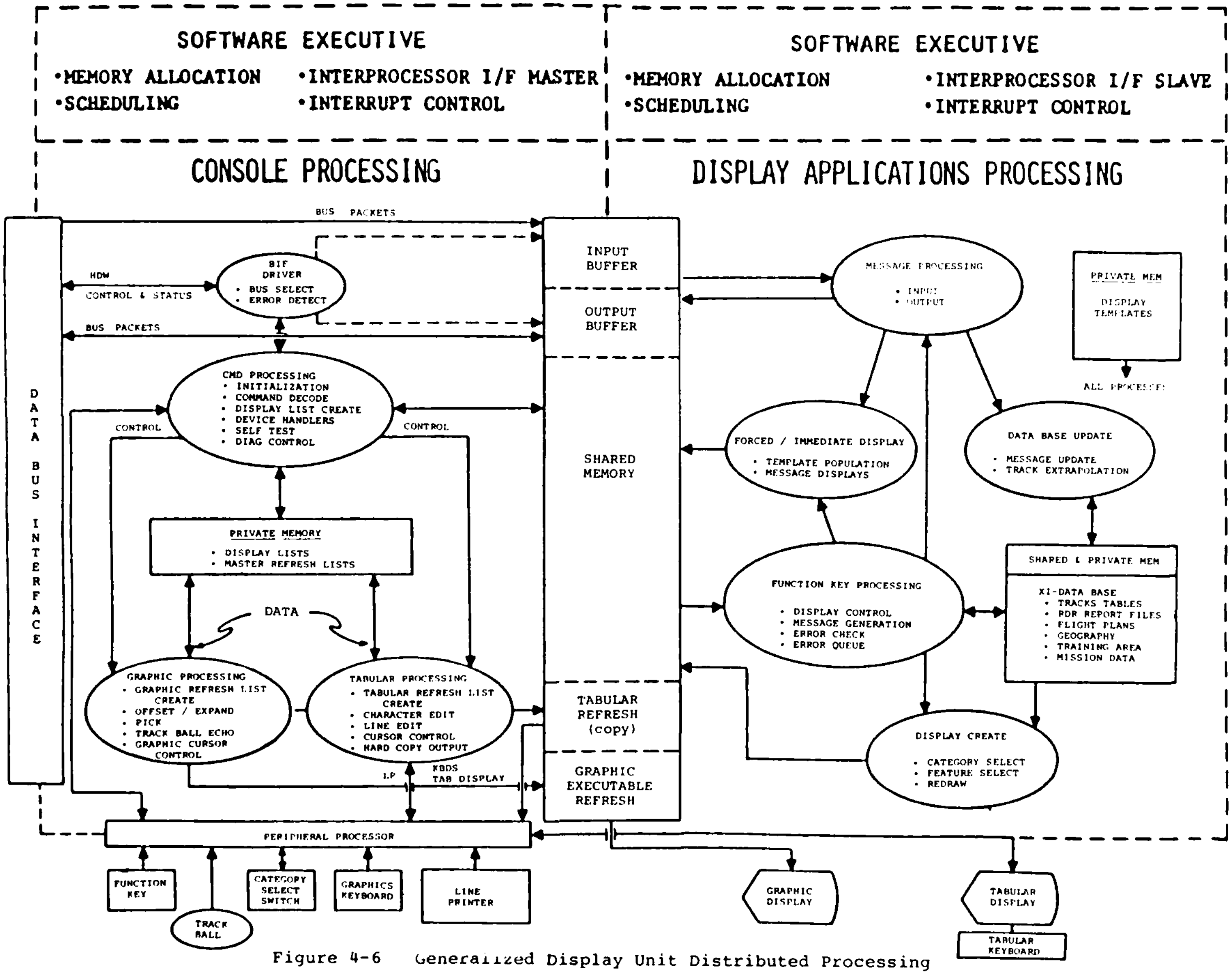


Figure 4-6 Generalized Display Unit Distributed Processing

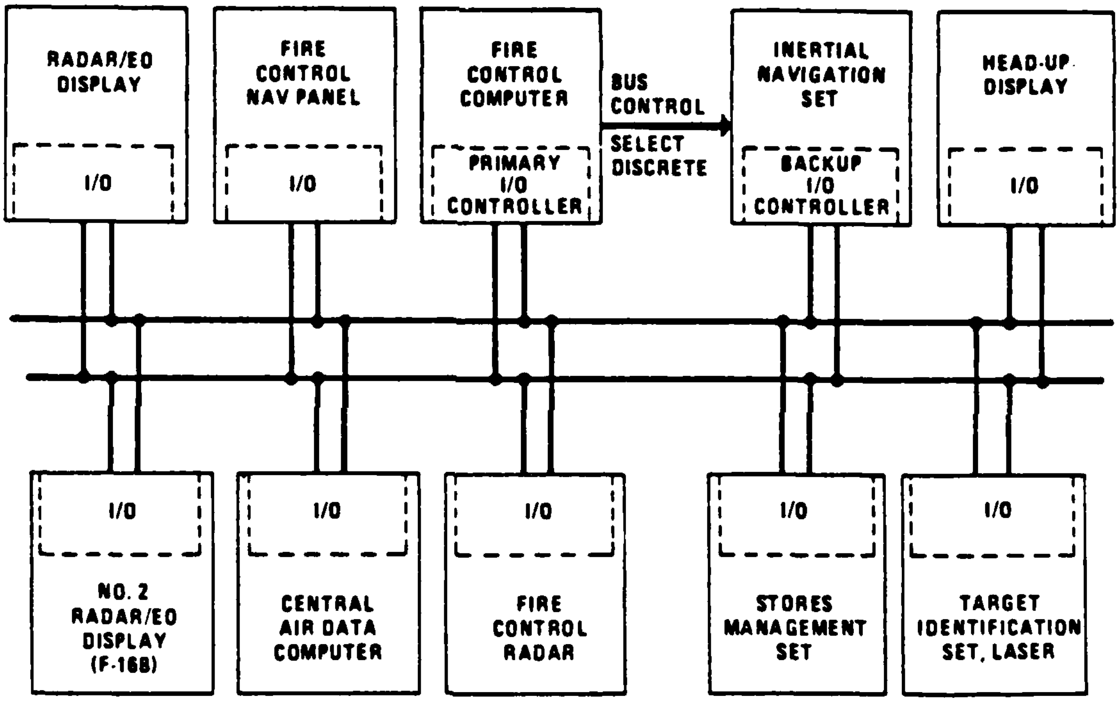


Figure 4-7 Distributed Data Processing Architecture for F-16 Aircraft

with MIL-STD-1553B Multiple Bus for the F-16 aircraft. The fire control computer is the systems integrator for this F-16 avionics and armament system. In this integration role, the fire control computer uses inputs from computer-controlled on-board sensor systems (e.g., radar, navigation, central air data computer, target identification set, etc.) to accomplish air-to-air and air-to-ground weapon delivery, navigation, fuel management, and stores management and control. Results of pertinent calculations are displayed on the radar E-O displays, NAV panels, and various other cockpit displays that are human-engineered for single-pilot operation. The F-16 system is unique in that it implements the standard military bus (MIL-STD-1553) with the latest version of this bus (MIL-STD-1553B) including additional subaddress modes, broadcast capability, improved noise rejection, and error-rate specifications. Use of this bus allows for distribution of the functional requirements to the various distributed computer systems and sensors.

As discussed previously, military computer systems span the spectrum from single microprocessors in "smart-bombs" or communications line controllers to multiple, distributed mainframes in world-wide Communication/Command/Control systems. These distributed processing systems can, however, be visualized as a region of a volume bounded by axes describing (1) distribution of hardware, (2) distribution of control, and (3) distribution of data bases. Figure 4-8 shows this volume as well as the relationship of uniprocessor systems to distributed processing systems. The particular use and performance, including reliability and maintainability, requirements of the computer system will determine where in the characteristic volume the system will be placed. Appendix A (Definition of the Scope of Embedded Distributed Processing Systems) discusses (1) the distribution of the various different types of military systems within this characteristic volume of Figure 4-8 and (2) the formulation of the various distributed processing systems into two high-level generic classifications: weapon systems and communication systems. Also presented in Appendix A (see Table A-1) are the key characteristics of the distributed processing systems which comprise these two generic categories.

The common thread linking the different types of distributed systems is the requirement to interconnect and communicate data and messages between the various processing elements. In many systems, serial communications lines are used as the interconnecting

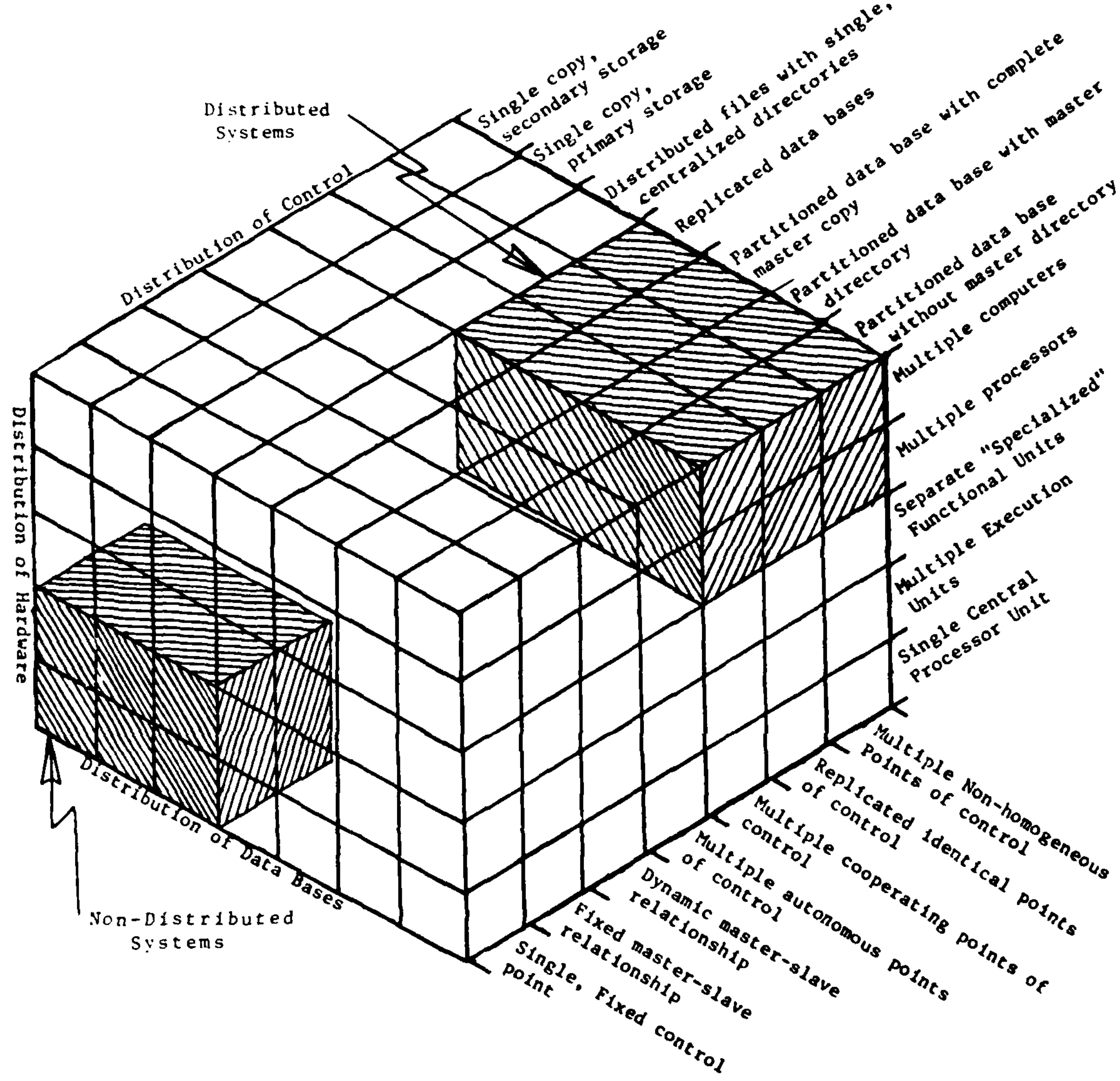

Distributed Systems
Distribution of Control
Single copy, secondary storage
Single copy, primary storage
Distributed files with single, centralized directories
Replicated data bases
Partitioned data base with complete master copy
Partitioned data base with master directory
Partitioned data base without master directory
Multiple computers
Multiple processors
Separate "Specialized" Functional Units
Multiple Execution Units
Single Central Processor Unit
Distribution of Hardware
Multiple Non-homogeneous Points of control
Replicated identical points of control
Multiple cooperating points of control
Multiple autonomous points of control
Dynamic master-slave relationship
Fixed master-slave relationship
Single, Fixed control point
Distribution of Data Bases
Non-Distributed Systems

Figure 4-8 Characteristic Volume of Computer System Capabilities

links. For geographically dispersed systems, these links are usually provided by the common carriers or dedicated microwave systems. Hence, the designer of distributed systems is faced with significant communications issues. Generally, the analysis of these communications issues and interconnect technologies, with associated characteristics, is aided by grouping the various interconnect techniques and architectures into three generic classifications: 1) computer buses (elements geographically dispersed within 200 feet), 2) local area networks (elements geographically dispersed within 6000 feet), and 3) long haul networks (elements geographically dispersed over many miles). Figure 4-9 depicts the physical distance relationship for these communication networks. Table 4-1 identifies 12 common interconnect technologies, together with their performance, reliability, geographic distribution, and modularity and expandability characteristics that support maintainability. In addition to these interconnect topologies, a logical structure or protocol must be used to allow for meaningful communications. Protocol can be classified in five levels, not all of which exist in all networks:

(1) Line control procedures. This is the lowest level of protocol. It administers the physical transmission medium.

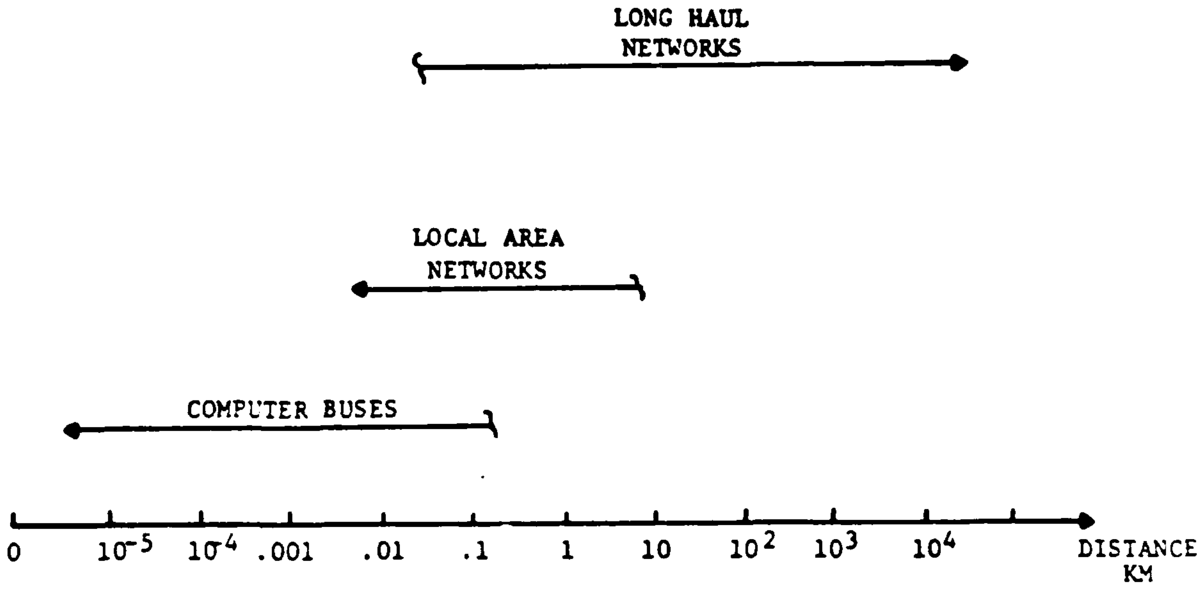
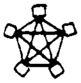
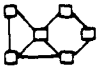
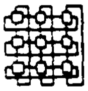
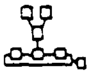
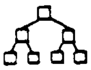
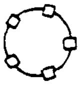
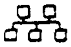
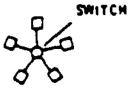
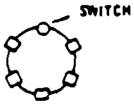
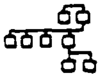
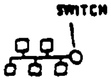
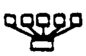


Figure 4-9 Communication Networks Versus Distance

Table 4-1 System Reliability Is a Driving Force for the Interconnect Technology

INTERCONNECT TECHNOLOGY	RELIABILITY		GEOGRAPHIC DISTRIBUTION	MODULARITY EXPANDABILITY	PERFORMANCE
COMPLETE INTERCONNECTION – THE COMPLETELY INTERCONNECTED ARCHITECTURE IS CONCEPTUALLY THE SIMPLEST DESIGN IN THIS DESIGN EACH PROCESSOR IS CONNECTED BY A DEDICATED PATH TO EVERY OTHER PROCESSOR COMMUNICATIONS COST BECOME PROHIBITIVE AS THE NUMBER OF PROCESSORS AND DISTANCES INCREASE 	ONLY LOCAL PROBLEM IF MINI FAILS REDUNDANT PATHS FOR SINGLE LINK FAILURES	MAXIMUM	UNLIMITED	FAIR NUMBER OF PORTS OF EACH MINI. • N 1	TYPICALLY 2400-4800 bps 50 Kbps AND 1 544 Mbps POSSIBLE
PACKET SWITCHED NETWORK – MESSAGES BROKEN INTO PACKETS AND TRANSMITTED VIA AVAILABLE NODES. AT LEAST TWO PATHS EXIST BETWEEN ANY TWO COMPUTERS IN THE SYSTEM 	ONLY LOCAL PROBLEM IF MINI FAILS		UNLIMITED	GOOD	TYPICALLY 56 Kbps
REGULAR NETWORK – EVERY COMPUTER IS CONNECTED TO ITS TWO NEIGHBORS AND TWO COMPUTERS ABOVE AND BELOW IT. THE NETWORK GETS COMPLICATED IF THERE ARE VERY MANY COMPUTERS. THE "TREE" IS A HIERARCHICALLY STRUCTURED VARIATION WITH ANY PROCESSOR ABLE TO COMMUNICATE WITH ITS SUPERIOR AND ITS SUBORDINATES AS WELL AS ITS TWO NEIGHBORS 	ONLY LOCAL PROBLEM IF MINI FAILS REDUNDANT PATHS FOR SINGLE CONNECT FAILURE		CAN BE UNLIMITED. TYPICALLY VERY LIMITED (10'S OF FEET)	POOR	TYPICALLY 3 5 Mbps
IRREGULAR NETWORK – THIS CONFIGURATION HAS NO CONSISTENT NEIGHBOR RELATIONSHIPS. IT IS COMMON IN GEOGRAPHICALLY DISPERSED NETWORKS WHERE COMMUNICATION LINKS CONTROL THE DESIGN 	PARTIAL REDUNDANCY FOR LINK FAILURES	INCREASED RELIABILITY	UNLIMITED	FAIR	TYPICALLY 2400 9600 bps
HIERARCHY – THIS CONFIGURATION IS USED IN PROCESS CONTROL AND DATA ACQUISITION APPLICATIONS. THE CAPABILITIES ARE SPECIALIZED AT LOWER LEVELS AND MORE GENERAL PURPOSE AT THE TOP 	SYSTEM OPERABILITY REDUCED WITH SINGLE POINT FAILURE. MORE SERIOUS THE HIGHER UP THE FAILURE OCCURS.		UNLIMITED	GOOD	TYPICALLY 2400-9600 bps
LOOP OR RING – LOOP ARCHITECTURE EVOLVED FROM THE DATA COMMUNICATIONS ENVIRONMENT. IN THIS CONFIGURATION, EACH COMPUTER IS CONNECTED TO TWO NEIGHBORING COMPUTERS. THE TRAFFIC COULD FLOW IN BOTH DIRECTIONS, BUT CIRCULATING TRAFFIC IN ONE DIRECTION IS LESS COMPLICATED 	SYSTEM UNAFFECTED WITH SINGLE LOOP FAILURE FOR TWO-LOOP SYSTEM CATASTROPHIC FOR SINGLE UNIDIRECTIONAL LOOP		LIMITED (100'S 1000'S OF FEET)	GOOD LIMITED BY MINI ADDRESSSING CAPABILITY	PARALLEL UP TO 500 KILO WORDS/SEC SERIAL 1 3 Mbps
GLOBAL BUS – THE USE OF A COMMON OR GLOBAL BUS REQUIRES SOME ALLOCATION SCHEME FOR SENDING MESSAGES FROM ONE COMPUTER TO ANOTHER 	ONLY LOCAL PROBLEM IF MINI FAILS CATASTROPHIC WITH BUS FAILURE		LIMITED (1000'S OF FEET)	GOOD	UP TO 50 Mbps TYPICAL 1 3 Mbps
STAR – THIS CONFIGURATION HAS A CENTRAL SWITCHING RESOURCE. EACH COMPUTER IS CONNECTED TO THE CENTRAL SWITCH. TRAFFIC IS IN BOTH DIRECTIONS 	ONLY LOCAL PROBLEM IF MINI OR BUS FAILS. CATASTROPHIC IF SWITCH FAILS. SWITCH POSSIBLY LESS RELIABLE THAN BUS OR LOOP		LIMITED (1000'S OF FEET)	GOOD UNTIL SWITCH SATURATES	UP TO 3 Mbps
LOOP WITH SWITCH – THIS REFINEMENT OF THE LOOP PROVIDES A SWITCHING ELEMENT THAT REMOVES MESSAGES FROM THE LOOP MAPS THEIR ADDRESSES, AND REPLACES THEM ON THE LOOP PROPERLY ADDRESSED TO THEIR INTENDED DESTINATION 	CATASTROPHIC IF EITHER SWITCH OR LOOP FAILS		LIMITED (100'S-1000'S OF FEET)	GOOD FAIR UNTIL SWITCH SATURATES	1 3 Mbps
BUS WINDOW – THIS CONFIGURATION HAS MORE THAN ONE SWITCH. MESSAGES MAY BE TRANSMITTED ON THE PATH THEY ARE RECEIVED OR ON ANOTHER. THE SWITCHES PROVIDE "WINDOWS" FOR PASSING MESSAGES BETWEEN BUSES. 	SERIOUS CONTENTION PROBLEMS. PARTIAL SYSTEM FAILURE IF SWITCH OR BUS FAILS	REDUCED RELIABILITY	VERY LIMITED (10'S OF FEET)	POOR	200-500 KILO WORDS/SEC
BUS WITH SWITCH – THIS IS MORE LIKE THE GLOBAL BUS BECAUSE EACH COMPUTER IS CONNECTED TO THE CENTRAL SWITCH, AND TRAFFIC FLOWS FROM THE ORIGINATING COMPUTER TO THE SWITCH, AND FROM THE SWITCH TO THE DESTINATION COMPUTER. THE COMPUTERS SHARE THE PATH (BUS) TO SHARE ACCESS TO THE SWITCH 	CATASTROPHIC IF BUS OR SWITCH FAILS		LIMITED (1000'S OF FEET)	GOOD FAIR UNTIL SWITCH SATURATES	UP TO 3 Mbps FOR SERIAL BUS
SHARED MEMORY – THE MOST COMMON WAY TO INTERCONNECT COMPUTER SYSTEMS IS TO COMMUNICATE BY LEAVING MESSAGES FOR ONE ANOTHER IN A COMMONLY ACCESSIBLE MEMORY THE KEY CHARACTERISTIC IS THAT THE MEMORY IS USED AS A DATA PATH AS WELL AS STORAGE. 	CATASTROPHIC IF MEMORY FAILS	MINIMUM	VERY LIMITED (10'S OF FEET)	POOR LIMITED TO NUMBER OF MEMORY PORTS	MEMORY SPEED 500 KW/SEC TO 3 MW/SEC

(2) Procedures to control data flow between communications processors (packet flow).

(3) Procedures to control data flow between a host computer and a communications processor.

(4) Procedures to allow flow between two distant host computers.

(5) Procedures to allow message flow between two user processes.

However, the particular categories of military systems most affected by technological advancements in embedded distributed processing include primarily the first two categories, local area networks and computer buses. The expansion of the von Neumann architecture within a multiprocess environment is typified by the diagram in Figure 4-10 and is characteristic of the interconnect technologies associated with the computer buses. Also in the computer bus technologies, the advances in Very-Large-Scale-Integration (VLSI) circuitry has offered new interconnect architectural alternatives which are shown in Figure 4-11 and are discussed in paragraphs 5.1 and 7.1. The key characteristic of the layered bus architecture is associated with its operation; in that, the communication between layers becomes probabilistic instead of deterministic as is the case with the von Neumann architecture. (Refer to paragraph 7.1.1 for a full explanation of this terminology.)

Local area networks (LAN) have evolved primarily from a need to provide data communication on a packet basis between increasingly intelligent terminals and host computer systems. The intelligent terminals are generally separated over larger distances than the internal workings of the computer buses, but at the same time these terminals are not nearly so distant as to require a communication link such as microwave or other long haul communication systems. Furthermore, LAN architecture and protocol is generally compatible with computer systems on a level not always achieved with long haul networks. Examples of the local area networ:. are shown in Figure 4-2 and 4-3 in terms of the dual serial data bus. Dual bus topology is generally the preferred topology used with LAN systems since they offer acceptable performance and high reliability. Figure 4-12 shows some of the key advantages and disadvantages associated with the LAN bus topology. In a like manner, bus control strategies, examples of bus systems, and examples of transmission media are shown in Figures 4-13, 4-14, and 4-15, respectively. Table

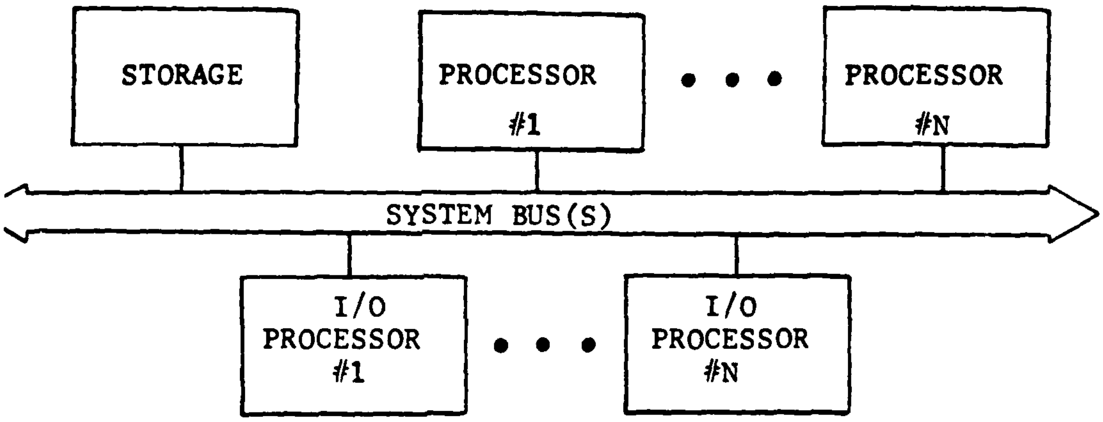


Figure 4-10 Expanded von Neumann Architecture

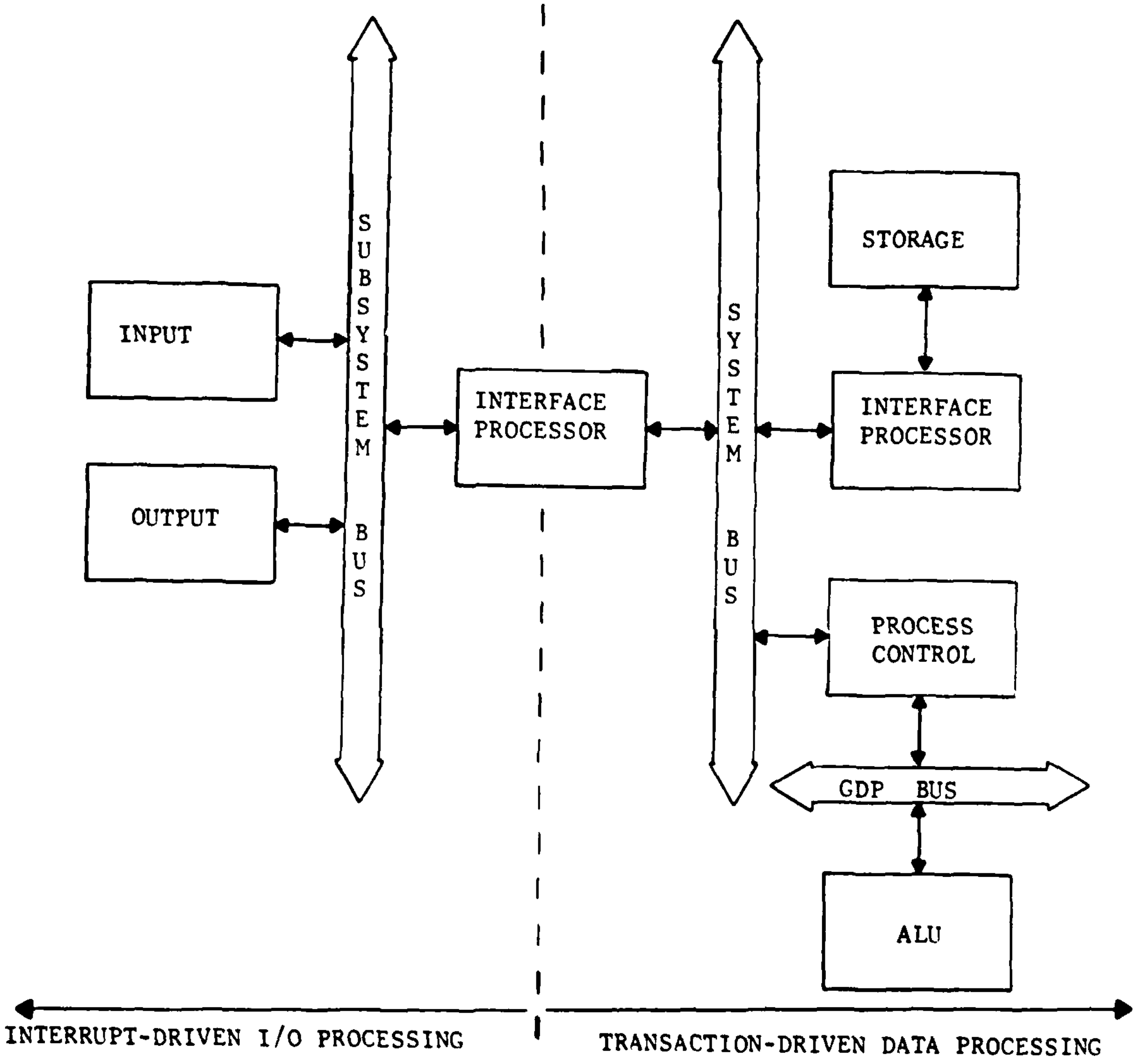


Figure 4-11 Layered Bus Architecture

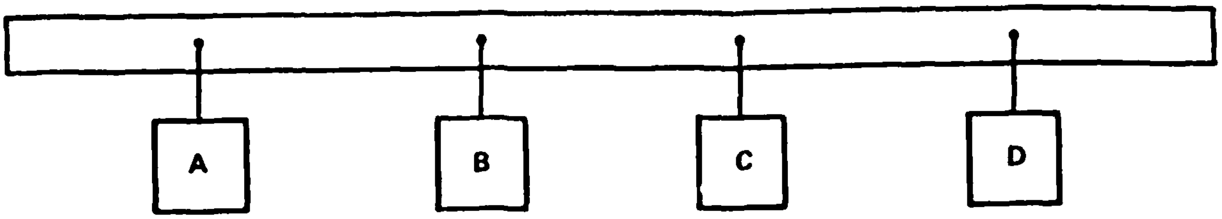
4-2 is a compilation of the majority of LAN's currently available.

Six additional distributed processing characteristics which serve as motivations for the continued development of parallel (concurrent) processing systems are:

- Response time
- Flexibility
- Resource sharing
- Reliability
- Availability
- Transportability

The common denominator and ultimate result of improvements in these characteristics is improved overall system performance, usually measured also on the basis of cost effectiveness. These characteristics are further discussed as follows:

(1) Reliability - Redundancy, which is related to reliability, can be achieved in a relatively inexpensive manner in a distributed system since the entire system does not have to be replicated as is the case with a single computer. Only an incremental number of processors must be added to insure the



ADVANTAGES	DISADVANTAGES
● No Active Taps	● Bandwith Limitations
● Interface Simplicity	● Probabilistic Access
● Fully Connected	
● High Reliability (Distributed Control)	

Figure 4-12 Linear Bus Subnetwork Topology

Contention

- Transmit
- Listen before talk
- Listen while talk

Back Off Strategies

- Fixed delay
- Adaptive delay
- Random delay

Figure 4-13 Bus Control Strategies

- Ethernet (Xerox)
- Mitre-bus (Mitre)
- Ford net (FACC)
- NSC-hyperchannel (Network Systems)
- LCN (UNIVAC)
- CABLENET (AMDAX CORP.)

Figure 4-14 Examples of Bus Systems

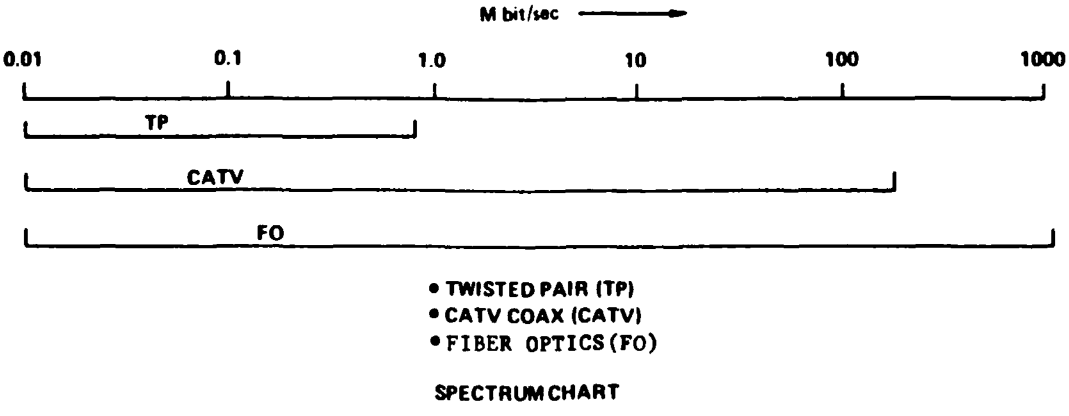


Figure 4-15 Transmission Media

Local Network	Company	Trans-Mission Medium	Max No. of Nodes	Network Architecture	Access Scheme
Attached Resource Computer (ARC)	Datapoint Corp.	Broadband coax/non-coherent infrared energy	255	Star	Proprietary
Cluster/One Model A	Nestar Systems Inc.	Baseband multi-conductor cable	64	Arbitrary (bus, star, etc.)	CSMA/CD
Distributed Operating Multi Access Inter Network (DOMAIN)	Apollo Computer Corp.	Broadband coax	Several hundred	Ring	Token passing
ETHERNET	Xerox Corp.	Baseband coax	100	Bus	CSMA/CD
HYPER-channel	Network Systems Corp.	Baseband coax	16 Low	Bus	CSMA/CA
HYPER-bus	"	"	"	"	Hybrid CSMA/CD (ACK)

Table 4-2 Typical Local Area Net

ess eme	Max length of Internodal transmission medium	Max Data Rate	Applications	Comments
prie- y	4 miles for coax. 1 mi.for infrared	2.5Mb/s	Office automation, data processing	Each processor node can participate in up to 6 ARCs
A/	1000 ft	250 kb/s Low	Office automation,personal computers, general-purpose	Free-form topology due to low data rates
n sing	3000 ft	12Mb/s Mb/s	Engineering scientific, CAD/CAM, general-purpose	Virtual file accessing, can page across network in a virtual environment
/	2.5 km	10 Mb/s	Office automation	Maximum separation between stations is 2.5 km.
/	Unavailable	50 Mb/s	Scientific, large computer centers	Link adapters allow a node to be attached to 4 independent trunk-to-trunk interfacing via microwave,fiber optics, and common carrier lines.
d /CD	"	6 Mbs		

rea Network Products

Local Network	Company	Trans-Mission Medium	Max No. of Nodes	Network Architecture	Acce Sche
Modway	Modcon Div. of Gould Inc.	Baseband or broadband coax, fiber optics	250	Arbitary (bus, star, etc.)	Toke pass
WANGNET	Wang Labs	Broad-Band coax		Open Loop	
Utility Band	Wang Labs	"	7 Channels		None
Interconnect Band	"	"	32 16	"	None
Wang Band	"	Broad Band coax	Many	"	CSMA/ (802)
Loosely coupled Network LCN	Control Data Corp.	Base Band coax	108	Trunk + Node	Rotat Prior Synch our TOKEN
IMUX	SCI, Systems	Base Band	100	Bus	Contetion

Table 4-2 (co

s	Network Architecture	Access Scheme	Max length of Internodal transmission medium	Max Data Rate	Applications	Comments
	Arbitary (bus, star, etc.)	Token passing	15,000 ft	1,544 Mb/s	Data processing, process control	Compatible with microwave and satelite-communications facilities for common-carrier transmissions
	Open Loop					
nels		None	N/A	N/A	Video	Supplies 7 channels to composite video equipment
	"	None	N/A	9.6Kb/J		Modem
				64 Kbs		Modem
	"	CSMA/CD (802)	N/A	12 MHz		X.25 SDLC compatible and CSMA/CD compatible
	Trunk + Node	Rotating Priority Synchronour mode TOKEN	3000 ft	50 MHz	Large Computers	Not x.25 compatible compares with Hyperchannel
	Bus	Contention	1000 ft.	10 MHz	Data Bus Systems	

Table 4-2 (continued)

1 2

Local Network	Company	Trans-Mission Medium	Max No. of Nodes	Network Architecture	Access Scheme	Max of dal mis med
Net/One	Ungermanu-Bass Inc.	Baseband coaz	250	Bus	CSMA/ CD	4000
Omnlink	Northern Telecom Inc.	Broadband coax	9 Low	Ring	Token passing	5000
Primenet	Prime computer Inc.	Baseband coax	15 Low	Ring	Token passing	750
Z-Net	Zilog Inc.	Baseband coax	255	Bus	CSMA/ CD	2 km

Table 4-2 (Continued)

ccess cheme	Max length of Internodal transmission medium	Max Data Rate	Applications	Comments
MA/	4000 ft	4 Mb/s	OEM systems, data processing,scientific office automation, process control	Intelligent network interface units can be programmed to interface to a wide variety of terminals
oken assing	5000 ft	40 kb/s	Data processing, office automation	Each node can have independent and accessible files peripherals and processors
oken assing	750 ft	10 Mb/s	Data processing, large computer centers	CCITT X.25-compatible for interfacing to other networks over long distances
MA/	2 km	800 kb/s Low	Office automation,small business computers	Emulator package allows data transfer between Zilog equipment and other vendors' equipment

(Continued)

Local Network	Company	Trans-Mission Medium	Max No. of Nodes	Network Architecture	Acce Sche
Localnet Systems 20 & 40	Sytek Inc.	Broadband coax	256/ channel	Bus	CSMA/ CD
Cablenet	Amdax Corp.	Coax Broad Band	16,000	Bus	Hybri TDMA conte tion
DPS	Litton			Loop	SDLC
MITRE X	MITRE	Coax Broad Band		Bus	Hybri
Cambridge Ring	Cambridge Univ. (Logica Ltd. (Toltec Data, Ltd)	Twisted Pair cable Fiber Optic cable	15 Expandable	Ring	Rotat Slots

Table 4-2 (Co

Network Architecture	Access Scheme	Max length of Internodal transmission medium	Max Data Rate	Applications	Comments
Bus	CSMA/CD	30 km	120 kb/s, 2 Mb/s	Distributed processing design automation	Each channel has CSMA/CD accessing. Up to 120 channels per cable
Bus	Hybrid TDMA contention	50 miles	14 MHz	Universal	Protocol Free
Loop	SDLC	200M	20 MHz		Main Computer tie-in using noses $100,000/node - see Electronics July 14, 1981
Bus	Hybrid		1 MHz		LWT
Ring	Rotating Slots	100M Long Distances	10 MHz	For terminals, Computer tie ins	Will interface with most

Table 4-2 (Continued)

required degree of availability. Also simpler, and hence more reliable, software structures may be achievable in a collection of small distributed processors.

(2) Response time - The distributed system can be more responsive because direct access to a computer or processing element can be provided to smaller user communities. This responsiveness can take the form of reduced turnaround time in a batch environment and faster response times in a real-time environment.

(3) Flexibility - A distributed system in danger of overload can be expanded incrementally at low cost by the addition of more processors. Also a host computer system in danger of overload can be preserved by offloading functions onto smaller processors.

(4) Resource sharing - A distributed network of computing systems allows users at one location to take advantage of resources that are available at other locations. These resources could consist of programs, data buses, and computational power. Resource-sharing

networks allow load balancing, backup, and reduced duplication of effort.

(5) Availability - Enhanced system availability can be achieved in a distributed system by means of improved reliability and maintainability. Specifically, system maintenance functions can be performed in parallel with system operation, if appropriate redundancy in key system elements, such as processors, memory, displays and peripherals is also provided.

(6) Transportability - Redundancy also supports system transportability by providing the means whereby the system processes can be shifted to the various distributed data processor elements.

In summary, the grouping of (1) the distributed processing systems, (2) the interconnect technologies, and (3) the overall system characteristics has benefitted the Distributed Processing Tools Definition study by allowing the requirements for tools and techniques to be layered by classes of generic military systems with associated interconnect technologies.

5.0 STATE-OF-THE-ART TECHNOLOGIES

The scope of embedded distributed processing systems has grown too broad for the computer specialists of today to understand. These specialists concentrate on a single aspect of the whole instead of the whole itself. Until a new group of computing professionals assumes the more general viewpoint the specialist of today must be used. Their use requires some degree of modularization. Different groups of specialized individuals should be able to maintain separate modules without interfering with one another. This requires clever design of both hardware and software. Furthermore, the dichotomy between hardware and software is not clear within embedded distributed processing systems. A clever design in hardware impacts software and vice versa. The current situation is doubly serious because of the highly specialized nature of existing computing personnel. An obvious polarity exists within these personnel. Some prefer the hardware issues and gravitate to an engineering orientation. Others prefer software issues and gravitate toward real-time systems, operating systems, compilers, security, network systems, etc. However, such a polarity only indicates the knowlcdge gap which must be overcome by a new group of computer professionals.

Current organizations generally group their computing design efforts together. Their span of control ranges from a purely hardware orientation to a software orientation. The subsequent section acknowledges such an organization. In the following section, hardware technologies will be addressed. After those, a section on software technologies is presented. The issues raised in either section impact the other. Such cross correlation should be kept in mind as the sections are read.

5.1 Hardware Technologies

Modern embedded distributed processing systems are being influenced primarily by two hardware circuit technologies: (1) Very-Large-Scale-Integrated (VLSI) circuits and (2) Very-High-Speed-Integrated Circuits (VHSIC). In fact, these hardware technologies in conjunction with the current software crisis (software cost, reliability, and management) has caused a flurry of research during the past few years; and, this research has resulted in a number of technological advances that relate to the embedded distributed processing systems. Figure 5-1 illustrates the industrial trends in VLSI and VHSIC. The key technological advances can, generally, be grouped as follows:

- Development of algorithmically specialized processors; e.g., nxn mesh of interconnected microprocessors

- Development of new computer architectures; e.g., Intel iAPX432 computer system

- Development of specialized embedded processors with appropriate protocol to support local area bus networks; e.g., VLSI chips to support Ethernet.

5.1.1 Algorithmically Specialized Processors

Examples of algorithmically specialized processors include designs for (1) Logical Unit (LU) matrix decomposition which is the main step in solving systems of linear equations; (2) tree processors which are used in searching, sorting and expression

- **INCREASINGLY HIGH RESOLUTION IN LITHOGRAPHY AND PROCESSING TECHNOLOGIES**
- **ENABLING VHSIC APPLICATIONS**
  - JTIDS
  - BIDS
  - Anti-Jam Modems
  - Multifunction Radars
  - Electronic Countermeasures
  - Target Management Systems
  - Image Processing
  - General Purpose Computers

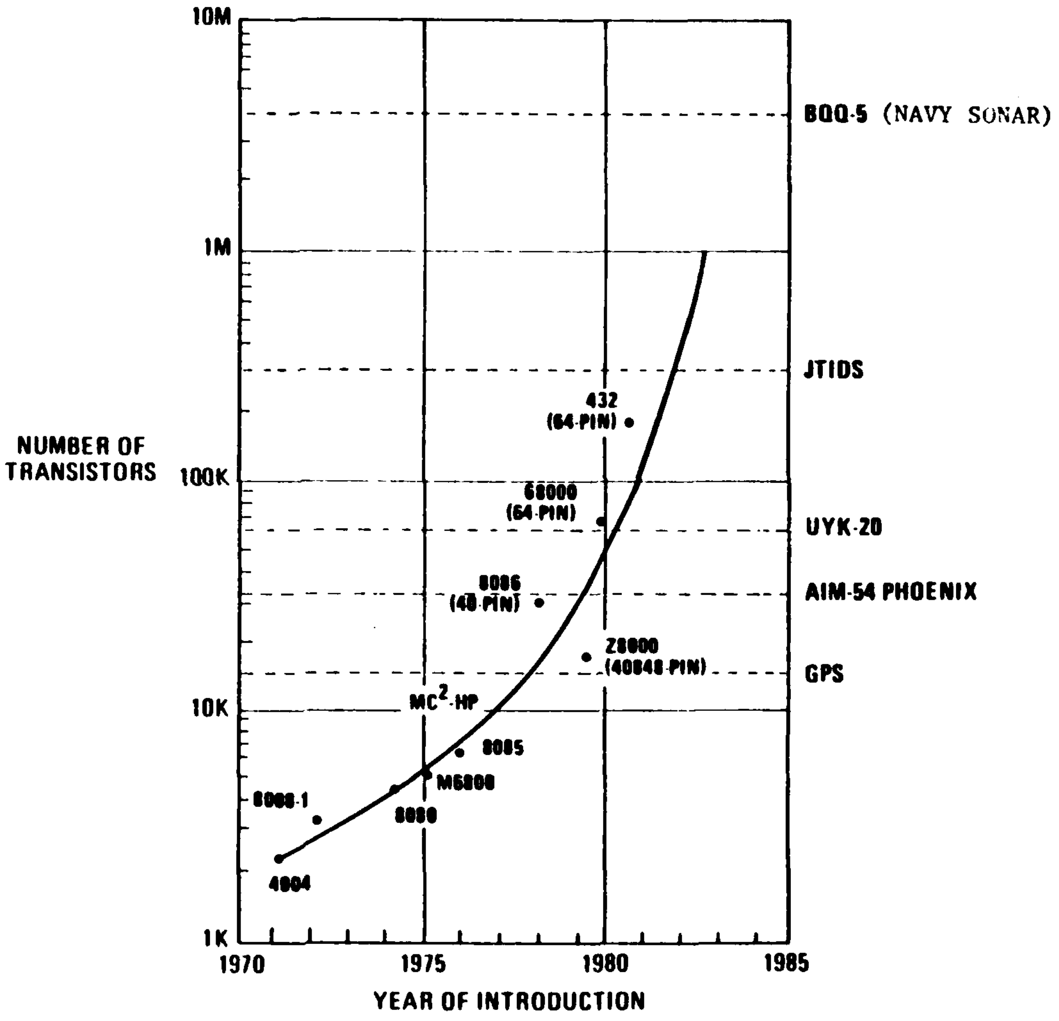
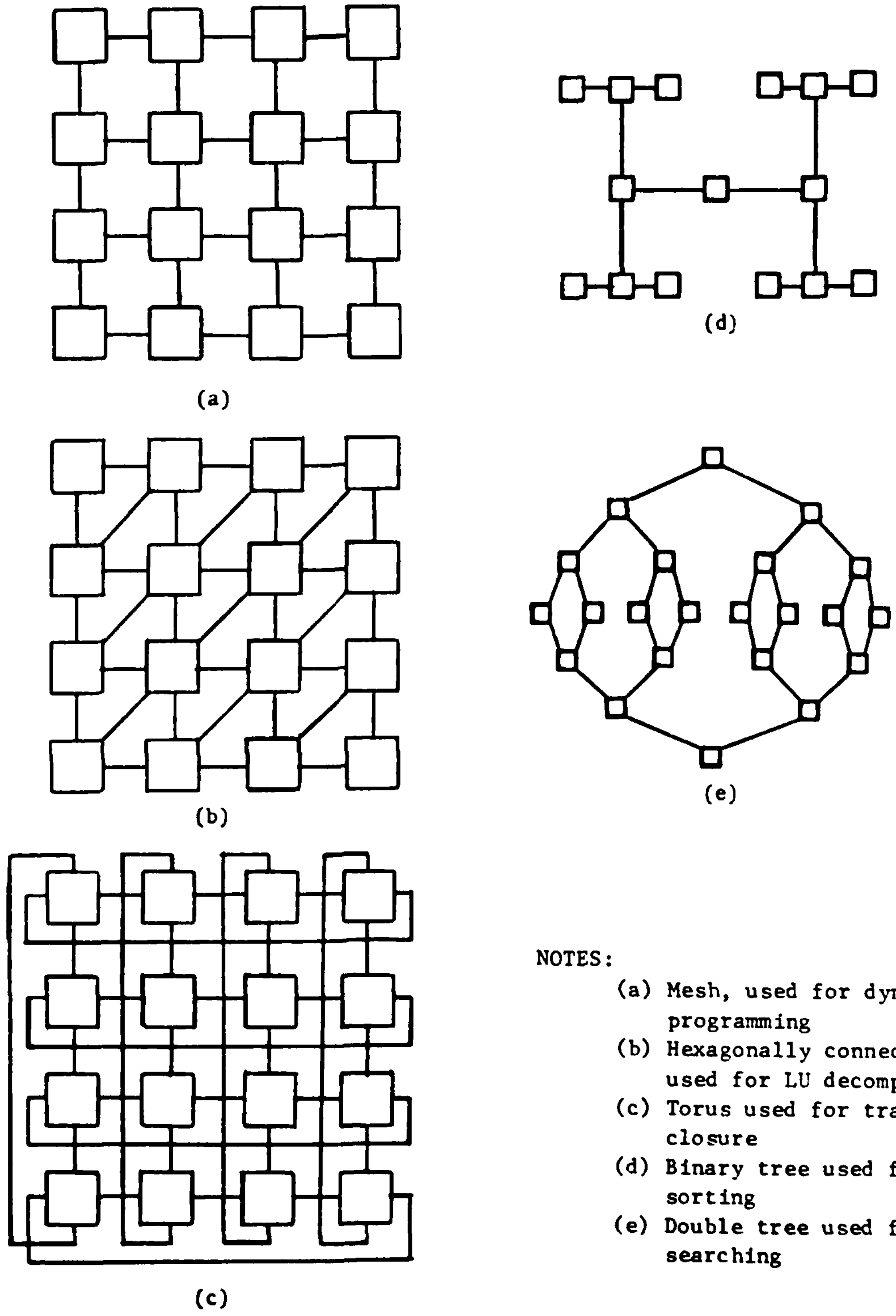


Figure 5-1 Industrial Trends in VLSI and VHSIC

evaluation; (3) dynamic programming matrix processors which are used for general problem solving; and (4) joint processors which are used for data base querying. Many researchers are, however, going to a more flexible approach which is to replace these dedicated processing elements with more general microprocessors and simply to program the algorithmically specialized processors. This solution is much more flexible since different components can use the same devices by changing programs and, with more recent research results, the interconnection patterns. Figure 5-2 shows some examples of the interconnection patterns used for specific functions. Figure 5-3 shows three examples of switch lattices which are used to reconfigure the matrices of general purpose microprocessor systems. The switch lattices are regular structures which are formed from programmable switches connected by data paths.

The Department of Computer Science personnel at Purdue University has developed a multimicroprocessor computer system (which is part of the research under the Blue CHiP Project) using this general processor and switching lattices network technology named the Configurable, Highly Parallel (CHiP)



NOTES:

(a) Mesh, used for dynamic programming

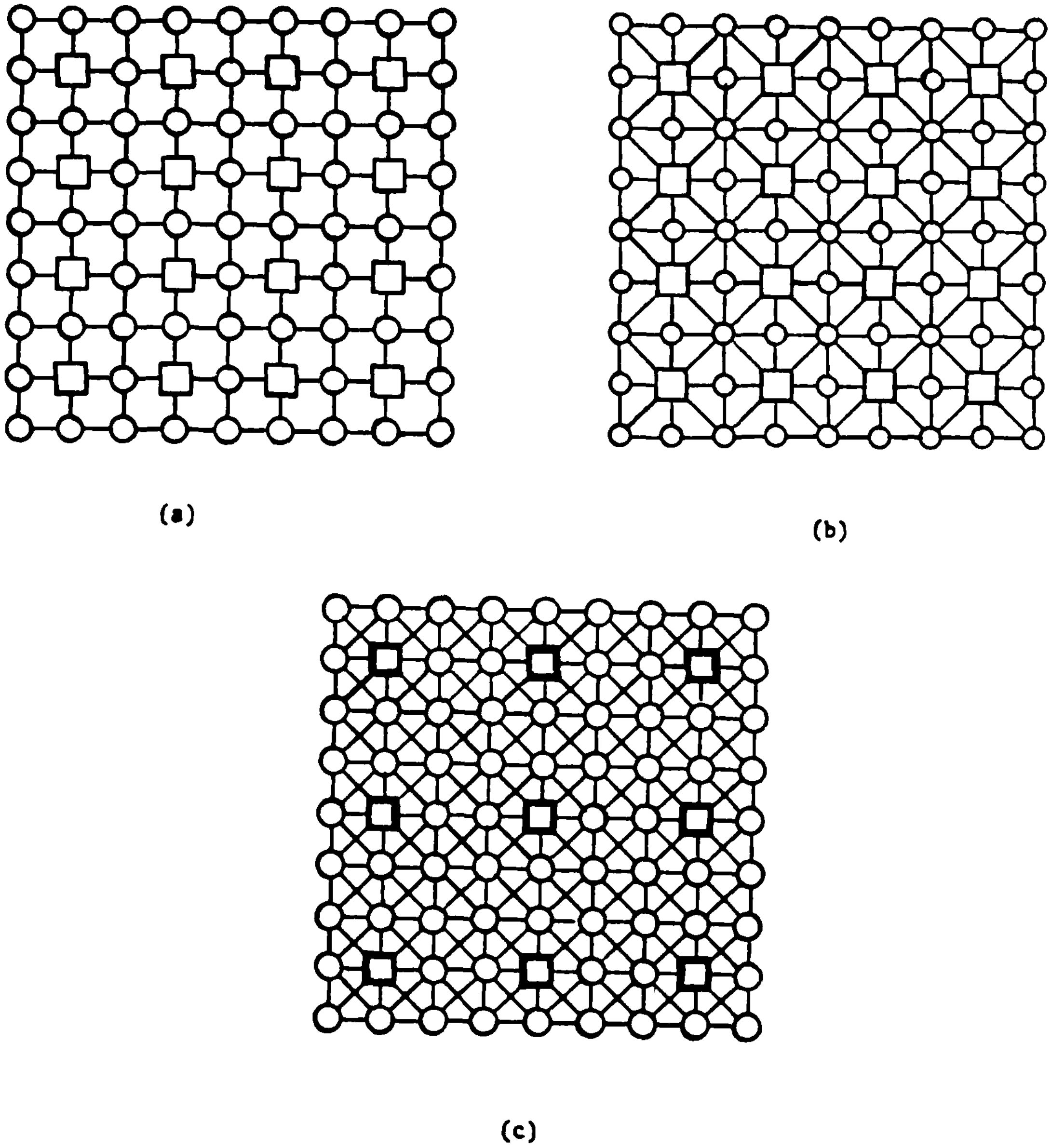
(b) Hexagonally connected mesh used for LU decomposition

(c) Torus used for transitive closure

(d) Binary tree used for sorting

(e) Double tree used for searching

Figure 5-2 Interconnection Patterns for Algorithmically Specialized Processors



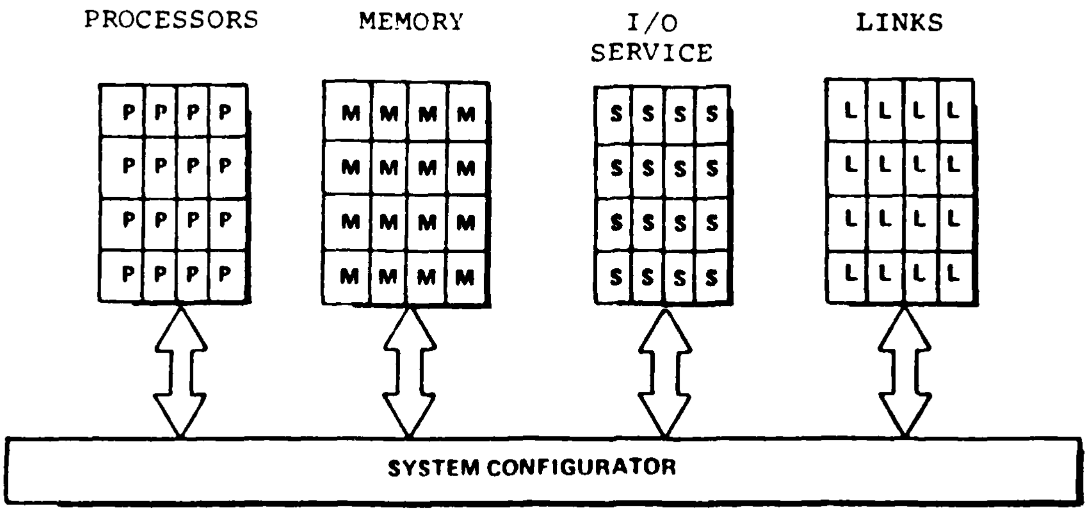
NOTE: Circles represent switches;
Squares represent processors

Figure 5-3 Three Switch Lattice Structures

Computer. The objective of this project is to provide the flexibility needed to compose general problem solutions while retaining the benefits of uniformity and locality that the algorithmically specialized processors exploit. The CHiP computer is a family of architectures each constructed from three components: (1) a collection of homogeneous microprocessors with associated memory, (2) a switch lattice, and (3) a controller. The switch lattice is the most important component and the main source of differences among family members; i.e., Figure 5-3(a), 5-3(b), and 5-3(c). The controller is responsible for loading the switch memory. CHiP processing begins with the controller broadcasting a command to all switches to invoke a particular configuration setting. For example suppose it is a mesh pattern (see Figure 5-2(a)) and a three switch lattice representation is used (see Figure 5-3). With the entire structure interconnected into a mesh, the individual microprocessor systems synchronously execute the instructions stored in their local memory. When a new phase of processing is to begin, the controller broadcasts a command to all switches to invoke a new configuration setting, say the one for a tree. With the lattice restructured into a tree interconnection pattern (see Figure 5-2(d)), the microprocessor systems resume processing, having spent only a single logical step in interphase structure reconfiguration. All three switch lattice structures of Figure 5-3 are capable of representing such an interconnection pattern. Other modes of operation include the operation of the microprocessor matrix with multiple instruction streams and multiple data streams. In this mode of parallelism, each processor takes its instructions and its data from its associated memory. As in the other mode, the interconnection network provides interprocessor communication. The overview of the CHiP computer family has been superficial, but it provided a context in which to present one hardware technological advancement category. Figures 5-4 and 5-5 show other system level architectures of the microprocessor matrix being used by the government, under support from System Development Corporation, at ARC Huntsville, Alabama.

5.1.2 Local Area Networks

In the area of other VLSI/VHSIC technology advancements, interconnect techniques/technologies are of vital importance to multiprocessor system



- OBJECTIVES
  - EMULATION OF DISTRIBUTED PROCESSING ARCHITECTURES
  - HIGH FIDELITY SIMULATION OF LWIRS SENSORS AND RADAR ANTENNA ELEMENTS
  - SUPPORT INTERFACES WITH SPECIAL-PURPOSE OPTICAL PROCESSORS
- FOURTH GENERATION MICROPROCESSOR HARDWARE
  - INTEL 8086
  - ZILOG Z8000
  - MOTOROLA MC 68000
- SYSTEM CONFIGURATOR
  - SWITCHING NETWORKS
  - PROGRAMMABLE INTERCONNECTION WITH BIT-SLICE

Figure 5-4 Multiple Microprocessor System

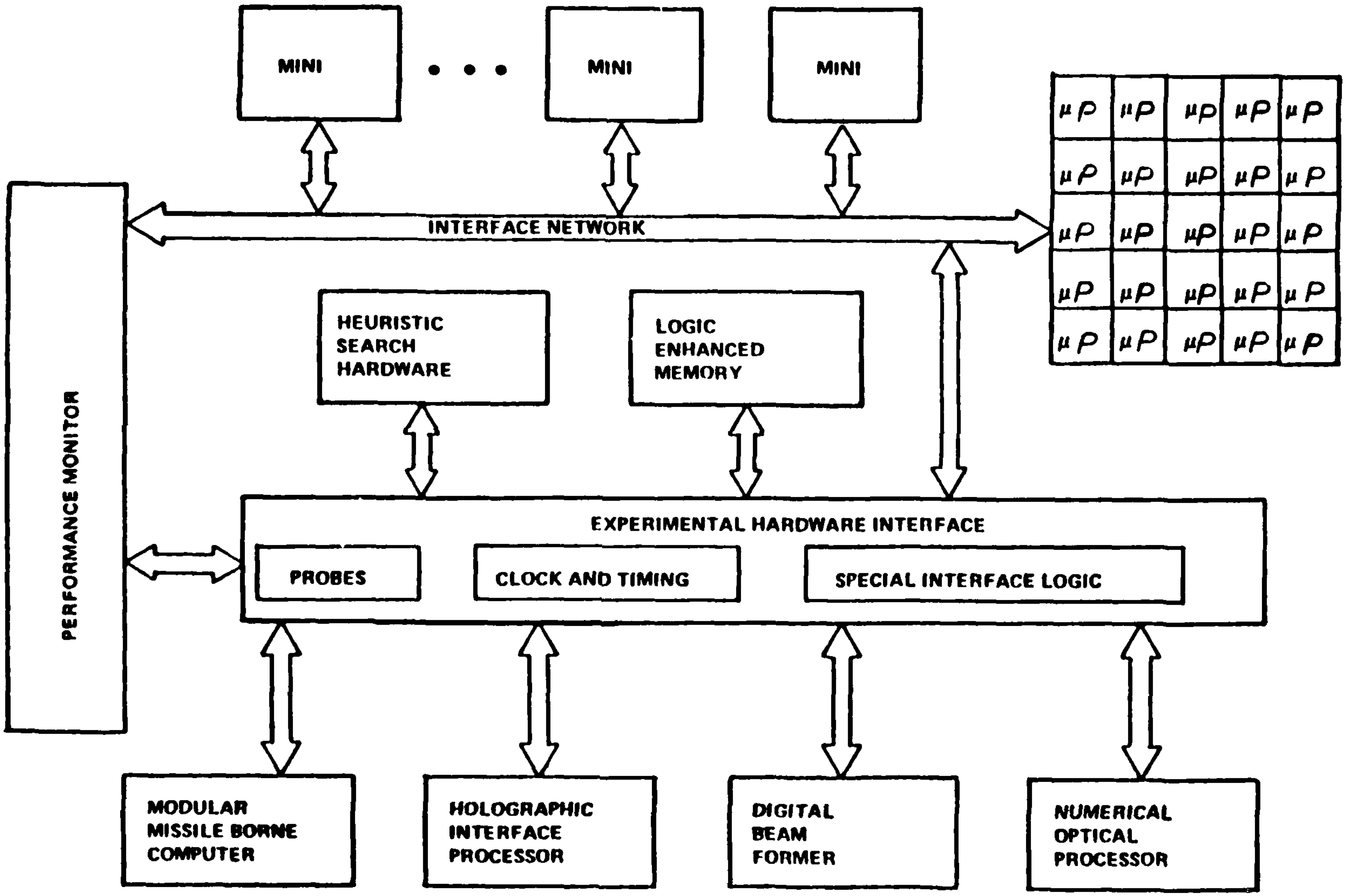


Figure 5-5 Advanced Data Processing Testbed -
Laboratory Model Hardware

operation. In fact, all distributed data processing systems are characterized by the requirements to interconnect and communicate data and messages between the various processing elements. However, local area networks and computer bus technologies have increasingly occupied the attention of research workers. As discussed in paragraph 4.0, local area networks are data communications systems for the interconnection of terminal and distributed data processing elements that are within one building, in several buildings on the same property, or in close proximity; as contrasted with the more familiar local and long-haul networks for private lines, public switched services, and private switched systems. The total extent of a local area network may thus be as little as a few hundred meters, or as great as several kilometers. Furthermore, the characteristic that sets recently-announced local area networks apart from conventional local and long-haul networks is bandwidth. It is feasible and relatively inexpensive to implement bandwidths or data rates of 10 megabytes per second (Mbps) in local area networks. Because of (1) the varying views of local network designers and users in regard to the diversity of types of devices to the connected, (2) the need for consistency of local network protocols with mainframe protocols, and (3) the desire to interoperate local networks and various external networks, such as the packet-switched common carrier networks, standardized, off-the-shelf local area networks are somewhat limited. DEC, Intel, and Xerox are, however, developing a specification with associated VLSI chips to support Ethernet Carrier Sense Multiple Access with Collision Detection (CSMA-CD) method of control. Xerox provides the basic local network design; DEC contributes the system design expertise in the area of communication transceivers and mini-computer networks; and Intel supplies the expertise in the partitioning of complex communications functions into micro-computer systems and VLSI components. The main problem with Ethernet occurs when two stations begin transmitting at the same instant. Such an event wastes the channel for an entire packet time. In this method a station wishing to transmit listens first for channel clear, and then transmits if such is the case. Collision detection is also implemented for the case where two stations transmit simultaneously. The characteristics of this network include a 10-Mbps data rate, coaxial cable medium with 500 meter Computer Interface Unit (CIU) spacing, and a datagram link-level protocol. The VLSI/VHSIC technologies are allowing many other companies, such as Zilog, to follow suit with their network versions as well. As discussed previously, detailed discussions of the characteristics of various interconnects/networks are presented in paragraph 4.0.

5.1.3 Bus Technologies

Primary advances in the computer bus technologies, as related to embedded distributed processing systems, are computer architectures which are characterized by multiple computer buses. Detail discussions of these multiple bus architectures are presented in paragraphs 6.3 and 7.1.1.

5.1.4 New Computer Architectures - iAPX432

The Intel iAPX432 computer architecture includes, in addition to multiple buses, the total impact of the VLSI/VHSIC technologies on modern computer architectures. Figure 5-6 presents the Intel iAPX432 structure along with key features/characteristics. As noted on the figure, bus bandwidth limits system performance. Figure 5-

- •INTERCONNECTIVITY USING BUSES
- •DATA STRUCTURE ENCODING
  - – Huffmann Code
  - – Assumed Frequency of Occurrence
- •BUS BANDWIDTH LIMITS PERFORMANCE
- •USES Ada MODULARIZATION
- •TRANSACTION DRIVEN ARCHITECTURE
  - – Interface Processors Control Bus Traffic
  - – Interface Processors Control Data FLow
- •HIGHLY-CONCURRENT ARCHITECTURE
- •VLSI PRODUCT

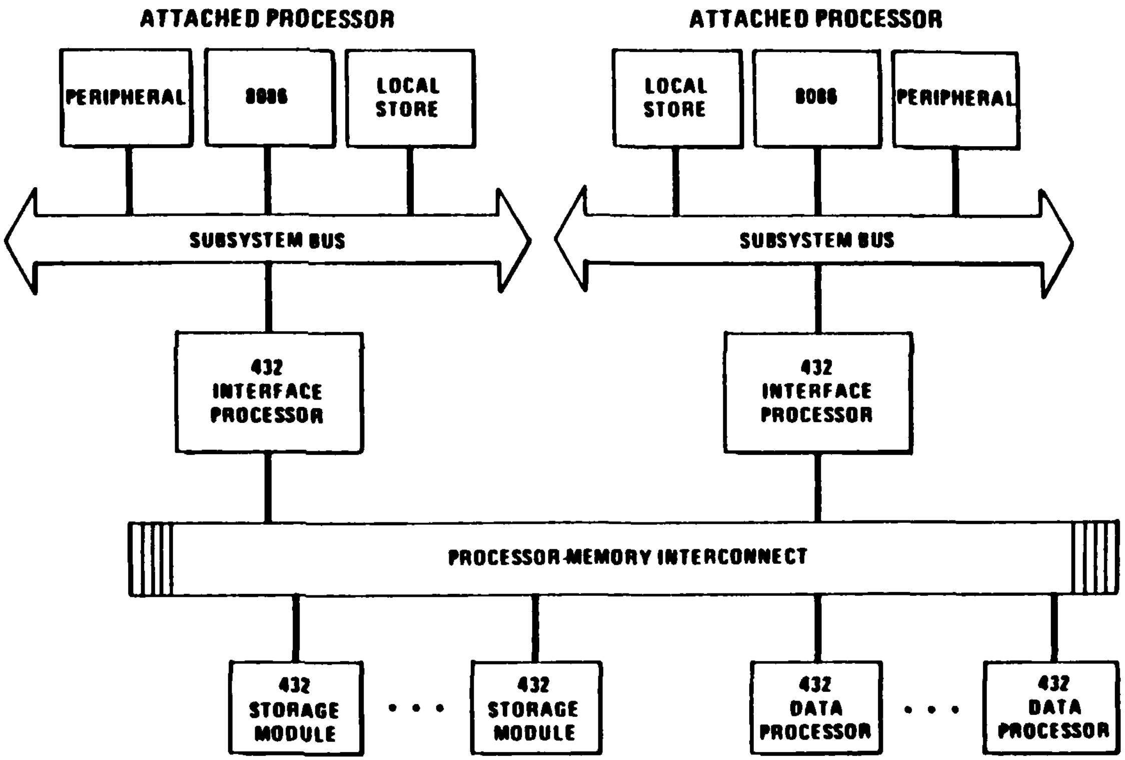
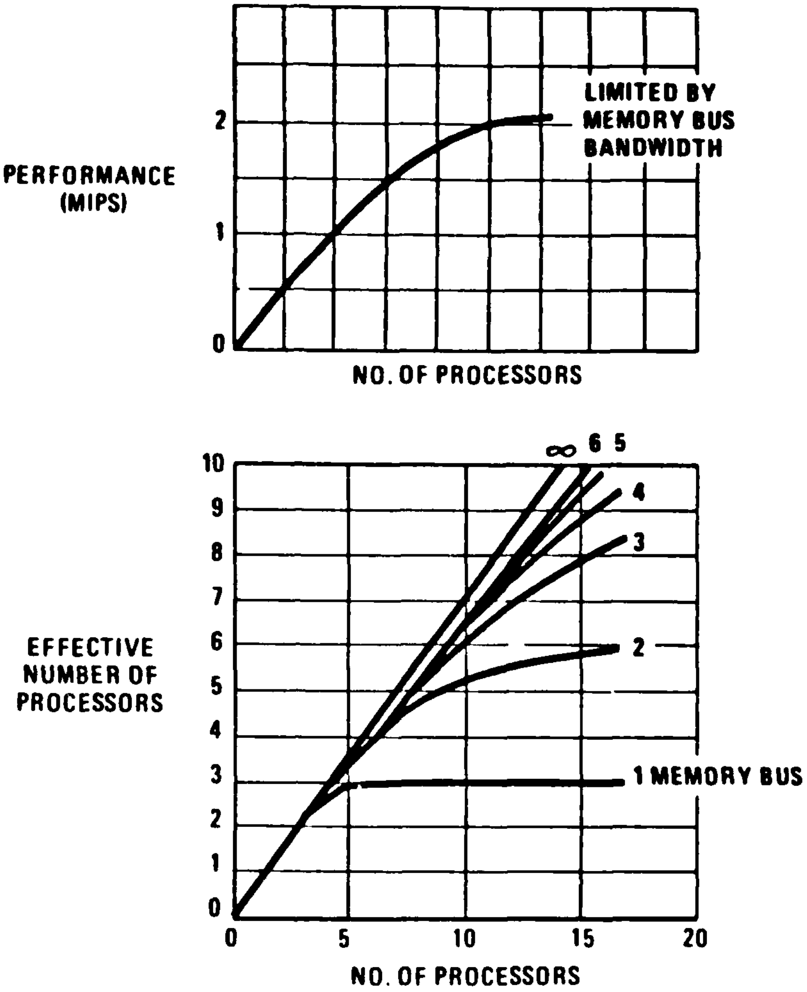


Figure 5-6 Intel iAPX432 Computer Structure



- BUS BANDWIDTH LIMITS PERFORMANCE
- INCREASING THE NUMBER OF BUSES, INCREASES INTERCONNECTIVITY PROBLEMS
- INCREASING INTERCONNECTIVITY, INCREASES CONCURRENT OPERATING CAPABILITY
- INCREASING CONCURRENT OPERATION, INCREASES DEMAND ON THE HOL FOR CONCURRENT SOFTWARE

Figure 5-7 Impact of Performance Planning for iAPX432

7 further illustrates this characteristic by showing that 2 million instructions per second (MIPS) is the upper throughput performance limit with current memory bandwidth and a single memory bus. The figure also shows the relationship of the effective number of processors versus the number of memory buses. As an example, a five processor configuration with only one memory bus would have throughput performance capabilities (measured in MIPS) of only three processors; whereas, two memory buses would increase the effective number of processors to about 3.5. However, a ten processor configuration would require two memory buses to achieve a five processor throughput. Figure 5-8 shows the performance of the Intel iAPX432 computer as compared to other computer types; and Figure 5-9 shows the new approach to hardware fault detection which can be implemented within Intel iAPX432 architecture. With this new hardware fault detection, the iAPX432 hardware can detect many different fault conditions, from attempting to execute data, to complex faults involving several processes. Once a fault is detected, the operation is aborted, and a complete description of the fault is reported. In a multiprocess system, a fault may cause one processor to suspend itself and begin

running diagnostics, but the other processors can usually keep the system operating.

In summary, advances in VLSI circuits and VHSIC technologies are having major impacts on modern embedded distributed processing systems. Specific impacts, on computer system capacity trends are itemized as follows:

- Increasing Programming Payloads
- Exceeding Requirements of Existing Military Standards
  - 1750A
  - 1553B
  - 1765 (Proposed)
- Increasing Tightly-Coupled Configurations
- Increasing Concurrent Hardware Operation
- Improving Error/Correction Capability

In a like manner, the impacts of VLSI/VHSIC technologies on computer timing trends are summarized as follows:

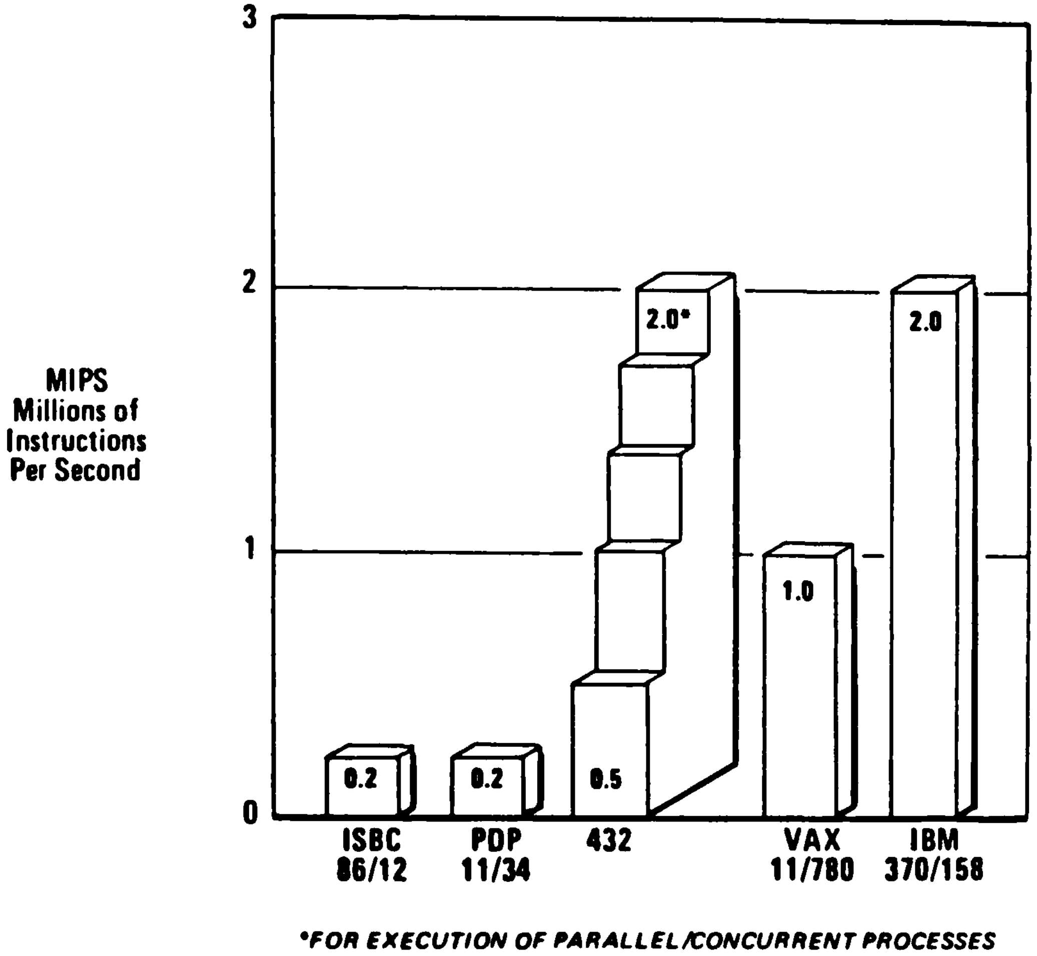


Figure 5-8 Performance Comparison

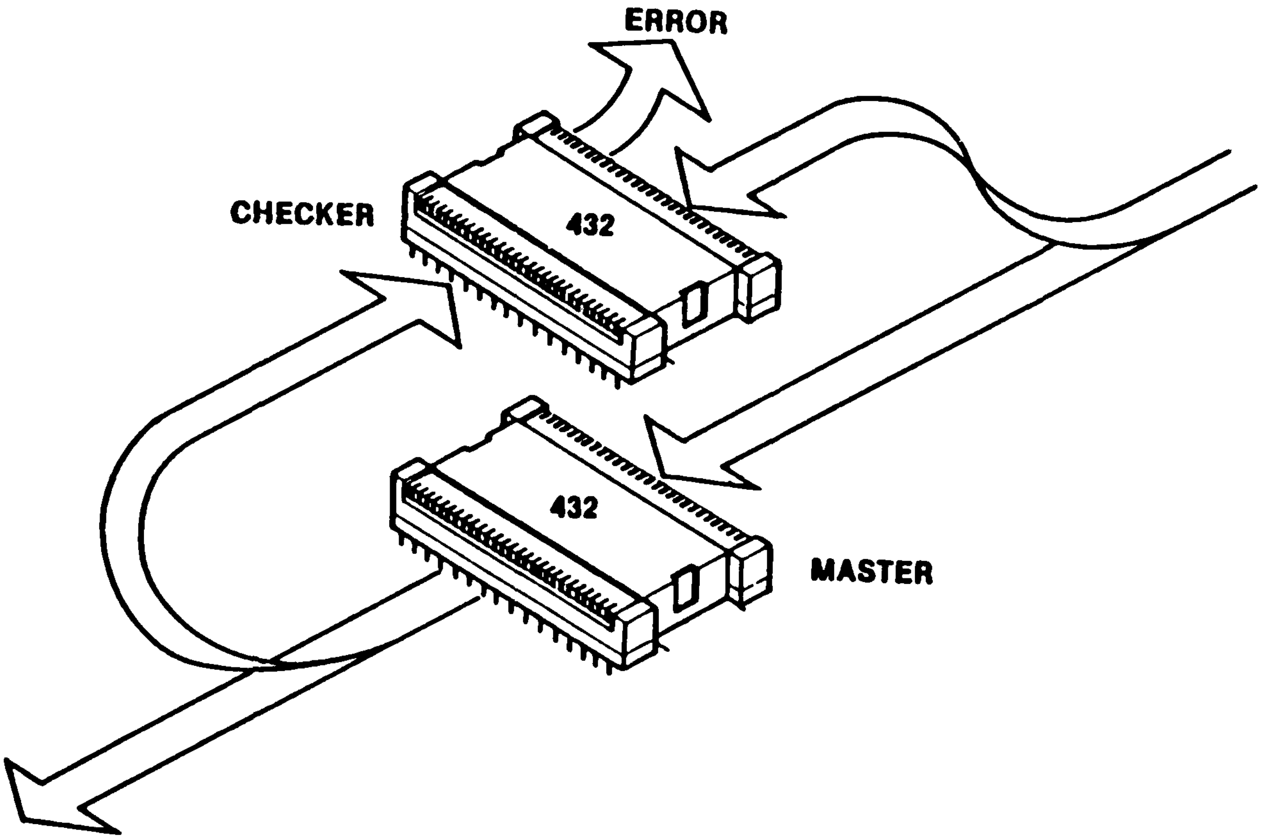


Figure 5-9 Hardware Fault Detection

- Faster Hardware Performance

- Increases in Concurrent Software

- Increases in Solution Complex Timing Problems

- Transparent Higher Order Languages (HOL) With Hardware-Related Timing Mechanisms

- Transaction-Driven Systems

- Highly-Distributed Intelligence with Independent Decision Making

5.2 Software Technologies

As the scope of embedded distributed processing systems continues to grow, their productivity becomes increasingly important. Software engineering is crucial to meeting that need. More manageable approaches to software development are essential. Important new aspects of software engineering address the following areas:

- Improved tools for software developments

- Improved facilities for software development
- Powerful specification and implementation languages
- Effective human interfaces with software
- Software performance engineering
- Appropriate modularizations
- Enhanced adaptability and reusability of modules
- Assurance of correctness.

Present embedded distributed processing systems are being implemented with VLSI components. Such components require their own internal operating systems which are usually implemented in microcode. Software development tools for these VLSI components can be characterized as inadequate. This characterization is not new. Users of large mainframes have long encountered inadequacies when they attempt to build integrated portfolios of applications programs upon a centralized data base. The availability of tools within the large mainframe environment has been scarce. In the VLSI environment such tools are virtually non-existent. Despite such obvious tool shortages embedded distributed processing systems continue to grow and expand. Underlying this growth and expansion is a universal desire to increase productivity. The problem comes in realizing that desire. The objective is difficult to achieve. Some degree of performance management must be applied in the initial stages of the design phase. Furthermore, performance management issues must continually be addressed throughout the remaining software development lifecycle phases. Subsequent sections will address the new aspects of software engineering which have been enumerated above in the present section.

5.2.1 Improved Tools for Software Development - Set(s) of Tools Covering Entire Lifecycle

The Ada Programming Support Environment (APSE) provides its own set of tools; i.e., compiler, debugger, linker-loader, editor, run controller, and configuration manager. With the growing use of object-oriented modularization such tools are not sufficient. Additional ones with carefully defined links to each phase of the software system lifecycle are required. Additional simulation tools would help. Examples would include simulators to simplify feasibility analyses, requirements languages, software specification languages, design languages, and static analysis. Most helpful would be formal verification tools, testing tools, change impact analyzers, and optimizers. Management aids for planning and control are also needed.

More than a single set of tools covering the entire lifecycle remains a distinct possibility. No single methodology seems to be emerging which means divergence might require several tool sets. Such divergence is not in concert with standardization. Consequently, unless a methodology offers an important feature which is unique, it should not be used as justification for an independent tool set.

Many individual tools establish development requirements of their own. Despite such need the subsequent concentration should be toward the complete tool set applicable throughout the lifecycle phases. The obvious benefit would be the reduction of errors but greater continuity would also be evident between the phases.

Several efforts have already been undertaken to develop a toolset for the entire lifecycle. One such effort is the Unix Programmer's Workbench (PWB) which possesses tools with crude compatibility. That compatibility derives from the byte-string nature of all Unix files which enables any tool to read the output from any other tool. Meaningful programming is another matter. Another effort is Maestro which exhibits clear compatibility as well as clear incompatibilities. Neither Unix nor Maestro contains the full spectrum of desired tools. Other efforts include the development environment CADES by ICL, USE by UCSF, Gandalf by CMU, and DREAM by the University of Colorado at Boulder.

5.2.2 Improved Facilities for Software Development - Programmer Workstation

The arrival of VLSI circuitry enables processing power and memory to be consolidated locally for use by programmers. The concept is to provide advanced graphics and displays in a single unit called a programmer workstation. These multi-media, multi-screen stations can provide powerful programmer/computer interfaces which can increase programmer productivity. One example of this approach is the SPICE workstation at Carnegie-Mellon University.

Workstations can provide an interface between the programmers and software engineers with their respective computer systems. The centralized computational power provides the data base management capability. Modularity of both hardware and software allows modification to match individual programmer need and the installation of updated technology. Standardized features and interfaces within the workstation can reduce training time for programmers assigned under new projects.

Considerable research into workstations is needed. Low cost configurations with appropriate modularity must be combined for ease of use by the programmer. Workstation software has yet to be established. Such software must be modified easily, portable, and capable of rapid installation.

Current interest in local area networks has heightened the interest in programmer workstations. Groups in human factors research and standardization are also interested in such workstations.

Many types of workstations are currently under development. Most prominent is SPICE at the Carnegie-Mellon University. Another is being developed by Xerox at its Palo Alto Research Center. The National Science Foundation is sponsoring Project Quanta at Purdue University to generate a problem solving environment.

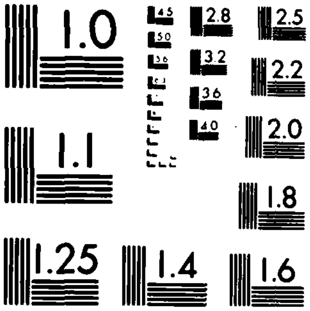
5.2.3 Powerful Specification and Implementation Languages - Configurable, Highly Parallel Computers

Under von Neumann architecture used by uniprocessors, computer functionality can be changed simply by changing programs. This ability to change has become so familiar that it is now considered to be obvious and is seldom discussed. Structured programming has produced a top-down methodology ideally suited for this uniprocessor architecture. However, programs can be viewed from a variety of directions. In their most basic form they simply are sequences of operations on a group of data structures. Consequently, programs can be typified by two sets: 1) a set of data structures and 2) a set of operations on those data structures. Obviously such a view of programs does not necessarily imply von Neumann architecture. Furthermore, the top-down methodology of structured programming does not enjoy its previously favored status. This basic and more general view of programs was not important until computers were configured around data structures and their operations. With the advent of VLSI circuitry that point has now been reached.

Currently VLSI circuit technology provides the potential for highly parallel computers which do not rely on von Neumann architecture. These devices have parallel functionality which can be changed by changing programs operating in parallel. The original approach used by structured programming needs modification. In its place is an object-oriented modularization which emphasizes data structures. Each data structure is carefully delineated and the operations allowed on that data structure is precisely defined. The data structures themselves are strongly typed as are their allowable operations. Each module is handled as if it were a single entity.

VLSI circuit technology raises the following issues which must be addressed.

AD-A137 686 DISTRIBUTED PROCESSING TOOLS DEFINITION VOLUME 1 2/3
HARDWARE AND SOFTWARE TE..(U) GENERAL DYNAMICS FORT
WORTH TX DATA SYSTEMS DIV H C CONN ET AL. JUN 83
UNCLASSIFIED RADC-TR-83-107-VOL-1 F30602-81-C-0142 F/G 9/2 NL


1.0
4.5
5.0
5.6
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

- Should algorithmically specialized processors be built focusing on computationally intensive problems?

- How should alternative architectures be evaluated?
- How should alternative software approaches be compared?

Several efforts are underway to examine the impact of configurable, highly parallel computers. One such effort is under Dr. Lawrence Synder of Purdue University under the sponsorship of the Office of Naval Research. Another is under Dr. Ron Krutz at Carnegie-Mellon University under the sponsorship of the VHSIC program. The impact of non-von Neumann architecture is not sufficiently known as far as software is concerned. Alternative software approaches become probabilistic under highly parallel configurations. Testbeds capable of comparing data structures and data structure operators need to be implemented immediately. Otherwise the problems will be addressed after they occur and under situations of duress.

5.2.4 Effective Human Interfaces with Software - Presentation and Manipulation

Human factors are important in achieving the best interface between machines and software engineers. These factors relate to the characteristics of problems being solved as well as the tasks being performed. Although such factors are not part of an automated program development environment, their presence or absence impacts that environment. Under a human factors approach the resources of local hardware, communications, and software tools are brought to bear on the basic needs of a programmer. The user interface is tailored to the semantics and usage patterns peculiar to each programmer.

The timesharing systems of today are not oriented to video and/or non-keyboard communications with high-bandwidth input/output. New systems must accommodate many different communications media, including audio, graphics, light pens, image processing, optical character recognition, and movable devices. As yet the necessary software to implement such highly interactive forms of human/computer interface has not been developed.

Several efforts are underway to improve the human factors associated with software engineering. Most are in the conceptual stage; e.g., the professional programmer based systems (PPBS) by DEC and the Automatic Software Generation System (ASGS) by John G. Rice. As concept becomes reality, the impact of improved human factors will complement software productivity.

5.2.5 Software Performance Engineering - Performance Management Techniques

Performance is one of the most important aspects of software quality. Among users of distributed processing services, it can be the difference between satisfaction and absolute rejection. High-performance internal support systems are vital to the routine operations of distributed networks. Without rapid response time, productivity is impacted. Extra time is required for implementation and extra effort is required to modify subsequent performance problems. Performance is not normally considered, but only when it becomes a problem. Performance management techniques must be applied from the initial design stage throughout the entire lifecycle.

Engineering for performance throughout the lifecycle has obvious advantages. The quality of a software product can be improved and productivity can be increased through such engineering. As a first analysis the following information is necessary:

- work-load specifications,
- software structure,
- execution environment,
- performance goal, and
- resource requirements.

The work-load specifications are derived from the users of distributed processing services. Software structure is established during the design phase. The execution environment anticipates a hardware configuration and an operating system. The performance goal is established by management in agreement with users. Resource requirements are derived projected usage levels. The relevance of results depends on the accuracy of information sighted previously.

Several efforts have been made to establish performance techniques. The longest standing such effort is under Dr. J. C. Browne at the University of

Texas and Information Research Associates. Under sponsorship of NASA Langley Dr. Browne has implemented a strategy to analyze software performance called ADEPT, i.e., A Design-based Evaluation and Prediction Technique. Extensions to ADEPT have been made and have been included in PAWS, a Performance Analyst Workbench System. Much remains to be done. More extensions need to be made if recent VSHIC architecture and software advances are to be accommodated. The impacts of non-deterministic, transaction-driven VLSI configurations are little understood. To avoid massive performance problems in the future the present trends must be completely understood in terms of performance management. To do otherwise would be unconscionable.

5.2.6 Appropriate Modularizations - Object-Oriented Modularization

Although the term object-oriented is new, its concept is not. Over a decade ago structured programming evolved a methodology which could produce partially independent modules of programming statements. The approach was top-down with semi-independent modules being broken down into submodules until program statements are eventually produced. Under an object orientation the characteristics of such a process change. Related programming statements are grouped around like data structures. The data structure is emphasized and the operations which can be allowed are carefully delineated. These structures are strongly typed as are their allowable operations. Data modules are handled as if they were single entities. The number of data types available are patterned to the needs of individual programs. The approach is bottom-up with data structures being combined to produce even more complex data structures. The structured programming and object-oriented techniques complement each other.

Object-oriented techniques have remained conceptual in scope since implementation requires support from a programming language. The strong typing capability complicates the language facilities. Each type of definition must support "visible" as well as "hidden" parts. Unless some parts remain "hidden" every user would be able to modify each definition. Consequently, the goal of an object-oriented implementation is to limit users to "visible" parts within definitions. This limited access has important implications. None of the popular languages implement such a concept with the exception of Ada. However, no validated Ada compiler has yet been produced. Complete implementations of object-oriented techniques await the arrival of these Ada compilers.

Ada packages for embedded distributed systems should receive special emphasis. The standardization of signal processing and navigational algorithms are distinct possibilities. Such standardizations have impact in command/communication/control systems. Packages for graphics can address line drawing, sub-screen manipulation, character manipulation, three-dimensional manipulation, shading, sealing, etc.

Proposed standards for certain usage areas such as data base management and graphics already exist. Other candidates for software standardization are already underway. The primary benefit to object-oriented modularization is the exploitation of commonality between various embedded computer systems. The resulting emphasis will be for rigid definition of language, portable compilers, and special language constructs for packaged software.

5.2.7 Enhanced Adaptability and Reusability of Modules - Integrated Software Support Environment

System adaptability is the ease with which changes can be made. At one extreme are systems with many capabilities which are fixed and not subject to change. The other extreme are systems with few features which are easy to modify. Such systems are good items for software toolkits. However, they do have problems. They must be recombined to fit the situation in which they are used. Depending upon the process, the recombination can be formidable. Beginning programmers find systems with many fixed capabilities easier to implement. Experts prefer systems with few, easily-modified features. Such capability for modification is offered by the integrated software support environment. Beginning programmers must undergo rigorous training in order to use it effectively. The integrated software support environment will evolve from the Ada Programming Support Environment. The environment should be easy to learn and equally easy to use. The thrust of such an environment is the generation of a compatible tool set covering the entire lifecycle. This thrust provides a framework for other thrusts.

The greatest potential exists for synergy between tools and in the cumulative improvement in production. The concept of monitoring accesses to specific tools is the beginning of a toolkit optimization process. The most frequently used tools can be cached for quick access. Furthermore, the more popular a tool becomes, the more general its access should be. The concept of reusable generic tools awaits the implementation of an integrated software support environment.

The foremost effort to implement an integrated software support environment is associated with the implementation of the Ada Programming Support Environment. A layer of structure will be needed between the APSE and sets of individual tools. Such a layer should support other languages besides Ada. It should provide standards for combined and hidden invocation, maintain data structures, and manage multiple representations. Within such an environment, different sets of tools based upon different methodologies can evolve.

5.2.8 Assurance of Correctness - Automated Verification Systems

The sooner an error is detected, the less it costs to repair. The activity associated with error detection is termed verification. Such activity validates the result of each successive step in the software development cycle. Validity is established by verifying the intent of the previous step has been satisfied by the results of the present step. The objective of an automated verification system is to detect and correct errors as rapidly as possible. New theory needs to be developed concerning the points at which various types of errors can be committed and detected. Practical methods need to be constructed for verification and detection. Errors obviously occur in all aspects of software development including requirements, design, and documentation. Knowing the earliest theoretical point at which verification can be done would help. If that earliest point could be established, the form of the verification itself could be determined. Once determined the verification might possibly be automated which would present the best of all possible worlds.

A number of efforts have already been undertaken to verify statements of requirements and design. Most prominent has been the University of Michigan and its Problem Statement Language/Problem Statement Analyzer (PSL/PSA). The approach acknowledges the best design and best code in the world will not do the job if the user requirements are not adequately stated. Without proper requirements definition, structured design and structured programming help disaster arrive more quickly. Consequently, PSL/PSA concentrates on the documentation associated with requirements definition and the difficulty of producing and managing manually generated documentation. The techniques used by PSL/PSA have been extended and revised by efforts originating elsewhere. These other efforts include TRW, Boeing, Hughes, and the Army Ballistic Missile Defense Advanced Technology Center. Additional efforts have been undertaken by High Order Software and Computer Sciences Corporation.

6.0 OBJECT-ORIENTED MODULARIZATION

The concept of object-oriented modularization provides a high-level abstraction of the essential characteristics of embedded distributed processing systems, described in section 4.0, and the key features of the state-of-the-art hardware and software technologies, described in section 5.0. This abstraction plays a central role between these observed features and the specific requirements and techniques needed to support each lifecycle phase of embedded distributed processing systems that are described in section 7.0. It will be shown in section 7.0 that particular manifestations of object-oriented modularization are directly relatable to support of the embedded distributed processing system lifecycle phases. Hence, in this section 6.0 the concept of object-oriented modularization will be introduced, developed, and explained. Subsequent subparagraphs of this section will (1) indicate the relationship of object-oriented modularization to abstract data structures and information hiding, (2) extend the object-oriented modularization abstraction to hardware and software for embedded distributed processing systems, and, finally, (3) discuss the benefits of this viewpoint to embedded distributed processing systems.

6.1 Description of Object-Oriented Modularization

This section begins by defining the terms object and modularization; then describes object-oriented modularization, and finally concludes with an example contrasting object-oriented modularization to conventional modularization.

An object is an entity that contains information in an organized manner. This definition is purposefully general to enable its application to both hardware and software. For example, in software an object can be thought of as a data structure; viz, a simple variable or an array or a complex record. In hardware an object could be a register, an I/O buffer, or even a VLSI component of a larger system. An object has three additional characteristics that permit segregation of similar objects.

(1) Each object has defined for it a set of operations, that manipulates the contained information.

(2) Each object can be addressed (referenced) as a whole.

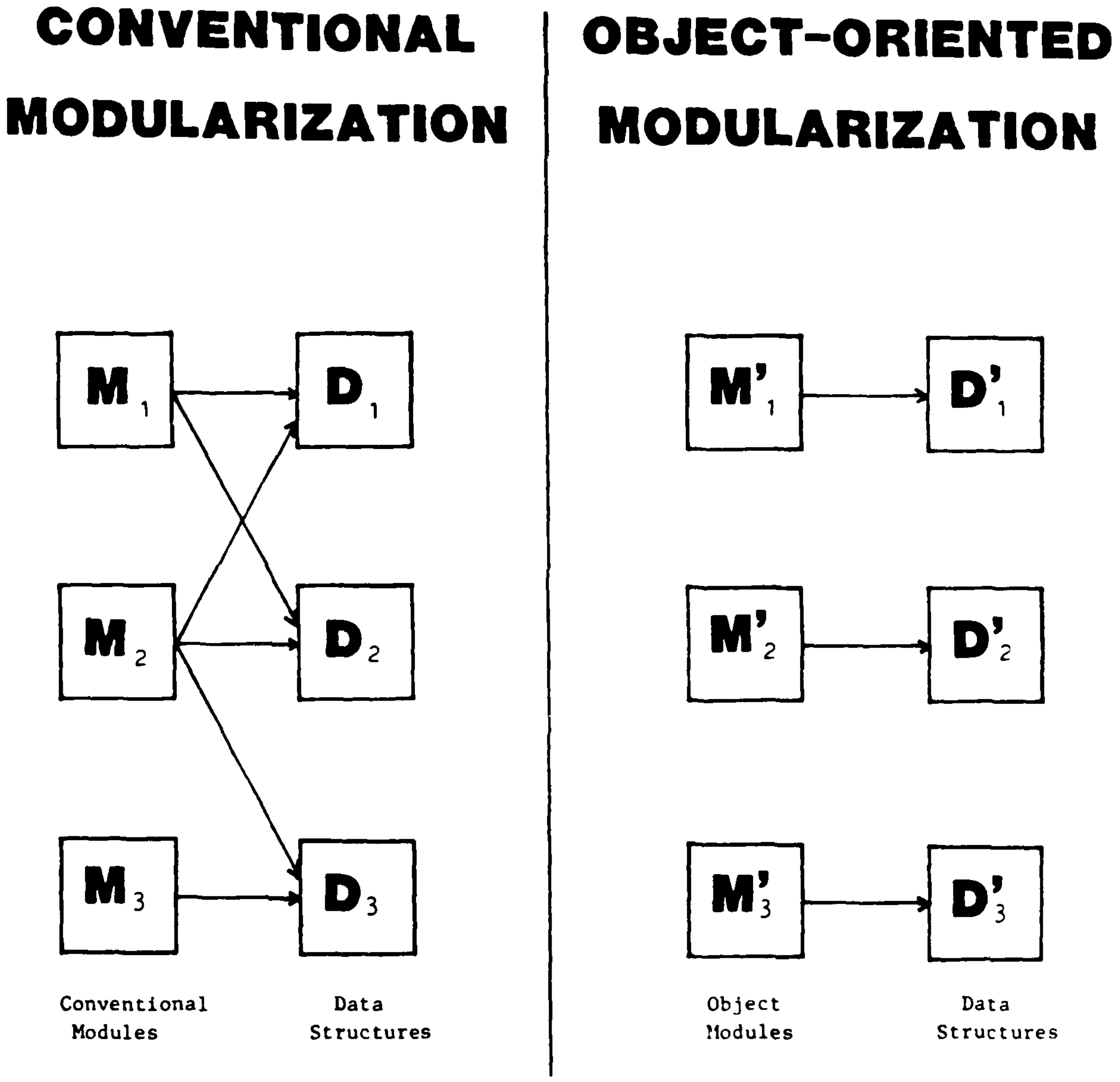
(3) Each object has a label that tells the object's type.

Objects are a very useful concept in dealing with distributed systems and increasingly sophisticated hardware designs. For example, communications ports, schedulers, and support software packages can all be considered as objects. This viewpoint creates a unified framework for the discussion of requirements for embedded distributed processing systems. Previously, communication ports, schedulers, and support software packages were considered as separate entities - as being inherently different. Objects provide a higher level, consistent method for analysis, design, and implementation of these kinds of entities for embedded distributed systems.

Modularization is traditionally viewed as the partitioning of the hardware/software task based upon a stated criterion. There are many ways to modularize; by function, by interface communications, by priority, etc. This traditional concept of modularization can be directly extended to include "modularization by object". Here, the basis for the partitioning of the tasks is the creation of objects. Modularization is now more than a simple partitioning; it is the recognition and assignment of a particular responsibility associated with the object. Modularity is now raised to a higher-level commensurate with the object concept.

Combining these two definitions, object-oriented modularization is the segmentation of the hardware/software task based upon the responsibility and domain of extent of the identified objects. The key feature of object-oriented modularization is that each object executes its responsibility without the need to know of the details of the internal structure of other objects. The objects communicate amongst themselves in a very well defined manner. In an object-oriented modularized system, the internal representation of any module could change, and the other object modules would not have to be changed. This feature is a great benefit in the software maintenance environment. As an example contrasting conventional modularization with object-oriented modularization, consider Figure 6-1.

The conventional modules (M's) could, for example, represent major processing steps and the data structures (D's) are accessed by the modules. In conventional modularization, it is possible for a particular data structure, for example D1, to be accessed by more than one module. The disadvantage of conventional modularization is that whenever a module or data structure is changed, several other modules or data structures may also have to be changed. This single fact decreases the system reliability. However, in object-oriented modularization, the central theme is a one-to-one correspondence between object modules and their data structure. A change in a module or data structure affects only that module or data structure - no others. The internal details of object modules and data structures are shielded from other system elements. The resulting benefits are easier maintenance and improved reliability.



Arrows indicate data structures directly manipulated by the module.

Figure 6-1 Conventional Modularization Versus Object-Oriented Modularization

6.2 Relationship of Object-Oriented Modularization to Abstract Data Structures and Information Hiding

An abstract data structure defines a class of attributes that is completely characterized by the operations available on those attributes. The goal of an abstract data structure is to permit the expression of relevant details and the suppression of irrelevant details. In modern higher order languages it is desirable to provide the capability for user defined abstract data structures. This capability eases the programming task, makes the resulting code easier to understand, and provides a mechanism for the user to more easily communicate in a natural manner. Consequently, system reliability is enhanced. The particular implementation of the abstract data structure in a language is accomplished by the procedures and utilities that embody the defined operations.

The principle axiom of the Theory of Information Hiding as proposed by D. L. Parnas is that a software design methodology should shield information developed at one level of design from its use on another level of design. Consider two procedures A and B which do not reference each other directly. The Theory of Information Hiding states that the fundamental output (considered as level 1) of procedure A should not depend on the detailed implementation (considered as level 2) of procedure B. For example, if procedure B implements a stack operation by using a linked list and if the output of procedure A is to return the top item of the stack, then procedure A should not depend upon procedure B's linked list implementation. Procedure A only needs to know that procedure B provides a stack; how the stack is implemented is unimportant to procedure A. The detailed information of procedure B is "hidden" from procedure A. The benefit of this methodology is that if the implementation of procedure B changes, for example the linked list is replaced by an array or utilization of PUSH and POP hardware capabilities, procedure A remains unchanged.

The object-oriented modularization described in section 6.1 incorporates the main features of both abstract data structures and information hiding. Recall that each object has a set of operations defined for it that manipulates its information. This aspect directly draws from the domain of abstract data structures. As shown in Figure 6-1, the key feature of object-oriented modularization is to partition module, data structure pairs from other module, data structure pairs and thereby minimize the access of a particular data structure by more than one module. The effect is the same as information hiding; namely, changes to a particular module do not affect other modules. Hence, the object-oriented modularization is in concert with the current theories of abstract data structures and information hiding.

Object-oriented modularization is more than simply the amalgamation of abstract data structures and information hiding. Both of these concepts have traditionally been limited to software. However, the object-oriented modularization can also be applied to hardware elements. The key benefit here is that for embedded distributed processing system, object-oriented modularization provides a unifying concept for analysis and design at the hardware/software system level.

6.3 Example Application of Object-Oriented Modularization to an Embedded Distributed Processing System

As a concrete example of the application of the object-oriented modularization to an actual embedded distributed processing system, we shall examine the Intel iAPX432 System. The iAPX432 is a new product from Intel. It is a high technology device that is anticipated to have a profound impact upon the design of future embedded distributed processing systems. The design of the iAPX432 itself is an outstanding example of object-oriented modularization. For these reasons, it has been chosen to illustrate object-oriented modularization.

Figure 6-2 shows the top-level architecture of a typical product that would contain an embedded iAPX432 system to accomplish the product's application task. In addition, the most important objects are indicated, and these objects will be discussed in greater detail in the following subparagraphs.

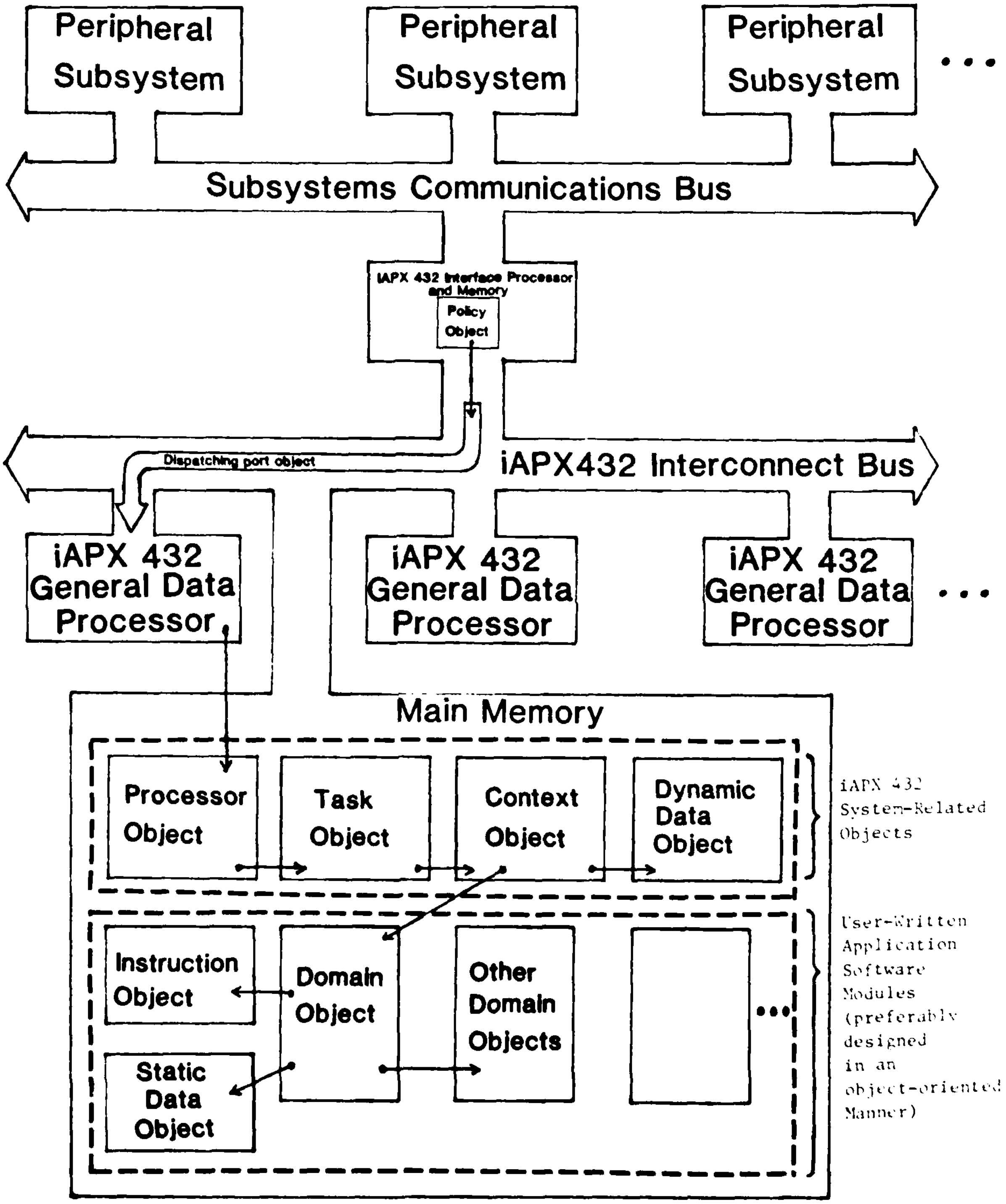

Peripheral Subsystem
Peripheral Subsystem
Peripheral Subsystem
• • •
Subsystems Communications Bus
iAPX 432 Interface Processor and Memory
Policy Object
Dispatching port object
iAPX432 Interconnect Bus
iAPX 432 General Data Processor
iAPX 432 General Data Processor
iAPX 432 General Data Processor
• • •
Main Memory
Processor Object
Task Object
Context Object
Dynamic Data Object
iAPX 432 System-Related Objects
Instruction Object
Domain Object
Other Domain Objects
•••
Static Data Object
User-Written Application Software Modules (preferably designed in an object-oriented Manner)

Figure 6-2 Top-Level Architecture and Objects in the iAPX432 System

The peripheral subsystems represent sensors, input/output devices, or functional units (e.g. inertial navigators, radars, line drivers, etc.), and these subsystems communicate via a common communications bus. The peripheral subsystems could also contain locally accessible memory. The interface processor is an intelligent link between the subsystems communications bus and the iAPX432 interconnect bus. The interface processor permits data transfer from the peripheral subsystems to iAPX432 main memory, and the interface processor contains the software for determining task execution priorities, scheduling, and dispatching (i.e., the policy object). Any number of iAPX432 general data processors can be put onto the interconnect bus and access main memory. Memory contention is resolved via the object-oriented modularization mechanisms in the iAPX432. The small arrow (--->) represents a reference of one object to another object. (Recall that two of the characteristics of an object are that it can be referenced as a whole and it has a label.) One object can access another object if and only if it contains an object reference. This mechanism provides protection of objects from other objects that do not have explicit access authorization.

The processor, task, context, and dynamic data objects are new concepts present in the iAPX432. The linkage of these objects minimizes erroneous access to software modules and provides a highly flexible capability that supports the dynamic environment needed for distributed processing. All of these objects are recognized by the iAPX432 hardware; hence, these object linkages do not degrade throughput. The application software modules reside, of course, in main memory. These modules are designed and coded by the user to accomplish the product's application task. The application modules are developed using standard practices of software engineering; preferably using the object-oriented modularization techniques described in section 6.1. The user application software (i.e. the domain objects) represents the bulk of main memory usage. The processor, task, context, and dynamic data objects are very minor tasks. The iAPX432 object-oriented modularization provides flexibility and protection for the user application software; it does not burden the software development activity.

6.3.1 Policy Object

The policy object has the responsibility of determining how the tasks to be executed are shared amongst the general data processors. Some typical criteria for policy decision are first-come-first-served, round-robin, priority, deadline, etc. Because the policy object must be tailored to the particular application, the policy object is implemented in software and resides in the interface processor. The scheduling and dispatching of tasks are accomplished in a manner consistent with the policy object. However, in order to increase throughput, the scheduling and dispatching functions are supported by hardware. The policy object contains a reference to the dispatching port object.

6.3.2 Dispatching Port Object

The dispatching port object is a hardware-recognized object that provides communications between the policy object and a particular general data processor. If all the processors are busy, the policy object can queue tasks in the dispatching port object awaiting a processor to become available. Likewise, if a processor is idle, it can wait at a dispatching port object for a task. Special hardware instructions SEND and RECEIVE provide rapid dispatching of tasks.

6.3.3 Processor Object

The processor object contains the information pertinent to a particular general data processor at each instance of time. Each processor has a processor object. The processor object contains information such as the processor status (e.g. running or waiting), diagnostic and machine check information, and an *object reference to the particular* task being executed. The object reference to the task being executed dynamically changes as the task being executed changes. Figure 6-3 shows an example of three processor objects at two points in time. At time t1, processor 1 is executing task B, processor 2 is halted, and processor 3 is executing task A. At a later time t2, processor 1 is still executing task B, processor 2 is now executing task C, and processor 3 is now halted. The processor objects dynamically changes as the tasks and processors are dispatched by the policy object. However, at each instance of time the status of each processor is completely determined by interrogation of its processor object. The processor object is referenced by the dispatching port object.

6.3.4 Task Object

The task object is the data structure that contains information about the task being executed; for example, the status of the task (running or waiting), how the task should be scheduled, and an object reference to the particular instance of the task being executed. The task objective is also a hardware recognized object to speed processing. Figure 6-4 shows how the task object changes as two tasks take turns running on a single processor. At time t1, task A is running and task B is waiting. At a later time t2, task A is now waiting and task B is now running.

6.3.5 Context Object and Dynamic Object

A task object can have more than one instance (or copy) of the procedures in memory at the same time; however, only one copy can be executing at a time in a particular general data processor. This

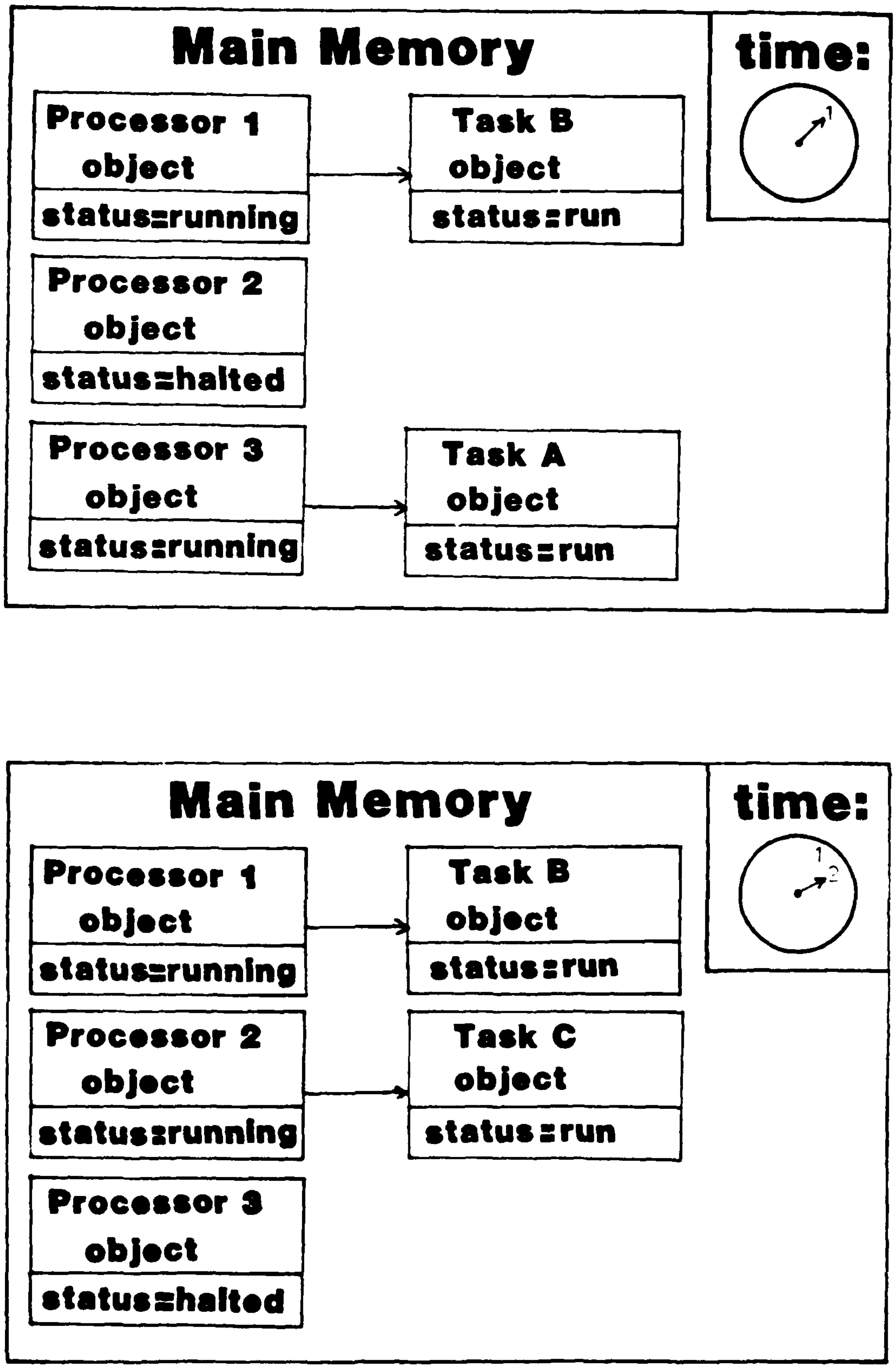


Figure 6-3 Processor Objects Snapshots at Two Instances in Time.

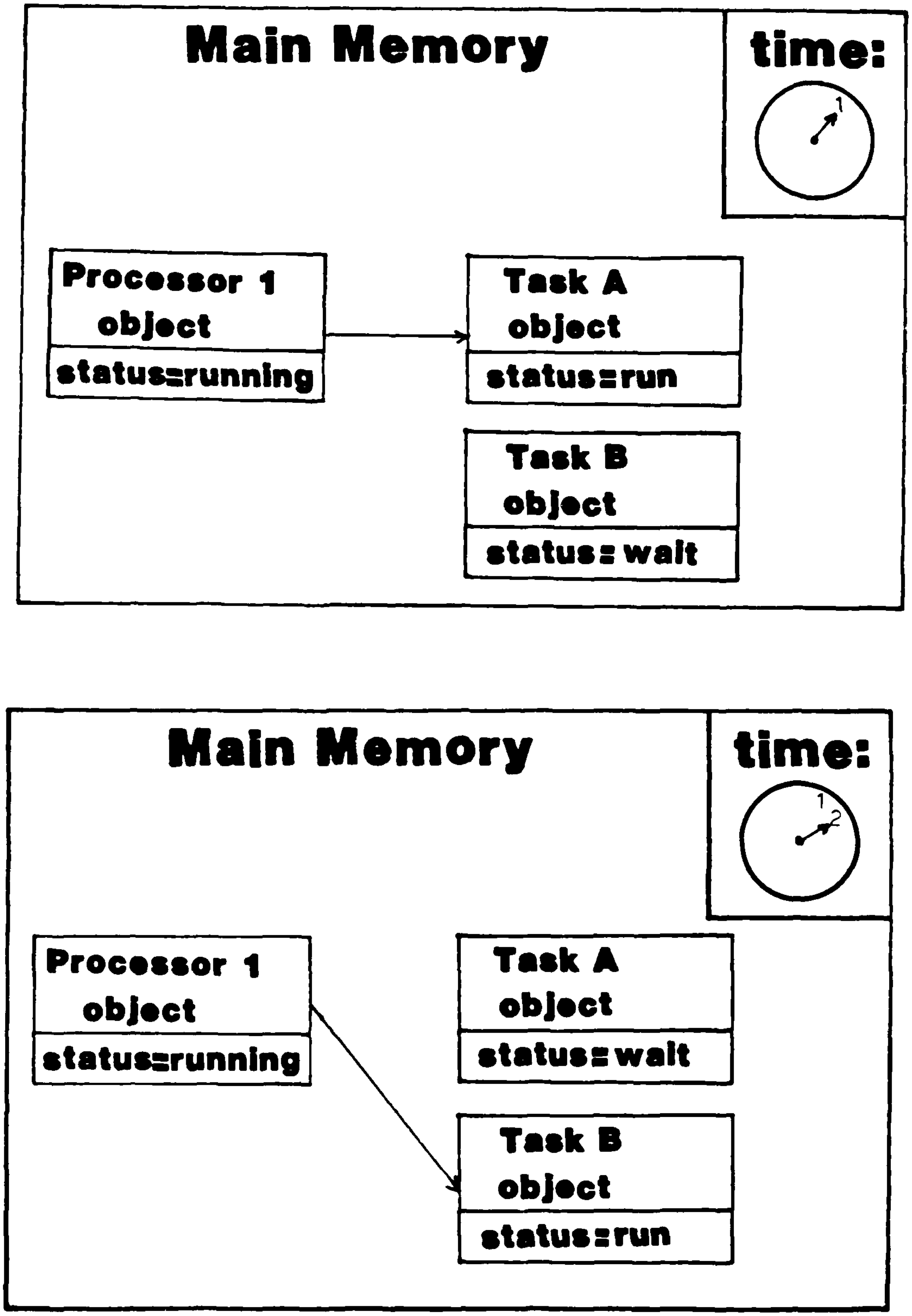
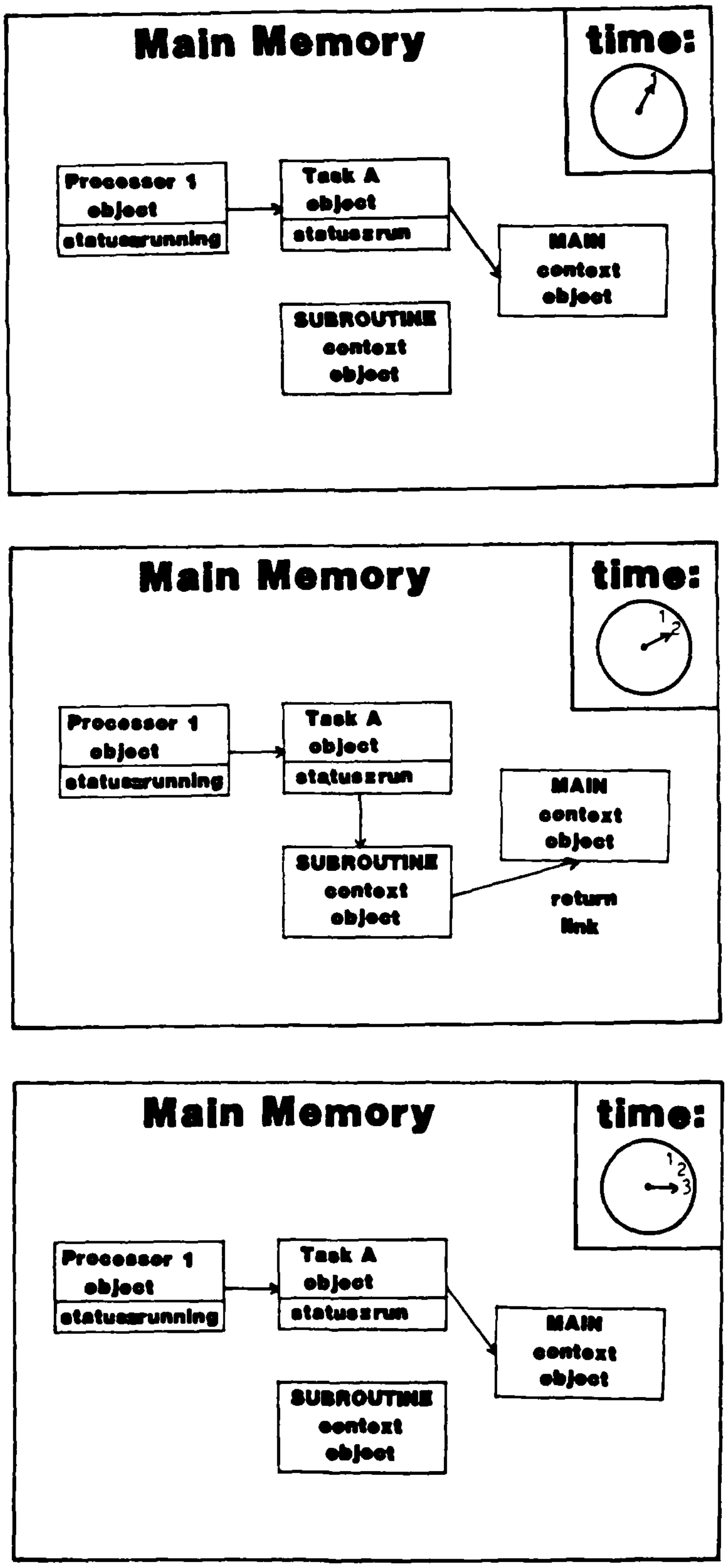


Figure 6-4 Task Objects Snapshots at Two Instances in Time.

situation commonly occurs for re-entrant procedures. A context object is the data structure that contains information pertinent to the particular instance of the task that is being executed. For example, the data contained in the context object includes an instruction pointer for this context, a stack pointer for this context, a return link to the task object, and references to all objects that can be accessed by this context. The context object is the fundamental vehicle for access of a particular instance of the task. Figure 6-5 shows how the context object changes in a typical subroutine calling sequence.

6.3.6 Domain, Instruction, and Static Objects

The domain object is a list of all the static object references to other applications modules, executable instructions, and static data. The object-oriented modularization is realized in the applications area by the structure of the modules and data structures written by the applications software engineer in accordance with the discussions presented in section 6.1. The domain object contains a reference to all of these objects. The instruction object contains only executable



MAIN executing; before call

SUBROUTINE executing; after call

MAIN executing; after return

Figure 6-5 Changes in the Context Object During a Subroutine Calling Sequence.

instructions. The general data processor only uses the instruction object as a source of instructions to fetch and execute. The static data object contains static data that remains in memory after a particular context execution is complete. The instruction and static data objects are typically the end leaves of the iAPX432 object tree.

6.3.7 Summary of Benefits

The object-oriented modularization as exemplified by the iAPX432 has three distinct benefits. First, the object-oriented modularization is used in both the hardware and software. This approach provides a unified viewpoint to the entire iAPX432 system. It is anticipated that future embedded distributed processing systems must address the hardware and software as a total system. Object-oriented modularization provides a convenient, consistent, and flexible systems methodology. Secondly, frequently used objects (e.g., dispatching port, processor, task, and context objects) are supported by hardware capabilities to provide improved performance. Finally, the use of object references provides a flexible system that still incorporates careful control of object access. The control of object access is a vitally important problem in embedded distributed processing systems. The iAPX432 illustrates many of the problems of embedded distributed processing systems, and it is felt that the iAPX432 offers many viable solutions to these problems.

6.4 Benefit of Object-Oriented modularization to Embedded Distributed Processing Systems

Embedded distributed processing systems are typically utilized in real-time, process control applications. These applications require a careful orchestration of the hardware and software; consequently, the hardware and software tasks must be viewed from a total systems standpoint. Object-oriented modularization as described in section 6 provides such a system level viewpoint. It is proposed that object-oriented modularization provide a central theme for the analysis, design, and implementation of embedded distributed processing systems. In section 7 we will explore specific manifestations of this object-oriented modularization theme and relate them to requirements and techniques to support the embedded distributed processing lifecycle phases.

7.0 IDENTIFICATION OF REQUIREMENTS AND TECHNIQUES FOR SUPPORT OF EDPS LIFECYCLE PHASES

Because of rapid advances in hardware technology both large and small computing systems will grow at an ever-increasing rate. The effective use of such systems will depend upon the ability of people to develop effective software. This software development must be of high quality and low cost. Because people have grown more expensive than machines, automation has an important role to play. The key element in that automation is the development of effective software tools. Major issues associated with the development of these software tools are as follows:

- the tools must form an integrated system which supports software throughout its lifecycle
- the role of breadboard models in requirements analysis and design phases of software development must be fully supported, and
- these tools must be developed under conditions which assure their successful use.

Much progress has been made in program development tools. However, much remains to be done on system construction tools. Tools are needed now to support independent specification and implementation of software modules.

The introduction of distributed systems on a large scale brings new challenges to the development of software. The design, development, testing and use of these systems demand increased simulation testing and more analysis tools in software development. In many hardware development activities, the product design phase requires the development of breadboard versions to investigate the difficult issues of system construction. A great need exists to quickly assemble and test breadboard versions. Performance needs to be analyzed and software design solidified before implementation begins.

7.1 Specific Manifestations of Object-Oriented Modularizations

Under object-oriented modularization the process of programming is transformed into a generic activity. Related programming statements are grouped on the basis of data structures. Operations are defined in terms of their impact on specific data structures. Both data structures and operations are strongly typed as to whether they are permissible or not. Furthermore, access to specific data structures can be tightly controlled. Once access has been accomplished, permission to change the accessed data structure may or may not be granted. If permission is not granted, the data structure is considered to be private. If allowed, the structure is visible. Graduations of access are a powerful tool under object-oriented modularizations. Data files and programs themself are handled as if they were individual data structures. A bottom-up orientation is possible with data structures being combined to form more complex structures. This emphasis on data structures demands an appreciation based upon experience. Novice programmers will undoubtedly prefer the simplicity of individualized programs and file structures. To program generically, more rigor is required. Object-oriented modularizations have many manifestations. Their consequences are only now being understood. The following paragraphs only begin to document such manifestations. Undoubtedly many more manifestations will be added as experience is gained. Most importantly an effort to gain needed experience must start immediately.

7.1.1 Deterministic Versus Probabilistic Systems

Von Neumann architecture within a uniprocessor environment is typified by Figure 7-1. Obviously the configuration is comprised of five functional components connected by a systems bus. The bus is somewhat misleading since its operation is not straightfoward. It is a combination of three distinctly different buses itself. Those three parts are a control bus, an address bus, and a data bus. Each has its characteristic architecture. The control bus is obviously the agent of the Process Control function within the von Neumann architecture. It instructs the actions to be taken by all the other functions within the architecture. Input and Output functions either place data on the data bus or take data off the data bus. Of course what action is performed is under the direction of the control bus. The Arithmetic Logic Unit or ALU performs either the arithmetic or logic required by the Process Control function. The results are usually placed on the data bus for subsequent usage within the architecture; e.g., Output or Storage.

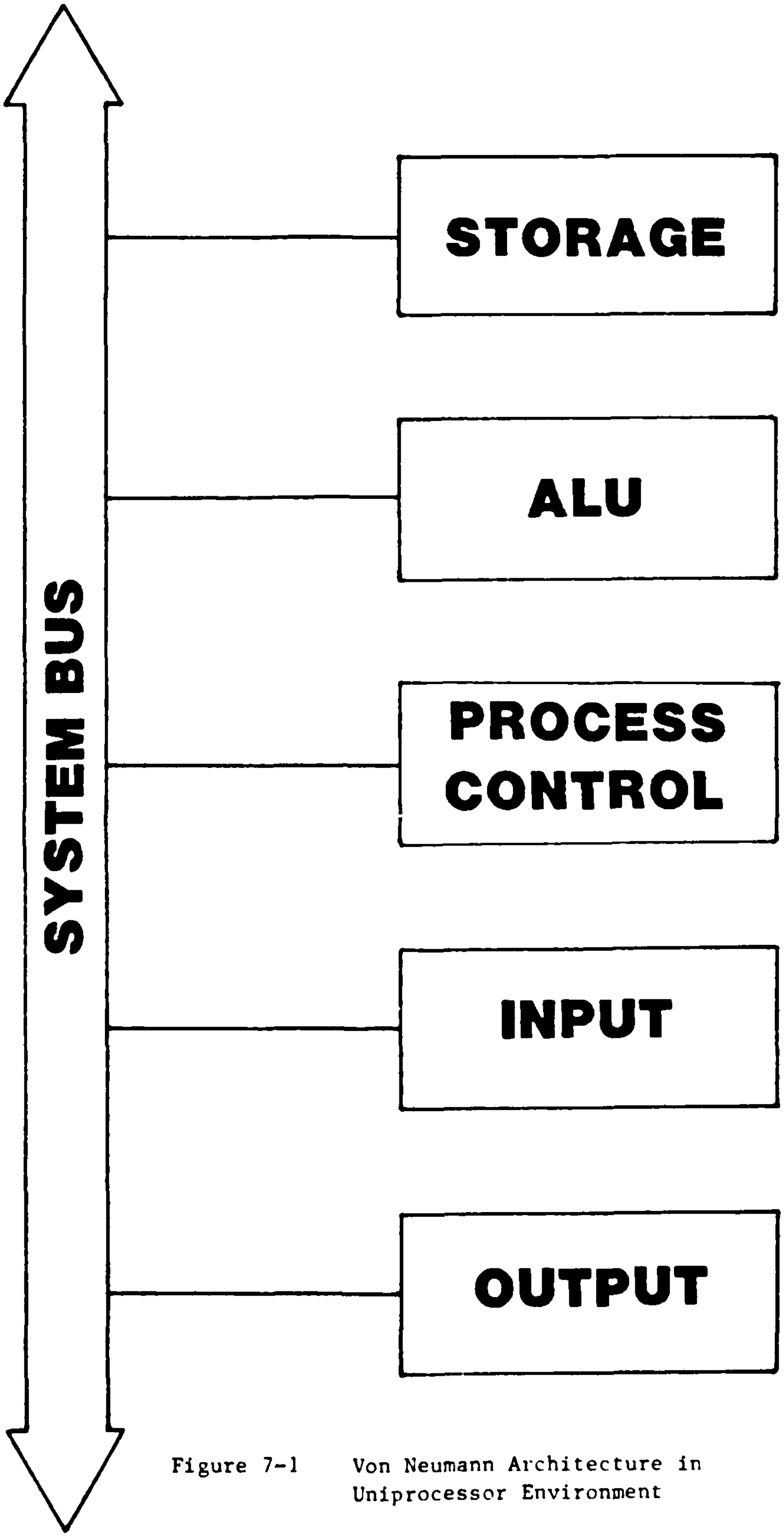


Figure 7-1 Von Neumann Architecture in Uniprocessor Environment

The Storage function either takes data from the data bus and places it in the location designated by the address bus or vice versa. The action taken is directed by the control bus. In summary the Systems Bus represents the concerted action of its three components.

Data flow between functions within a von Neumann architecture requires the use of registers. In effect such registers represent the beginning and ending points for the System Bus function. Since each register can be thought of as a "box" whose contents can be filled or emptied, the System Bus can be visualized as a postal service system. Each "box" is a mailbox in which letters are either delivered or dispatched. Such an analogy is valuable to illustrate certain architectural problems. When a letter is needed, it may not have arrived at its mailbox. Worse yet, the letter which was previously dispatched may not have been picked up and erroneously be misinterpreted as newly-arrived. In uniprocessor configurations which tend to be von Neumann in nature, the architectural problems are easy to avoid. The straightforward architecture illustrated previously can be expanded to exhibit the necessary mailboxes. Figure 7-2 illustrates this expanded von Neumann architecture.

Such expansions offer new architectural alternatives. The advances by Very-Large-Scale-Integration (VLSI) circuitry are exploring these alternatives. As a result, concurrent operations are becoming commonplace. Intelligence is being expanded to new architectural locations within von Neumann architecture. As an example, the mailboxes alluded to previously can assume their own intelligence. More appropriately they can be termed intelligent Interface Processors. Each such processor has its own von Neumann architecture with its own Input/Output being provided by either the System Bus or other von Neumann components. Furthermore, the actual computational process within a von Neumann architecture can be separated from the input/output process. In effect, two buses are introduced under such an arrangement: a system bus and a periphery bus. Figure 7-3 illustrates such an architecture. It is exemplified in the marketplace by the INTEL iAPX432 micromainframe. What emerges in such architectures is a layering of specialized buses. Figure 7-3 represents the architecture of a layered bus architecture.

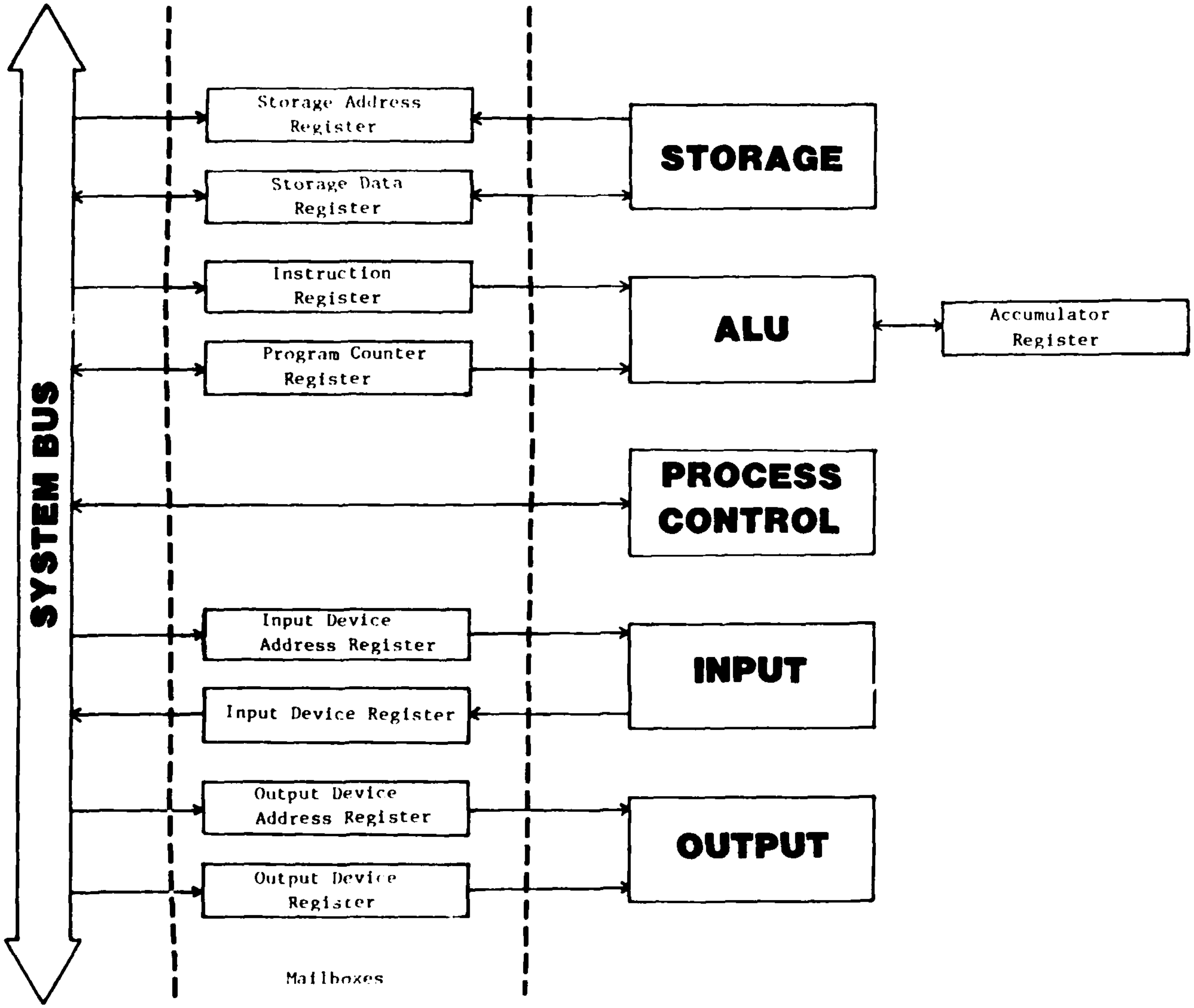


Figure 7-2 Von Neumann Architecture with Registers

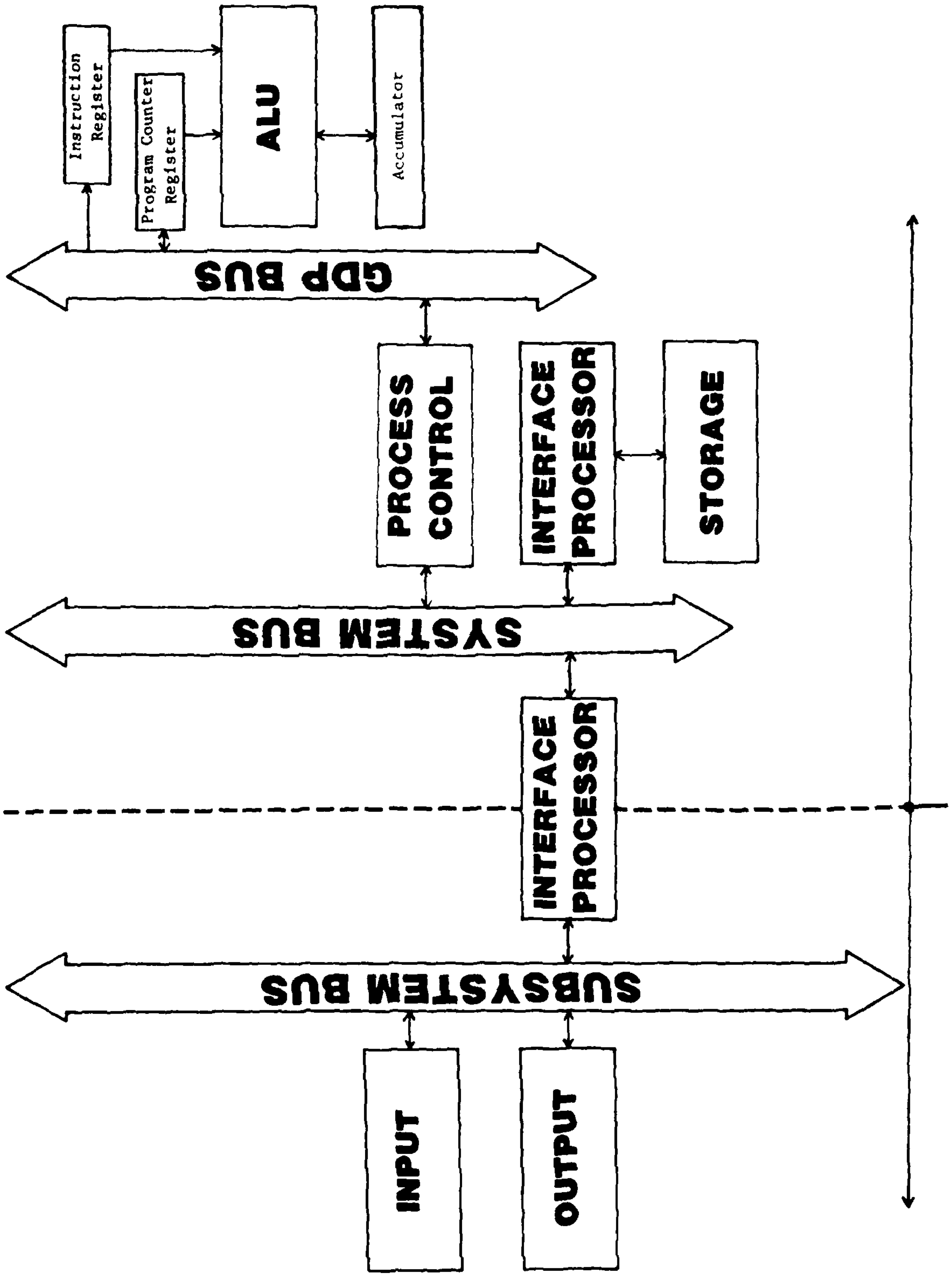


Figure 7-3 Layered Architecture of the iAPX432

The important property of layered bus architecture concerns its operation. The communication between layers becomes decidedly non-von Neumann or probabilistic. Such a phenomenon is enabled by the way the different buses are operating. Each is capable of operating independently of the other with communication occurring within mailboxes. At the topmost level the device appears to be interrupt-driven. However, at lower levels the device becomes transaction-driven which is the preferred architecture for distributed processing applications. Transactions occur independently of one another without an assumed interrupt schema. Such a philosophy has been adopted by the communications used with satellites and packet switching radio networks. New channel allocations have been implemented around open system, transaction-driven interconnect architectures. Projected advances in VLSI circuitry and Very-High-Speed-Integrated Circuitry (VHSIC) are adopting open system architectures to accommodate distributed systems. As a consequence, the operation is becoming more probabilistic. Various layers within the architecture can be operating independently of one another. Furthermore, components within a single interconnect level may also be operating independently of one another. To comprehend the operation of such configurations requires a knowledge of an underlying probability distribution. Not only must the transactions to be processed be known but their probability of occurrence must also be known. This requirements is new and very necessary in the emerging non-von Neumann architecture. The more layers of independently operating buses implemented, the more probabilistic subsequent operation becomes. Tools developed for von Neumann architecture are confronted with a non-von Neumann operation. The applicability of assumed von Neumann tools has yet to be established. Software tools are lagging behind VLSI and VHSIC developments. The situation needs to be corrected.

7.1.2 Global Timing

Advances within VLSI circuitry and the emerging VHSIC technology are enabling transaction-driven architectures to be implemented. In such architectures the predictability of time divided multiplexing is not available. Transactions either occur or they do not. Such operation accommodates a degree of spontaneity never before achieved.

When spontaneity is accommodated, a whole new approach must be applied. Transactions can overlay one another and compete for the same system resources. In the worst case transactions occur simultaneously with one another. Under such circumstances not only must transactions be processed but their overlap or coincidence must be recognized. In the case of either overlap or coincidence, the offending transactions must be reissued. If they are reissued, some means to assure they do not overlap or coincide a second time must be implemented.

Configuration throughputs have traditionally been modelled through the use of queuing networks. Such networks implement multiple job classes, mixed architectures, and hierarchical models. However, the underlying assumption remains von Neumann architecture which connotes a uniprocessor structure. As that architecture becomes more probabilistic and less von Neumann, the problem of timing becomes more acute. Characteristically the problem is either addressed directly or ignored completely. The easier solution is to ignore it completely which amounts to a "fix-it-later" attitude. By far the more rigorous approach is to confront it directly. Performance and global timing must be considered in the initial design stage and complicates subsequent software effort. The requirement for simulating different Poisson distributions is evident. Poisson mathematics is normally not addressed during the system lifecycle phases. However, Poisson distributions are mathematical tools available for the consideration of different probability density functions. Such functions underlie messages transmitted between independently operating bus structures within distributed architectures. As layered buses become more prevalent, the importance of the underlying message distribution functions is increased.

Presently available queuing networks can simulate multiple job classes, mixed architectures, and hierarchical models. However, each assumes an underlying Poisson distribution which does not vary. What is needed is the ability to vary the Poisson as well. When that variability is accomplished, the consequences of global timing can be appreciated. In the absence of such capability, purely software techniques are assumed to be valid with little or no justification. An example is the software technique called rendezvous within the Ada syntax definition. By virtue of its syntactic definition, such a technique may or may not accommodate subsequent operational distributions within the bus architecture of a particular system. In effect, by defining the rendezvous from a syntactic standpoint the whole issue of global timing is relegated to a "fix-it-later" approach. The current situation can be likened to the absence of ability to measure. Since global timing is not measured, the assumption is that it does not pose a problem. Until an architecture throttles itself there may not be a problem. But if the architecture ever does throttle, the problems are immense. They are much larger than necessary since their occurrence could have been avoided by including global timing as a design phase activity. The subsequent absence of problems should serve as justification for putting timing in the design phase within the system lifecycle phases.

7.1.3 Rapid Prototyping

System requirements have always been difficult to formulate. The difficulty is even encountered when similar automated applications are attempted. However, the similarity does provide information that ncrmally is not available during the system requirements phase. It enables a quick prototype to be constructed to aid in the development of additional system requirements. This alleviates some of the problems usually faced by users and analysts when they attempt to specify complete sets of requirements. However, the present situation concerning such prototyping is not clear. The proliferation of differing software support environments has obscured its usefulness. When software support environments begin to standardize their capability, the importance of rapid prototyping will increase. This standardization is scheduled to proceed under the Ada implementation effort with the DOD. Once the Ada Program Support Environment (APSE) is implemented, rapid prototyping can demonstrate a reduction in time needed to produce requirements as well as an improvement in their quality.

An underlying question in rapid prototyping is what can best be ignored by the prototype. Many design details seem to be extraneous during the requirements phase. As Ada techniques become more widely known, this question becomes even more subtle. Because of the syntactic approach taken to global timing considerations by Ada, the design details previously thought to be extraneous may, in fact, not be. A suspicion begins to grow that clever deferring by Ada definers may complicate subsequent rapid prototyping efforts. As an adjunct to this line of reasoning, the applicability of rapid prototyping itself may be changing. As Ada usage grows, the capability for rapid prototyping becomes far more important. Not only must Ada programs be written but their subsequent interaction with the Integrated Software Support Environment (ISSE) must be known completely prior to the design phase; i.e., during the requirements phase. Obviously the language of choice to implement a rapid prototyping system should be Ada. Furthermore, once implemented, the prototyper should become part of the APSE.

A parallel effort to the development of a rapid prototyper for an ISSE should be the capability for simulating the ISSE. Such simulation would provide numbers or measurements which could be used to determine if an application is feasible or not. If not feasible, the effort could stop at that point. If feasible, the requirements phase could be addressed through the use of a rapid prototyper.

Such an approach simplifies the design of a prototyper since that prototyper does not have to handle impossible situations.

The use of rapid prototyping promises to improve the quality of requirements analysis. Subsequent systems should demonstrate improved relevance and usefulness.

Presently many companies are claiming competence in rapid prototyping based upon their efforts in general data base management systems. Components of such systems which come into play are configuration managers and specialized data storage and retrieval mechanisms. In most instances the storage involves hierarchical structures containing simulation parameters which are subsequently compared to each other. Rapid prototyping capability is closely related to the verbs contained in the data base management systems. However, such expertise may not apply when the APSE is implemented. Much of the capability claimed within a general data base management system will be offered by the APSE which offsets some of their expertise claimed in rapid prototyping. In effect, Ada ushers in an entire new set of circumstances for rapid prototyping.

7.1.4 Static Analysis Techniques

Static analysis techniques refer to the validation and verification techniques in which the analyst examines software without executing it. In its general sense it applies to all types of software products like designs, specifications, etc. In its restricted sense it applies only to data flow through a program in an attempt to detect anomalies like references to uninitialized variables. Until static analysis can be made to work in the restricted sense, it can not work in a general sense. Consequently, the following comments are limited to the restricted sense.

The best known early work on static analysis has been done by Osterweil and Fosdick. They built a system called DAVE which analyzed Fortran programs for data flow anomalies. It detected possible references to uninitialized variables and the assignment of value to a variable not referenced by the remainder of a program. Several criticisms have been levelled at DAVE. First, it carries out data flow analysis one variable at a time rather than simultaneously for all variables. Second, it non-selectively prints out too much data. Third, it is too large which may, in fact, pertain to the use of Fortran. Fourth, it is too slow and requires too much computer time. However, these criticisms can be ameliorated by viewing DAVE as a prototype product of a research project as opposed to a production line tool.

The research project which produced DAVE has enabled Fosdick and Osterweil to study the relationship between data flow in static analysis and data flow in program optimization. They were the first to observe data flow algorithms developed for optimizations could also be used for static analysis. The basic idea is that variables can be in different states and that operations such as value assignment and referencing can change those states. The set theory resulting from their observations is useful to the extent it indicates how existing well defined and efficient data flow algorithms can be used for static analysis. It is not useful as an end unto itself. The gen, kill, live, and avail sets are not good vehicles for the discussion of static analysis in general. However, this is not just a shortcoming of the work done with DAVE. No general problem-oriented approach to static analysis is presently available. Several ideas are beginning to emerge. First, static analysis can be viewed as a kind of program execution mechanism which operates in different programming language semantics. The analysis done on HAL/S for NASA is an example of such an effort. Such an approach needs to be applied to Ada constructs before its exception handling capability creates semantic problems within existing hardware configurations. Another idea is the abstract computation type which includes patterns of operators. Applying a set theory of its own will serve to verify and validate static data flow through programs. The end result would be to discriminate legal operator pattern sets from illegal operator pattern sets.

Distributed processing introduces massive complications in static analysis. The reason involves asynchronous processes. This enables referencing and defining of variable values in parallel processes. This, in turn, enables ambiguous variable definitions due to multiple variable definitions within parallel tasks. Underlying these problems is the problem of synchronization. Taylor has attacked the problem in a stepwise manner. He first examined concurrent programs with no interprocess communication. Upon completion of that highly restrictive case, Taylor examined the rendezvous technique of Ada. What has resulted from this effort has been the determination of whether or not a syntactically possible rendezvous violates the semantics of synchronization. The end result has been the realization that arbitrary systems of concurrent processes can not be analyzed efficiently using static analysis techniques. Although static analysis has been demonstrated to be an effective error detecting mechanism for analyzing single programs, they may or may not be applicable to concurrent programs. Their applicability rests upon the synchronization properties of the distributed system. An interesting adjunct to such a conclusion would be a future study of different kinds of process scheduling with different capabilities for distributed processing structures.

7.2 Relationship of Manifestations to EDPS

The increasing density of electronically active devices on VLSI semiconductors is fueling a revolution. More and more compute power is being squeezed into less and less space. The limiting factor is the speed of light which means quicker response times for new, smaller devices. VHSIC technology optimizes response times by shrinking circuitry within specialized hardware structures. In a sense, compute power is being distributed within its own architectures. So much power is being distributed that problems are generated in three general directions. First, access to all this new power is becoming increasingly difficult. Controlling that access is no longer trivial. Access can be used to perform a task within a given architecture or it can be used to modify the operating characteristics themselves. These operating characteristics are functions of hardware distribution which introduces another general consideration. How the compute power is distributed within hardware is becoming more important. As single buses are replaced by multiple buses, how information flows within an architecture is sometimes obscured. The von Neumann characteristics of a single processor are transformed into configurations displaying probabilistic characteristics. Working around these probabilistic characteristics emphasizes the importance of a third general consideration; i.e., the distribution of data bases. The architecture of a particular data base should parallel the operation of the hardware architecture which contains it. Obviously magnetic tape data structures would not be expected to operate optimally within disc architectures. Taken as a group, considerations concerning the distribution of control, the distribution of hardware, and the distribution of data bases are becoming increasingly important. The subsequent paragraphs will examine specific manifestations of such considerations within embedded distributed processing systems.

7.2.1 Support of EDPS by Object-Oriented Architecture

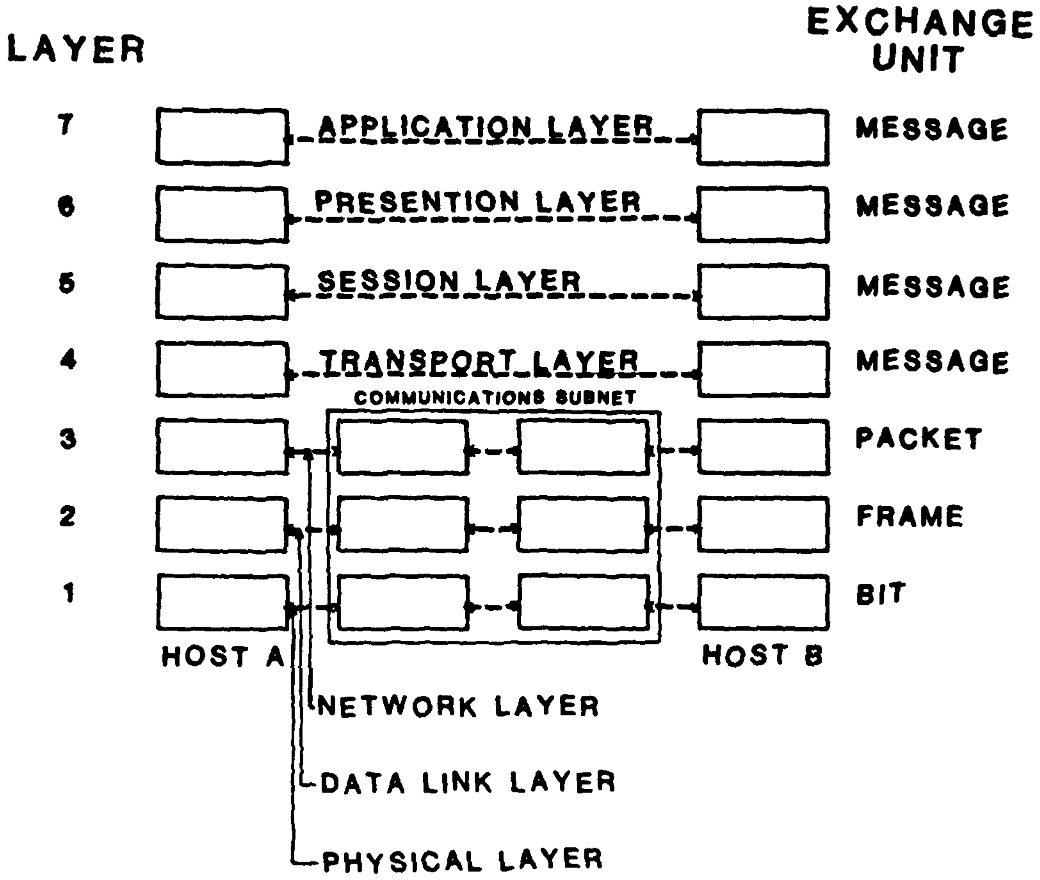
Current trends indicate significant departures from the usual von Neumann architecture of the present. Multiple processor configurations are becoming more prevalent and interconnect networks are offering unprecedented increases in their throughput. How to take advantage of these advances is becoming a growing problem. First, users must comprehend the advantages offered by such architectures. Such knowledge is in short supply. Second, suppliers of such architectures must explain their advances to the marketplace. These explanations are becoming more subtle and the required communication talent increasingly rare. Third, present commitments to existing architectures make revolutionary innovation difficult to learn. Simply maintaining existing architectures is a full time job requiring great technical expertise. Such workforces have little time to spend assessing revolutionary breakthroughs. In the absence of assessment the breakthroughs continue with a quickened pace. The suppliers are driven into popularity contests within the marketplace. In some instances single suppliers produce products which compete with one another; e.g., the IBM System/38 and the IBM 4300. Such antics exacerbate the problem. When single suppliers introduce innovative products competing with one another in identical marketplaces, the situation is out of control. Assessments which should originate within the marketplace have not occurred. Consequently, fragmentation occurs driving the suppliers to more innovation, more breakthroughs, and more chaos. One of the true ironies of such circumstances is the emerging role of colleges and universities. In attempts to remain abreast of the accelerating breakthroughs, groups of faculty and students concentrate on the assessment of specific revolutionary breakthroughs. In some instances they may even design and implement them; e.g., Ada and Diana. When the breakthroughs are introduced to the marketplace, the workforces acquired by that marketplace to assess them are recruited from the originating faculty and students. Attempts to adapt the breakthrough to specific needs within the marketplace rest upon these newly acquired workforces. Such workforces seldom appreciate the needs of the marketplace. Producing the revolutionary breakthroughs is a full time job with little time available for assessments within unknown marketplaces. As new generations of innovative researchers proceed to the marketplace, they are replaced by new faculty and new students. Neither has allegiance to existing architectures or ways of doing things. Consequently, they offer ideal test-beds for the development of revolutionary innovation. The present cycle works so well that colleges and universities have increasing difficulty in attracting and keeping competent computer faculty members. The marketplace pays a premium for the talent which understands revolutionary breakthroughs.

The job migration phenomenon mentioned above is particularly evident within the embedded distributed processing marketplace. Talent to maintain existing commitments is extremely scarce. Consequently, talent to assess technological breakthroughs is virtually non-existent. Significant departures from existing methodologies have few advocates but in the absence of independent assessment, revolutionary breakthroughs are being produced and adopted. The reorientation of existing workforces is becoming imperative. The adoption of an object-oriented modularization approach expedites this reorientation. Only with such an orientation can be consequences of non-von Neumann architecture be grasped. The understanding of a distributed system becomes more generic. The conceptualization of interconnect buses and their architectures is heightened. The emerging importance of distributed data bases is clarified. In fact, the object-orientation of the data bases themselves becomes evident. In summary, the performance of embedded distributed processing systems will be enhanced by object-oriented modularization. This will be evident in faster response times, increased flexibility, massive resource sharing, increased reliability, wider availability, and highly transportable systems. Failsafe architectures and operation will be the hallmark of distributed systems of the future. Underlying all such architectures and operations will be object-oriented modularizations.

7.2.2 Object-Oriented Modularizations and State-of-the-Art Technologies

The multi-layered architecture produced by VLSI circuits and VHSIC technologies has consequences far beyond simple departures from the von Neumann architecture of uniprocessors. It affects geographic location, data base partitioning, and system control. When multi-layered architecture is combined with multiple processors, the situation can become non-deterministic. Many of the operating characteristics of the past are being transformed in the present. Genuine multiprocessing hardware now exists. Each component operates independently of the other configuration components. They operate simultaneously. The network connecting all components is highly sophisticated and operates at a variety of levels. Each layer typifies a different level of abstraction. In the International Standards Organization's (ISO) Open Systems Interconnection (OSI) there are seven levels of abstraction. These are presented in Figure 7-4. The lowest level is physical and involves the movement of bits within a network.

Obviously, the next lowest level is grouping bits into frames. The frames can be grouped into packets for the third level of abstraction. Taken as a group these first three levels comprise the communication subnet boundary which exists for every architecture. Four additional layers have been provided for higher levels of abstraction. At the highest level is the application layer which is not transparent to network users. This is the level at which most users interface with the network. Ethernet is an example of communications alternatives which embody the OSI suggested by the ISO. As multi-layered architectures become prevalent, the operation of tightly-coupled buses approximates the operation of an OSI model. This layering effect is evident in the architecture of the iAPX432



Layer 7 - the application layer encompasses information peculiar to the network's end users. It is the only layer which is not transparent to the user.

Layer 6 - the presentation layer translates messages between the various formats, codes and languages generated by different network residents.

Layer 5 - the session layer handles the logical exchange of messages between network stations.

Layer 4 - the transport layer manipulates message transport between end users like computers and communication networks.

Layer 3 - the network layer controls the switching and routing of messages between nodes to effect transparent data delivery.

Layer 2 - the data link layer regulates the coding and decoding of data packets for reception and delivery over data communication lines. It also performs error detection and can provide correction services.

Layer 1 - the physical layer incorporates the mechanical, electrical and functional characteristics of the line between network nodes.

Figure 7-4 International Standards Organization's Open Systems Interconnection Model

and is presented in Figure 7-5. In the previous analysis of the iAPX432 architecture the layers above the interface processor were interrupt-driven while those below were transaction-driven. Such an observation is tantamount to stating levels 4 to 7 are interrupt-driven and levels 1 to 3 are transaction-driven. Much of the knowledge needed to operate the multi-layer architecture of the iAPX432 is provided by a thorough knowledge of the OSI model.

In effect, the breakthroughs of VLSI and VHSIC technologies are forcing an understanding of the OSI model and its ramifications upon the marketplace. Object-oriented modularization is valuable because it enables applications (i.e., layer 7 issues of the ISO model) to be implemented without loss of control. As more power is distributed throughout EDPS networks the issues remain unchanged. The object-oriented approach by Ada will prove extremely valuable in future multi-layer, distributed architectures. Hence, immediate effort should concentrate upon thorough understandings of multi-layered architectures, open system interconnects, and Ada application packages. The subsequent integrated software support environments

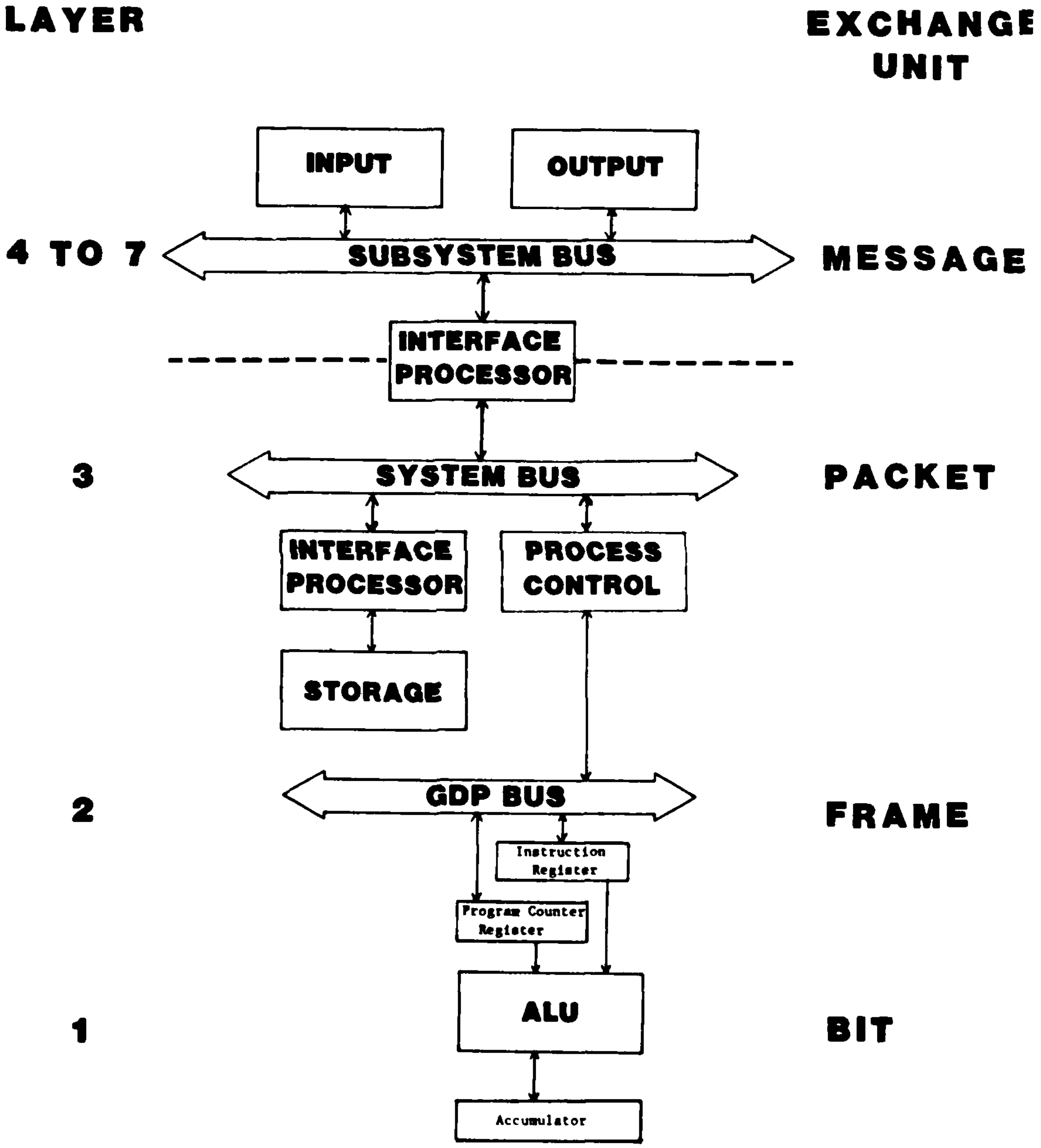


Figure 7-5 Open Systems Interconnections of the iAPX 432

should accommodate effort from all three directions. The largest unresolved problem remains distributed data base structures and their parallel algorithms. Such environments have simply not been sufficiently available to resolve the problem. Advances in VLSI and VHSIC technologies virtually assure the ready availability of such environments in the immediate future. The subsequent section addresses the highly technical and complex issue of tractability within such environments. The problem is so new that basic issues must again be examined.

7.2.3 Benefits of Object-Oriented Modularization to EDPS

The evolving environments of multi-layered architecture loosen the present constraints placed on computing power. Access to computational power is becoming commonplace. Communication networks with parallel structures are being implemented on a global basis. Data bases and their inherent structures are stored wherever needed. The cost of software has become greater than the cost of hardware. All these factors impact embedded distributed processing systems.

To combat costs and expedite development, an integrated software support environment remains essential. Before such an environment is realized the Ada program support environment is necessary. The commitments to the APSE have been made. Ultimately all advances are contingent upon the applications which are to be implemented. The most valuable technique during these implementations is object-oriented modularizations. However, this technique has its shortcomings as well. Most significant is the matter of tractability; i.e., whether the solution can be automated under all conditions. In effect, the absence of constraints introduces the tractability problem. The order of computation becomes increasingly important. Fewer constraints mean more alternatives for state transitions. Some orders of computation become indeterminant. The technical term is NP-complete. This problem was not encountered as often in uniprocessor architectures. Static analysis techniques had the problem in the uniprocessor environment. The problem with static analysis has increased with distributed systems.

In summary, object-oriented modularization solves many software problems within embedded distributed processing. It also raises other issues concerning tractability. The problems can be solved if they are recognized but this recognition is becoming increasingly subtle. New techniques are beginning to emerge and will be discussed in the subsequent phases of the Distributed Processing Tools Definition study.

7.3 Categorization of EDPS Requirements and Techniques by Lifecycle Phase

The Ada Programming Support Environment (APSE) provides an initial set of tools. They include a compiler, a debugger, a linker-loader, an editor, a run controller, and a configuration manager. Additional tools will be needed. Furthermore, each phase of the software system lifecycle must be addressed. As each new application is developed, where the software is used must be determined. Once that environment has been established, whether the requirements can be achieved by allocating an acceptable level of resources must be determined. If an unacceptable level of resources is required, why continue? Assuming the requirements phase issues can be resolved, the design phase should carefully weigh several more issues. An expected level of performance should be calculated. Such a calculation can be used to satisfy an expected level of performance. It verifies a proposed configuration and isolates what parts of the subsequent system must be monitored very closely. As the design phase continues, the impact of each change on the expected levels of performance must be carefully documented. Careful documentation will refine the expected performance. With each refinement the expected performance will become more realistic. During the coding phase, alternative ways to achieve the same operating expectations should be compared. As unforeseen problems are encountered, they should be carefully documented. Each critical component should be carefully monitored during the testing phase. The trade-off between resource requirements and critical component performance should be established as a matter of record. The documentation phase is aided by the compilation of adequate documents throughout the previous phases. When modifications are requested, the maintenance phase should assess the effect of each modification. From these assessments a long-range configuration requirements plan can be accumulated for the future.

In summary, performance is a primary consideration throughout the lifecycle. When requirements are being defined and the initial software design is being formulated, a performance analysis verifies the feasibility and desirability of the functional architecture. Once feasibility and desirability have been verified, the actual configuration required to support the new application is determined. Such a determination establishes the power required from the support hardware as well as the operating system software. Configuration and design are not separate issues. Design depends upon requirements while configuration depends upon design. Therefore, several iterations of requirements-design-configuration activities are usually needed before the best combination is known. The subsequent sections will illustrate this process.

7.3.1 Static Analysis of Concurrent Programs

The problems of referencing undefined program variables are compounded by asynchronous processes. Related problems include:

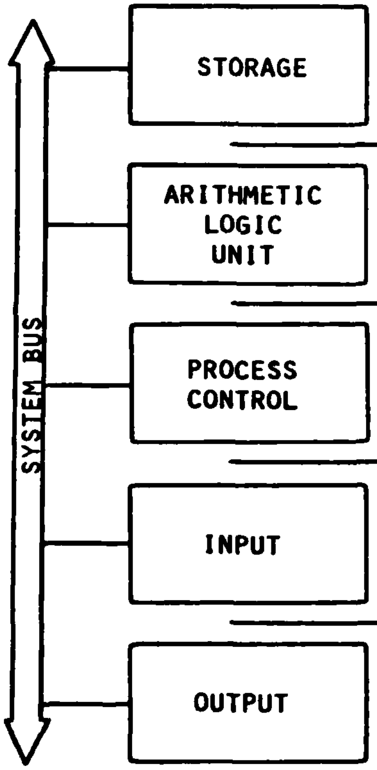
- referencing and defining variable values in parallel processes
- ambiguous variable definition due to multiple variable definitions in parallel tasks
- waiting for synchronization with a process which has already been guaranteed to have terminated
- waiting for synchronization with a process which may never have been scheduled, and
- the illegal scheduling of a process in parallel with itself.

Taylor studied concurrent programs in which there was no interprocess communication in his first paper on the subject. Such programs could schedule and then wait for completion of processes but concurrently running processes could not explicitly communicate with each other or affected each other's progress while running except through access to shared global variables.

Subsequent effort by both Taylor and Osterweil on concurrent programs can be carried out in an efficient way -- provided the programs require no interprocess communication. Furthermore, the programs disallow run-time determination of which processes will be scheduled. A related restriction is that the directed graph model of the process invocation structure of either program must be acyclic. Recursive subroutine calls are not allowed.

In addition to his work with Osterweil, Taylor has investigated different synchronization primitives. In particular, he studied the rendezvous mechanism of Ada. This mechanism allows interprocess communication and synchronization precluded previously. Taylor argues a flow model describing the set of possible flows of control through the system of simultaneously operating processes must be completed before static analysis can be performed. When single programs are examined, many of their syntactically possible flows are semantically infeasible. In single programs a flow path is infeasible if no set of input data exists which can satisfy the set of branch conditions occurring along its path. When concurrent programs are examined, their infeasible paths occur because of the semantics in the synchronization primitives. An example would be a process with two "calls" on an entry in another process and that second process waiting at the entry point for a call. Ir the two processes contain no cycles, the second call can never be synchronized with the entry point. In general, efficient static analysis for arbitrary systems of concurrent processes can not be constructed.

Depending upon the synchronization properties of the distributed system, the construction of data flow analysis algorithms may or may not be feasible. The work by Taylor indicates efficient algorithms for the general case can not be constructed. Taylor and Osterweil have shown efficient algorithms can be constructed for special situations in which the run-time determination of process scheduling is tightly constrained. When the scheduling of processes and the process interaction can be made deterministic, efficient static analysis algorithms can be constructed to detect a wide variety of possible data flow and process scheduling anomalies. Table 7-1 indicates the anticipated effect of static analysis techniques on EDPS lifecycle phases.



	REQUIREMENTS	DESIGN	CODING	TESTING	DOCUMENTATION	MAINTENANCE
STORAGE	1 / 3					
ARITHMETIC LOGIC UNIT	3 / 3					
PROCESS CONTROL	2 / 2					
INPUT	3 / 1					
OUTPUT	3 / 1					

COUPLING refers to number of connections between a calling and a called module as well as to the complexity of those connections.
"1" means low coupling. "2" means average coupling. "3" means high coupling.

COHESION describes how tightly bound together the instructions are within a module. If module does more than one discrete task the instructions in that module are not bound very closely to each other.
"1" means strong cohesion. "2" means average cohesion. "3" means weak cohesion.

Table 7-1 Effect of Static Analysis Techniques on EDPS Lifecycle Phases

Additional work concerning different kinds of process scheduling (e.g., non-deterministic or probabilistic) and interaction capabilities is required. Such scheduling and interaction presently exist within various kinds of distributed processing structures.

7.3.2 Branch Testing

Branch testing is the most common form of testing in which program structures rather than black box functional specifications are used to guide the testing efforts. The goal is to construct test data in such a way that every program branch is executed at least once. The method is appealing and tests all parts of the program. Furthermore, it is easy to audit and provides each programmer with criterion for a complete set of tests. However, many errors go unfound because every branch is usually tested only once. Attempts have been made to extend the method by requiring the testing of combinations of branches or classes of program paths. The problem with these extensions is that their number becomes very large, very quickly. The number can be reduced by relying upon the data flow relationships between program constructs. As an example, some test may cause the execution of two statements s1 and s2 if the first (s1) defines the values required by the second (s2). Otherwise s1 and s2 could be tested separately with two different tests.

The difficulties of extending branch testing to more powerful methods have caused a re-examination of functional testing. If a systematic approach to requirements specification is used, rules for identifying functions to be tested can be developed. Empirical evidence indicates that many errors can be found using a systematic approach to functional testing. Such errors cannot be found by branch testing alone. The best approach is probably a complimentary one using both functional testing and branch testing.

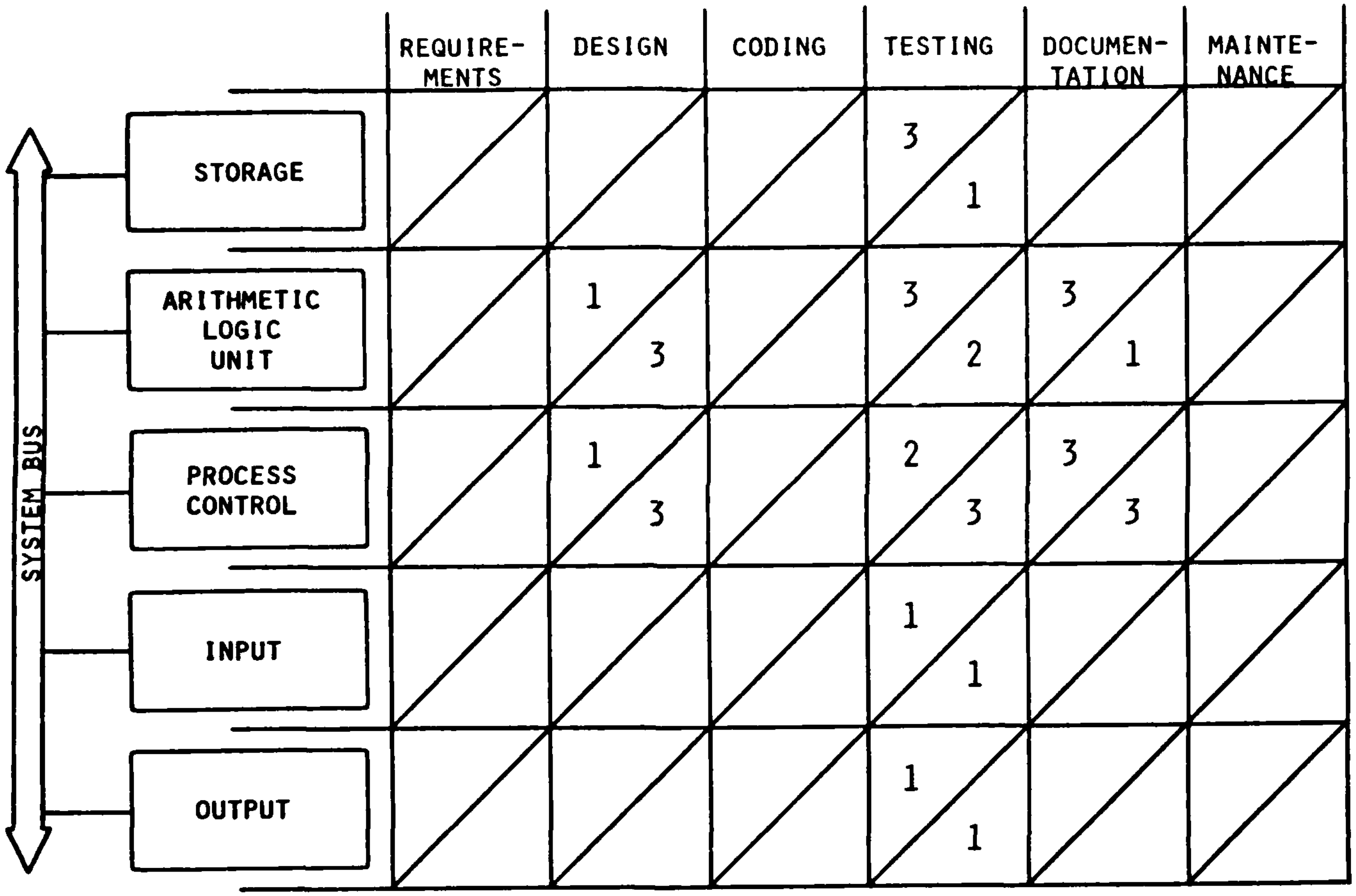
The techniques that have been developed for single, non-distributed programs or systems can obviously be applied to individual components within a distributed system. The special properties of distributed systems make them more difficult to test and require the development of additional test data generation techniques. Single programs can be thought of as being at a certain point in computation when the flow of control reaches that associated point in the program. To test such computations, test data must be constructed to reach that point in the program. In a distributed system several programs must cooperate to produce the desired effect. The state of the system becomes increasingly important because certain computations can only be performed while in an appropriate state.

The role of computation states in a distributed system affects testing in two principle ways:

- detailed systematic documentation of system states is absolutely necessary and

- the condition under which a system or program can change states must be known.

Much such information comes from design. Consequently, systematic design specification is more important for distributed systems than for non-distributed ones. The anticipated effects of branch testing techniques on EDPS lifecycle phases are indicated in Table 7-2.



	REQUIREMENTS	DESIGN	CODING	TESTING	DOCUMENTATION	MAINTENANCE
STORAGE				3 1		
ARITHMETIC LOGIC UNIT		1 3		3 2	3 1	
PROCESS CONTROL		1 3		2 3	3 3	
INPUT				1 1		
OUTPUT				1 1		

COUPLING refers to number of connections between a calling and a called module as well as to the complexity of those connections.
"1" means low coupling. "2" means average coupling. "3" means high coupling.

COHESION describes how tightly bound together the instructions are within a module. If module does more than one discrete task the instructions in that module are not bound very closely to each other.
"1" means strong cohesion. "2" means average cohesion. "3" means weak cohesion.

Table 7-2 Effect of Branch Testing Techniques on EDPS Lifecycle Phases

7.3.3 Impact Analysis

When a system object is altered there is always the danger that the change will have unforseen effects due to a forgotten relationship between the object which was changed and other objects. Automated impact analysis can be used to avoid this problem if relationships between objects are recorded in a machine readable format. There is a wide variety of kinds of objects and relationships.

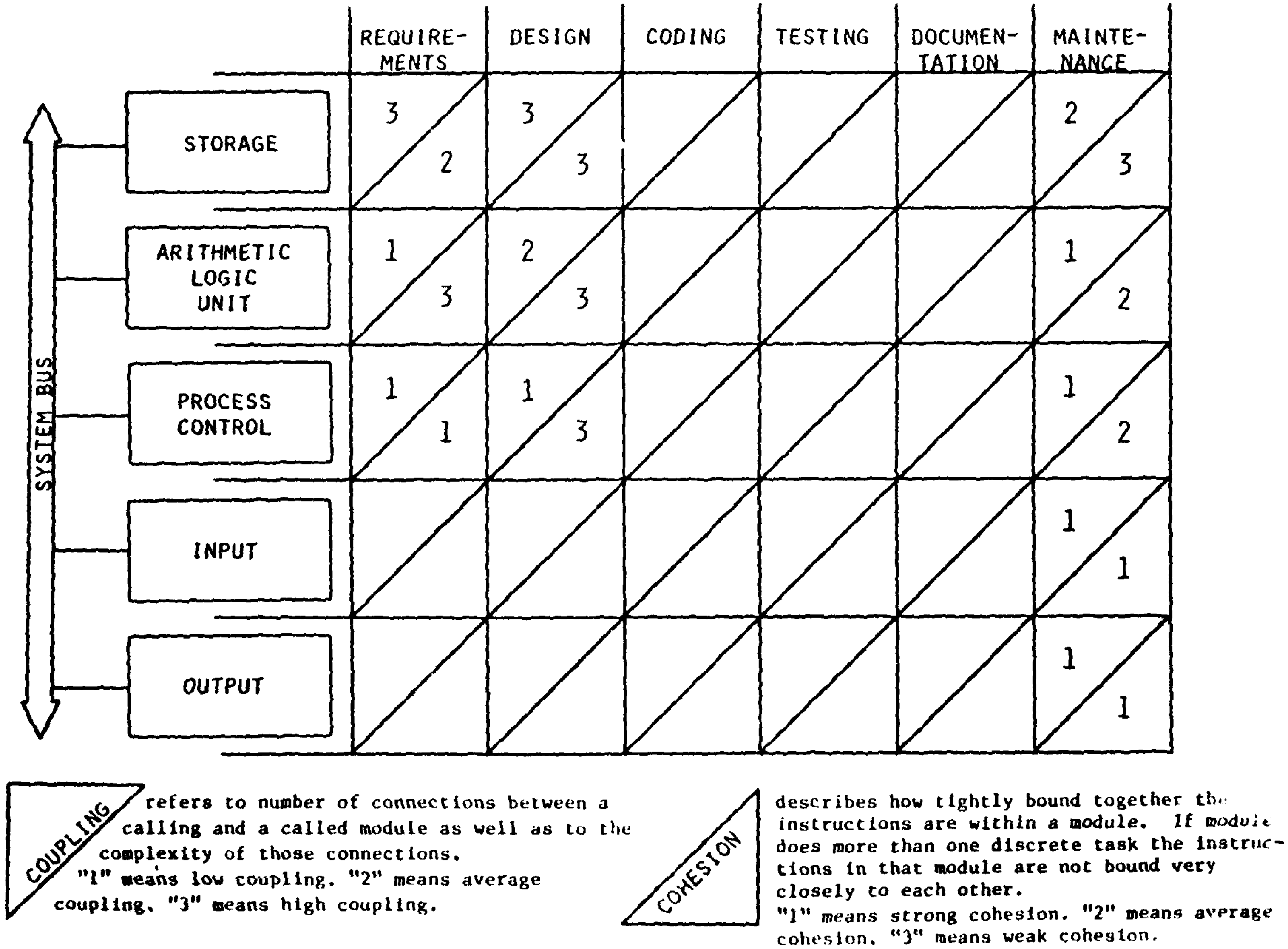
Source code is the most commonly available object for impact analysis. A traditional cross reference testing tool can be thought of as a very simple impact analysis aid. Impact analysis tools can range in sophistication from tools that are as simple as cross reference listers to tools which are capable of complex data flow analysis.

Interesting, powerful tools can be built for source code impact analysis. It is important, however, to construct tools which are cost effective for dealing with real maintenance problems that really occur rather than with imagined problems whose principal appeal is that they can be attacked using an elegant methodology. For this reason it is suggested that research on impact analysis tools focus on studies of maintenance problems and on studies of the types of changes that are commonly made to source code.

Maintenance problems that arise due to changes in source code can be due not only to change itself but to bad design and imprecise specifications. Studies of maintenance problems should consider how the problems can be avoided with different design and specification methods. This is necessary in order to avoid building impact analysis tools for dealing with problems which might be avoidable through the use of better software development techniques. Structured design, for example, emphasizes the use of modular decompositions in which there is low inter-module coupling and high module cohesion. Parnas' design emphasizes the hiding of design decisions (e.g., data structure implementations) inside modules. Both methods are useful for reducing the potential effects of change and in reducing the need for elaborate impact analysis techniques and tools.

The special problems of doing impact analysis on distributed systems source code will include those of doing static analysis on distributed systems source code. In order to determine data dependencies between parts of a system it is necessary to do data flow analysis. The special problems of doing data flow analysis on distributed systems code are described in the static analysis critique. The limitations on distributed system structure which are necessary to allow data flow analysis to be carried out efficiently must be very carefully considered in any proposal to construct a change impact tool for distributed systems.

In distributed system design it is necessary to consider not only software design but also the hardware configuration onto which the software must be mapped. The effects of changes on hardware are likely to impact the design of a distributed system more than the design of a non-distributed system. It is necessary to model both the software system and the hardware resources in order to do automated impact analysis. It is also necessary to consider timing and synchronization, and to construct models for these. Table 7-3 shows the anticipated effect of impact analysis upon the lifecycle phases of EDPS. Little research has been completed on the representation of this kind of information for distributed systems and any proposed research into impact analysis for distributed systems should include resources for studying first the primary problems of design and requirements representations.



SYSTEM BUS
COUPLING
COHESION

	REQUIREMENTS	DESIGN	CODING	TESTING	DOCUMENTATION	MAINTENANCE
STORAGE	3 2	3 3				2 3
ARITHMETIC LOGIC UNIT	1 3	2 3				1 2
PROCESS CONTROL	1 1	1 3				1 2
INPUT						1 1
OUTPUT						1 1

COUPLING refers to number of connections between a calling and a called module as well as to the complexity of those connections. "1" means low coupling. "2" means average coupling. "3" means high coupling.

COHESION describes how tightly bound together the instructions are within a module. If module does more than one discrete task the instructions in that module are not bound very closely to each other. "1" means strong cohesion. "2" means average cohesion. "3" means weak cohesion.

Table 7-3 Effect of Impact Analysis Techniques on EDPS Lifecycle Phases

8.0 REMAINING WORK TO BE ACCOMPLISHED IN THE DISTRIBUTED PROCESSING TOOLS DEFINITION STUDY

The Distributed Processing Tools Definition (DPTD) study is divided into three phases:

Phase I - Study of Hardware and Software Technologies

Phase II - Survey of Existing Tools and Techniques

Phase III - Analysis of Problem Areas and Recommendation of Candidates for Research and Development Efforts.

This report includes the results of the Phase I study only. The hardware and software technologies pertinent to embedded distributed processing systems have been analyzed and their requirements and impacts have been categorized with respect to the software life cycle phases. This categorization of requirements and impacts forms the basis of the Phase II survey. In Phase II, industrial, university, and Department of Defense software tools and techniques will be researched, and we will identify those tools and techniques that satisfy the lifecycle phase support requirements established in Phase I. Also in Phase II we will denote as problem areas those Phase I lifecycle requirements that are not supported by tools or techniques. Finally, in Phase III these problem areas will be analyzed and a prioritized list of candidate research and development efforts to solve the problem areas will be recommended. These candidate efforts will be fully described with estimates of manhours, schedules, technical feasibility, benefits, and probable users. Phase III completes the DPTD study.

The results of the tools survey will be submitted to RADC in an interim technical report upon the completion of Phase II. This report will be concatenated to the Phase I report. The benefits of this approach are twofold. First, the phases of the DPTD study are closely linked, and each phase builds upon the preceding phase. The understanding of the work completed to date on the DPTD study requires an understanding of the previously accomplished work. It is easier for a reader to understand the results if they are all bound in a single volume with smooth transition between phases. Secondly, in the course of the study modifications and updates can be easily incorporated. These updates arise because of recent technological advances or deeper insight gained in a particular area during the course of the study. Modifications to the text of previously submitted reports will be summarized in a list of page changes and change bars will be inserted in the text denoting the affected portions. Therefore, the final technical report for the DPTD study will be submitted to RADC in a single volume that reports the results of Phases I, II, and III.

APPENDIX A

DEFINITION OF THE SCOPE OF EMBEDDED DISTRIBUTED PROCESSING SYSTEMS

Military computer systems span the spectrum from single microprocessors in "smart-bombs" to multiple, distributed mainframes in world-wide communications systems. The particular categories of military systems most affected by technological advancements in embedded distributed processing include: (1) armament, (2) aeronautical, (3) missile and space, (4) command/control/communication, and (5) mission/force management systems. The analysis of the impact of technological advancements upon these five categories of systems is aided by grouping the five categories into two, higher-level, generic classifications. The armament, aeronautical, and missile and space systems will be classified as weapon systems, and the command/control/communication and mission/force management systems will be classified as communication systems. These generic classifications are based upon common characteristics of the members within each class. These common characteristics are defined later in this appendix. Hence, in this technical report we will use the generic classifications of weapon systems and communication systems, and it is understood that observations and conclusions pertinent to a generic classification apply to all members of the class.

Distributed processing systems can be visualized as a region of a volume bounded by axes describing (1) distribution of hardware, (2) distribution of control, and (3) distribution of data base. Figure A-1 shows this volume.

The portion of this volume characterized by a single CPU, a single, fixed executive controller, and a single copy of the data base represents the common uniprocessor systems of today. Moving toward multiple computers, multiple operating systems, and partitioned data bases is characteristic of fully distributed processing systems. Weapon systems and communication systems occupy different regions within the characteristic volume, but are definitely within the distributed processing domain. Figure A-1 shows the relationship of uniprocessor systems to distributed processing systems and where weapon and communication systems fit in these descriptions.

The differences between weapon systems and communication systems are rooted in implementation of the

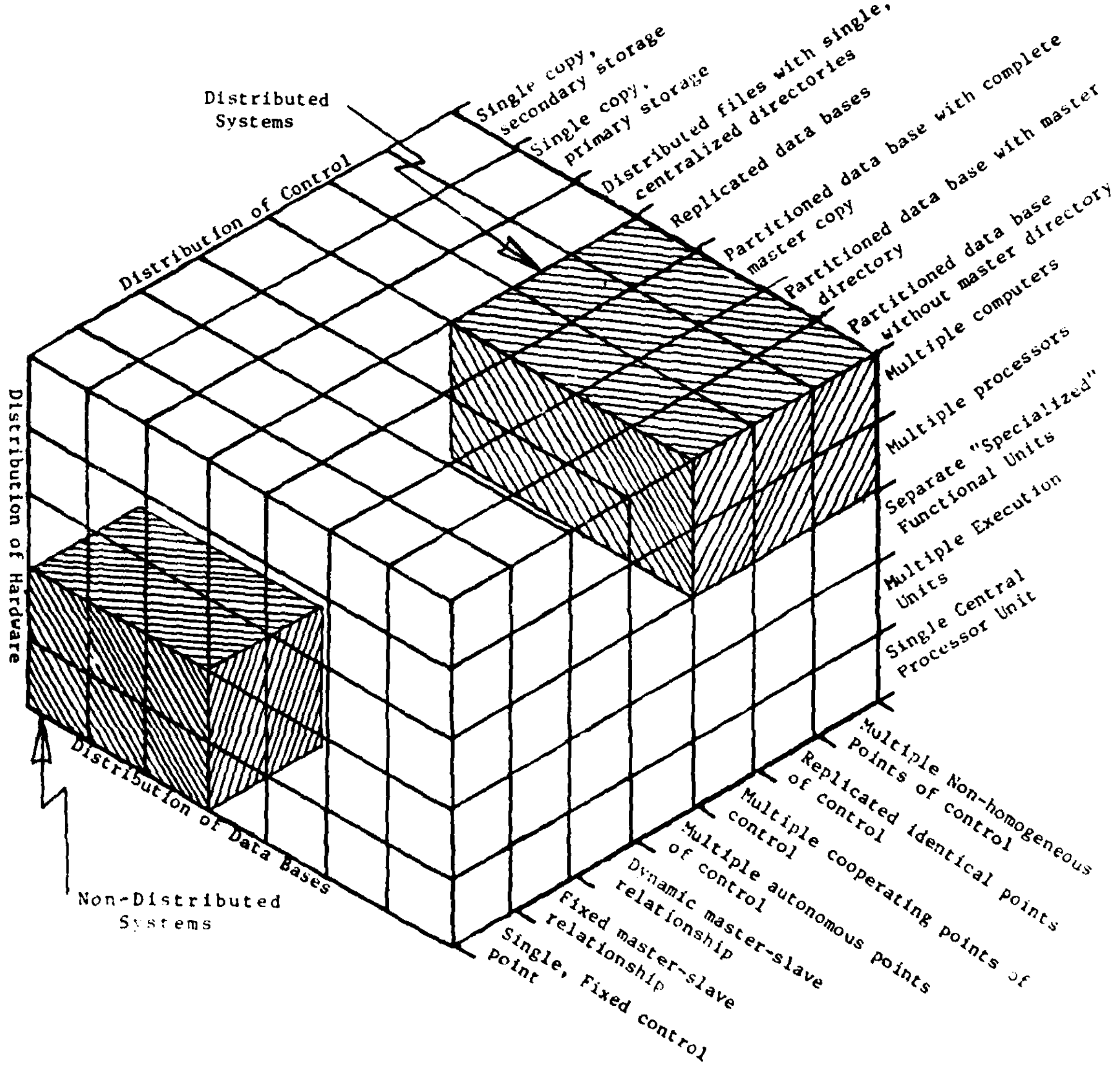

Distributed Systems
Distribution of Control
Single copy, secondary storage
Single copy, primary storage
Distributed files with single, centralized directories
Replicated data bases
Partitioned data base with complete master copy
Partitioned data base with master directory
Partitioned data base without master directory
Multiple computers
Multiple processors
Separate "Specialized" Functional Units
Multiple Execution Units
Single Central Processor Unit
Distribution of Hardware
Distribution of Data Bases
Non-Distributed Systems
Multiple Non-homogeneous Points of control
Replicated identical points of control
Multiple cooperating points of control
Multiple autonomous points of control
Dynamic master-slave relationship
Fixed master-slave relationship
Single, Fixed control Point

Figure A-1 Characteristic Volume of Computer System Capabilities following two key features of distributed processing environments:

(1) the degree of human interaction required by the hardware/software system

(2) the timing constraints of system operation.

Weapon systems must require human decision before a weapon is released; for safety requirements a system should not automatically switch from a non-attack situation to an attack with weapons launch capability without explicit human consent. However, once this consent is acknowledged, the hardware/software system must execute in real-time because of the speeds associated with targets and launch platforms. Therefore, prior knowledge of system state transitions is a prerequisite for weapon systems mode design. Other characteristics of weapons systems that are commensurate with these two features are shown in Table A-1.

In contrast, communication systems permit a higher degree of automation without human intervention. If no human response is received the system can automatically queue messages or resend the transmissions to provide backup capability. Consequently, the system mode design is often interrupt driven with queuing permitted. The system timing constraints are often near real-time; that is, between one millisecond and one second. Table 'A-1 lists the characteristics of communication systems. A study of Table A-1 shows that both weapon systems and communication systems strongly exhibit features of distributed processing; however, there are significant differences within this distributed processing domain to warrant the two generic classifications of weapon systems and communication systems that we have identified.

In summary the grouping of armament, aeronautical, and missile and space systems into the generic classification of weapon systems and the grouping of command/control/communication and mission/force management into the generic classification of communication systems recognizes the differences between the groups and benefits the distributed processing tools definition study by specifying the requirements for tools and techniques for a larger class of generic military systems.

Table A-1

Characteristics of Weapon and Communication

Embedded Distributed Processing Systems

	CLASSIFICATION	
CHARACTERISTICS	WEAPON SYSTEMS (Armament & Aeronautical & Missile & Space)	COMMUNICATION SYSTEMS (Command/Control Communications & Mission/Force Management)
Timing Constraints	Real-Time (usually less than a few milliseconds)	Near Real-Time (usually between milliseconds and several seconds)
Spatial Constraints	Close physical proximity (often less than 1000 feet) of components	Large geographical separations (often more than 100,000 feet) of components
System Mode Design	All transitions between system states are a priori known to satisfy safety requirements	System state transitions are often interrupt/transaction driven with queuing permitted and may be random
Data Transfer	Usually relatively small quantities of data transferred at high data rates	Very large amounts of data transferred

Table A-1 (Continued)

	CLASSIFICATION	
CHARACTERISTICS	WEAPON SYSTEMS (Armament & Aeronautical & Missile & Space)	COMMUNICATION SYSTEMS (Command/Control Communications & Mission/Force Management)
Hardware/Software Resources	Hardware and software functions are often distributed to special-purpose, dedicated units	Usually a multiplicity of general purpose hardware components (disks memories, CPU, etc.) and software must be transparent to hardware peculiarities
Operating System	Minimum size with limited capabilities for support of system mode designs	High-level, very capable system that integrates and controls the distributed components
System Component Interconnection	Highly standardized communication protocols to permit easy add-on of new units to bus	"Cooperative autonomy" whereby the various system components work largely independently but in a coordinated manner under the control of the operating system
Cohesiveness	Tightly-coupled	Loosely-coupled to fully distributed

Table A-1 (Continued)

	CLASSIFICATION	
CHARACTERISTICS	WEAPON SYSTEMS (Armament & Aeronautical & Missile & Space)	COMMUNICATION SYSTEMS (Command/Control Communications & Mission/Force Management)
Backup Capability	System degradation must be predictable	System degradation is minimized by redundant data processing elements and communication links
Security	Security must be maintained at the entire system level because the weapon system must be isolated from external manipulation	Security must be maintained within the individual components because of extensive communication links to the external environment

APPENDIX B

RATIONALE FOR THE SELECTION OF ADA AS THE HIGHER ORDER LANGUAGE TO STUDY FOR EDPS

Ada has now been adopted by the Department of Defense. As far as the United States Air Force is concerned, Ada is scheduled for introduction and use by 1983. By 1986 all new embedded distributed processing systems must use Ada. Present strategy relies upon the continued use of JOVIAL (J73) until sufficient development time has been provided for Ada.

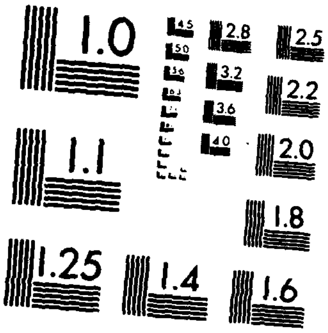
The Office of the Under Secretary of Defense for Research and Engineering (OUSDRE) has been assigned responsibility for planning the Software Technology Initiative (STI) of the Department of Defense. Subsequent coordination also rests with OUSDRE. The present is particularly propitious for a concerted and concentrated effort on Ada. All candidates for short-term research initiatives must emphasize technology transfer, standardization of software environments, tools, packages, workstations, and preliminary results from thrusts whose payoffs occur later. In summary they must produce reusable Ada packages within integrated software support environments which are themselves dependent upon Ada. As presently envisioned, the integrated software support environment of the future will evolve from or be incorporated by the Ada Programming Support Environment (APSE). Active efforts are underway to design and implement the APSE. All future software tools concerning embedded distributed processing systems must link to the APSE. The consequences of Ada will be felt throughout the entire industry as well as all the Federal Government. The integrated software support environment provided by Ada will provide important contributions for many years.

Meeting the overall goals of the Ada effort depends upon the wide availability of effective support environments. In particular these environments must be rigorously engineered to support Ada throughout the software life cycle phases. Specific features of such environments include:

- layered approaches to maximize subsequent portability of tools;
- completely engineered Ada compilers which produce high quality object code;
- comprehensive basic toolsets;
- careful separation of roles between hosts and targets;

- user friendly interfaces;
- sophisticated database techniques; and
- complete sets of control tools.

Many attempts at integrated software support environments have been made. Few have succeeded. The fragments of both the successful and unsuccessful attempts still exist and are constantly being used. All these remainders need to be converted to the integrated software support environment that Ada enables. The tools of this emerging environment have yet to be developed. Most are generic in nature and will probably exhibit unique properties. All will be developed through the system lifecycle phases. Consequently, of primary importance to embedded distributed processing systems is a complete understanding of Ada and what its impact will be on the software development lifecycle. Subsequent effort within the Distributed Processing Tools Definition Study for RADC will concentrate upon Ada. The justification for such a concentration is offered in the preceding remarks.



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

LIST OF ABBREVIATIONS

ADEPT	- A Design-based Evaluation and Prediction Technique
ADGE	- Air Defense Ground Environment
ALU	- Arithmetic Logic Unit
APSE	- Ada Programming Support Environment
ARC	- Advanced Research Center
ASGS	- Automatic Software Generation System
CADES	- Computer Analytical Design Evaluation System
CHiP	- Configurable, Highly Parallel
CIU	- Computer Interface Unit
CMU	- Carnegie Mellon University
CPU	- Central Processor Unit
CSMA-CD	- Carrier Sense Multiple Access with Collision Detection
DAVE	- Documentation, Analysis, Validation and Error-detection
DEC	- Digital Equipment Corporation
DEDS	- Data Entry Display Stations
DOD	- Department of Defense
DPTD	- Distributed Processing Tools Definition
DREAM	- Design Requirements Evaluation Analysis Method
EDPS	- Embedded Distributed Processing Systems
E/O	- Electro/Optical
GPU	- General Processor Unit

HAL/S	- Hierarchical Analysis Language/System
HOL	- Higher Order Language
ICL	- International Computers Limited
I/O	- Input/Output
ISO	- International Standards Organization
ISSE	- Integrated Software Support Environment
LAN	- Local Area Network
LU	- Logical Unit
Mbps	- Megabytes per second
MIPS	- Million Instructions Per Second
NASA	- National Aeronautics and Space Administration
NAV	- Navigation
NP	- Non-deterministic Polynomial
OOM	- Object-Oriented Modularization
OSI	- Open Systems Interconnection
OUSDRE	- Office of the Under Secretary of Defense for Research and Engineering
PAWS	- Performance Analyst Workbench System
PPBS	- Professional Programmer Based System
PSA	- Problem Statement Analyzer
PSL	- Problem Statement Language
PWB	- Programmer's Work Bench
STI	- Software Technology Initiatives
TRW	- Thompson Ramo Woolridge
UCSF	- University of California at San Francisco
VHSIC	- Very High Speed Integrated Circuits

VLSI	- Very Large Scale Integration

BIBLIOGRAPHY

1. Aho, A.V., Hopcroft, J.E., and Ullman, J.D., The Design and Analysis of Computer Algorithms, Addison-Wesley Publishing Co., Reading, Mass., Oct., 1975.

2. Andrews, D.M. and Bensen, J.P., An Automated Testing Methodology and Its Implementation, Proceeding of the Fifth International Conference on Software Engineering, San Diego, 1981.

3. Barbe, D.F., VHSIC Systems and Technology, Computer, Feb., 1981.

4. Barnes, J., et al, United Kingdom Ada Study Final Technical Report, London, Dept. of Industry, 1982.

5. Bernstein, P.A. and Goodman, N., Concurrency Control in Distributed Data Base Systems, ACM Computing Surveys, June, 1981.

6. Boehm, B.W., Some Experience With Automated Aids to the Design of Large Scale Reliable Software, IEEE Transactions on Software Engineering, Vol. SE-1, 1975.

7. Browne, J.C., The Performance Analyst's Workbench System, Information Research Associates, Austin, Texas, Apr., 1981.

8. DeMarco, T., Structured Analysis and System Specification, Yourdon, N.Y., 1978.

9. Deutsch, M.S., Software Project Verification and Validations, IEEE Computer, Apr., 1981.

10. Ferris, David, Micro Software Trends in the Business and Professional Environment Conference, Anaheim, Calif., Nov., 1981.

11. Fosdick, L.D. and Osterweil, L.J., Data Flow Analysis in Software Reliability, Computing Surveys, Vol. 8, 1976.

12. Gutz, S., Wasserman, A.I., and Spier, M.J., Personal Development Systems for the Professional Programmer, Computer, Vol. 14, No. 4, Apr., 1981.

13. Hansen, P.B., Testing a Multiprogramming System, Software Practice and Experience, Vol. 3, 1973.

14. Hennell, M.A., Woodward, M.R., and Hedley, R., On Program Analysis, Information Processing Letters, Vol. 5, 1978.

15. Horejs, J., Finite Semantics for Program Testing, Scripta Fac. Sci. Nat. U.J.E.P. Brunensis, Mathematical 1, Vol. 9, 1979.

16. Howden, W.E., Methodology for the Generation of Program Test Data, IEEE Transactions on Computers, C-24, 1975.

17. Howden, W.E., Life Cycle Software Validation, Computer, Feb., 1982.

18. Howden, W.E., Applicability of Software Validation Techniques to Scientific Programs, ACM Transactions on Programming Languages and Systems, Vol. 2, 1980.

19. Howden, W.E., Functional Program Testing, IEEE Transactions Software Engineering, Vol. SE-7, 1980.

20. Howden, W.E., Functional Testing and Design Abstraction, Journal of Systems and Software, Vol. 1, 1980.

21. Howden, W.E., Contemporary Software Development Environments, Communications of the ACM, Apr., 1982.

22. Ichbiah, J.D., et al, Reference Manual for the Ada Programming Language, U.S. DoD, July, 1980.

23. Intel Corporation, iAPX432 Object Primer, manual order number 171858-001 Rev. B., Santa Clara, CA., 1981.

24. Intel Corporation, Introduction to the iAPX432 Architecture, manual order number 171821-001, Santa Clara, CA., 1981.

25. Korel, B. and Laski, J.W., A Data Flow Oriented Program Testing Strategy, Institute of Control Systems, Ul.Armii Czerkuonej 101, Katowice, Poland, 1980.

26. Krutz, R.L., A Hierarchical Design Approach for VHSIC, VHSIC Briefs (Phase 3 Projects), OUSDRE, Oct., 1981.

27. LeBlanc, R.J. and MacCabe, A.B., An Introduction to Data Abstraction, School of Information and

Computer Science, Georgia Institute of Technology, Atlanta, Georgia.

28. Liskov, B. and Zilles, S., Programming with Abstract Data Types, Project MAC, Massachusetts Institute of Technology, Cambridge, Massachusetts.

29. Mhatre, G., VLSI: The Challenge, Electronics Engineering Times Technology Strategies, Nov., 1981.

30. Miller, E.F. and Melton, R.A., Automated Generation of Test Case Data Sets, Proceedings 1975 International Conference on Reliable Software, L.A., 1975.

31. Myers, G.F., The Art of Software Testing, Wiley Interscience, N.Y., 1979.

32. Ntafos, S.L., On Testing With Required Elements, Dept. of Math. Science, U. of Texas at Dallas, 1981.

33. Osterwald, L.J., A Software Lifecycle Methodology and Tool Support, U. of Colorado at Boulder, Boulder, Colorado, Apr. 1979.

34. Osterweil, L.J. and Fosdick, L.D., DAVE - A Validation, Error Detection, and Documentation System for FORTRAN Programs, Software Practice and Experience, Vol. 6, 1976.

35. Panzl, D.J., Automatic Software Test Drivers, Computer, Vol. 11, 1978.

36. Parnas, D.L., On the Criteria to be Used in Decomposing Systems in Modules, Communications of the ACM, Vol. 15, No. 12, Dec., 1972, p. 1053.

37. Parnas, D.L., Information Distribution Aspects of Design Methodology, IFIPS-71, North Holland, 1971.

38. Redwine, Samuel T., Siegel, Eric D., and Berglass, Gilbert R., Candidate R & D Thrusts for the Software Technology Initiative, DoD, May, 1981.

39. Reifer, D.J. and Trattner, S., A Glossary of Software Tools and Techniques, Computer, Vol. 10, No. 7, July, 1977.

40. Rice, J.G., Build Program Technique: A Practical Approach for the Development of Automatic Software Generation Systems, John Wiley & Sons, 1981.

41. Riddle, W.E. and Fairley, R.E., Software Development Tools, Berlin, Springer-Verlag, 1980.

42. Ryer, M. and Belmont, P.A., Building the First Ada Compiler - Planning Ahead, AIAA Computers in Aerospace III Conference, Oct., 1981.

43. Siegel, H.J. and Smith, S.D., An Interconnection Network for Multimicroprocessor Emulator Systems, IEEE, June, 1979.

44. Smith, Connie, Software Performance Engineering, Computerworld, Dec. 7, 1981.

45. Smith, C.U., The Prediction and Evaluation of the Performance of Software from Extended Design Specifications, IPAD Project Report No. 2, NASA, Aug., 1980.

46 Snyder, L., Introduction to the Configurable, Highly Parallel Computer, ONR Report CSD-TR-351, May, 1981.

47. Stucki, L.G., Automatic Generation of Self-Metric Software, Proceedings IEEE Symposium on Computer Software Reliability, N.Y., 1973.

48. Stucki, L.G., *New Directions in Automated Tools for Improving Software Quality*, Current Trends in Programming Methodology, Vol. 27, Prentice-Hall, 1977.

49. Taylor, R.N. and Osterweil, L.J., *Anomaly Detection in Concurrent Software of Static Data Flow Analysis*, IEEE Transactions on Software Engineering, Vol. 6, 1980.

50. Taylor, R.N., *Complexity of Analyzing the Synchronization Structure of Concurrent Programs*, U. of Victoria, Dept. of Comp. Sci., 1981.

51. Teichroew, D. and Bastarache, M.J., *PSL User's Manual*, U. of Michigan, ISDOS Working Paper No. 98, March, 1975.

52. Yourdon, E. and Constantine, L., *Structured Design*, Prentice-Hall, Englewood Cliffs, N.J., 1979.

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.

FILME

3-8