THESIS

THE INTEL 432/670 and ADA PERFORMANCE BENCHMARKS

by

David James Applegate
and
Robert Abbott Coates

December 1982

Thesis Advisor: U.R. Kodres

Approved for public release; distribution unlimited
# THE INTEL 432/670 and ADA PERFORMANCE BENCHMARKS

## Abstract
The INTEL 432/670 microcomputer system contains the iAPX-432 microprocessor which executes compiled ADA programs. The compiler resides on a host VAX 11/780, and compiled programs are downloaded to an INTEL MDS 800 system where they are transferred to the 432/670 for execution. This thesis describes a preliminary performance evaluation of the INTEL 432/670 through the use of selected benchmark algorithms from the Computer Family Architecture (CFA) study.
A description of the hardware components of both the MDS 800 and 432/670 is provided, including the modifications made to the operating system to allow compatibility with existing hardware. Additionally, the benchmark program source code and a user's manual are appended.
The INTEL 432/670 and ADA Performance Benchmarks

by

David Applegate
Lieutenant, United States Navy
B.A., St. Cloud State University, 1975

Robert Coates
Captain, United States Marine Corps
B.S., University of Idaho, 1976

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

 NAVAL POSTGRADUATE SCHOOL
 December 1982

Author: David Applegate

Author: Robert Coates

Approved by: Chris R. Kodres
Thesis Advisor

Bradford Menu
Second Reader

Chairman, Department of Computer Science

Dean of Information and Policy Sciences
ABSTRACT

The INTEL 432/670 microcomputer system contains the iAPX-432 microprocessor which executes compiled ADA programs. The compiler resides on a host VAX 11/780, and compiled programs are downloaded to an INTEL MDS 800 system where they are transferred to the 432/670 for execution. This thesis describes a preliminary performance evaluation of the INTEL 432/670 through the use of selected benchmark algorithms from the Computer Family Architecture (CFA) study. A description of the hardware components of both the MDS 800 and 432/670 is provided, including the modifications made to the operating system to allow compatibility with existing hardware. Additionally, the benchmark program source code and a user's manual are appended.
# TABLE OF CONTENTS

## I. INTRODUCTION

A. THESIS DESCRIPTION ........................................ 8
B. EVALUATION OF COMPUTER ARCHITECTURES .............. 9
C. THESIS ORGANIZATION .................................. 10

## II. THE MILITARY COMPUTER FAMILY ARCHITECTURE .... 12

A. CFA/NCF PROJECT MOTIVATION ......................... 12
B. CFA/NCF PROJECT DESCRIPTION ......................... 12
C. CFA SELECTION CRITERIA ............................. 14
   1. Qualitative Criteria ............................. 14
   2. Quantitative Criteria ......................... 15
D. MEASUREMENT PARAMETERS .......................... 16
E. BENCHMARK EVALUATION DESCRIPTION ............. 17

## III. THE INTEL IAPX-432 MICROPROCESSOR .......... 20

A. ARCHITECTURE DESCRIPTION .......................... 20
   1. Object-Oriented .................................. 21
   2. Transparent Multiprocessing ................... 25
   3. Capability Based Protection ................... 28
   4. Operating System Support ..................... 32
B. ADA LANGUAGE SUPPORT ............................. 36
   1. Object Typing .................................. 36
   2. Domain Objects - Package Objects ........... 38
   3. Procedure Objects - Procedures ............... 39
   4. Activation Records .......................... 40
   5. Tasking ...................................... 40
<table>
<thead>
<tr>
<th>Appendix Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Appendix A (Hardware Description)</td>
<td>75</td>
</tr>
<tr>
<td>Appendix B (Operating System Modifications)</td>
<td>77</td>
</tr>
<tr>
<td>Appendix C (Ada Source Code)</td>
<td>78</td>
</tr>
<tr>
<td>Appendix D (CFA Benchmark Algorithms)</td>
<td>114</td>
</tr>
<tr>
<td>Appendix E (Cds 432/670 Users Manual)</td>
<td>134</td>
</tr>
<tr>
<td>List of References</td>
<td>149</td>
</tr>
<tr>
<td>Initial Distribution List</td>
<td>150</td>
</tr>
</tbody>
</table>
I. INTRODUCTION

A. THESIS DESCRIPTION

The development of new engineering tools is accompanied by the perceived need to find applications for those tools. Microprocessors are no exception. When a new computer is introduced it is important to know what, if any, significant benefits can be realized through its use. Things to consider in evaluating a microprocessor include several quantitative items, such as, instruction execution speed, memory capacity, and overall performance. Less tangible, but equally important qualities of multiprocessor support, user protection, and ease of programming also need to be measured. The introduction of the INTEL iAPX-432 in 1981 represented a radical change from traditional computer architectures. Previous advances in integrated circuits have primarily focused on larger memory, and faster execution. The iAPX-432 has addressed these issues, but has also tackled many of the problems found in software engineering.

This thesis involved the setup and evaluation of a modified INTEL 432/670 cross development system to measure the overall performance of the programming language ADA executing on a companion vehicle, the INTEL iAPX-432
microprocessor. The motivation for this investigation was straightforward. Since the Department of Defense has spent considerable time and effort in developing the ADA language it would be interesting to observe how the language performs on a processor that was designed with identical goals. It is not often that a language and processor are developed in parallel. More importantly, the INTEL iAPX-432's unique architecture directly supports many of the important ADA language features. Such as:

1. Access protection for packages.
2. Automatic maintenance of activation record stacks.
3. Multiprocessor support for multitasking.

This support may provide for less expensive, easier to maintain software, a common objective of both hardware and software designers.

B. EVALUATION OF COMPUTER ARCHITECTURES

Evaluation of computer architectures and computer languages has traditionally been an investigative process directed toward a specific application. This study involved the general purpose applicability of the language and the processor. The choice of measurement methods used followed an earlier effort performed by the Computer Family Architecture committee in 1976 concerning general purpose computer application evaluations. In particular, some of the
benchmark programs used by the committee were coded in ADA and then executed and timed on the IAPX-432. Although no provisions have been made to eliminate the effects of compiler efficiency, or inefficiency, the results should give an indication of the execution speed available to the end user. This method of testing was chosen since the processor is designed to be programmed in a high level language (ADA). No assembler is under development or planned for by the manufacturer. Therefore, if the language and processor are to be used as designed, then the performance needs to be evaluated in a working environment. That is, programmers programming in ADA and compiled code executing on the processor.

C. THESIS ORGANIZATION

This thesis is composed of six chapters and five appendices. Chapter II is a brief discussion of the work done by the Computer Family Architecture committee (CFA) and its applicability to this investigation. Chapter III is an introduction to some of the unique architectural aspects of the IAPX-432 and how these new features support the language ADA. The benchmark program descriptions and timing results are in Chapter IV. Included in that chapter is a description of the parameters passed and the calling conventions used. An attempt has been made to give an impartial evaluation of the CD3 432/670 system in Chapter V. Finally, in Chapter VI.
the reader will find what basic conclusions have been drawn about the IAPX-432 and the CDS 432/670 system as a result of this study. The appendices are filled with the material necessary to repeat any of the results obtained in Chapter IV. They include a description of the hardware and operating system modifications performed and a listing of all the ADA source code. As a convenient reference the algorithms used by the CFA are provided in Appendix D. A users manual is included in Appendix E to allow a new user to quickly become familiar with the system.
II. THE MILITARY COMPUTER FAMILY ARCHITECTURE

The Military Computer Family Architecture (MCF) refers to the architecture standard defined in a study done by the Computer Family Architecture committee (CFA) between October of 1975 and August of 1976. The initial study concluded that the PDP-11 best met the criteria for a military computer family standard. Since that time another CFA related study by Dietz(1) suggested several improvements in the algorithms used to evaluate architectures. An overview of the CFA project follows.

A. CFA/MCF PROJECT MOTIVATION

The CFA/MCF project was a joint ARMY/NAVY effort to develop a method of comparing computer architectures for use on a general class of applications. The enormous sums of money that the Department of Defense was spending on data processing prompted the investigation into the possibility of defining a standard computer architecture. Decreasing the life cycle costs of computer systems played a major role in the committees selection criteria.

B. CFA/MCF PROJECT DESCRIPTION

One of the first items the CFA examined was the reason for skyrocketing data processing costs. The answers they
obtained were not too surprising. That is, computer selections often are based on local schedules, funding, and profit considerations with little regard for the impact these decisions have on long term hardware/software logistics costs. Consequently, incompatible military systems are contributing to the problems of development and maintenance of software. Although a formal movement in standardizing a language was underway (ADA), there was no method for standardizing an architecture. It was with this mandate that the CFA committee pursued the evaluation of several available computer architectures, with the goal of selecting a standard.

A standard architecture does not mean specific numbers of registers, accumulators etc., but rather the structure of the machine that a programmer needs to know to write his programs. For example, if the architecture standard requires stack relative addressing, then any machine having that instruction (and the other required instructions!) can be programmed by a given programmer without his having any knowledge of how the instruction is implemented. The programmer knows there's a stack and a stack relative address instruction; the hardware implementation is transparent to him. In this fashion, any two computers having the standard architecture can run the same software. The advantage realized is that new hardware with faster,
more efficient physical characteristics, can run the same software with little or no modification.

C. CFA SELECTION CRITERIA

The CFA committee initiated the selection process by specifying nine absolute qualitative criteria and several quantitative criteria that they felt an architecture must satisfy to meet the needs of a military computer system.

1. **Qualitative Criteria**

   The nine qualitative criteria were:

   1. **Virtual Memory**: The architecture must support a virtual address to physical address translation mechanism.

   2. **Protection**: The capability must exist to add new experimental programs without endangering the liable operation of existing programs. Architectures with privileged modes of operation generally meet this criteria.

   3. **Floating Point Operations**: The explicit support of floating point data types with more than 10 decimal digits of significance.

   4. **Interrupts and Traps**: The capability to write a trap handler to respond to any trap condition with the program resuming operation of the program. Additionally, the architecture needs to be capable of resuming execution following any interrupt.

   5. **Subsettability**: Some of the components of the architecture must be able to be factored out of the full architecture.

   6. **Multiprocessing**: Support of communication and synchronization of multiple processors.

   7. **I/O Controllability**: A processor must have the ability to exercise absolute control over any I/O processor.
8. Extensibility: Some method needs to exist to add new instructions to the architecture consistent with existing formats.

9. Read Only Code: It must be possible to execute programs from read only memory.

These nine criteria were definitely subjective in nature but did provide a good initial screening for any standard architecture candidate. Although the study was done before the introduction of the INTEL iAPX-432, most of the criteria are met or exceeded by the iAPX-432 with the exception of the interrupt capability. The iAPX-432 has no hardware interrupt, however, it is designed to operate with an attached processor which does have an interrupt capability.

2. Quantitative Criteria

The quantitative criteria judged by the CFA committee included the following items:

1. Virtual address space.
2. Physical address space.
3. Fraction of address space unassigned.
4. Size of the central processor state (amount of CPU information stored on interrupts).
5. Usage base (number of units in operation).
6. I/O initiation (efficiency of peripheral device accessibility).
7. Virtualizability (support of virtual machines).
8. Direct instruction addressability.
9. Maximum interrupt latency (time from receipt of interrupt to processing).

D. MEASUREMENT PARAMETERS

The quantitative criteria were evaluated, in part, by the use of twelve benchmark programs. These programs were hand assembled by several different programmers, and then statistically analyzed for program use of computer space and time. The measurement parameters used were:

S: Measure of space, the number of bytes used to represent a test program.

M: Measure of execution time, the number of bytes transferred between primary memory and the processor during execution of the test program.

R: The number of bytes transferred among internal registers of the processor during execution of the test program.

Although the S, M, and R measures are useful in evaluating conventional architectures, they are not readily applied to the INTEL iAPX-432. In fact, the microprocessor's manufacturer has stated that there is no intention of supplying an assembler, nor is one under development. This would make the measurement of S and M difficult and the measurement of R virtually impossible. For this reason, the evaluation of the INTEL iAPX-432 was primarily based on the execution timing of selected benchmark programs.
E. BENCHMARK EVALUATION DESCRIPTION

The original CFA committee developed twelve benchmark programs to evaluate the selected criteria. A brief description of the programs follows with a complete algorithmic description in Appendix D.

1. I/O kernel, four priority level interrupts.
2. I/O kernel, FIFO processing.
3. I/O device handler.
4. Large fast Fourier transform of a large vector.
5. Character search.
8. Linked list insertion.
10. ASCII to floating point conversion.
12. Virtual memory space exchange.

These programs tested many of the items considered to be of value by the CFA committee, however, a later study by Dietz [1] determined that the number and types of test programs should be expanded. The proposed set of benchmark programs consisted of sixteen programs organized into four groups as follows:
A. Interrupts and traps.
1. Terminal input driver.
3. Multiple priority interrupt handler.
4. Virtual memory exchange.

B. Miscellaneous.
5. Scale vector display.
6. Array manipulation-LU decomposition.
7. Target tracking.
8. Digital communications processing.

C. Address manipulation.
10. Linked list insertion.
11. Presort.

D. Character and bit manipulation.
13. Character search.
15. Record unpacking.
16. Vector to scan line conversion.

A complete algorithmic description of these benchmark programs can also be found in Appendix D.

These sixteen algorithms were thought to more rigorously test specific features of the computer family architecture standard. None of the above benchmark programs are
necessarily firm algorithms that must be adhered to. However, they do provide some guidance in the type of tasks that must be performed by a computer in order for it to fulfill the minimum requirements of an architectural standard. In the original evaluation the PDP-11 was selected as the best candidate architecture for the military computer family. Since that time several major advances in both hardware and software have occurred. The unique architecture of the INTEL IAPX-432 provides a different test platform for the execution of the benchmark programs. Those programs which were supported by the current INTEL ADA-432 compiler were coded, executed, and timed. The results are summarized in Chapter IV of this thesis.
III. THE INTEL IAPX-432 MICROPROCESSOR

A. ARCHITECTURE DESCRIPTION

Computer architectures in the majority of commercial systems available today can be viewed as refined descendants of the often termed Von Neumann computer architecture. A Von Neumann computer architecture usually includes the following properties [2]:

1. A single, sequentially addressed memory which contains both program and data.

2. No explicit distinctions between instructions and data. Rather, instructions and data are distinguished by the operations directed towards them.

In 1981, Intel announced a 32-bit VLSI microprocessor incorporating several architectural innovations [3]. This announcement stated:

"The Intel IAPX 432 represents a dramatic advance in computer architecture: it is the first computer whose architecture supports true software transparent, multiprocessor operation; it is the first commercial system to support an object-oriented programming methodology; it is designed to be programmed entirely in high-level languages; it supports a virtual address space in excess of a trillion bytes; and it supports on the chip itself the proposed IEEE-standard for floating point arithmetic."
The next few pages will be devoted to providing a brief overview of the following architectural aspects of the IAPX-432:

1. Object-Oriented.
2. Transparent Multiprocessing.
3. Capability-Based Protection.

1. Object-Oriented

What does it mean to be an object-based computer? Unlike the classical Von Neumann architecture described in the introduction, memory is not accessed as a single, contiguous block. Rather, the memory is considered as a collection or set of smaller units called objects, each of which occupies some contiguous amount of memory. Very important and fundamental to this concept is the object's recognition. This can occur in software, or as in the majority of cases for the 432, in hardware. This recognition enables the object to be typed or classified as to the operators which are allowed to act upon the particular object. Since the 432 architecture can determine the classification of an object it can prevent incidents such as instructions (e.g. instruction objects) being interpreted as data, and conversely, data (e.g. data objects) being executed as instructions.
At the machine level, objects can be thought of as being segments, a segment being a set of contiguous memory locations which in the 432 case can range from 1 to 65,536 bytes in length. However, there can be some differences in the 432 case. Specifically, an object can be any one of the following:

1. A single segment.
3. A part of a segment.

This latitude in object abstraction gives compiler designers a powerful base on which to build object oriented compilers (ADA).

Intel has moved the recognition of specific object types into the 432 hardware, as alluded to above. Additionally, certain operators on these objects are incorporated directly into hardware, while other operations must be done via software. The net effect of this decision is twofold:

1. Increased reliability of all operations.
2. Increased execution speed of certain functions.

Figure 1 illustrates some typical iAPX-432 hardware recognized objects:
The incorporation of an object-based programming methodology, in the manufacturer’s own words, "...raises the level of the hardware/software interface". The justification for this statement can be found in the following example.

Early computers had very simple hardware operations. These early machines were not capable of supporting floating-point manipulations directly. If you wanted floating-point operations you had to implement them in
software. With the passage of time and increased technological progress, computer hardware gained functionality. What were once software functions found themselves migrated into hardware, a classic example being floating point operations. This evolution of software into hardware is generally regarded as raising the level of the hardware/software interface in a computer architecture. The 432 carries this progression one step further by placing system management operations, such as process scheduling, memory management, and interprocess communication into the hardware also. Referring back to Figure 1, the importance of such objects as processor object, process object, etc. should now take on greater significance. Naturally, more than these basic system objects will be needed to implement the operations listed above. The processor must be able to manipulate these objects in an appropriate way so that what is traditionally done in a series of program steps is now accomplished with a single instruction. The net effect of such hardware instructions is to increase processing speeds.

Recalling the example of floating point operations, we find that their incorporation into hardware increased their speed of operation. Furthermore, speed and reliability are significantly enhanced when an operation is implemented in hardware. However, the capability based architecture adds a significant amount of execution time to each instruction and consequently the performance of a processor is reduced.
The choice of an object-based computer architecture, besides raising the hardware/software interface, integrates ideas that have developed over the last decade in software engineering. These include data abstraction, domain based protection, information hiding, and high-level system functionality. The IAPX 432 is an attempt to bring these notions coherently together in a single architecture.

Summarizing, an object can be regarded as possessing the following properties:

1. A data structure that contains information in an organized manner.

2. A set of basic operations may manipulate an object. The 432 hardware ensures that these are the only operations that can manipulate the data structure.

3. An object can be addressed as a single entity.

4. An object has a label which specifies the object's type (e.g., processor vs. process).

Lastly, as regards the relationship between segments and objects, a segment refers to the physical structure of data in memory, i.e., where the structure is located. An object refers to the logical structure of data in memory, i.e., how the memory is used.

2. Transparent Multiprocessing

One of the most highly promoted features of the IAPX-432 is its software transparent multiprocessing capabilities, also called "incremental computing power".

25
What this means is that the number of physical processors (GDP boards) in the 432/670 system can be changed without any corresponding changes in application software. That is, a user's application program never has to be concerned with the number of physical processors present. The only visible effect of having more than one physical processor is the increase in system throughput. This kind of flexible performance is not usually associated with microcomputers. As applications become more complex and more dynamic, it becomes increasingly difficult to predict how much processing power a system will need to meet its performance goals. This uncertainty can be a serious source of risk. An application may have to commit itself to a processor some time before any code has actually been written. This problem is solved by the iAPX-432 through the use of processor objects. Processor objects are abstractions of physical processors and hence their behavior can be manipulated like any other object.

Transparent multiprocessing is accomplished through the use of the processor object. The existence of a particular physical processor is immaterial. System throughput can be increased by adding physical processors (GDP boards) and therefore creating more processor objects. More processor objects means that more user processes can execute. Similarly, the removal of a physical processor results in the removal of a processor object and a
subsequent reduction in the total performance. Fault
tolerance can thus be said to be improved by the fact that
in a multiple processor environment, if a processor fails,
it is simply removed from the system. The only effect should
be some reduction in throughput. In order to describe how
this "software transparent" multiprocessing is achieved,
other 432 objects besides processor objects and process
objects, will be introduced. Process objects can be equated
with user programs in the discussion which follows.

The term dispatching refers to the assignment of a
432 processor to some process which is waiting to execute.
In the 432 case, this is the pairing-up of a processor
object with a process object. The manner in which this is
done is through the aid of another particular type of object
called a dispatching port object. Since this is an object,
it also has certain unique operators which apply to it. The
dispatching port object can be thought of as a queue-like
data structure which can contain process objects or
processor objects, but never both. Processors, and hence
their processor objects, are self dispatching on the 432.
Therefore, when a processor completes its current task or
process it examines the dispatching port object to determine
if there is a waiting process, represented by a process
object, enqueued at the dispatching port. If there is a
process object present, the process object is "bound" to the
processor object, that is, a link is formed between the
processor object and process object. The processor then dequeues the process object from the dispatching port, and then proceeds to execute the process. Conversely, if there are no processes (process objects) enqueued at the dispatching port, the processor enqueues its processor object at the dispatching port, in effect waiting for the next ready process. Processes are not dependent on specifically which processor is executing it, or how many processors are present in the system. Processes ready for execution are simply enqueued at the dispatching port. The presence of more physical processors simply means that the average time a process is queued up at a dispatching port should be decreased.

3. Capability Based Protection

Sharing data among a computer system's users in a carefully controlled way has been a subject for much investigation in computer systems. Implementation techniques aimed at providing for this controlled information flow have run from introducing privileged and user instructions (e.g., IBM 360/370) to hierarchical protection systems as classically illustrated in the MULTICS ring structure. Intel's approach to this problem in the 432 architecture has been to implement what are termed capabilities.

Capabilities can be thought of as tickets, the possession of which conveys privileges, normally the privilege to access a segment. In the 432 case, to think of
them as a pointer plus access rights pair would be an even closer analogy. Possession of a capability means that access to a segment is allowed under the access rights associated with that capability. Access rights are: read, write, both, or neither. In order to ensure protection, certain processes should not be permitted to possess capabilities which grant non-discriminate access to certain portions of memory. For example, user processes should not have access to the memory where the operating system is contained. Therefore, because of their function and inherent potential to be used maliciously, capabilities must be unforgeable. In the IAPX-432, capabilities are recognized and operated on by hardware to assure this needed protection. The set of capabilities accessible to a process at any one time is called the domain of protection. As a process runs, the domain of protection will change. The ideal to be realized is that the domain of protection should always be exactly matched to the requirements of the process; that is, it should contain capabilities for all the segments that the process needs to access and nothing more. This satisfies the principle of "minimum privilege" in secure systems jargon.

The original reasons that led to the desire to design a computer with a capability based architecture may be summarized under ruggedness and security. Ruggedness in this sense means the ability of the system to survive the consequences of hardware failures or software bugs [4].
Security, on the other hand, can be thought of as ensuring that access to memory is determined exclusively by the access rights of the particular process in question.

There are basically two distinct ways of implementing capabilities in hardware. These can be termed the tagged and partitioned approach [5]. In the former, all words in the system carry a 'tag' bit which plays no part other than to indicate whether the particular word is a capability or not. In the partitioned approach, words carry no tag, so it is not possible by examining a word in memory to determine whether it is a capability or data word. Instead, the type of segment is important, i.e. there must be capability segments which contain capabilities and nothing more, and 'data' segments which contain anything but capabilities. The iAPX 432 uses the partitioned approach.

Intel's decision to implement the partitioned approach causes us to slightly refine the concept of an object as discussed earlier. As was previously stated, objects in their physical form are equated with segment(s). A combination of an object-based architecture with capabilities implemented in the partitioned approach means that each object is composed of two distinct parts, a data part and a capability part. Indeed, in the 432 architecture there are two fundamental segment base types. These base types are called data segments and access segments. A data segment can contain anything except capabilities, whereas an
access segment can contain only capabilities. Therefore, an object should now be correctly envisioned as being comprised of these two segment types. An example of how this is actually implemented for some of the system objects is shown in Figure 2.

![Object Representation](image)

**Figure 2. Object Representation**

Summarizing, Intel has implemented capability-based support for memory protection in the 432. These capabilities can be thought of as an address of, or pointer to, an object with an attached type describing the classification of the
referenced object (e.g., process object, context object, etc.) and an attached protection mode (e.g., read only or read/write). In the 432, Intel has decided to call capabilities access descriptors because of their similarity in concept to pointer implementation in ADA which is termed an "access". Furthermore, objects in the 432 system are seen to be comprised of both data segment(s) and capability (access descriptor) segment(s). The data segment of an object could be thought of as containing information intrinsic to the particular type of object. The capability segment on the other hand, contains capabilities for all the other objects it may need to reference. Additionally, capabilities are seen to enforce the principle of minimum privilege. Perhaps providing an important insight into 432 performance, M.V. Wilkes has said [6]:

"Compared with a conventional computer system, there will inevitably be a cost to be met in providing a system in which the domains of protection are small and frequently changed. This cost will manifest itself in terms of additional hardware, decreased run-time speed, and increased memory occupancy. It is at present an open question whether, by the adoption of the capability approach, the cost can be reduced to reasonable proportions."

4. Operating System Support

Like the 432 hardware, Intel has created an object-oriented operating system for the IAPX 432 called iMAX. It has been designed as a multiprocessor operating system, and consequently it accommodates any number of running
processors. As a result, all synchronization within the system must be explicit. Furthermore, as the manufacturer has pointed out [7], the 432 and IMAX are products primarily intended to be used by original equipment manufacturers in the construction of their products. Related to this is the fact that IMAX does not provide a command language or a human interface, rather it is designed to provide executive services to user-provided software which makes calls to IMAX.

Many traditional operating system primitives are implemented as hardware functions in the 432. In an effort to elaborate on the relationship between the IMAX operating system and the IAPX-432 functions, a digression is in order. As pointed out earlier, the IAPX 432 architecture provides a higher level of functionality in hardware than conventional computers. Important system management functions are realized through hardware-recognized representations, i.e. objects. High level operations on these system objects (see Fig. [1]), such as sending a message between processes, are provided as single machine instructions. These features of the 432 are referred to as the Silicon Operating system. These features are not in themselves an operating system, but contribute greatly to the building of one.

The relationship between IMAX and the hardware might best be described as cooperation. IMAX doesn't simply run on the hardware, rather the hardware acts autonomously to
provide important services, such as processor self-dispatching as pointed out earlier. A good example of the division of labor which occurs between IMAX and the 432 hardware can be found in storage management. Hardware defines system objects used for storage management, provides single instructions that allocate new objects, and sets flags needed for storage reclamation and virtual storage management. IMAX will then provide services which will create and reclaim local storage pools and will provide processes which compact storage and reclaim unreferenced objects.

- Probably the most notable point about IMAX is that the user may view IMAX as a set of ADA package specifications, each of which corresponds to a particular service provided by the system. Additionally, there is no distinction between IMAX packages and user-written packages. IMAX operations and user operations are invoked in the same way. There is no special calling convention, no "Supervisor Call" instruction, as is the case in many current commercial systems. The effect of this particular implementation is twofold:

1. Users can create subsets of IMAX by omitting unused packages.

2. Users can create supersets of IMAX by adding their own packages.
iMAX also benefits from the 432's capability protection mechanism described earlier. References for system objects can be passed to user processes without fear of damage or system compromise because the rights associated with these user process capabilities have been modified by iMAX appropriately (e.g., read only). User processes cannot corrupt these references passed from iMAX.

Like the 432 hardware, iMAX is in a continual state of change by Intel. Version 1.0, which this thesis worked with, is a preliminary version intended to get potential users quickly acquainted with it in order to acquire the ability to execute ADA programs on the 432. As a result, the number of ADA packages which the user can tailor to his or her application are relatively few. As advertised, the following services are provided by iMAX, V1.0:

1. Configure and initialize a multiple-GDP system.
2. Read from and write to the debugger console ONLY.
3. Create and start multiple user processes defined at compile time.
4. Communicate between user processes by exchanging messages.
5. Inspect type, rights, and storage information contained in access descriptors and object descriptors.
6. Inspect context and process dependent information in a running program's environment.
Later versions are supposed to support Attached Processors which are essentially the means by which the 432 can communicate with the outside world. When this support is finally implemented, the current, severely limited I/O (i.e. debugger console only) will be replaced by a variety of conventional I/O devices.

B. ADA LANGUAGE SUPPORT

As was previously mentioned, there was a considerable amount of parallel development between the ADA language and the INTEL IAPX-432. Both the ADA language and the 432 architecture address many of the problems associated with large scale software development projects. This resulted in several architectural constructs which directly support many ADA language features.

1. Object Typing

The object orientation of the architecture plays a major role in language support. Every object is typed by the compiler or by the hardware to indicate its intended use. This allows a natural separation of procedure objects from data objects. In addition to "intended use" typing, the objects are also classified as to their internal structure. This structure can be one of two types, access objects or data objects. The access object is an array of access descriptors (to other objects) while data objects are structured blocks of data information. Access objects
contain only access descriptors and data objects contain only data. This is represented in Figure 3.

As shown in Figure 3, any set of LAPX-432 objects can be represented by a directed graph containing access object nodes and data object nodes. This notational convention serves as a useful model for representing execution time objects and their relationships to corresponding ADA programs. It is important to realize that an object can exist as the subpart of another object and yet be logically different. Such an object that is physically contained inside a parent object is termed a refinement of the parent.
object. The refined objects are physically sub-parts of the
parent object, yet they can inherit the full privileges of
objects, as if they were physically distinct from the
parent. In the case of multiple refinements, they can behave
as if physically distinct from other refinements of the
parent. This is illustrated in Figure 4.

![Figure 4. Refinements](image)

2. **Domain Objects - Package Objects**

Common data structures and procedures can be grouped
together using the ADA package construct. The INTEL IAPX-432
uses a domain object to represent an ADA package. The domain
object, like a package, is a collection of data objects and
procedure objects (hence it is of type access). This can
best be illustrated by the following example of an ADA
package definition and the corresponding IAPX-432 object
representation shown in Figure 5.
Since objects can be refined, it is possible to refine a domain object to create domains of a package with different access rights. This mechanism very nicely supports the public and private access rights defined in ADA. A user is given access to public information by creating a refined object with access descriptors to a refined domain which contains only public data.

3. Procedure Objects - Procedures

An IAPX-432 procedure object consists of executable code corresponding to an ADA procedure. The procedure object
also contains information required to form the activation record or context object which is created on procedure invocation. Procedures may be invoked in either interdomain or intradomain contexts. The interdomain context means that a procedure in one package (domain) is calling a procedure in another package (domain). Intradomain procedure calls are simply calls within the same package.

4. Activation Records

A block structured language such as ADA can make efficient use of activation records. The iAPX-432 supports the use of activation records via context access objects and context data objects. The context access object represents local reference variables and the context data object represents local data variables of the activation record. The iAPX-432 instructions 'procedure call' and 'procedure return' create and destroy context objects.

5. Tasking

One of the important multiprocessing features of the ADA language is the concept of a task. Tasks are directly supported in the iAPX-432 through the use of dispatching and communication port objects. The communication port object is a message queue that acts as a buffer between processes that may be executing concurrently. Its function is to allow inter-process communication. A dispatching port is a special form of a message queue in which a process object may spend time waiting for the arrival of an available processor, or
where a processor object awaits the arrival of a process. These operations are performed in hardware which allows for very efficient coding of the ADA tasking model.

It may be surmised from the previous discussion that the language ADA and the INTEL 1APX-432 have several common foundations. This was undoubtedly intentional. The microprocessor is designed to be programmed using high level languages such as ADA as the development language. No assembler is planned or under development by the manufacturer.
IV. BENCHMARK PROGRAM TIMING RESULTS

A. BENCHMARK PROGRAMS

The benchmark programs were obtained, for the most part, from the CFA algorithms referenced in Chapter II, Section E "Benchmark Evaluation Description". Some programs from a non-CFA related study were also used so that an objective timing comparison could be made with other processors.

1. Methods Used

The programs were coded in ADA, compiled using the INTEL ADA-432 compiler on a VAX-11/780 host computer, linked on the VAX-11/780 using the INTEL 432 linker, and downloaded to a floppy disk via the INTEL asynchronous communications link. Execution of the downloaded object code was performed using the INTEL Debugger and Execution software package operating on a INTEL MDS System 800. The INTEL MDS system is required to load the executable code into the INTEL 432/670 system for execution on the iAPX-432 microprocessor. The system setup is shown Figure 6.

<table>
<thead>
<tr>
<th>VAX 11/780</th>
<th>VAX 11/780</th>
<th>MDS-800</th>
<th>432/670</th>
</tr>
</thead>
<tbody>
<tr>
<td>compile</td>
<td>link</td>
<td>download</td>
<td>execute</td>
</tr>
</tbody>
</table>

Figure 6. COS System Overview
In order to actively simulate large scale software development (and to exercise some unique ADA features) all the coded CFA programs were developed in such a way that the program specifications were separate from the program body. The effect of this decision was twofold:

1. Programs could be written and debugged independently by both authors.
2. The concept and value of using a separate program specification construct could be demonstrated.

A careful inspection of each benchmark program will reveal that it consists of three primary parts. These parts are:

1. Package specification.
2. Package body.
3. Main or driver procedure.

The driver routine is needed to initiate a user process in the 432/670 system. The programs were designed so that the user could control the number of times the benchmark was invoked. This allowed for an effective averaging method. For example, the benchmark could be executed 100,000 times, accurately timed with a stopwatch, and then the total elapsed time could be divided by 100,000 to obtain the average execution time for the procedure. Each program writes a start and a stop message, including an audible 'bell' to indicate when to commence and end timing. In order to effectively isolate the procedure invocation timing
overhead from the benchmark timing, there were usually two different driver routines with each benchmark program. Each program, when executed, would request the number of times to perform the algorithm in question. This request could come from the driver routine or from the benchmark procedure. If it came from the former then the time measured included the time required to invoke the procedure. A timing request from the benchmark procedure included only the timing required to perform the algorithm. The difference in the two times was then a measure of the procedure invocation overhead. Note that this method would not work with a recursive procedure. Further discussion of these methods and the mechanics involved can be found in Chapter V. "CDS 432/670 User Evaluation."

2. Applicable Algorithms

The ADA-432 compiler (Version 1.0) does not support the full ADA language. The manufacturer has added some extensions to the compiler but it presently lacks many important ADA features. Some of the significant compiler limitations are as follows:

1. Fixed point and floating point types are not implemented.

2. Tasking, as defined in the Reference Manual for the ADA Programming Language, is not implemented.

3. Array operations, such as concatenation, assignment, and boolean operations are not implemented.
4. Dynamic arrays and dynamic strings are not implemented.
5. Run time checks are not operational.
6. Exceptions are not implemented.
7. Record representations for records containing fields of type access are not implemented.

Although the above compiler limitations are rather severe it was still possible to code several of the CFA algorithms in ADA-432 and most of those coded could be executed on the iAPX-432. The lack of a hardware interrupt prevented many of the CFA benchmarks from being coded. Future releases of the 432/670 system are supposed to provide the facility of an interrupt through the use of an attached processor. This feature was not available in this release of the 432. A short description of each of the executable programs follows. The complete source code can be found in Appendix C.

a. Character Search

This program searched a given string for the occurrence of an argument string and returned the location of the argument string, if it was located. The program was coded from the algorithm in the original CFA study. The algorithm is listed in Appendix D. The strings were read into a variable of type STRING80, which is an ADA-432 predefined type required for text I/O. The strings were then decomposed into individual characters and assigned to a
I by 256 character array. This method was necessary because of the primitive state of the current ADA-432 text I/O package. The program was made interactive to allow for many searches to be performed in any given debugging/execution session. The data structures, calling conventions, and a sample expected result are shown in Figure 7.

Search String:

```
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>Mo</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>----------------------------------------------------------</td>
</tr>
</tbody>
</table>
1 2 3 4 5 6 7 8 9 . . . . . . . . . . . . . . . . . . . 22
```

Argument string:

```
|-|-|-|
d|a|y|
|-|-|-|
1 2 3
```

Search length := 22
Argument length := 3

SEARCH(Search_length,Arg_length,Search_str,Arg_len,loc)

expected result : loc = 3

Figure 7. Character Search
Two versions of the program were used. One version included the time required to invoke a procedure while the other version did not include procedure invocation overhead. As will be shown in the timing results section of this chapter, procedure invocation is expensive.

b. Quicksort

This program performed a quicksort on a given array of records. The program was coded using the CFA quicksort algorithm in Appendix D. The records sorted consisted of an integer key field (to be sorted on) and a character field associated with each key. A pictorial representation of the data structure and the sorting process and calling convention is shown in Figure 8.

```
|---|---|    |---|---|
| 4 | D |    | 1 | A |
|---|---|    |---|---|
| 1 | A |    | 2 | B |
|---|---|    |---|---|
| 5 | E | SORT | 3 | C |
|---|---|----> |---|---|
| 2 | B |    | 4 | D |
|---|---|    |---|---|
| 3 | C |    | 5 | E |
|---|---|    |---|---|

Calling convention: SORT(Array1,Array2)

expected result: array1 is sorted on the integer field to make array2.
```

Figure 8. Quicksort
The program was written to act interactively with the user to allow for several different runs per debugging session. Two versions of Quicksort were used. One was an iterative sort, the other a recursive sort. The timing results show that the procedure invocation overhead of the recursive sort was significant.

c. Hashtable

This program located the position a key would occupy in a hash table. An example of the data structure used and the calling conventions are shown in Figure 9. The algorithm for this program was obtained from the second CFA study by Dietz[1] and can be found in Appendix D.

```
TABLE

<table>
<thead>
<tr>
<th>HASHES(key)</th>
<th>--</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td></td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>v</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td></td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td></td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>
| position := HASHES( key )
```

Figure 9. Hashtable Data Structure and Calling Convention

Since this program used a function, there was only one version written. The procedure invocation overhead is included in the timing results.
d. Digital Communications Processing

This program sent a message to a given output buffer. The algorithm was taken from the second CFA study by Dietz [1] and is located in Appendix D. The data structures used for the program and a typical calling convention are shown in Figure 10.

<table>
<thead>
<tr>
<th>msg_ptr: A pointer to a message record.</th>
</tr>
</thead>
<tbody>
<tr>
<td>message:</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>destination_connection_message_data</td>
</tr>
<tr>
<td>----------------------------------------</td>
</tr>
<tr>
<td>destination_tbl</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>----------------------------------------</td>
</tr>
<tr>
<td>destination_indexes</td>
</tr>
<tr>
<td>destination_tbl</td>
</tr>
<tr>
<td>which then points to a specific buf_tbl array.</td>
</tr>
<tr>
<td>----------------------------------------</td>
</tr>
<tr>
<td>calling convention:</td>
</tr>
<tr>
<td>FORWARD(destination,connection,msg_ptr)</td>
</tr>
</tbody>
</table>

Figure 10. Digital Communications
The program interactively queried for the message destination, message connection and the message text. This allowed for several sample runs to be performed during a debugging session.

e. Memory Usage

A close inspection of the ADA source code in Appendix C shows that many of the data structures are quite small. This is intentional, and necessary. Early in the course of this investigation it was discovered that programs would compile correctly but execute in an unpredictable manner. The problem was found to be in the amount of heap memory allotted to a user process in Version 1.0 of the iMAX-432 operating system. The memory allocated was not extremely large, and could often be used up without any indication to the user what was wrong. The program Eat-Memory was written to demonstrate how fast memory was used up. The program was fairly simple in that all it did was to create an array of 50 characters and a pointer to the array. This program was also written in two versions, one that created the arrays recursively, the other iteratively. The expense of context creation in a recursive procedure was shown to be very great. Only nine recursive calls could be made before the program used all of the available memory and the system crashed. The iterative version did much better and 199 separate data structures were created before all available memory was exhausted. Of particular interest to
the user is that there is no indication as to what is wrong when the memory is used up. The display is "blank" and all efforts to use the debug facility resulted in a system response of "no current process". In summary, the user must laboriously inspect the program object code (the MAP file) and arbitrarily set breakpoints in the code to determine what the cause of the fatal error is. This problem is elaborated in Chapter V of this thesis.

3. CFA Coded but not Executed Programs

Two programs from the first CFA study were coded in ADA and executed on an ADA-ED compiler to check for correct program execution. These programs were:

1. Linked List Insertion.
2. Runge-Kutta Integration.

Unfortunately the present ADA-432 compiler does not support the floating point data type necessary for the integration program; nor does the ADA-432 compiler support records with access types, which is necessary for the linked list insertion program. The ADA source code for these programs is located in Appendix C for easy reference to allow for possible conversion when a more complete compiler is released.

4. Non-CFA Related Programs

Since the CFA study never actually timed the benchmark programs in terms of execution speed, it was
decided that a physical comparison of the IAPX-432 with other processors would be useful. A previous evaluation of the IAPX-432 by Hansen [3] in June 1982 provided three convenient ADA programs to use. These programs were obtained from the Computer Science Department at U.C. Berkeley, modified slightly to conform with the current ADA-432 compiler requirements, and then executed and timed on the 432/670 system. The three programs used were:

1. Search: Search a 120 character string for a 15 character sub-string.
2. Sieve: Compute prime numbers.
3. Acker: Calculate Ackerman’s function with arguments 3 and 6. This is a recursive computation requiring more than 170,000 procedure calls.

The complete ADA source code for the programs can be found in Appendix C. The timing results are summarized in Chapter IV.

B. TIMING PROCEDURE AND RESULTS

All the CFA programs were written so that the user could write the arguments from the keyboard and select the number of times the program was to execute. Dividing the total elapsed time by the number of times the program executed gave an average value of execution time for the particular benchmark. Procedure invocation overhead was subtracted from the non-recursive procedures and both timing values are shown in the following discussion. In addition, the
parameters used and the number of executions are also listed.

1. **CFA Benchmark Program Results.**

The following sections describe the parameters used for each benchmark executed, the number of executions performed, the total elapsed time (in seconds), and the calculated execution time for a single run. Note that the program name corresponds to the ADA-432 source code for the respective program in Appendix C.

a. **Character Search**

The parameters used in this benchmark timing were:

SEARCH STRING: Monday, June 7th, 1976
ARGUMENT STRING: day
SEARCH LENGTH: 22
ARGUMENT LENGTH: 3

<table>
<thead>
<tr>
<th>Program name</th>
<th>Number of executions</th>
<th>Elapsed time (seconds)</th>
<th>Time (msec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHARS1</td>
<td>100,000</td>
<td>315.6</td>
<td>3.2</td>
</tr>
<tr>
<td>CHARS2</td>
<td>100,000</td>
<td>142.3</td>
<td>1.4</td>
</tr>
</tbody>
</table>

Figure 11. Character Search Results
The program CHAR31 included the time required for 100,000 procedure invocations whereas CHAR32 did not. For this benchmark, Figure 11 shows that the procedure overhead was more than twice the time to perform the algorithm.

b. Quicksort

Two forms of the quicksort algorithm were used, one recursive, the other iterative. A twenty element array was sorted. The worst case array was chosen, that is, all the elements were inversely ordered. Figure 12 represents the parameters passed: unsorted array1 was passed to the procedure and the sorted array2 resulted.

<table>
<thead>
<tr>
<th>Array1</th>
<th>20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array2</td>
<td>1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Program name</th>
<th>Number of executions</th>
<th>Elapsed time (seconds)</th>
<th>Time (msec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>QUICK1 (recursive)</td>
<td>1000</td>
<td>55.8</td>
<td>.56</td>
</tr>
<tr>
<td>QUICK2 (non recursive)</td>
<td>1000</td>
<td>40.5</td>
<td>.41</td>
</tr>
</tbody>
</table>

Figure 12. Quicksort Results
As expected, the recursive procedure took considerably longer to execute. This is not too surprising since the overhead of procedure invocation is included.

c. Hashtable

The hashtable algorithm was implemented as a function. The timing results therefore include the function call overhead. This function used the sample hash table from the CFA study and a key value of 41 was used as the argument of the function. The hashtable's initial values, calling convention, and timing results are shown in Figure 13.

<table>
<thead>
<tr>
<th>key</th>
<th>0 183 11 1035 1035 183 86 0 193 183</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>0 1 2 3 4 5 6 7 8 9</td>
</tr>
</tbody>
</table>

position := HASHES(41)

<table>
<thead>
<tr>
<th>Program name</th>
<th>Number of executions</th>
<th>Elapsed time seconds</th>
<th>Time msec</th>
</tr>
</thead>
<tbody>
<tr>
<td>HASH1</td>
<td>100,000</td>
<td>252</td>
<td>2.5</td>
</tr>
</tbody>
</table>

Figure 13. Hashtable Results

d. Digital Communication Processing

This procedure sent a thirty character message to the output buffer. Figure 14 represents the data values passed to the procedure for processing.
In this case the procedure overhead was nearly one third that of performing the algorithm.

2. Non CFA Program Results

An earlier study of the 432 was performed by Hansen[3] at U.C. Berkeley. Several benchmark programs were coded and executed on various machines and in several different languages. A summary of those results is shown in Figure 15.
<table>
<thead>
<tr>
<th>machine</th>
<th>language</th>
<th>program name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Search</td>
</tr>
<tr>
<td>432 4 MHZ</td>
<td>ADA</td>
<td>14.2</td>
</tr>
<tr>
<td>8086 5 MHZ</td>
<td>PASCAL</td>
<td>7.3</td>
</tr>
<tr>
<td>68000 16 MHZ</td>
<td>PASCAL</td>
<td>1.3</td>
</tr>
<tr>
<td>VAX 11/780 (VMS)</td>
<td>PASCAL</td>
<td>1.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All times are in msec</td>
</tr>
</tbody>
</table>

These results are from a study by HANSEN(3) which were performed on a 432 version 2. The processors manufacturers were: 8086 - INTEL, 68000 - MOTOROLA, VAX 11/780 - DEC.

Figure 15. Previous Non CFA Timing Results

An attempt was made to duplicate the results from the earlier study by executing the benchmark programs on the CNS 432/670 system. The programs that were received from U.C. Berkeley would not compile under Version 1.0 of the compiler supplied with the 432/670 system. No parameters were passed in using these tests, they were included in the code. An examination of the ADA source code in Appendix C will also reveal that no effort was made to separate program body from program specification. The results from our timing are shown in Figure 16.
<table>
<thead>
<tr>
<th>machine</th>
<th>language</th>
<th>program name</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEARCH</td>
<td>SIEVE</td>
<td>ACKER</td>
</tr>
<tr>
<td>432</td>
<td>ADA V 1.0</td>
<td>21.7</td>
</tr>
</tbody>
</table>

*Figure 16. Non CFA Timing Results*  
*ADA Version 1.0*

Extreme caution must be exercised when comparing these values to the previous study. Specifically in the case of the SIEVE and ACKER programs. The limited stack heap available prevented implementing the code exactly as done by U.C. Berkeley. The results of the SEARCH benchmark are very interesting. The three programs received from U.C. Berkeley required some modification before they would compile successfully on the Intel ADA-432 compiler. More importantly, our results generally include the time required for procedure invocation. In some instances, notably our algorithm implementing the character search, we also have results which do not include procedure invocation overhead. Lastly, whereas we used the concept of packages in arriving at the coding of our benchmarks, the U.C. Berkeley programs
did not. These differences are easily seen by referring to Appendix C.

It is not clear whether the results by Hansen[3] include procedure invocation overhead. However, since the 432 used in this thesis had a 5 MHZ clock rate (with an 8 MHZ system clock) as opposed to a 4 MHZ clock rate in the Hansen study, one would suspect that running the same program with the same data would give at the least, comparable results. To our surprise, this was not the case. Initially, we timed the SEARCH algorithm sent from Berkeley "as is". This was timed at 23 milliseconds, quite a difference from 14.2 in the previously cited study. We then modified the Berkeley algorithm so as not to include string initialization each time. Since our first timing was so different from the Berkeley study we thought that string initialization should not be included in the results. The second test was made by just timing the Berkeley search function alone. This included procedure invocation overhead. The result is listed in Figure 16.

C. SUMMARY OF RESULTS

The data in the previous figures pertinent to the CFA studies, is summarized in Figure 17. It is believed by the authors that the following times represent realistic execution speeds available to a user performing in the working environment of the present 432/670 system.

59
<table>
<thead>
<tr>
<th>Program description</th>
<th>execution speed (msec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character Search</td>
<td>1.4</td>
</tr>
<tr>
<td>Quicksort (recursive)</td>
<td>0.56</td>
</tr>
<tr>
<td>Quicksort (non-recursive)</td>
<td>0.41</td>
</tr>
<tr>
<td>Hashtable lookup</td>
<td>2.5</td>
</tr>
<tr>
<td>Digital Communication</td>
<td>2.0</td>
</tr>
</tbody>
</table>

Figure 17. Execution Speed Result Summary

The data reported above does not include the procedure invocation overhead, with the exception of the recursive Quicksort and the function Hashtable Lookup. It needs to be emphasized that the numbers are only "rules of thumb" that should be used in describing the execution speed of the iAPX-432. Compiler differences, and just as importantly the argument used in the algorithm, can significantly affect the
execution speed. For example, if the character string searched for in the Character Search is near the beginning of the search string vs. near the end of the search string, the results can vary by as much as a factor of ten. (The length of the string searched also plays a significant role in determining execution time). The exact arguments passed and the calling conventions used have been described in detail (Chapter IV.A) for future reference and comparison.

The values in Figure 17 represent an approximation to the time required to perform a given algorithm. In order to cross check and verify the timing results, an effort was made to time a single iAPX-432 instruction. This was accomplished by writing two test programs, T100 and T101, which differed by a single line of source code. That is, T100 executed "A := B - C" one hundred times and T101 executed "A := B - C" one hundred and one times. An examination of the MAP file (the compiler output) revealed that the code differed by one statement. That statement was "sub_i", an iAPX-432 mnemonic for subtract integer. The time difference between the two programs could then give a figure for the execution speed of the single sub_i instruction. The measured speed could be directly compared with a previous study [8] which timed individual instruction speeds on a 4MHZ iAPX-432/100 Version1. The results are summarized in Figure 18.
<table>
<thead>
<tr>
<th>Program name</th>
<th>Number of sub_i executions</th>
<th>time/sec</th>
<th>difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>T100</td>
<td>40,000,000</td>
<td>777.8</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>6.90</td>
</tr>
<tr>
<td>T101</td>
<td>40,400,000</td>
<td>784.7</td>
<td></td>
</tr>
</tbody>
</table>

Execution time

\[
\text{sub}_i \quad = \frac{6.90}{400,000} = 1.73 \times 10^{-5} \text{ sec}
\]

<table>
<thead>
<tr>
<th>sub_i Version 1 5MHZ</th>
<th>sub_i Version 2 5MHZ</th>
</tr>
</thead>
<tbody>
<tr>
<td>estimated cycles</td>
<td>measured cycles</td>
</tr>
<tr>
<td>77</td>
<td>86</td>
</tr>
</tbody>
</table>

Estimated cycles are from an earlier study [8] on a 432/100 system and represent a projection based on measured results. Version 2 measured cycles are the result of the product of execution time and the clock rate.

Figure 18. Individual Instruction Timing

As can be seen in Figure 18, the measured speed of the sub_i instruction in this study is in good agreement with the previous results. The differences can possibly be accounted for in the fact that two different versions of the microprocessor are being compared.

An attempt was also made to eliminate the effects of "dead time", or "time out" in the execution of a program. This time out is the period during which a process is suspended while the dispatching port is checked for another process to be assigned to a processor. Normally a process is
given a default value of 0.2 seconds of dedicated processor time between time outs. Since only one program was executing at a time, it was not believed that the program timing results would be significantly affected by the dispatching port check overhead. To verify this, a modification was made to the INTEL supplied ADA package PSERP.MBS. The modification increased the time slice from 0.2 seconds to 2 seconds. Similar programs that differed only in the time slice period (0.2 seconds vs 2 seconds) executed within 0.5 seconds of each other over a total execution time of 200 seconds. This confirmed that the time slice period between dispatching port checks was not significantly interfering with the benchmark results.
V. CDS 432/670 USER EVALUATION

In the process of working on this thesis both authors felt that a section devoted to constructive criticism of the INTEL Cross Development System would be appropriate. By Cross Development System we mean the INTEL ADA compiler, linker, downloading and execution software and corresponding documentation. Additionally we conclude with some of our thoughts on ADA. We understand that many of the problems addressed here are not permanent, and very likely many of the items we have found to be mysterious or irksome may have already been corrected in a later release.

The INTEL 432/670 system can be conveniently divided into four major components:

1. Compiler.
2. Linker.
3. Downloading and Asynchronous Communication.
4. Debugging and Execution.

The following discussion will treat each component in turn, stating what positive and negative attributes we found.

A. COMPILER

The ADA-432 compiler does not support full ADA. The language limitations are listed in Chapter IV. Of these, the
lack of floating point number support was felt to be extremely burdensome to this thesis. A great many of the CFA measures are focused on floating point manipulation, as are many real world applications. At the machine level, the IAPX-432 has outstanding floating point support, such as multiply, divide, and square root machine instructions. The lack of compiler support for floating point operations prevented us from testing programs in an area where the IAPX-432 should provide outstanding performance.

The present text I/O package provided in the ADA-432 compiler can best be described as primitive. The user is given a choice as to how messages can be input and output to and from the screen, that is, the message can be 1, 10, 20, 30 or 80 characters long, and of no other length. Counting the number of characters in one’s input and output text significantly detracts from the art of programming.

Compilation of a user’s ADA source code is performed on the host VAX 11/780 and it proceeds at a respectable rate, the turn around time was always less than a minute. The number of compilation errors is displayed at the end of compilation, however the reason for the errors is not. To evaluate the compilation errors, INTEL has supplied a very useful report facility which is an image of the original ADA source code with errors identified by a diagnostic message and code number. Unfortunately, many of the error code
numbers in the INTEL reference manual just repeat the same diagnostic error message, with no further elaboration.

There was one very frustrating aspect of the compiler output to the screen. That is, after compilation is complete, there is no message as to what unit was just compiled. Since the compiler output often scrolls the screen, this leaves it up to the user to remember what unit has just been compiled. ADA programs consist of many units, and in more than one instance we found ourselves recompiling a unit that had just been compiled. A very simple solution to this would be to output the compiled unit's name as the last line of output along with the error messages.

As with most new compilers, there are some errors. The more significant of these are the type that allowed compilation of code representing features that are not yet implemented. For example, array assignments are not yet operational, yet a source code program containing them compiles with no error messages. Execution, as expected, does not occur. Most of the ADA restrictions are well documented in the error report file, however, it only takes one or two which are not identified to cause significant problems in debugging a program. At least one type of error crashes the compiler. That is, a program which needs a large data structure may never compile and furthermore the user will never be informed as to the reason for the failure. This problem occurred with the following program unit:
type item is
  record
    key : integer;
    data : character;
  end record;
type array_one is array(1..2000) of item;
begin
  
When array_one had 2000 elements the program unit crashed the compiler. Lowering the number of elements to 200 allowed satisfactory compilation.

B. LINKER

The linking process of a users program is tedious. A separate link program needs to be written for each program that is to be linked. The time to link a program is considerable, usually in the range of two to three minutes. Many default parameters occur in the linking process which can be changed by directives in the users link program. No problems were experienced with the default values, but depending on a default value for proper program execution can easily lead to difficult debugging errors in future program maintenance. In our opinion all the directives should be required to be explicitly stated.

The linker has at least one ambiguous characteristic. After a successful linkage, a message is written to the
screen which states "LINKAGE SUCCESSFUL". This message may also be accompanied by one or more warning messages. In every case that we experienced, if a warning message occurred during linking then the program would not execute. The message "LINKAGE SUCCESSFUL" can be very misleading.

C. DOWNLOADING

The process of downloading programs from the host VAX 11/780 system is probably the biggest drawback to the 432/670 system. Since the iMAX operating system is part of the downloaded object files (EOD), the files requiring transfer are quite large. A typical small ADA program (less than 100 lines of source code) takes nearly twenty minutes to download at 2400 baud. This makes program changes very time consuming. Even if a 9600 baud line were used, the entire process of correcting source code, re-compiling affected modules, and then downloading them, requires a significant amount of time. There is a program called UPDATE for merging a recompiled module of a program with the existing EOD file. The smaller re-compiled module is much faster to download, about seven minutes, but the UPDATE program takes about 3 to 4 minutes to execute. The time saved was not considered significant to warrant use of the UPDATE feature. Especially since a new link program would have to be written each time it was desired to recompile a portion of a program.
D. DEBUGGING AND EXECUTION

Our impression of the debug facility was favorable. It allowed for access to the program structure at an assembly language level. This did not allow any type of assembly-like programming, but did provide a means to locate errors in our source code by mapping the error location to a source code statement number. A very useful utility is the LOG program. This allows everything that was input or output at the terminal to be logged for future reference. The debug facility could be made much more user friendly by implementing the ADA exception features. At present, the lack of exceptions means that run time errors may not be reported, and indeed may cause the system to crash with no indication to the user as to the cause. An example of this occurred when one of our programs attempted to index an array outside the declared array bounds. No error messages were reported, and the system crashed.

The execution of a program was difficult to initiate. The following sequence of commands represents the minimum time required to execute a program after the power is turned on and the ISIS-II operating system is booted. The times are approximate and they depend on the size of the program that is going to be executed.
command | time required
-------|-----------------
RUN WORK IF01 | .5 min.
RUN DEB432      | 1 min.
INCLUDE DEB432.TEM | 1 min.
INIT           | 1 min.
DEBUG "userprogram" | 1 min.
START          | 1 min.

Once the system debugger is loaded (once per session) things proceed a little faster. Only the last three commands of INIT, DEBUG, and START are required per program.

E. ADA IMPRESSIONS

One of the many interesting facets of working on this thesis was the exposure to the new DoD language ADA. Inasmuch as our use of ADA was limited to the benchmarks in this thesis, plus the fact that we dealt with a compiler which did not fully implement the language, our impressions are limited. However, the features of ADA we did exercise left us with some favorable impressions.

The feature we used and liked most was the ability to separate the specifications of a program from the corresponding body of the program. The package feature of ADA was used to do this. A specification package is simply the formalizing in ADA of what the interface of the program is to be, i.e., the "what" of the program. The body package on the other hand is the formalizing in ADA of the manner in
which one plans to implement the program, i.e., the "how" of the program. The contribution of this separation is twofold:

1. Given a specification package, a programmer is free to implement the program in the manner he or she sees fit, so long as it satisfies the specification, or interface.

2. Users of a particular program or programs need only be given the specification package in order to discern what the particular code can do for them. The "how" of the code, or the body package, need not concern them.

Using this technique in very large software projects should have a significant effect on software development and maintenance. In our small scale projects the separation of specification and body allowed for easy parallel development of the benchmark programs. The acceptance of ADA by DoD computer personnel could be seen to lead to:

1. The growth of software libraries with specification packages as the user interface to the library.

2. Greater productivity among programmers. For instance, suppose a decision is reached on what a particular piece of software is to do. This "what" is formalized in ADA, and given to the programmer(s). The programmer is now free to bring all of his or her abilities to bear on successfully implementing the body, or the "how" of the piece of software.

Both of these abilities are generally regarded to be very worthwhile, something which up to now has been pursued with no great degree of success. Supporting and thereby
facilitating this feature of packages is the separate compilation ability of ADA, while still enforcing strong type-checking of interfaces. That is, making sure that parameters in the body package are of the exact same kind as those delineated in the specification, which may have been compiled some time before actual coding was ever begun on the body.
VI. CONCLUSIONS

In its present state the INTEL Cross Development System (CDS) is very much a development tool. Areas which we feel could be changed to improve the user friendliness of the system have been presented in the previous chapter.

As an execution vehicle for the ADA language, the processor seems especially well suited. However, the incompleteness of the compiler did not permit us to rigorously exercise the 432 as much as we wanted to. Though the 432 and ADA seem especially well matched, it is not reflected in program execution speed. An object-oriented architecture, which also incorporates system management facilities in hardware, undoubtedly must have some drawbacks. In this version of the 432, this was unfortunately reflected in execution speed. As an aside, when the compiler comes to support floating point operations, benchmarks which exercise floating point manipulations should provide some interesting results. As elaborated previously, hardware support for floating point operations in the 432 are outstanding.

The lack of a hardware interrupt is a handicap that should be capable of being overcome through the use of the attached processor. This feature was not operational on the 432/670 system and therefore could not be tested.
The timing performance of the system, at first glance, does not present a very favorable impression. The benchmark programs that were compared with the previous study by Hansen[3] confirmed that the 432 is slow in its execution speed. Execution speed is but one of many measures of any computer architecture. It is, however, a measure which readily lends itself to numerical analysis as opposed to qualitative features which do not. This subjective qualitative category can include such items as the amount of fault tolerance and protection available.

The multiprocessor capabilities of the 432 provide a case study in some of the issues which must be addressed by any system using multiprocessing. Moreover, the system in general permits one to analyze the more basic concepts of an operating system. Processes, inter-process communication, ready, running, and blocked states are all generic terms to the architecture. Any study of the processor’s architecture cannot help but to provide an excellent insight into these concepts.

Finally, the architecture has been designed to be programmed in a high level language only. As the compiler inefficiencies are removed and the cost of procedure invocation is lowered the 432 should show a marked improvement in its overall performance.
APPENDIX A
HARDWARE DESCRIPTION

This thesis used a modified INTEL MDS SYSTEM 800 interfaced with the IAPX-432 execution vehicle. This setup required a special circuit board to allow communication between the MDS 800 system and the 432/670. The chassis name, slot number, and board number of the system components used in this evaluation follow.

Card cage number to circuit board identification

MDS-800 board description:


432/670 board description:

1. 2. 3. 4.
5. MEMORY INTEL 112340-004 REV C 112354-001 REV C
   S/N 000279
6. MEMORY INTEL 112340-004 REV C 112354-001 REV C
   S/N 000262
7. MEMORY CONTROLLER INTEL 172075-005 REV E
   S/N -xp-000033
8. GDP INTEL 432/601 005 REV F S/N=xp-000187
9. GDP INTEL 432/601 MF=006 REV H S/N=xp-000104
10. GDP INTEL 11/16/82 432/601 MF=005 REV F
    S/N=xp-000095 MD=17-0003
11.
12. IP-LINK INTEL 432/603 172028-004 REV E
    S/N=xp-000-227
APPENDIX B
OPERATING SYSTEM MODIFICATIONS

The iMAX-432 operating system supplied with the 432/670 was not compatible with the hardware configuration. Specifically, interface processors are not yet supported, even though the iMAX-432 operating system is configured for them. This necessitated a change to the ADA package body that describes the system processor configuration. The name of this package is PSORS.MBS. The code referring to the number of processors and interface processors in the package body PSORS.MBS must be changed to reflect the current physical state of the 432/670 system. For a three GDP board configuration with no IPL boards, the PSORS.MBS would include the following description:

```ada
-- Define GDP boards present
package psor1 is new GDP_Def(psor_num => 1);
package psor2 is new GDP_Def(psor_num => 2);
package psor3 is new GDP_Def(psor_num => 3);
processor1: processor retypes psor1.psor;
processor2: processor retypes psor2.psor;
processor3: processor retypes psor3.psor;
-- Define empty slots
processor3: constant processor := null;
processor4: constant processor := null;
processor5: constant processor := null;
```

A complete discussion as to how these changes can be incorporated in the PSORS.MBS package can be found in Reference 7.
APPENDIX C

ADA SOURCE CODE

All of the benchmark programs that were coded in ADA follow. Most programs are composed of three parts. That is, a package specification, package body, and a driver or main routine. The respective parts are labeled accordingly. The programs obtained from U.C. Berkeley are composed of just a single main routine. For easy cross reference the program name and the corresponding benchmark program are listed below.

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Program Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHARS1</td>
<td>Character search with procedure overhead.</td>
</tr>
<tr>
<td>CHARS2</td>
<td>Character search without procedure overhead.</td>
</tr>
<tr>
<td>QUICK1</td>
<td>Quicksort iterative</td>
</tr>
<tr>
<td>QUICK2</td>
<td>Quicksort recursive</td>
</tr>
<tr>
<td>HASH1</td>
<td>Hash function</td>
</tr>
<tr>
<td>DCOM1</td>
<td>Digital Communication with procedure overhead.</td>
</tr>
<tr>
<td>DCOM2</td>
<td>Digital Communications without procedure overhead.</td>
</tr>
<tr>
<td>MEM1</td>
<td>Recursive memory test</td>
</tr>
<tr>
<td>MEM2</td>
<td>Iterative memory test</td>
</tr>
<tr>
<td>SEARCH</td>
<td>U.C. Berkeley character search</td>
</tr>
<tr>
<td>SIEVE</td>
<td>U.C. Berkeley prime number generator</td>
</tr>
<tr>
<td>ACKER</td>
<td>U.C. Berkeley Ackerman's function</td>
</tr>
</tbody>
</table>
In addition to the programs above, two other programs were
coded in ADA but were not executed due to compiler
limitations. The Runge-Kutta integration was coded and the
source code appears under the program name RUNGE. Some of
the programs were extensively tested under an ADA-ED
interpreter. The linked list insertion program was written
and tested in ADA-ED and the source code for it is under the
program name LINK. The reader is warned that these two
programs, RUNGE and LINK have NOT been tested under ADA-432
and some modifications may be necessary to get them to
execute.
-- CHARSI package specification

--

-- This is the ADA specification package for the
-- CFA character search benchmark.

-- CHARSI

package SCHAR is
  subtype subint is integer range 1..256;
type ttxtarray is array(1..256) of character;
array1, array2 : ttxtarray;
procedure RDFIL;
procedure SEARCH(srcstrnlen, arglen : IN subint;
                 array1, array2 : IN ttxtarray;
                 loc : OUT subint);
end SCHAR;

-- CHARSI package body

--

pragma environment("ACS:TEXTIO.MLE","INTIO.MSE","SCHAR,MSE");
with text+io,intio; use text'io,intio,ascii;
package body SCHAR is
  procedure RDFIL is
    line+of+inout : string80;
    char : character;
    i,j : integer;
    begin
      skip+line;
      new+line();
      put+line+30("Enter Srch-strnq, $ ends.......");
      i:=1;
      j:=1;
      while j < 256 loop
        line+of+inout := Get+line+80();
        exit when line+of+inout(1) = '$';
        for j in 1..80 loop
          exit when line+of+inout(j) = ' ' and
                line+of+inout(j+1) = ' ';
          array1(i) := line+of+inout(j);
          i := i+1;
        end loop;
      end loop;
    -- fill array 2

new-line();
one-line=30("Enter Srch-arg, $ ends.........");
i := 1;
while i < 256 loop
    line-of-inout := Get-line+80();
    exit when line-of-inout(1) = '$';
    for j in 1..80 loop
        exit when line-of-inout(j) = '' and
        line-of-inout(j+1) = ''; 
        array2(i) := line-of-inout(j);
        i := i+1;
    end loop;
end loop;

    -- check the array's contents
    --
    new-line();
    for i in 1..80 loop
        out(array1(i));
    end loop;
    new-line();
    for i in 1..80 loop
        out(array2(i));
    end loop;
    out-line=10("end RDFIL ");
end RDFIL;
procedure SEARCH(srchlen, arglen : IN integer;
array1, array2 : IN txtarray;
loc : OUT integer) is
i, j : integer;
begin
    i := 1;
    j := 1;
    loc := -1;
    while i <= srchlen loop
        if array1(i) = array2(j) then
            if j+1 <= arglen then
                i := i+1;
                j := j+1;
            else
                loc := i-j;
                exit;
            end if;
        else
            i := i+1;
            j := 1;
        end if;
    end loop;
end SEARCH;
end SCHAR;

CHARS1 driver routine

--
pragma environment("ACS:TEXTIO.MLE","INTIO.MSE","SCHAR.MSE",
"MAIN.MSE");
with text+io, intio, schar; use text+io, intio, schar, ascii;

--
-- RDFIL and SEARCH contained in the same package
-- Timing also includes time for procedure
-- invocation.
-- 14 Oct. 1982
--
package body USER+PROCESS+1 is
procedure MAIN is
  i, loc, src+length, src+arg, timer+loop : integer;
  forever : boolean :=true;
  answer : character;
begin
  while forever loop
    -- initialize the arrays
    for i in 1..256 loop
      array1(i) := ' ';
      array2(i) := ' ';
    end loop;
    -- get the search arguments
    new+line();
    out+30("Character search Q=Quits........");
    get(answer);
    exit when answer = 'Q';
    RDFIL;
    new+line();
    out+30("Length of string to search?
    get(src+length); new+line();
    out+30("Length of string to search for");
    get(src+arg); new+line();
    out+30("Number of loops to time........");
  end loop;
end MAIN;
get(timer+loop);
new+line();
out+20(" Start of Search.....");
out(BEL);
for i in 1.. timer+loop loop
SEARCH(src+length, src+array, array1, array2, loc);
end loop;
out(BEL);
new+line();
out+20("end the search.....");
new+line();
out+10("Location= ");
out(loc);
skip+line;
end loop;
end MAIN;
end USER+PROCESS+1;
package SCHAR is
  type txtarray is array(1..256) of character;
  array1, array2 : txtarray;
procedure RDFIL;
procedure SEARCH(srcrlen, arglen : IN integer;
  array1, array2 : IN txtarray;
  loc : OUT integer);
end SCHAR;

package body SCHAR is
  procedure RDFIL is
    line_of_input : string80;
    char : character;
    i, j : integer;
begin
  skip_line;
  new_line();
  out_line30("Enter Srch-string, $ ends.......");
  i := 1;
  j := 1;
  while i < 256 loop
    line_of_input := Get_line80();
    exit when line_of_input(1) = '$';
    for j in 1..80 loop
      exit when line_of_input(j) = ' ' and
      line_of_input(j+1) = ' '; 
      array1(i) := line_of_input(j);
      i := i+1;
    end loop;
  end loop;

  fill array 2
  new_line();
  out_line30("Enter Srch-array, $ ends.......");
  i := 1;
  while i < 256 loop

84
line+of+input := Get+line+80();
exit when line+of+input(1) = '$';
for j in 1..80 loop
exit when line+of+input(j) = ' ' and
line+of+input(j+1) = ' ';
array2(i) := line+of+input(j);
i := i+1;
end loop;
end loop;

-- check the array's contents
--
end RDFIL;

--
procedure SEARCH(srcshlen, arlen : IN integer;
array1, array2 : IN txtarray;
loc : OUT integer) is
i, j, k, timer+loop : integer;
begin
new+line();
out+30("Number of loops to time......."); get(timer+loop);
new+line();
out+20("Start of search.......");
out(BEL);
for k in 1..timer+loop loop
i := 1;
j := 1;
loc := -1;
while i <= srcshlen loop
if array1(i) = array2(j) then
if j+1 <= arlen then
i := i+1;
j := j+1;
else
loc := i-j;
exit;
end if;
else
i := i+1;
j := 1;
end if;
end loop;
end loop;
out(BEL);
out+20("end the search.......");
new+line();
end SEARCH;
end SCHAR;
--
-- CHARS2 driver routine
--
pragma environment("ACS:TEXTIO.MLE","INTIO.MSE","SCHAR.MSE","MAIN.MSE");
with text+io,intio,schar; use text+io,intio,schar,ascii;
--
-- RDFIL and SEARCH contained in the same package
-- Timing is for the SEARCH only. Prompts are from
-- the SEARCH procedure
--
package body USER+PROCESS+1 is
  procedure MAIN is
    i,loc,srch-length,srch-arg : integer;
    forever : boolean :=true;
    answer : character;
  --
  begin
  --
  put+30("chars2 with 4 odo configurat..");
  new+line();
  --
  while forever loop
  --
  -- initialize the arrays
  --
  for i in 1..256 loop
    array1(i) := ' ';
    array2(i) := ' ';
  end loop;
  --
  -- get the search arguments
  --
  new+line();
  put+30("Character search Q=Quits......");
  get(answer);
  exit when answer = 'Q';
  RDFIL;
  new+line();
  put+30("Length of string to search?..." );
  get(srch-length);
  new+line();
**Length of string to search for**: 
```
put+30("Length of string to search for");
get(srcb+arg);
new+line();
SEARCH(srcb+length,srcb+arg,arrow!,arrow2,loc);
put+10("Location= ");
put(loc);
skip+line;
end loop;
end MAIN;
end USER+PROCESS+1;
```
-- QUICK1  package specification
--
-- QUICKSORT  package specification (Iterative)
--
package QUICKSORT is
  type item is
    record
      key : integer;
      data : character;
    end record;
  type inarray is array(1..20) of item;
--
procedure SORT(arg : IN OUT inarray);
--
end QUICKSORT;
--
--
-- QUICK1  package body
--
-- QUICKSORT  package body (Iterative)
--
pragma environment("ACS:TEXTIO.MLE","INTIO.MSE", "QUICK.MSE");
with textio,intio,quicksort;
use textio,intio,quicksort;
package body QUICKSORT is
  procedure SORT(arg : IN OUT inarray) is
    m : constant := 20;
    i,j,l,r : integer;
    mid,temp : item;
    type stack-frame is
      record
        l,r : integer;
      end record;
    stack : array(1..m) of stack-frame;
  s : integer;
    begin
      l := 1;
      r := 20;
      s := 1;
      stack(1).l := 1;
      stack(1).r := 20;
      loop
        l := stack(s).l;


```
r := stack(s).r;
s := s+1;
loop
  i := 1;
  j := r;
  midpt := arg((1+r)/2);
  loop
    while arg(i).key < midpt.key loop
      i := i+1;
    end loop;
    while midpt.key < arg(j).key loop
      j := j-1;
    end loop;
    if i <= j then
      temp := arg(i);
      arg(i) := arg(j);
      arg(j) := temp;
      i := i+1;
      j := j-1;
    end if;
    exit when i > j;
  end loop;
  if i < r then
    s := s+1;
    stack(s).l := i;
    stack(s).r := r;
  end if;
  r := j;
end loop;
exit when s = 0;
end loop;
end SORT;
end QUICKSORT;
--
-- QUICK! driver routine
--
-- QUICKSORT package body for Driver (Iterative)
--
pragma environment("ACS:TEXTIO.MLE","QUICK.MSE",
"INTIO.MSE","MAIN.MSE");
with quicksort,text+io,intio;
use quicksort,text+io,intio,ascii;
package body USER+PROCESS+1 is
  procedure MAIN is
    arg,temp+array : inarray;
```
i, loop+val, j : integer;
data : boolean := true;

Begin
  for i in 1..20 loop
    arg(i).key := 0;
    arg(i).data := 'a';
  end loop;

new-line();
out-line-20("QUICKSORT BENCHMARK ");
out-line-20("Iterative Version...");
out-20("Enter key, followed ");
out-20("immediately by data.");
out-line-20(" O terminates....... ");
new-line();
i := 1;
while data loop
  get(arg(i).key);
  exit when arg(i).key = 0;
  skip-line;
  get(arg(i).data);
  i := i+1;
  skip-line;
end loop;
new-line();
out-line-10("Your Input");
for i in 1..20 loop
  out(arg(i).key);
  out(arg(i).data);
end line();
end loop;

Loop
out-30("Number of loops to time........ ");
get(loop+val);
exit when (loop+val) = 0;
new-line();
for i in 1..20 loop
  temp-array(i).key := arg(i).key;
  temp-array(i).data := arg(i).data;
end loop;
out-20("Start of Quicksort...");
out(bel);
for i in 1..(loop+val) loop
  for j in 1..20 loop
    arg(j).key := temp+array(j).key;
    arg(j).data := temp+array(j).data;
  end loop;
end loop;
end loop;

end loop;
SORT(arg);
end loop;
out(bel);
newline();
putline(20("End the Quicksort...");
putline(10("The Output");
for i in 1..20 loop
  put(arg(i).key);
  put(arg(i).data);
  newline();
end loop;
end Loop;
end MAIN;
end USER+PROCESS+1;
package specification

package specification (Recursive)

package QUICKSORT is

  type item is
    record
      key : integer;
      data : character;
    end record;

  type inarray is array(1..20) of item;

  subtype subint is integer range 1..20;

procedure SORT(left, right : in subint; arg : in out inarray);

end QUICKSORT;

package body QUICKSORT

procedure SORT(left, right : in subint; arg : in out inarray) is

  i, j : subint;
  mid, temp : item;

begin
  i := left;
  j := right;
  mid := arg((left+right)/2);

  loop
    while arg(i).key < mid.key loop
      i := i+1;
    end loop;

    loop
      while mid.key < arg(j).key loop
        j := j-1;
      end loop;

  if i <= j then
    temp := arg(i);
    arg(i) := arg(j);
    arg(j) := temp;
  end if;
  end loop;

end QUICKSORT;

package body QUICKSORT

pragma environment("ACS:TEXTIO.MLE","INTIO.MSE", "QUICK.MSE");

with text+io, intio, quicksort; use text+io, intio, quicksort;

procedure SORT(left, right : in subint; arg : in out inarray) is

  i, j : subint;
  mid, temp : item;

begin
  i := left;
  j := right;
  mid := arg((left+right)/2);

  loop
    while arg(i).key < mid.key loop
      i := i+1;
    end loop;

    loop
      while mid.key < arg(j).key loop
        j := j-1;
      end loop;

  if i <= j then
    temp := arg(i);
    arg(i) := arg(j);
    arg(j) := temp;
  end if;
  end loop;

end QUICKSORT;
arg(i) := arg(j);
arg(j) := temp;
i := i+1;
j := j-1;
end if;
exit when i > j;
end loop;
if left < j then
SORT(left, j, arg);
end if;
if i < right then
SORT(i, right, arg);
end if;
end SORT;
end QUICKSORT;

-- QUICK2 driver routine
-- QUICKSORT package body for Driver (Recursive)
--
pragma environment("ACS:TEXTIO.MLE","QUICK.MSE",
"INTIO.MSE","MAIN.MSE");
with quicksort, text+io, intio;
use quicksort, text+io, intio, ascii;
package body USER.PROCESS+1 is
procedure MAIN is
arg, temp+array : inarray;
left+index, right+index : subint;
i,1000+val, j : integer;
data : boolean := true;

Begin
for i in 1..20 loop
arg(i).key := 0;
arg(i).data := 'a';
end loop;

new+line();
out+line+20("QUICKSORT BENCHMARK ");
out+20("Enter key, followed ");
out+20("immediately by data,");
out+line+20(" 0 terminates....... ");
new+line();
i := 1;
while data loop

get(arg(i).key);
exit when arg(i).key = 0;
skip line;
get(arg(i).data);
i := i + 1;
skip line;
end loop;
new line();
put line+10("Your Input");
for i in 1..20 loop
out(arg(i).key);
out(arg(i).data);
new line();
end loop;

Loop
out+30("# of loops to time?..0 exits ");
get(loop+val);
exit when (loop+val) = 0;
new line();
for i in 1..20 loop
    temp+array(i).key := arg(i).key;
    temp+array(i).data := arg(i).data;
end loop;
out+20("Start of Quicksort...");
out(bel);
for i in 1..(loop+val) loop
    for j in 1..20 loop
        arg(j).key := temp+array(j).key;
        arg(j).data := temp+array(j).data;
    end loop;
    left+index := 1;
    right+index := 20;
    SORT(left+index, right+index, arq);
end loop;
out(bel);
new line();
out+line+20("End the Quicksort...");
out+line+10("The Output");
for i in 1..20 loop
    out(arg(i).key);
    put(arg(i).data);
    new line();
end loop;
new line();
end Loop;
end MAIN;
end USER+PROCESS+1;
package HASH is
  size : integer := 10;
  table : array(0..9) of integer;
  function HASHES(key : IN integer) return integer;
end HASH;

package body HASH is
  function HASHES(key : IN integer) return integer is
    check, i : integer;
    Begin
      -- compute the first place to look
      check := key mod size;
      for i in 1..size/2 loop
        if table(check) = key or table(check) = 0 then
          return check;
        else
          check := (check+i) mod size;
        end if;
      end loop;
      return 0;
    end HASHES;
end HASH;

package body HASH is
  function HASHES(key : IN integer) return integer is
  end HASH;

package HASH is
  hash table search benchmark
  hash1.end on disk
  timing includes procedure invocation overhead
  pragma environment("ACS:TEXTIO.MLE","INTIO.MSE","HASH.MSE");
  with text+io, intio, HASH;
  use text+io, intio, HASH, ascii;
package body user+process+1 is

procedure main is
  timer+loop, position, key; : integer;
  answer : character;
  forever : boolean := true;
begin
  new+line();
  out+20("HASH1 benchmark.....");
-- fill the hash table with CFA sample entries
  table(0) := 0;
  table(1) := 183;
  table(2) := 11;
  table(3) := 1035;
  table(4) := 1035;
  table(5) := 183;
  table(6) := 86;
  table(7) := 0;
  table(8) := 183;
  table(9) := 183;
while forever loop
  new+line();
  out+20("Continue?  Q: quits.");
  get(answer);
  exit when answer = 'Q';
  new+line();
  out+20("enter an integer key");
  get(key);
  new+line();
  out+30("number of loops to time........");
  get(timer+loop);
  new+line();
  out+20("start hash lookup..."UBLISH
  end loop;
  for j in 1..timer+loop loop
    position := HASHES(key);
  end loop;
out(bel);
new-line();
out+20("end of hash lookup...");
new-line();
out+20("hash position = ....");
out(position);
skip-line;

end loop;

end main;
end user-process=1;
package specification

Digital Communication Processing Program

19 Oct 82

pragma environment("ACS:TEXTIO.MLE");
with textio; use textio;

Package DIG+COM is
constant c1 := 10;
c2 := 10;
subtype dest-type is integer range 1..c1;
subtype con-id-type is integer range 1..c2;
type message is record
  destination : dest-type;
  connection : con-id-type;
  size : integer;
  data : string30;
end record;
subtype buf-index is integer range 1..c2;
type buf-tbl is array(1..c2) of buf-index;
type buf-tbl-otr is access buf-tbl;
destination-tbl : array(1..c1) of buf-tbl-otr;
buffer-arrav : array(1..c2) of string30;
procedure forward(msg : in message-otr);

end DIG+COM;

pragma environment("ACS:TEXTIO.MLE","DCOM.MSE","INTIO.MSE");
with textio,intio; use textio,intio,ascii;

package body DIG+COM is
procedure forward(msg : in message-otr) is
  i,j : integer;
  buffer-index : buf-index;
  line : buf-tbl-otr;
  buff-arrav : buf-tbl;

98
begin
    line := destination tbl(msg, destination);
    buf+array := line.all;
    i:=1;
    while buf+array(i) /= msg,connection loop
        i:= i+1;
    end loop;
    buffer+index := buf+array(i);
    buffer+array(buffer+index) := msg.data;
end forward;
end DIG+COM;

--

-- DIG.COM: driver routine
--
digital communication benchmark
--
DCOM12,EOD on disk
timing includes procedure invocation overhead
--
26 Oct 1982
--
pragma environment("ACS:TEXTIO.MLE","INTIO.MSE","DCOM.MSE", "MAIN.MSE");
with text+io,intio,DIG+COM;
use text+io,intio,DIG+COM,ascii;
--
package body user+process+1 is
procedure main is
    i,j : integer;
    timer+loop : integer;
    k : buffer+index;
    buf+table+ptr := buf+table+ptr;
    msg+out := message+ptr;
    forever := boolean := true;
    answer := character;
--
    begin
        out+30("chars*. 4 qdp configuration...");
        --
        newline();
        out+30("timing includes proc ovhd.....");
        -- initialize the destination table
        newline();
        --

99
out*30("init destination table........");
for i in 1..c1 loop
  destination+tbl(i) := new buf+tbl;
end loop;
-- initialize all buf+tbl's
new+line();
out*30("init buf+tbl's...................");
for i in 1..c1 loop
  buf+table+ptr := destination+tbl(i);
  for j in 1..c2 loop
    buf+table+ptr(j) := j;
  end loop;
end loop;
-- initialize buffer
new+line();
out*20("init the buffer.......");
for k in 1..c2 loop
  buffer+array(k) := "..................";
end loop;
new+line();
while forever loop
  out*10("continue? ");
  get(answer);
  exit when answer = 'N';
  msg+out := new message;
  msg+out+size := 0;
  new+line();
  out*20("start digit comm....");
  new+line();
  out*30("enter destination, conn,data...");
  get(msg+out+destination);
  skip+line;
  get(msg+out+connection);
  skip+line;
  msg+out+data := get+line*30();
  new+line();
  out*30("number of loops to time........");
  get(timer+loop);
  out*10("sending...");
  out(be1);
  for i in 1..timer+loop loop
    forward(msg+out);
  end loop;
100
out(be1);  
out+20("...done sending.....");
newline();
out+20("buffer flush is.....");
for k in 1..c2 loop
  put(k);
  put+line+30(buffer+array(k));
  newline();
end loop;
skip+line;
end loop;
newline();
out+20("end of decom1........");
end main;
end user+process+1;
-- DCOM2 package specification

-- Digital Communication Processing Program

-- 19 Oct 82

pragma environment("ACS:TEXTIO.MLE");
with text'io; use text'io;

package DIG#COM is

  c1 : constant := 10;
  c2 : constant := 10;

  subtype dest$type is integer range 1..c1;
  subtype con4-id$type is integer range 1..c2;

  type message is record
    destination : dest$type;
    connection : con4-id$type;
    size : integer;
    data : string30;
  end record;

  subtype buf4-index is integer range i..c2;
  type buf4-tbl is array(1..c2) of buf4-index;
  type buf4-tbl4-otr is access buf4-tbl;
  type destination4-tbl is array(1..c1) of buf4-tbl4-otr;
  type buffer4-array is array(1..c2) of string30;

procedure forward(msq : IN message4-otr);

end DIG#COM;

pragma environment("ACS:TEXTIO.MLE","DCOM.MSE","INTIO.MSE");
with text4io, intio; use text4io, intio, ascii;

package body DIG#COM is

  procedure forward(msq : IN message4-otr) is
    i, j : integer;
    timer4-loop : integer;
    buffer4-index : buf4-index;
    line : buf4-tbl4-otr;
    buf4-array : buf4-tbl;

end DIG#COM;
begin

new+line();
put+30("number of loops to time.......");
get(timer+loop);
put+10("sending....");
out(bel);
for j in 1..timer+loop loop
   line := destination+tbl(msg.destination);
   buf+array := line.all;
j:=1;
   while buf+array(i) /= msg.connection loop
      i:= i+1;
   end loop;
   buffer+index := buf+array(i);
   buffer+array(buffer+index) := msg.data;
end loop;
out(bel);
out+20("...done sending.....");
new+line();
end forward;
end DIG+COM;

---

---

---

---

--

DCOM2 driver routine

--

digital communication benchmark

--

DCOM2.EOD on disk

timing does not include procedure invocation overhead

--

26 Oct 1982

pragma environment ("ACS:TEXTIO.MLE","INTIO.MSE","DCOM.MSE",
"MAIN.MSE");
with text+io,intio, DIG+COM;
use text+io,intio,DIG+COM,ascii;

--

package body user+process+1 is

procedure main is
   i,j : integer;
   k : buf+index;
   buf+table+otr : buf+tbl+otr;
   msg+out : message+otr;
   forever : boolean := true;
   answer : character;

103
begin
out$30("chars*. 4 gdo configuration...");
--
new$line();
out$30("timing does not include proc.");
--
initialize the destination table
new$line();
--
out$30("init destination table...........");
for i in 1..c1 loop
  destination$tbl(i) := new buf$tbl;
end loop;
--
initialize all buf$tbl's
--
new$line();
out$30("init buf$tbl's.................");
for i in 1..c1 loop
  buf$table$otr := destination$tbl(i);
  for j in 1..c2 loop
    buf$table$ptr(j) := j;
  end loop;
end loop;
--
initialize buffer
--
new$line();
out$20("init the buffer......");
for k in 1..c2 loop
  buffer$array(k) := ".........................";
end loop;
--
new$line();
while forever loop
  out$10("continue? ");
  get(answer);
  exit when answer = 'n';
  msg$out := new message;
  msg$out.size := 0;
  new$line();
  out$20("start digit comm....");
  new$line();
  out$30("enter destination, conn,data...");
  get(msg$out.destination);
  skip$line;
  get(msg$out.connection);
  skip$line;
  msg$out.data := get$line$30();
forward(msg+out);
gout=20("buffer flush is....");
for k in 1..c2 loop
    put(k);
    putline+30(buffer+array(k));
    newline();
end loop;
skio+line;
end loop;
new+line();
put+20("end of Jecom1.......");
end main;
end user+process+1;
--- VEM1 package specification

--- VEM1 recursive memory test package specification
pragma environment("ACS:TEXTIO,MLE");
with text+io; use text+io;
package EAT+MEMORY is
  size : constant := 50;
  i : integer :=0;
  type small+table is array(1..size) of character;
  type small+table+ptr is access small+table;

  procedure FOREVER;
end EAT+MEMORY;

---

--- VEM1 package body

--- VEM1 recursive memory test body
pragma environment("ACS:TEXTIO,MLE","EAT,MSE","INTIO,MSE");
with intio,text+io;
use intio,text+io;

package body EAT+MEMORY is
  procedure FOREVER is
    table+ptr : small+table+ptr;
    begin
      i := i+1;
      out(i);
      newline();
      table+ptr := new small+table;
      FOREVER;
    end FOREVER;
end EAT+MEMORY;

---

--- VEM1 driver routine

--- VEM1 recursive memory test driver routine
pragma environment("ACS:TEXTIO,MLE","EAT,MSE","MAIN,MSE");
with text+io,EAT+MEMORY;
use text+io,EAT+MEMORY;
package body user+process+1 is
procedure main is

--

begin

--

put+30(" start of eat memory..........");
FOREVER;
end main;
end user=process+1;
-- MEM2 package specification
-- MEM2 interactive memory test package specification

pragma environment("ACS:TEXTIO.MLE");
with text*io; use text*io;
package EAT*MEMORY is
  size : constant := 50;
  i : integer := 0;
  type small+table is array(1..size) of character;
  type small+table+otr is access small+table;

  procedure FOREVER;
end EAT*MEMORY;

-- MEM2 package body
-- MEM2 interactive memory test body

pragma environment("ACS:TEXTIO.MLE","EAT.MSE","INTIO.MSE");
with intio,text*io;
use intio,text*io;

package body EAT*MEMORY is
  procedure FOREVER is
    table*otr : small+table+otr;
    infinite : boolean := true;
    begin
      while infinite loop
        i := i + 1;
        put(i);
        new+line();
        table*otr := new small+table;
      end loop;
      end FOREVER;
end EAT*MEMORY;

-- MEM2 driver routine
-- MEM2 interactive memory test driver routine

pragma environment("ACS:TEXTIO.MLE","EAT.MSE","MAIN.MSE");
with text+io,EAT+MEMORY;
use textio, EAT+MEMORY;
package body user+process+1 is
  procedure main is
    begin
    put+30(" start of eat memory...........");
    FOREVER;
    end main;
end user+process+1;
pragma environment("ACS:TEXTIO.MLE","INTIO.MSE","MAIN.MSE");
with textio, intio;
use textio, intio, ascii;

package body USER.PROCESS+1 is
procedure MAIN is
  type strin is array(integer range 1..120) of character;
  numiterations : integer;
  position,ns,nk : integer;
  s,k : strin;

  function STRSCH(s,k : IN strin;
                   ns,nk : IN integer) return integer is
    i,j : integer;
    base,ksave,cont : integer;
    kend,ssave : integer;
    r : integer;

    begin
      base := 1;
      ksave := 1;
      cont := ns-nk+base;
      kend := ksave + nk-1;
      i := 1;
      j := 1;
      <<top>>
      while s(i) /= k(j) loop
        if i >= cont then
          r := -1;
          goto finish;
        end if;
        i := i+1;
      end loop;
      ssave := i;
      j := j+1;
      while j <= kend loop
        i := i+1;
        if s(i) /= k(j) then
          i := ssave + 1;
          j := ksave;
          goto too;
        end if;
      end while;
    end function;
end USER.PROCESS+1;
j := j + 1;
end loop;
r := ssave - base + 1;
<<finish>>
return (r);
end STRSCH;

Begin
s(1..60) := "0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
pragma environment("ACS:TEXTIO.MLE","INTIO.MSE","MAIN.MSE");
with text+io, intio,
use text+io, intio, ascii;
package body USER+PROCESS+1 is
  procedure MAIN is
    size: constant integer := 200;
    flags: array(O..size) of boolean;
    prime, k, count, loop+val: integer;
    loop
      out+30("# of loops to time?..0 exits ");
      get(loop+val);
      exit when loop+val = 0;
      new+line();
      put(bel);
      for iter in integer range 1..(loop+val) loop
        count := 0;
        for i in O..size loop
          flags(i) := true;
        end loop;
        for i in O..size loop
          if flags(i) then
            prime := i + i + 3;
            k := i + prime;
            while k <= size loop
              flags(k) := false;
              k := k + prime;
            end loop;
            count := count + 1;
          end if;
        end loop;
      end loop;
      put(bel);
      out+line+10(" End Sieve");
      out(count);
      out+10(" Primes ");
      new+line();
    end loop;
  end MAIN;
end USER+PROCESS+1;
ACKER

-- Courtesy Prof. Patterson, Computer Science Division,
-- Department of Electrical Engineering & Computer Sciences
-- Univ. of California, Berkeley, CA.

pragma environment("ACS:TEXTIO,MLE","INTIO,MSE","MAIN,MSE");
with textio, intio;
use textio, intio, ascii;

package body USER+PROCESS+1 is
  procedure MAIN is
    a, i, arg1, arg2 : integer;
    function ACKER(x, y : IN integer) return integer is
      begin
        if x = 0 then
          return (y + 1);
        elseif y = 0 then
          return ACKER(x - 1, 1);
        else
          return ACKER(x - 1, ACKER(x, y - 1));
        end if;
      end;
    begin
      put_line(20("Ackermann Benchmark ");
      put_line(20("To Exit, Enter 0 ");
      put_line(30("Begin time when bell sounds ");
      loop
        put_line(30("Enter ACKER Arguments ");
        get(arg1);
        exit when arg1 = 0;
        skip_line;
        get(arg2);
        out(bel); a := ACKER(arg1, arg2);
        out(bel); put(10("Output of ");
        put(arg1);
        put(,, );
        put(arg2);
        new_line();
        put(a);
        new_line();
      end loop;
      end MAIN;
    end USER+PROCESS+1;
APPENDIX D
CFA BENCHMARK ALGORITHMS

The twelve benchmark program algorithm descriptions used in the first CFA study follow. A more detailed discussion of these can be found in Reference 8.

1. I/O INTERRUPT KERNEL, FOUR PRIORITY LEVELS

The interrupt kernel will be activated by an I/O interrupt with priority level 0, 1, 2 or 3 from one of four devices. Actual interrupt processing will be simulated by counting the occurrences of each type of interrupt. Higher level interrupts will be able to preempt processing of lower priority interrupts. The interrupt handler must provide for resumption of processing of the preempted lower level interrupt from the point of preemption. As much processing as possible will be done with higher priority I/O interrupts enabled.

2. I/O INTERRUPT KERNEL, FIFO PROCESSING

The interrupt kernel will be activated by an I/O interrupt from one of four devices which will be placed in a service queue for first-in-first-out (FIFO) processing. Actual interrupt processing will be simulated by counting the occurrences of each type of interrupt. Space should be
provided to handle at least ten queued interrupts at one time.

3. INPUT/OUTPUT DEVICE HANDLER

After an I/O request is issued by an application program, and after the executive queues an input control block, this test program is initiated and it performs the following actions:

1. Check status of the tape drive. If device is busy exit. If the device is not operable branch to an error routine. If the device is available, set up and initiate the requested transfer.

2. After completion of the transfer, and a consequent interrupt, the device handler is reentered and the following processing is performed:
   a. Store status information (device type and identification).
   b. If transfer was unsuccessful, abort further processing.
   c. If a successful transfer occurred and all requested transfers accomplished then exit.

The application programs perform high level logical I/O calls that cause the queuing.

4. FAST FOURIER TRANSFORM

The following variables are used in the algorithm:

\( N \): The number of data points \( 0 \leq N \leq 2^{16} \).
\( X \): A vector holding the \( N \) samples as complex numbers.
\( W \): A vector holding the first \( N/2 \) powers of \( \exp(-2\pi i \times N) \).
\( \text{work} \): Auxiliary working storage.
procedure FFT(N,X,W)
GROUPS := N
for PASS := 0 by steps of 1 until log2(n) = 1
  do for all ELEMENT such that
     0 <= element <= N/2
     "generate complex addend"
     WEXP := 0
     if PASS > 0
       then WEXP := ((ELEMENT*N)/2) / 2**PASS MOD (N/2)
     end-if
     if WEXP <> 0
       then TEMP1 := X(ELEMENT+N/2)*WEX
         else TEMP1 := X(ELEMENT+N/2)
     end if
     "generate 2 element entries in data vector"
     X1(ELEMENT) := X(ELEMENT) + TEMP1
     X1(ELEMENT + N/2) := X(ELEMENT) - TEMP1
   end-do
  if PASS < (log2(N) - 1)
    then
      "execute perfect card shuffle on data vector"
      P := 2**PASS
      GROUPS := GROUPS/2
      do for all I such that
           0 <= I < GROUPS
         do for all J such that
            0 <= J < P
              INDEX1 := 2*P*I + J
              INDEX2 := P*I + J
              X(INDEX1) := X1(INDEX2)
              X(INDEX1+P) := X1(INDEX2+N/2)
            end-do
        end-do
  else
    do for all I such that 0 <= I < N
      X := X1(I)
    end-do
  end-if
end-do
5. CHARACTER SEARCH

The variables used in this algorithm are:

- SRCHSTR : pointer to a string of characters to be searched.
- SRCHLENGTH : length of that string.
- SRCHARG : pointer to a string of characters.
- ARGLENGTH : length of that string.
- LOC : an integer return code.
- WORK : pointer to any needed storage.

procedure CHARSEARCH(SRCHSTR,SRCHLENGTH, SRCHARG,ARGLENGTH,LOC,WORK)

integer I

LOC := 1

do for all I such that 0 <= I <= SRCHLENGTH-SRCHARG
or until LOC <> -1

if the substring of SRCHSTR from I to
I+ARGLENGTH-1 = SRCHARG

then LOC := I

end-if
end-do

6. BIT TEST, SET, OR RESET

The variables used are:

- F : Function code, 1=test, 2=set, 3=reset.
- N : Relative bit to be tested.
- A1 : Pointer to tightly packed bit string.
- RC : Return code indicating original bit status.
- WORK : Pointer to any needed work storage.

procedure BITTEST(F,N,A1,RC,WORK)

integer ABIT,D

ABIT := A1 + N/(word length)
D := N mod (word length)

if D’th bit at address ABIT = 1
then RC := 1
else RC := 0
end-if
if \( F = 2 \)
then 3’th bit at address ABIT := 1
else if \( F = 3 \)
then 3’th bit at address ABIT := 0
end-if

7. RUNGE-KUTTA INTEGRATION

This algorithm solves the differential equation \( F(t, y) = t + y = \frac{dy}{dt} \) using a third order Runge-Kutta integration.

The variables used are:

\[
\begin{align*}
T0 & : Initial value of T, single precision. \\
YO & : Initial value of Y, single precision. \\
H & : Interval of integration, single precision. \\
TMAX & : Final value of T, single precision. \\
YMAX & : Final value of Y returned, single precision. \\
\end{align*}
\]

procedure RUNGEKUTTA(T0,Y0,TMAX,YMAX,WORK)
real K1,K2,K3

YMAX := Y0

for all T from T0 incremented in steps of H until T > TMAX

\[
\begin{align*}
K1 & := H \times (T + YMAX) \\
K2 & := H \times (T + H/2 + Y + K1/2) \\
K3 & := H \times (T + 3H/4 + Y + 3K2/4) \\
YMAX & := YMAX + 2*K1/9 + K2/3 + 4*K3/9 \\
\end{align*}
\]
end-do

8. LINKED LIST INSERTION

This algorithm inserts an element into a doubly linked list. Variables used are:

\[
\begin{align*}
\text{LISTCB} & : Pointer to a list control block containing: \\
\text{HEAD} & : pointer to first node. \\
\text{TAIL} & : pointer to last node. \\
\text{NUMENTRIES} & : number of entries. \\
\text{NEWENTRY} & : pointer to new entry to be inserted. \\
\end{align*}
\]
procedure LISTINSERT(LISTCB, NEWENTRY)

"the notation POINTER.FIELD is used to access a particular field of the structure pointed to by POINTER"

pointer PRESENT
if LISTCB.NUMENTRIES = 0 then "list is empty, so initialize"
    LISTCB.HEAD := LISTCB.TAIL := NEWENTRY
    LISTCB.NUMENTRIES := 1
    NEWENTRY.NEXT := NEWENTRY.PREV := 0
else "list not empty"
    PRESENT := LISTCB.HEAD
    LISTCB.NUMENTRIES := LISTCB.NUMENTRIES + 1
    "determine position of new entry"
    while NEW.KEY >= PRESENT.NEXT <> 0 do
        PRESENT := PRESENT.NEXT
    if PRESENT.PREV = 0 and NEW.KEY < PRESENT.KEY then
        "new list head"
        LISTCB.HEAD := NEW
        NEW.PREV := 0
        PRESENT.PREV := NEW
        NEW.NEXT := PRESENT
    else
        if NEW.KEY => PRESENT.KEY then
            "new list tail"
            PRESENT.NEXT := LISTCB.TAIL := NEW
            NEW.NEXT := 0
            NEW.PREV := PRESENT
        else
            "insert in middle"
            NEW.NEXT := PRESENT
            NEW.PREV := PRESENT.PREV
            PRESENT.PREV := NEW
    "back up and link with predecessor"
    PRESENT := NEW.PREV
    PRESENT.NEXT := NEW
end-if
end-if

119
9. QUICKSORT

This algorithm performs a quicksort on an array of records. The variable used are:

- \( N \): The number of records to be sorted.
- \( M \): Integer parameter specifying the changeover point between quicksort and a simple insertion.
- \( REC \): Pointer or the first element of the array to be sorted.
- \( WORK \): Pointer to any needed working storage.

```plaintext
procedure QUICKSORT(N, REC, M, WORK)
    integer L, R, I, J, K
    integer array STACK[0:2*f(N)-1]
    character string V

    REC[N+1] := infinite
    L := 1; R := N
    do forever
        I := L; J := R+1; V := REC[L]
        do forever
            do I := I+1 until REC[I] => V end-do
            do J := J-1 until REC[J] <= V end-do
            if J > I then
                swap REC[I] with REC[J]
            else goto end-first
            end-if
        end-do
        end-first:
        swap REC[L] with REC[J]
        if both subfile sizes (J-L and R-J) <= M then
            if stack empty then goto end-outer
            else pop L and R from stack end-if
        else
            if smaller subfile size <= M then
                set L and R to lower and upper limits of larger subfile
            else push lower and upper limits of larger subfile onto stack set L and R to limits of smaller subfile end-if
        end-if
    end-do
end-outer:
```
do for I from N=1 to 1 in steps of 1
    if REC[I] > REC[I+1] then
        V := REC[I]; J := I+1
        do forever
            REC[J-1] := REC[J]; J := J+1
            if REC[J] => V then goto end-last end-if
        end-do
    end-last: A[J-1] := V
end-if
end-do

10. ASCII TO FLOATING POINT CONVERSION

The following variables are used in this algorithm:

N : Number of characters in the string.
A1 : Address of the character string.
A2 : Address of floating point number where the result will be placed.

procedure AFP(N,A1,A2)
    integer NUMBER, POSITION
    real RESULT, DIVISOR
    boolean ISNEGATIVE

    ISNEGATIVE := false
    POSITION := 0
    if first character of A1 is a sign character
        then
            if sign character is "-" then
                ISNEGATIVE := true
                POSITION := 1
            end-if
        end-if
    NUMBER := integer equivalent of characters POSITION to J-1 of A1 where
              character J of A1 is "."
    RESULT := floating point equivalent of NUMBER
              "the following two steps can be done in parallel if desired"
    NUMBER := integer equivalent of characters J+1 to N of A1
    DIVISOR := floating equivalent of 10**(N-J)
    A2 := RESULT + (floating point equivalent of NUMBER) / DIVISOR
11. BOOLEAN MATRIX TRANSPOSE

The following variables were used in this algorithm:

- $A_1$: Pointer to a word of storage.
- $A_2$: bit number within word $A_1$ where the matrix begins.
- $N$: Size of the boolean matrix.

procedure BMT($N, A_1, A_2$)
integer I, J
boolean $B[i:N, i:N]$ beginning at bit $A_2$ of word $A_1$
do for all $I$ and $J$ such that $(i \leq J \leq N)$ and $(J+1 \leq I \leq N)$
    swap $B[I, J]$ and $B[J, I]$end-do

12. VIRTUAL MEMORY SPACE EXCHANGE

This algorithm performed a virtual memory space exchange through the use of a supervisor call. There are two functions which must be provided by the algorithm.

1. CALL: saves enough information to restore the entire state of the caller.

2. RETURN: restores the environment active before the previous call.

The sixteen benchmark programs written by the second CPA study group follow. A complete discussion of them can be found in reference 1.

1. TERMINAL INPUT DRIVER

This algorithm inputs one line of ASCII characters from a terminal device. ASCII rubouts should delete the character. A carriage return terminates the line. The program need not be reentrant.
Algorithm: A subroutine TTYIN(BUFFER) initiates the transfer. It has a single reference parameter, the buffer to be filled. The buffer consists of:

ADDRESS TERMADDR

CHARACTER CBUF[1:7]

The buffer is assumed to be large enough for the line. The transfer is started and the routine returns. The interrupt service routine collects the line in some machine dependent manner. The terminal interface is assumed to be a minimal one, (it does the serial-parallel conversion). When a carriage return is entered, the terminal input is disconnected and a transfer to the buffer TERMADDR is made.

2. MESSAGE BUFFERING AND TRANSMISSION

This algorithm queues a message buffer and then transmits the message over a DMA link in FIFO order.

RECORD BUFR(ADDRESS NEXT, ADDRESS TERMADDR, INTEGER SIZE, INTEGER DATA[1:SIZE]);
POINTER BUFR END, START
ADDRESS TEMP;
!QUEUE SUBROUTINE
PROCEDURE QUEUE(REFERENCE BUFFER) =
BEGIN
IF START NEQ 0 THEN END, NEXT <- ADDRESS(BUFFER) FI;
END <- ADDRESS(BUFFER);
!QUIT IF CHANNEL ALREADY RUNNING
IF START NEQ 0 THEN RETURN ELSE
START <- ADDRESS(BUFFER);
TEMP <- 0;
GOTO RESTART
FI;
END;

INTERRUPT:
BEGIN
123
Programmers should insert here device and machine dependent code to terminate the device transfer.

\[
\text{TEMP} \leftarrow \text{START.TERMADDR}; \\
\text{START} \leftarrow \text{START.NEXT}; \\
\]

**RESTART:**

IF \( \text{START} = 0 \), THEN

GO TO TEMP

ELSE

Programmers should insert here device and machine dependent code to initiate the device transfer.

FI:

IF \( \text{TEMP} = 0 \), THEN RETURN

ELSE GO TO TEMP

FI

END

3. **MULTIPLE PRIORITY INTERRUPT HANDLER**

This test program is designed to process interrupts from four devices in priority order. Upon receiving an interrupt, the processor will branch to the appropriate device service routine. All interrupts from lower priority devices will be disabled. Device priority is equal to device number, device number 1 has lowest priority, device 4 has highest. After the device dependent service the device ID is added to the executive queue for user scheduling purposes. This program need not be reentrant. Each device service routine will be simulated by the algorithm below.

```
!DEVICE SERVICE ROUTINE INTEGER OWN A; FOR I <= 1 TO A[012] DO A <= (A*899) MOD 123757 OD;
```

124
4. VIRTUAL MEMORY SPACE EXCHANGE

This algorithm will involve a supervisory call handler which will provide the functions "call" and "return". The supervisor is to implement protected procedure calls with parameters. "call" will select index into a table of address space descriptors maintained by the supervisor. The "call" performs the following:

1. Save the caller's state.
2. Determine the callee's address space.
3. Set up the memory mapping and protection to access the callee's address space.

The "return" function takes no parameters. It restores the environment active before the previous call.

5. SCALE VECTOR DISPLAY

This algorithm scales a list of graphic vectors about a given center. The vectors are represented as:

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>function</td>
<td>4</td>
</tr>
<tr>
<td>x coordinate</td>
<td>12</td>
</tr>
<tr>
<td>intensity</td>
<td>4</td>
</tr>
<tr>
<td>y coordinate</td>
<td>12</td>
</tr>
</tbody>
</table>

PROCEDURE SCALEADJUST(REF DLIST, VALUE LEN, VALUE XCENTR, VALUE YCENTR, VALUE SCALE)=

BEGIN
10 LEQ XCENTR, YCENTR LEQ 2047
1SCALE IS THE ACTUAL SCALE FACTOR TIMES 128
INTEGER LEN,XCENTR,YCENTR,SCALE,I,XTMP,YTMP;
RECORD VECTOR(INT4 FUNCT,INT 12 X, INT4 INTEN, INT 12 Y);
VECTOR DLIST[1:LEN];
FOR I <- 1 TO LEN DO

125
XTMP ← DLIST.X[I]*SCALE;
YTMP ← DLIST.Y[I]*SCALE;
IF DLIST.FUNCT[I] NEQ 0 THEN
  XTMP ← XTMP+XCENTR*(128-SCALE);
  YTEMP ← YTEMP+YCENTR*(128-SCALE);
FI;
DLIST.X[I] ← XTMP/128;
DLIST.Y[I] ← YTMP/128;
OD;
RETURN
END

6. ARRAY MANIPULATION - LU DECOMPOSITION

This algorithm factors a square matrix into an upper and lower triangular matrix.

LUDECOMP(REFERENCE A, VALUE N)=
BEGIN
REAL ARRAY A[1:N,1:N];
REAL MULT;
INTEGER DIAG, ROW, COL;
FOR DIAG =1, N-1 DO
  FOR ROW = DIAG+1, N DO
  FOR COL = DIAG+1, N DO
  OD
OD
END

7. TARGET TRACKING

This algorithm takes the coordinates of an unknown object and finds in a table sorted by x coordinate the closest entry.

PROCEDURE TARGET(REFERENCE TABLE, VALUE LEN, VALUE X
VALUE Y, REFERENCE FOUND)=
BEGIN
INTEGER LEN,START,END,MID,UP,DOWN;

126
REAL MINDIST;
ADDRESS FOUND
RECORD TENTRY(REAL X, REAL Y, REAL DAT1, REAL DAT2);
TENTRY TABLE[1:LEN]
START <- 1; END <- LEN;
WHILE START <- END DO
  MID <- (START + END) / 2
  IF TABLE.X[MID] < X THEN
    START <- MID + 1
  ELSE
    END <- MID
  END;
!Compute distance of nearest x entry
MINDIST <- DIST(TABLE[MID], X, Y);
FOUND <- ADDRESS(TABLE[MID]);
!Search neighborhood for a nearer entry
UP <- MID + 1; DOWN <- MID - 1;
WHILE UP > 0 OR DOWN > 0 DO
  IF UP > 0 THEN CHECK(UP); UP <- UP + 1 FI;
  IF DOWN > 0 THEN CHECK(DOWN); DOWN <- DOWN - 1 FI;
END;
RETURN;
!Check an individual entry against closest found
PROCEDURE MACRO CHECK(J) =
BEGIN
  IF J < 1 OR J > LEN OR ABS(TABLE.X[J] - X) >= MINDIST THEN J <- 0; RETURN FI;
  IF DIST(TABLE[J], X, Y) < MINDIST THEN
    MINDIST <- DIST(TABLE[J], X, Y);
    FOUND <- ADDRESS(TABLE[J])
  FI;
RETURN
END
!DIST() is the metric defined in the problem
END

8. DIGITAL COMMUNICATIONS PROCESSING

This algorithm is given a message with a header which contains the destination and connection ID, and places the message in the appropriate transmission line's output buffer.
PROCEDURE FORWARD(REFERENCE MSG) =
BEGIN
RECORD MESSAGE(INT16 CID,INT16 DEST, INT16 SIZE
   CHARACTER MSG[1:7]);
BUFTABLE(INTEGER CID,ADDRESS BUFFER);
MESSAGE MSG;
POINTER BUFTABLE LINE;
EXTERNAL ADDRESS DESTABLE[1:7];
!Find BUFFER table for destination line
LINE <= DESTABLE[MSG,DEST];
!Find ring buffer for this connection
I <= 1;
WHILE LINE.CID[I] NEQ MSG.CID
   DO I <= I + 1 OD;
BUFFER <= LINE,BUFFER[I];
!Copy the message to the buffer
MOVE(ADDRESS(MSG),BUFFER,MSG,SIZE);
RETURN
END

9. HASH TABLE SEARCH

This program locates the position a key would occupy in
a hash table.

PROCEDURE HASHLOOK(REFERENCE TABLE, VALUE SIZE,
   VALUE KEY, REFERENCE POSITION,
   REFERENCE FULL) =
BEGIN
ADDRESS POSITION
INTEGER SIZE,KEY,CHECK;
BOOLEAN FULL;
RECORD TENTRY(INTEGER KEY, INTEGER DATA);
TENTRY TABLE[0:SIZE-1];
!Compute first place to look
CHECK <= KEY MOD SIZE;
FULL <= FALSE;
FOR I <= 1 TO SIZE/2 DO
   IF TABLE.KEY[CHECK] = KEY OR TABLE.KEY[CHECK] = 0
      THEN
      POSITION <= ADDRESS(TABLE.KEY[CHECK]);
      RETURN
   FI;
CHECK <= (CHECK + I) MOD SIZE
OD
FULL <= TRUE
RETURN
END

128
10. LINKED LIST INSERTION

This algorithm inserts a node in an ordered doubly linked list.

PROCEDURE LISTINSERT(VALUE LISTCB, VALUE NEWENTRY) = BEGIN
  RECORD LCB(ADDRESS HEAD, ADDRESS TAIL, INTEGER NUMENTRIES);
  RECORD LISTENTRY(INT32 KEY, ADDRESS NEXT, ADDRESS PREV)
  POINTER LCB LISTCB;
  POINTER LISTENTRY, NEWENTRY, PRESENT;
  IF LISTCB, NUMENTRIES = 0 THEN
    LISTCB, HEAD <- LISTCB, TAIL <- NEWENTRY;
    LISTCB, NUMENTRIES <- 1;
    NEWENTRY, NEXT <- NEWENTRY, PREV <- 0
  ELSE
    PRESENT <- LISTCB, HEAD;
    LISTCB, NUMENTRIES <- LISTCB, NUMENTRIES + 1;
    WHILE NEWENTRY, KEY GEO PRESENT, KEY AND
    PRESENT, NEXT NEQ 0 DO PRESENT <- PRESENT, NEXT OD;
  IF PRESENT, PREV = 0 AND NEWENTRY, KEY < PRESENT, KEY THEN
    LISTCB, HEAD <- NEWENTRY;
    NEWENTRY, PREV <- 0;
    PRESENT, PREV <- NEWENTRY
    NEWENTRY, NEXT <- PRESENT;
  ELSE
    IF NEWENTRY, KEY GEO PRESENT, KEY THEN
      PRESENT, NEXT <- LISTCB, TAIL <- NEWENTRY;
      NEWENTRY, NEXT <- 0;
      NEWENTRY, PREV <- PRESENT
    ELSE
      NEWENTRY, NEXT <- PRESENT;
      NEWENTRY, PREV <- PRESENT, PREV;
      PRESENT, PREV <- NEWENTRY;
      PRESENT, NEXT <- NEWENTRY
    FI
  FI
RETURN END
11. PRESORT ON A LARGE ADDRESS SPACE

This algorithm takes an array of records in random order and rearranges them to form a heap. The heap is a binary tree in which each node is greater than or equal to its descendents.

HEAPIFY(REFERENCE REC, VALUE N) =
BEGIN
INTEGER ARRAY REC[1:N];
INTEGER CHECK, NEW;
FOR NEW <= 2, N DO
  CHECK <= NEW;
  WHILE CHECK NEQ 1 AND REC[CHECK] > REC[CHECK/2] DO
    REC[CHECK] <=> REC[CHECK/2];
    CHECK <= CHECK/2
  OD
END

12. AUTOCORRELATE ON A LARGE ADDRESS SPACE

This algorithm computes the autocorrelation of the vector A from 1 to T.

PROCEDURE AUTO(REFERENCE A, VALUE N, VALUE T, REFERENCE RES) =
BEGIN
INTEGER N, T, TAU;
REAL A[1:N], RES[1:T];
FOR I <= 1 TO T DO RES[I] <= 0 OD;
FOR I <= 1 TO N DO
  FOR TAU <= 1 TO T DO
    IF I + TAU-1 > N THEN EXITLOOP FI;
  OD
OD
RETURN
END
13. CHARACTER SEARCH

This algorithm searches a given string to see if it contains a substring that exactly matches the given argument string.

```plaintext
PROCEDURE CHARSRC(REF SRCHSTR, VALUE SRCHLENGTH, 
                  REF SRCHARG, VALUE ARGLNGTH, REF LOC) =
BEGIN
  INTEGER I, SRCHLENGTH, ARGLNGTH;
  BYTEVECTOR SRCHSTR[0:SRCHLENGTH-1], SRCHARG[0:ARGLNGTH-1]
  LOC <- -1;
  IF ARGLNGTH LE 0 THEN LOC <- 0; RETURN FI;
  FOR I IN 0, SRCHLENGTH-ARGLNGTH DO
    IF SRCHSTR[I;I+ARGLNGTH] LE SRCHARG
      THEN LOC <- I; RETURN FI;
  OD;
RETURN
END
```

14. BOOLEAN MATRIX TRANSPOSE

This algorithm computes the transpose of a given N by N matrix in place.

```plaintext
PROCEDURE BMT(VAL N, VAL A1, VAL A2) =
BEGIN
  INTEGER I, J;
  BOOLEAN B[1:N, 1:N]
  FOR I IN 1, N-1 ; J IN I+1, N DO
    B[I, J] <=> B[J, I]
  OD;
RETURN
END
```
15. RECORD UNPACKING

This algorithm unpacks the fields of a record into an integer array.

```
PROCEDURE UNPACK(REF RECORD, REF FORMAT, VALUE LEN
REF RESULT)=
BEGIN
BITSTRING RECORD[i:7];
INTEGER LEN,START,RESULT[i:LEN],TEMP,I;
ARBITRARY FORMAT[i:LEN];
START <= 0
FOR I <= 1 TO LEN DO
  TEMP <= RECORD[START:START+FORMAT[I]-1];
  START = START + FORMAT[I];
  IF FORMAT[I] IS A DISTINGUISHED VALUE
  THEN
    TEMP <= SIGNEXTEND(TEMP)
  FI;
  RESULT[I] <= TEMP;
Od;
RETURN
END
```

16. VECTOR TO SCAN LINE CONVERSION

This algorithm takes a list of vectors and produces a raster scan line conversion.

```
PROCEDURE VECSCAN(REF DLIST,VALUE LEN, REF TEMP)=
BEGIN
RECORD DISPLAY(INT16 XS, INT16 YS, INT16 XE, INT16 YE),
WORKLIST(INT16 XS,INT16 XE,INT32 Y,INT32 SLOPE);
DISPLAY DLIST[i:LEN]
WORKLIST TEMP[i:LEN+1];
INTEGER I, START, LINE, DENOM;
BITSTRING BIT(1:1024);
!Generate working vector
FOR I <= 1 TO LEN DO
  TEMP,XS[I] <= DLIST,XS[I];
  TEMP,XE <= DLIST,XE[I];
  TEMP,Y[I] <= DLIST,YS[I]*1024;
  DENOM <= (DLIST,XE[I] - DLIST,XS[I] + 1);
  TEMP,SLOPE[I] <= (DLIST,YE(I) - DLIST,YS[I])*1024/DENOM
END
```
OD;
TEMP.XS[LEN+1] <= 1025
! Generate the scan image
START <= 1;
FOR LINE <= 1 TO 1024 DO
  BIT <= 0
  I <= START;
  WHILE TEMP.XS[I] LEQ LINE DO
    FOR K <= TEMP.Y[I]/1024 TO (TEMP.Y[I] +
       TEMP.SLOPE[I])/1024
      DO BIT[K] <= 1 OD;
    TEMP.Y[I] <= TEMP.Y[I] + TEMP.SLOPE[I];
    IF TEMP.XE[I] = LINE
       THEN TEMP[START] <=> TEMP[I];
           START <= START + 1;
    FI;
    I <= I + 1;
  OD;
OD
RETURN
END
APPENDIX E
CDS 432/670 USERS MANUAL

The following is an effort to enable someone with no prior knowledge of the 432/600 system to be able to compile, link, and execute programs on the 432 in a minimum amount of time and 'fuss'. A knowledge of ADA is assumed, as is familiarity with VMS (e.g. the VMS editor).

Referring back to Figure[6] it can be seen that a variety of hardware and software is involved in simply getting a program to "run" on the 432. This variety of needed hardware/software is collectively referred to as the 432 "Cross Development System", or "CDS" for short. Not surprisingly, those functions needed first in order to achieve the desired result of a program executing on the 432 are accomplished on the VAX 11/780 host. Briefly, the steps required, plus their CDS 'companion elements' are:

1. Program Creation/Editing — VAX/VMS
2. Compilation — VAX/VMS
3. Linking — VAX/VMS
4. Downloading — MDS 800
5. Program Load/Execution — MDS 800/432

134
1. PROGRAM CREATION/EDITING

Creation of a login file with at least the following commands will substantially add to the ease of your terminal sessions while working with those CDS parts which reside on the VAX/VMS host:

`sADA432`

`smpo ::= del *,mso;**,mbo;`

`smpnc ::= del *,msc;**,mbc;`

`smpoe ::= del *,mse;**,mbe;`

The reason for these commands will become evident as we continue.

ADA source files to be compiled by the Intel ADA cross compiler must have a file extension type of either:

1. `<filename>,SS` => An ADA source specification file.
2. `<filename>,MS` => An ADA source body file.
3. `<filename>,MCS` => Both specification and body.

In our opinion, dividing source code into separate specification (,SS) and body (,MS) files was in keeping with some of the original philosophies behind ADA, i.e., encapsulation and information hiding. Unfortunately, the compilation efforts, of necessity, must double (2 files to compile vs. 1 in the MCS case). What follows next are figures of a sample program. Figure 19 illustrates the
division into specification and body. Figure 20 illustrates
the combined (MCS) format. Besides the distinction of
working with two separate files as opposed to one, take
special note of the line, common to the "body", which begins
with "pragma environment...".

```
package EXAMPLE1 is
  procedure SIMPLE;
end EXAMPLE1;  

The specification filed as EXAMPLE1.MSS

The body filed as EXAMPLE1.MBS

pragma environment("ACS;TEXTIO,MLE","EXAMPLE1.MSE","INTIO.MSE");
with text_io,intio;
use text_io,intio,ascii;
package body EXAMPLE1 is
  procedure SIMPLE is
    --
    x,y,z : integer;
    --
    begin
      x := 10;
      y := 15;
      put_line_10(" SIMPLE ");
      put(bel);
      -- this rings the bell, use ascii enables this
      z := x+y;
      put(z);  -- "intio" allows you to do this
      put(bel);
      put_line_10("END SIMPLE");
    end SIMPLE;
end EXAMPLE1;
```

Figure 19. Specification and Body Format (Separate)
Information is conveyed to the ADA compiler system by means of pragmas. The environment pragma specifies the names of external environment files (or library units) that constitute the compilation environment for the current compilation unit(s). If the current compilation depends on other compilation units from other compilations, then the environment files from these compilations must be listed in the ENVIRONMENT pragma in the current compilation. These
environment pragmas enable separate compilation while still maintaining strong type checking of interfaces, two features which ADA is supposed to fulfill. In these examples the compilation of the body depends on:

-- ACS:TEXTIO.MLE => so the package can perform character I/O.
-- INTIO.MSE => so the package can perform integer I/O.
-- EXAMPLE1.MSE => the corresponding specification file.

To alleviate confusion on file extensions, the following is a list of VMS file extensions used in the 432 ADA Compiler System (ACS).

1. First character:
   M -- The file contains a library unit. M stands for module.
   S -- The file contains a SEPARATE stub.

2. Second Character:
   S -- The file contains a program unit specification.
   B -- The file contains a program unit body.
   C -- The file contains the combination of a program unit specification and a program unit body.
   L -- The file is a program library file supplied by Intel.

3. Third (last) Character:
   S -- The file is an ADA source text file.
E -- The file is an environment file.
R -- The file is a REPORT file.
O -- The file is an object code (EOD) file.
L -- The file is a REPORT listing file.
C -- The file is an object code listing file.
M -- The file is a specification file for the COMBINE utility and contains a list of environment files that are to be merged.
I -- The file is an integrated environment file created by the COMBINE utility.
T -- The file is a listing file produced by COMBINE and contains the file table listing of the integrated environment.

For added clarification:

e.g. <filename>.MSS -- An ADA source text file which corresponds to a specification.
e.g. <filename>.MBS -- An ADA source file containing program unit bodies,
e.g. TEXTIO.MLE -- A library environment file supplied by Intel.

2. COMPILATION

The Intel compiler is invoked by the command "IDA", followed by the filename. If the filename is omitted, the compiler will prompt for it. Our input to the compiler consisted either of <filename>.MSS>, for specification files, or <filename>.MBS>, for the implementation, i.e., body, files. Output from a successful compilation consists of files of type:
1. .MBE or .MSE -- The environment file representation.

2. .MBC or .MSC -- The object code listing file. It is utilized when debugging on the 432.

3. .MBO or .MSO -- The object code. This is input to the linking process.

Unsuccessful compilation results in files of the type:

1. .MBL or .MSL -- A report listing file. We generally never used this.

2. .MBR or .MSR -- A file which when prefixed by the command "REPORT", e.g., REPORT prog.mbr, allows one to scan through one's program on the terminal. More importantly, all errors detected by the compiler are flagged with their corresponding diagnostic message.

A typical session on VAX/VMS consists of the following:

1. Code and compile the specification file for the problem, i.e., the program, at hand. Since a specification file is essentially just a means of formalizing in ADA what one considers the interface to be, it usually needs no environment pragma statement.

2. Code and compile the body, which is the means by which one implements the program. Since the body depends on what the interface is, the environment file representation of the corresponding specification file must be included. Additionally, if I/O is to be performed in the body, which is generally the case, the general I/O, Intel-supplied package (TEXTIO), must also be included in the pragma environment statement.

An example of all this can be found in Appendix C, which shows the ADA source code for the programs done in this thesis.
In case it wasn't made clear in the above discussions, compilation order is important. Any modules included in the pragma environment statement or referenced in the standard ADA constructs, "WITH..." and "USE..." must be successfully compiled beforehand, otherwise unsuccessful compilation is all the reward one will get for one's efforts in the current compilation attempt.

Successful compilation means the creation of three new files in addition to the original source file. Directory space in VMS is quickly exhausted if one is performing many compilations. Without adequate directory space, the INTEL compiler and linker will abort. Therefore, when asking for an account, the system managers must be informed that more directory space than is normally given a VMS user is needed. Furthermore, in an attempt to provide a quick means of deleting unneeded environment, object-code listing, and object-code files, the commands, mope, mopc, and mopo will automatically delete all files of the corresponding filetype in the current directory.

Once one has successfully defined one's interface, coded it, compiled it, and has done the same with the corresponding body or bodies, one has reached the point where in most traditional systems one is ready to link the object code in preparation for actual program execution. In the 432 case, additional compilation must still be performed before the linking process may begin.
First, a module termed PSERP,MBS must be compiled. An example of this is included in Appendix C. Its function is to initialize the user process(es). It essentially marks which module is to begin execution first. For instance, a Driver routine which invokes all other subroutines is usually executed first. In our case, PSERP always initialized the Driver routine, which we always termed MAIN, in an attempt to cut down on our coding/compilation efforts.

Secondly, as pointed out in the architecture overview on operating system support, users can tailor some of the iMAX O.S. packages. In this thesis, modification of the system configuration package, PSORS,MBS, was implemented. Hence, the successful compilation of this modified package was also needed. This package is also included in Appendix C.

3. LINKING

The .MSO or .MBO files produced by a successful compilation are input to the 432 linker by being listed in a user created directives file. The output from a successful link is of filetype .EOD. EOD stands for "External Object Description". Actually, the respective MSO and MBO output files from the compiler are in this EOD format. The choice of using EOD as the filetype of the output from the linker is an arbitrary one.

The 432 linker combines a set of compiled EOD's (e.g. the .MSO and .MBO files) into a single linked EOD. Compiled
EOD's, generated by the ACS, contain program modules. These modules, in turn, contain a collection of compiler-generated objects, such as segments, refinements, etc. The output from the linking process, a single file, is then downloaded to the MDS 800 system.

The 432 linker performs the following traditional functions:

1. Resolves inter-module references.
2. Assigns physical memory addresses to all segments contained in the input modules.
3. Verifies the compatibility of modules that are linked together.
4. Produces a linked EOD that may be loaded into the System 432/670 main memory and executed.
5. Generates error messages for abnormal conditions encountered during processing.
6. Generates a linker listing that summarizes the results of the linker operation and address assignment.

In addition, the linker performs the following 432-specific actions:

1. Version checks the input EODs for compatibility.
2. Assigns object table directory indices and object table indices (known as object coordinates) for objects contained within the input modules.
3. Builds the physical 432 access segments described symbolically within each input module.
4. Builds object tables and the object table directory associated with the objects in the input modules.
5. Generates initialization object tables, access descriptors, and storage allocation information.

The net result of all this is an EOD which, when loaded into 432 memory, will execute as one has programmed it. The input or directives file to the 432 linker should be a file created on VMS with a file extension of LKD. This file, an example of which is provided in Figure 21, may have other file extensions or types. However, if that is the case, then the full file name must be given to the linker, i.e., LKD is the default file type. For example, given a link file which we call "TEST.LKD", to link this file, the following command would be entered:

LINK432 TEST

The linking process can be appreciably longer than compilation. However, if linkage is successful, a single, simple message of:

LINKAGE SUCCESSFUL

should be the only message which appears on the console. Warning messages, not error messages, accompanied by "LINKAGE SUCCESSFUL", do not really mean a successful linkage! At least this has been true in our experience. A detailed explanation of the different directives which can appear in the linker file, plus their meanings, can be found in the manual, "VAX/VMS Host User's Guide". With the
culmination of a successful linking, one is ready to download the output file generated by the linker to the MDS 800 system. For a detailed explanation of the linking process and the available directives, i.e., commands included in the link file, refer to "VAX/VMS Host User's Guide".

```
; An example of a link file which serves as input to the 432 linker. The semicolons which precede ; these statements signify comments. Link, free, ; output, print, and objectmap are examples of ; linker directives. The blank lines which occur ; between directives MUST be present!

link ACS:IMAXv1.eod
ACS: textio.mlo
example1.mso
example1.mbo
main.mso
main.*bo
pserp.mbo
osors.mbo

free(1 in directory)

output example.eod

print example.map

objectmap

This could be filed in VMS as TEST.LKD
```

Figure 21. A Linker Input File

4. DOWNLOADING

Downloading is performed on the MDS 800 system. In order for downloading to be accomplished, the VAX must be
operating under VMS. A cable, marked with a tag which reads "VAX", is the transmission facility for downloading. The following steps comprise the procedure to follow when downloading a file:

1. Attach the VAX cable to the ADM36 terminal. Logon to VMS as you normally would. Enter the following command: "SET TERM/SPEED=2400". This is done because the MDS 800 system is currently modified to support only 2400 baud communication rates unless hardware/software changes are implemented.

2. Remove the VAX cable from the ADM terminal, connect one end to a null modem. Connect the other end of the null modem to the MDS 800 TTY port located on the control unit.

3. Insert into drive 0 of the MDS 800 system the ASYNCH LINK diskette.

4. Insert into drive 1 the diskette one wishes to download to. Root the system.

5. On the MDS 800 terminal, enter the following command: "DNLOAD <VMS EOD file> TO :Fl:<new or same file name>. For instance, assume one has an EOD file named TEST.EOD in the VMS directory. Furthermore, one wishes to call this file TEST1.EOD on the MDS 800 system. One would enter the following command: "DNLOAD TEST.EOD TO :Fl:TEST1.EOD", quotes not included.

We have experienced average download times of approximately 20 minutes. Any errors in transmission mean that downloading must be redone until a complete error-free download is accomplished. We have not experienced any errors in downloading to date. The conclusion of a successful download marks the beginning of the next step, execution on the 432.
5. PROGRAM LOAD/EXECUTION

Now that a linked EOD file is on a diskette, all that remains is to load it into 432 memory and execute it. The following procedure assumes that the MDS'800 system and the 432/670 execution vehicle are powered up and have no hardware faults. In the following discussion, commands which are to be entered at the MDS 800 terminal (termed the "debugger console" by INTEL) will be printed in capital letters and enclosed in quotes. This is for illustration purposes only. Capital letters are not necessary, and quotes will result in an error message.

1. Insert into drive 0 of the MDS 800 system, the diskette labeled UPDATE-432/DEBUG-432.

2. Insert into drive 1 the diskette which contains the executable program. Boot the system.

3. Enter the following command: "RUN WORK :FO:"

4. When the ISIS prompt (\) returns, enter: "RUN DEB432". This should result in the display of "SERIES III 432 Systems Level Debugger, V1.00".

5. Once in the debugger, the ISIS prompt will be replaced by a "?" as the prompt symbol. Enter the command: "INIT".

6. When the prompt returns, enter: "INCLUDE DEB432.TEM".

7. When the prompt returns, enter: "DEBUG :Fi:<filename, filetype >". For example, suppose one has downloaded the file TEST,EOD which one wishes to execute. Here, one would enter: "DEBUG :Fi:TEST,EOD".

8. Enter: "START". This command initiates program execution.
This command will result in program execution on the 432. For an in-depth explanation of debugging facilities available on the 432, in case the program does not execute as planned, refer to "Workstation User's Guide".
LIST OF REFERENCES


<table>
<thead>
<tr>
<th>No.</th>
<th>Copies</th>
<th>Initial Distribution List</th>
</tr>
</thead>
</table>
| 1.  | 2      | Defense Technical Information Center  
Cameron Station  
Alexandria, Virginia  22314 |
| 2.  | 2      | Library, Code 0142  
Naval Postgraduate School  
Monterey, California  93940 |
| 3.  | 2      | Department Chairman, Code 52  
Department of Computer Science  
Naval Postgraduate School  
Monterey, California  93940 |
| 4.  | 2      | Associate Professor Uno R. Kodres, Code 52Kr  
Department of Computer Science  
Naval Postgraduate School  
Monterey, California  93940 |
| 5.  | 2      | Capt. Bradford D. Mercer, Code 52ZI  
Department of Computer Science  
Naval Postgraduate School  
Monterey, California  93940 |
| 6.  | 1      | RCA AEGIS Data Repository  
RCA Corporation  
Government Systems Division  
Mail Stop 127-327  
Moorestown, New Jersey  08057 |
| 7.  | 1      | Library (Code E33-05)  
Naval Surface Warfare Center  
Dahlgren, Virginia  22449 |
| 8.  | 1      | Daniel Green (Code N20E)  
Naval Surface Warfare Center  
Dahlgren, Virginia  22449 |
| 9.  | 1      | CDR J. Donegan, USN  
PMS 40085  
Naval Sea Systems Command  
Washington, DC  20362 |
Fleet Systems Department  
Applied Physics Laboratory  
Laurel, Maryland 20810

11. Lt. Dave Applegate  
413 Exeter Place  
Marina, California 93933

12. Capt. Robert Coates  
5840 Avenida Jinette  
Bonsall, California 92003
END
DATE FILMED
5 - 83
DTIC