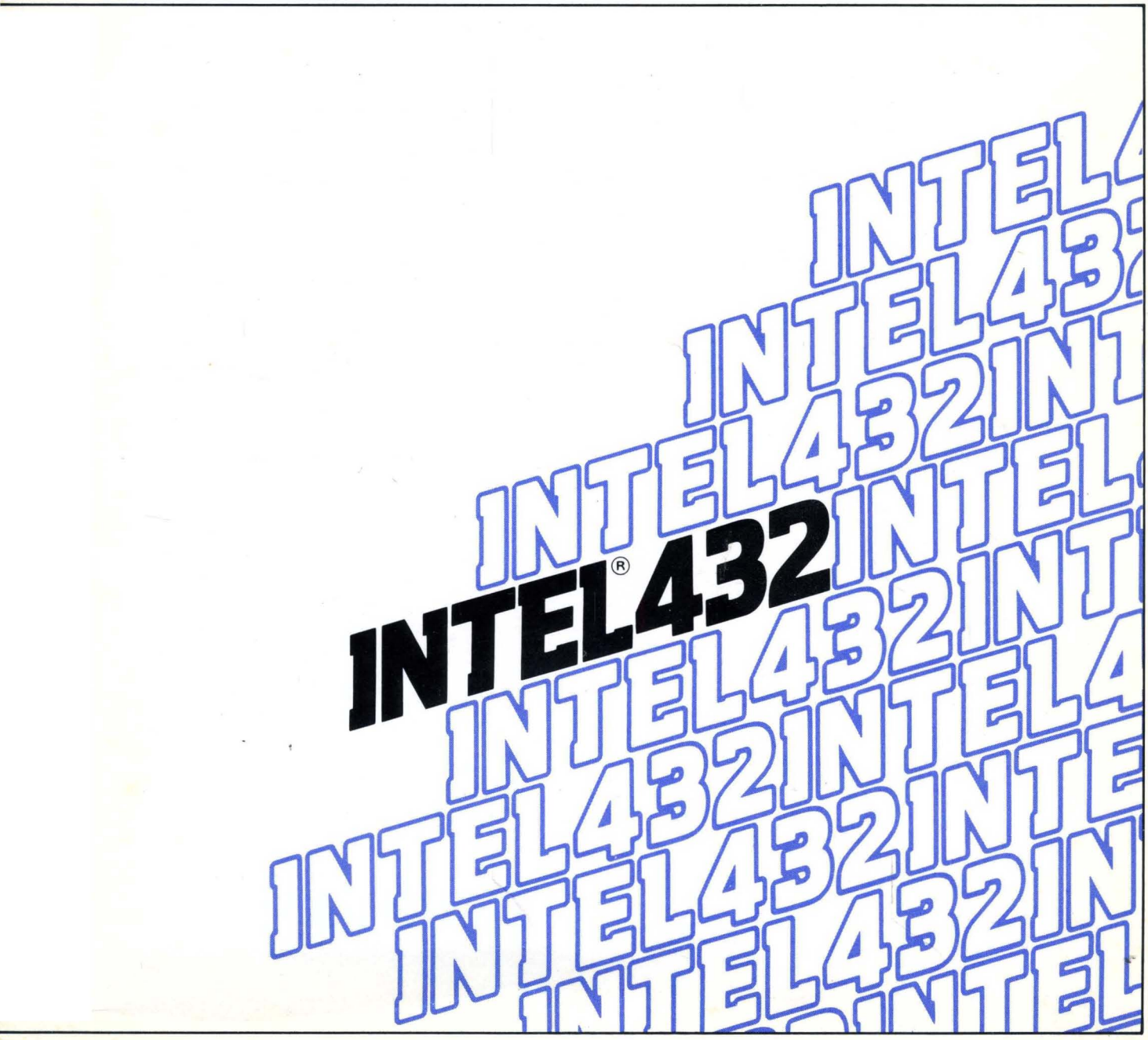
intel

Introduction to the iAPX 432 Architecture



171821-001

INTRODUCTION TO THE iAPX 432 ARCHITECTURE

Manual Order Number: 171821-001

Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051

Additional copies of this manual or other Intel literature may be obtained from:

Literature Department
Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95051

PREFACE

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 of over a trillion (2^{40}) bytes; and it supports on the chip itself the proposed IEEE-standard for floating-point arithmetic. Because it is so advanced, a discussion of the iAPX 432 architecture will unavoidably introduce concepts that are new to many readers.

The purpose of this document is to provide an accurate and comprehensive overview of the architecture, recognizing that not only are many of the concepts new, but that readers with widely divergent backgrounds are likely to be interested in the iAPX 432. Consequently, a considerable amount of background information will be supplied.

The Introduction is organized as follows: Chapter 1 introduces the concept of architecture and the various elements of an iAPX 432 system. Then the focus is narrowed to the architecture of the principal processing element, the General Data Processor (GDP).

Chapters 2, 3, and 4 provide an overview of three broad areas where the GDP architecture represents a significant advance over contemporary architectures, namely memory organization, data manipulation, and hardware support of state-of-the-art programming methodologies. These chapters motivate and explain most of the architectural features.

The *iAPX 432 General Data Processor Architecture Reference Manual* contains complete, detailed descriptions of all aspects of the architecture. It should be consulted whenever more information is required on any of the topics covered in this document.

CONTENTS

TABLES

ILLUSTRATIONS

CHAPTER 1
COMPUTER ARCHITECTURE

1.1 What is Computer Architecture?

The term *computer architecture* is often used as if it meant simply "the organization and design of computers." This rather loose definition must be made considerably more precise before we can study the differences between the architecture of the iAPX 432 and that of more conventional computers.

In practice, there are several distinct architectures within a computer system, each defined by a boundary between different levels of the system. Figure 1-1 shows an abstract picture of a computer system, with the simplest operations and functions on the bottom and the most complex, user-dependent operations on top. Each level makes use of the functions provided by the level below. The whole effect is like a stack of bricks, with each of the higher bricks supported by the ones below.

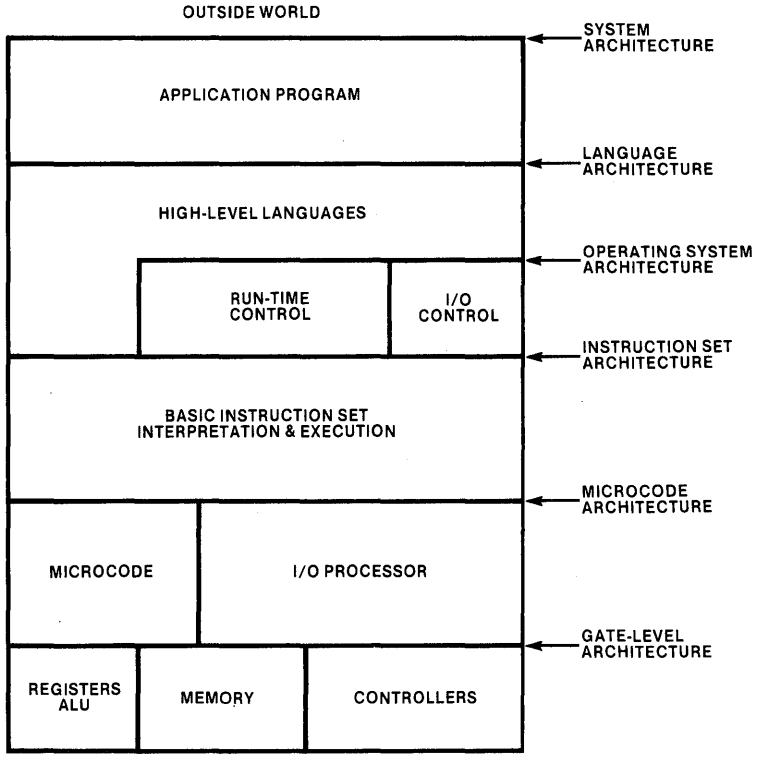


Figure 1-1. Computer System Architectures 171821-01

An architecture is the boundary or interface between two of these functional modules or bricks. It might be defined as *the functional appearance of the system below an interface to a user above the interface*. Designers above an architectural interface are generally not concerned with the details of the system below the architecture or with any still lower-level architectures. They are only concerned with the function of the system at the interface immediately below them.

In his classic book, *The Mythical Man-Month*, Frederick Brooks, project manager of the IBM System 360, summed up the notion of architecture as "the complete and detailed specification of the user interface."

The highest architecture is the interface between the whole computer system and the outside world; this can be called the *system architecture*. Somewhat lower down is the interface between the application program and the high-level programming language itself (assuming the application is written in a high-level language). This boundary can be said to define the *programming language architecture*. Since more and more programs are written in high-level languages, this architecture is the one most programmers will be concerned with. Still lower is the interface between the language and various run-time resource management functions that are usually provided by operating systems. We can call this boundary the *operating system architecture*.

The next interface is especially significant, because in conventional computers it defines the boundary between hardware and software. It is the level at which elementary, machine-recognized instructions are decoded and executed. We shall call this the *conventional instruction set architecture*. The two lowest-level architectures (*microcode architecture* and *gate-level architecture*) define even more primitive functions and are not of real concern to most programmers.

Out of this multiplicity of architectures, the one usually called *the* computer architecture is defined by the boundary between hardware and software. To be precise, the computer architecture is the level of the computer system that is seen by an assembly language programmer or compiler writer.

For each computer, the computer architecture can be seen as a kind of horizontal "slice" through the diagram in figure 1-1. Everything below the slice is performed by the hardware in this computer; everything above the slice is performed by the software. The major task of the computer architect is to decide which functions are best performed below the boundary and which above the boundary. Naturally, this boundary is not the same for all computers. Over the years, as computer technology has developed, the boundary between hardware and software has changed. When we compare the architecture of the iAPX 432 with other computer architectures, we will be discussing where this boundary between hardware and software has been drawn.

Vertical slices through the system diagram in figure 1-1 may be considered as well. A vertical slice separates components into a multiprocessor or distributed processing system. For example, a cut that separates out I/O functions defines an I/O processor (sometimes called a front-end processor). Other vertical cuts might separate several general purpose processors and so define a multiprocessor system. These vertical cuts may be said to define a *configuration architecture*. (See figure 1-2.)

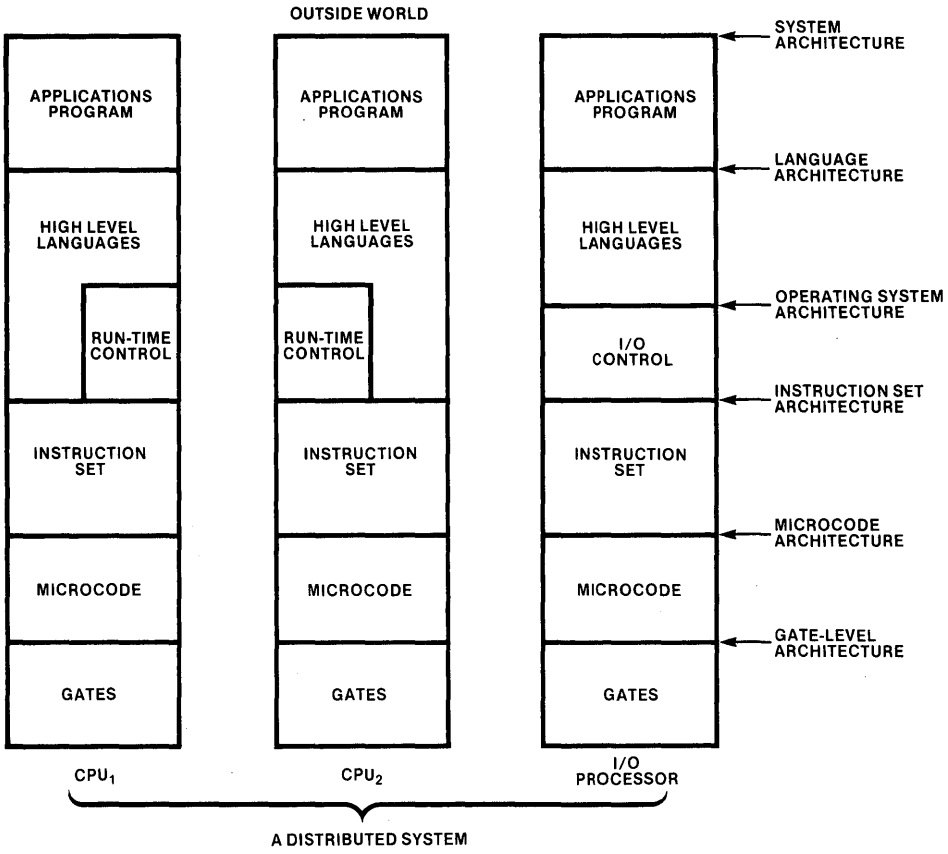


Figure 1-2. Configuration Architecture 171821-02

1.2 The Hardware-Software Interface

Thirty years ago computers were very expensive to build; the hardware filled whole rooms and a single gate might cost several hundred dollars. Because of these costs, early computers could have only very simple hardware operations, such as "Add the contents of register X to register Y." Consequently, it made good economic sense to substitute a program for a piece of equipment whenever possible. If more elaborate operations were needed, they were implemented with a library of subroutines. Thus, the architecture of these early machines represented a fairly low-level cut through the diagram in figure 1-1.

As technology has advanced, hardware costs have declined rapidly. Meanwhile, for a variety of reasons which we shall examine in Chapter 4, the overall software costs of computer systems have increased astronomically. Today, as much as 80% of the total system cost can be due to software. Thus, the architectural motivations of the 1950s and 1960s—trying to minimize hardware costs at the expense of software—make little sense today.

But a recent study concluded:

> If one compares the architecture of most current, widely-used machines (e.g., IBM System/370 and System 32, DEC PDP-10 and PDP-11, CDC 6600, Univac 1108, and Intel 8080) to . . . the first electronic stored-program computers (built in the 1940s), all the significant differences will be found to have originated in the 1950s . . . . Although current systems differ significantly in terms of cost, speed, reliability, internal organization, and circuit technology, the computer architecture . . . has not advanced beyond the concepts of the 1950s.*

This lack of advance is why we can use a single line in figure 1-1 to represent "conventional instruction set architecture" (i.e., conventional computer architecture).

The question is this: If hardware costs are not a factor, which software functions should be done in the hardware, where they can be done faster and more reliably? Some functions can be added to the hardware with little or no controversy. Instead of libraries of floating point arithmetic subroutines, manufacturers now build a floating point unit into the computer itself. But before more complicated functions can be added to the hardware, decisions must be made about high-level languages, operating systems, and even programming methodologies, since raising the hardware-software interface means that the structures and operations in the architecture will be more closely related to the structures and operations in the programming language and operating system.

Consider the following historical example. In the late fifties, dissatisfaction with Fortran led computer scientists to try to formulate a better computer language. This research culminated in the design of Algol 60. At about that time, the preliminary specifications for the Burroughs B5000 computer were being drawn up. The architects of the B5000 decided to design an architecture specifically to support Algol instead of a conventional architecture. They committed themselves to high-level language programming instead of assembly language programming, and they chose the best language available to them.

The B5000 architecture had a number of advanced features, including segmented memory to support the static block structure of Algol and a stack mechanism to support its dynamic behavior. In fact, the B5000 and its successors were among the few important commercial machines developed during the 1960s and 1970s whose architectures represented any advance over the concepts of the fifties.

The B5000 had the right approach; it attempted to raise the level of the architecture using the best available programming methodology (c. 1960), which largely reduced to "use Algol", and the architecture supported Algol very effectively. But in the 1970s and 1980s problems have arisen for which Algol and the programming methodology of the early 1960s offer no solution.

These problems have led other manufacturers, whose earlier computers had more conventional architectures, to recognize the wisdom of raising the level of the hardware-software interface. Consider, for example, the IBM System 38, IBM's most recent architecture. Not only have the designers of the System 38 followed the Burroughs approach in architectural support for high-level languages, they have also included most of the operating system in the hardware as well. It seems inevitable that the fundamental problems facing the computer industry will force more and more manufacturers to take this approach.

---

*Glenford J. Myers, *Advances in Computer Architecture* (New York: 1978)

1.3 Current Problem Areas

The basic problem facing the computer industry today is the software crisis; software cost as a fraction of total system cost has dramatically increased over the last two decades. It is an odd kind of crisis, because it is in a sense a crisis caused by good fortune: decreasing hardware costs have made possible more and more ambitious projects. Unfortunately, our ambitions greatly exceed our current programming abilities. There are four fundamental problems: the size of modern programs, the demands for security, the use of concurrency, and the need for expandable systems. These four topics are discussed in detail in Chapter 4. The software crisis has thrown a spotlight on the deficiencies of current architectures. We can now see that conventional architectures were designed to solve hardware problems that are no longer relevant. The current problems also identify three areas where the hardware-software interface can be profitably raised: memory organization, data manipulation, and support for the programming environment. (The topics that follow are discussed in much greater detail in Chapters 2, 3 and 4, where all the key terms are defined.)

Memory Organization: The logical organization of memory and the memory protection mechanisms should reflect the organization of programs, not the arbitrary limitations of the current hardware technology. Large virtual memory systems are needed to free users from concern over the size limitations of physical memory. Efficient use of memory will also require hardware support for dynamic storage allocation mechanisms, so that programs take only the memory they need, and memory that is no longer in use can be quickly reallocated.

Data Manipulation: The instruction set should support high-level language statements easily and yet produce compact code. The instructions should have a complete set of operators for a full range of data types, from bits to long floating-point numbers (typically at least 64 bits).

Support for the Programming Environment: A new programming methodology has been developed over the last decade which can make programming cheaper and more reliable. This new "object-oriented" methodology is based on the concepts of abstract data types and protection domains, concepts which are difficult to implement at an acceptable performance level unless they are supported by the architecture. In addition, many operating system mechanisms should be implemented in the architecture, where they can be handled more rapidly and securely.

1.4 iAPX 432 Architecture

The iAPX 432 represents one of the most significant advances in computer architecture since the 1950s.

Figure 1-3 shows a diagram similar to the one in figure 1-1, but with lines drawn to represent the level of the hardware-software interface in conventional micro-, mini-, and mainframe computers, and in the iAPX 432.

1.4.1 Main Features

The iAPX 432 is the first computer whose architecture supports software-transparent, multi-processor operation. Processors can be added to or removed from a system to select the desired price/performance level, without requiring any software changes. If one processor fails, the rest of the system can usually continue to operate.

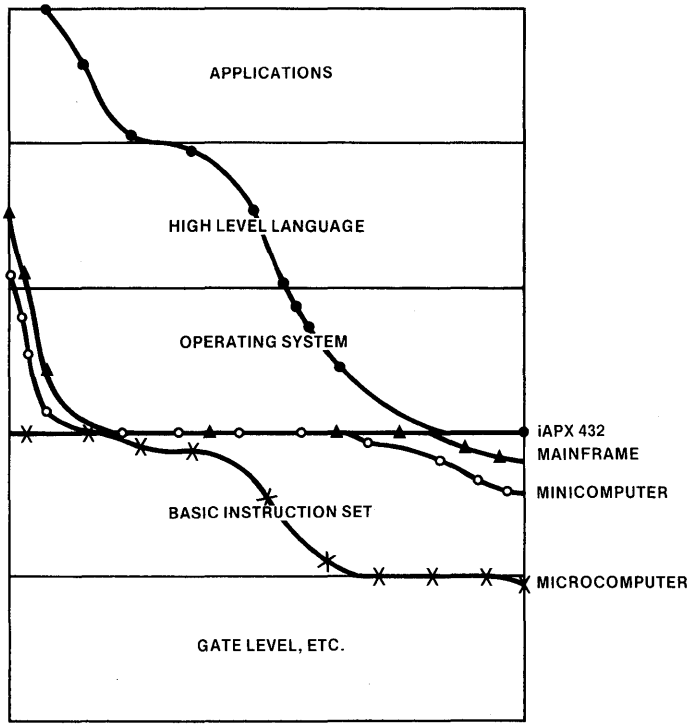


Figure 1-3. Raising the Hardware-Software Interface 171821-03

The iAPX 432 is the first commercial computer whose architecture fully supports the new object-oriented programming methodology. This new methodology can significantly reduce the enormous cost of software in contemporary systems. Both abstract data types and objects are supported by hardware recognized, hardware protected, and hardware manipulated structures. The iAPX 432 architecture also includes the Silicon Operating System, which provides the hardware mechanisms to support operations such as process scheduling, interprocess communication, and dynamic storage allocation that are implemented by the operating software in most computers. (See Chapter 4 for more details.)

The iAPX 432 is designed to be programmed entirely in high-level languages. To facilitate compiler writing, the instruction set is completely symmetric. (All addressing modes are available for every operand, and all required operators are available for every data type—see Chapter 3 for more details.) The iAPX 432 has also implemented the proposed IEEE floating-point arithmetic standard.

The iAPX 432 has an extremely large virtual address space (2^{40} bytes) and hardware-supplied mechanisms for implementing virtual memory systems that can exploit this address space. The memory architecture is segmented, with enough segments (over 16 million) for every meaningful program unit to have its own segment. Thus the memory protection mechanisms, which are based on segments, can give hardware protection to the logical structure of programs. (See Chapter 2 for more details.)

The iAPX 432 hardware can detect hundreds of different fault conditions, from attempting to divide by zero or attempting to execute data, to complex faults involving several processes. Most computers do not detect these faults at all, even at the operating system level, so it is common for the system to crash or for data to be destroyed.

On the iAPX 432, if a fault is detected, the operation is aborted, and a complete description of the fault is reported. In a multi-process system, a fault may cause the current process to suspend itself, but the other processes can continue to execute. In a multi-processor system, a fault may cause one processor to suspend itself and begin running diagnostics, but the other processors can usually keep the system operating. (Fault handling is described in Chapter 12 of the *iAPX 432 General Data Processor Architecture Reference Manual*.)

Table 1-1 summarizes the principal differences between conventional architectures and the architecture of the iAPX 432.

Table 1-1. A Comparison of the iAPX 432 Architecture and the Architecture of Conventional Mainframes

Feature of Architecture	Conventional Mainframe Architecture	iAPX 432 Architecture
MEMORY ORGANIZATION		
Organization, size	Linear $2^{24} - 2^{32}$ bytes	Structured segmented 2^{24} segments 2^{16} byte displacement 2^{40} byte virtual address space
Logical to physical address translation	Single-level map Page-based relocation and virtual memory	Two-level map Segment-based relocation and virtual memory
Protected memory unit	Fixed-size page	Individual program module or data structure
DATA MANIPULATION		
Expression evaluation	General register	Stack or memory-to-memory
Primitive data types	Characters, unsigned integers, integers, reals	Characters, unsigned integers, integers, reals temporary reals
Floating point hardware	Yes	Yes
Addressing modes	Some modes not available for all operands	Symmetrical: all modes available for every operand
PROGRAMMING ENVIRONMENT SUPPORT		
Operating system	No multi-process support	Multi-process mechanisms in hardware
	No support for dynamic storage allocation	Dynamic storage allocation mechanisms in hardware
	Very limited multi-processor operation, if any	Software-transparent, multi-processor operation
High level language	Assembly language-oriented instruction set	Oriented toward high level languages
Programming methodology	No support at all	Object-based architecture

1.4.2 Configurations

The iAPX 432 configuration architecture is defined by the components that make up an iAPX 432 system. In order to achieve high performance in both general purpose computation and input/output operation, the iAPX 432 has a distinct type of processing unit for each of these functions. The General Data Processor (GDP) handles all program decoding, computation, and address generation. The Interface Processor (IP) handles all communication with peripheral devices. Communication among the GDP, IP, and memory is provided by a packet-based interconnect bus. The IP is also connected to an interrupt-driven I/O subsystem bus, to which all peripherals are interfaced. A conventional processor, called the Attached Processor (AP), provides processing power in the I/O subsystem.

Figure 1-4 gives an overview of a small iAPX 432 system, showing GDP, IP, AP, memory, the interconnect bus, and the I/O subsystem bus.

The iAPX 432 is the first computer whose architecture allows for true software-transparent multiprocessor operation. This is perhaps the most revolutionary feature of the iAPX 432 architecture. The number of processors in a system is totally invisible to the software. Several processors (both GDPs and IPs) and memory controllers can be attached to a single interconnect bus. Figure 1-5 shows a system with two GDPs and two IPs.

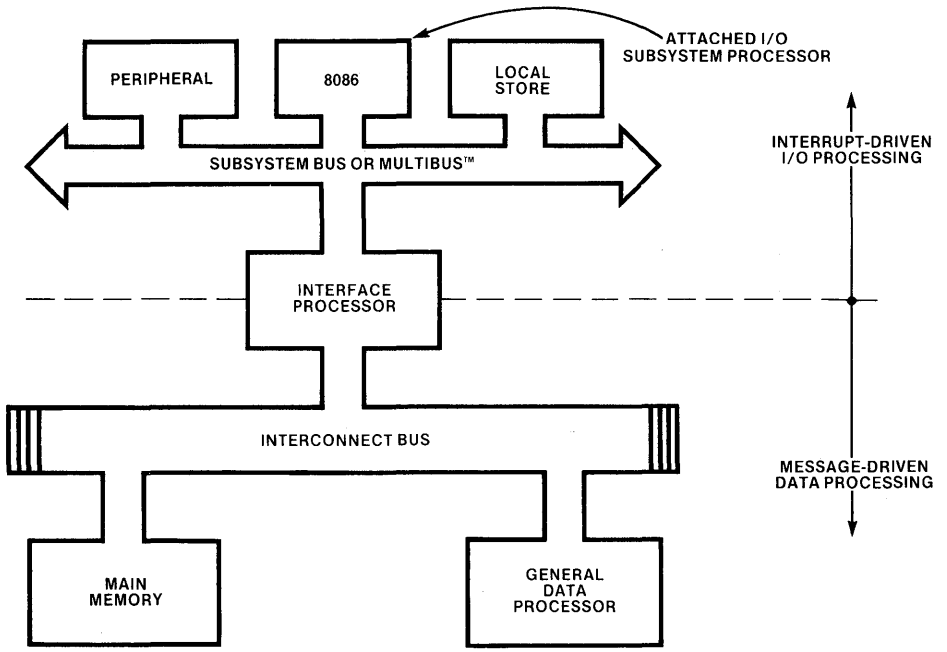


Figure 1-4. A Small iAPX 432 System 171821-04

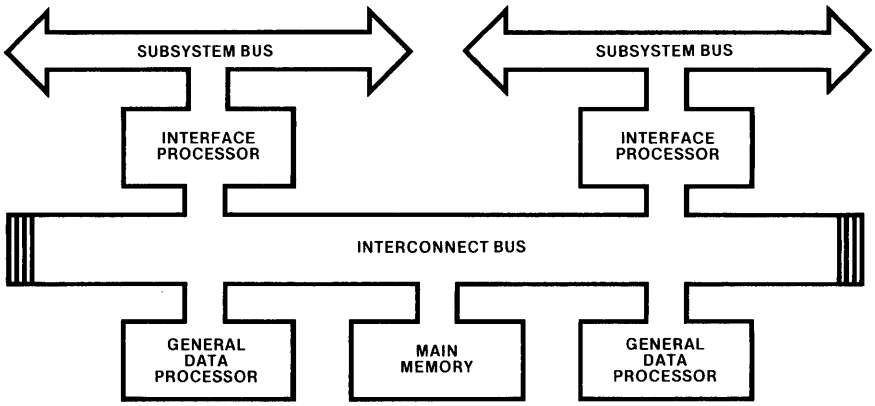


Figure 1-5. A Multiple-Processor System 171821-05

The Intel iAPX 432 computer system family covers a wide range in processing power, from the iSBC 432/100 single-board computer, which contains a single GDP (with peripheral interfacing provided by discrete components), to systems with several GDPs, IPs, and APs.

1.5 Topics Covered in this Document

Although system configuration is an important topic, this document will deal solely with the architecture of the General Data Processor, the principal component in an iAPX 432 system. System configuration, the architecture of the IP, and the related topics of communication with the outside world, initialization, program downloading, and interface to an interrupt-driven external processor will be covered in separate documents.

The next three chapters will examine three broad areas where the level of the hardware-software interface has been raised in the iAPX 432 architecture. Chapter 2 describes the memory architecture of the iAPX 432 and compares it to the memory organization of conventional computers. Chapter 3 describes the data manipulation instruction set of the iAPX 432 and shows how it supports high-level languages. Chapter 4 explains what objects are and how an object-oriented architecture can be used to support the new software methodology and the Silicon Operating System.

CHAPTER 2
MEMORY ORGANIZATION

2.1 Fundamentals

One of the most important characteristics of a computer architecture is the way memory is organized and the way information in memory is accessed.

The main memory of a computer is organized as a set of storage locations numbered consecutively, beginning with zero. The number associated with one of these physical storage locations is called a *physical address*, and the set of all physical addresses is called *physical address space*.

A *logical address*, on the other hand, is an address in an instruction—an address as used by a programmer. The *logical address space*, the set of all logical addresses, is thus the set of addresses that can be used by a program. The organization of the logical address space defines the *memory architecture*.

The organization of physical address space is determined by memory technology and by cost, but the logical address space should not necessarily be constrained by either of these considerations. Instead, the organization of logical memory should be determined by the structure of the programs that will run in memory. And in fact, the history of advanced memory architectures shows an evolution toward this goal. Initially, the logical address space was identical to the physical address space; now, in the iAPX 432, the level of the memory architecture has been raised much closer to the level of user program organization. In this chapter we will explore some of this history and show how it has led to the iAPX 432 memory architecture.

2.2 Linear Memory

The most common organization of logical address space; that is, the most common architecture for memory, is a linear, contiguous address space. In this kind of architecture addresses start at location zero and proceed in linear fashion (i.e., no holes or breaks) to the upper limit imposed by the total number of bits in a logical address. A program, often consisting of several procedures, and its data are all located within this single address space. The logical address space of a linear memory thus has the same basic organization as the physical memory.

The simplest kind of linear address space is exemplified by the 8080. 8080 programs can generate 2^{16} (65,536) distinct addresses. (See figure 2-1.) The addresses generated by 8080 programs are used directly by the memory hardware to locate data. Thus, a Load Accumulator instruction that references address 50000 will cause data to be read from physical memory location 50000.

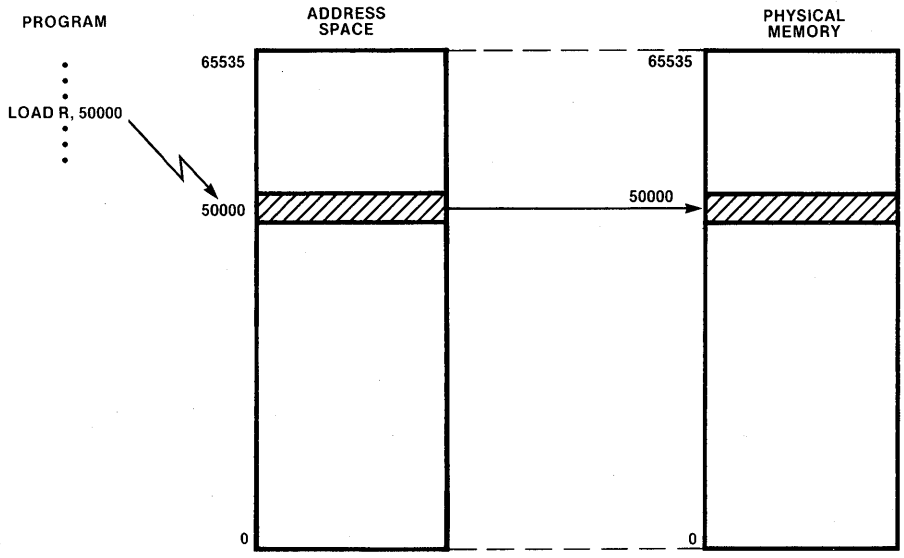


Figure 2-1. Unmapped Linear Memory 171821-06

The limitations of this memory architecture are readily apparent when several programs share the same machine in order to use efficiently the system's resources. It is difficult to implement such a multiple-program system on systems with a memory organization as simple as the 8080's.

First, nothing prevents one program from accessing or even destroying data in another program. Since all programs share the same logical address space, any program can access any location in memory, and no program is protected from unauthorized access by another program. Second, all the programs have to fit inside the relatively constricted logical address space of the 8080.

To illustrate the problem of uncontrolled access of this simple memory organization, examine figure 2-2. A program *can* reference any location in memory, even if it isn't supposed to. In this simple system nothing prevents program A from overwriting program B or other parts of its own code or data, for that matter. A simple programming error by A can wipe out B.

Thus, even though complex software partitioning schemes were employed, multiple-program systems never operated reliably or securely with this memory organization. To overcome these liabilities without abandoning linear architecture, new mechanisms for memory management were invented: logical-to-physical address translation (also called *mapping*), memory access control based on fixed-sized pages, and virtual memory. These mechanisms are explained in the next four subsections.

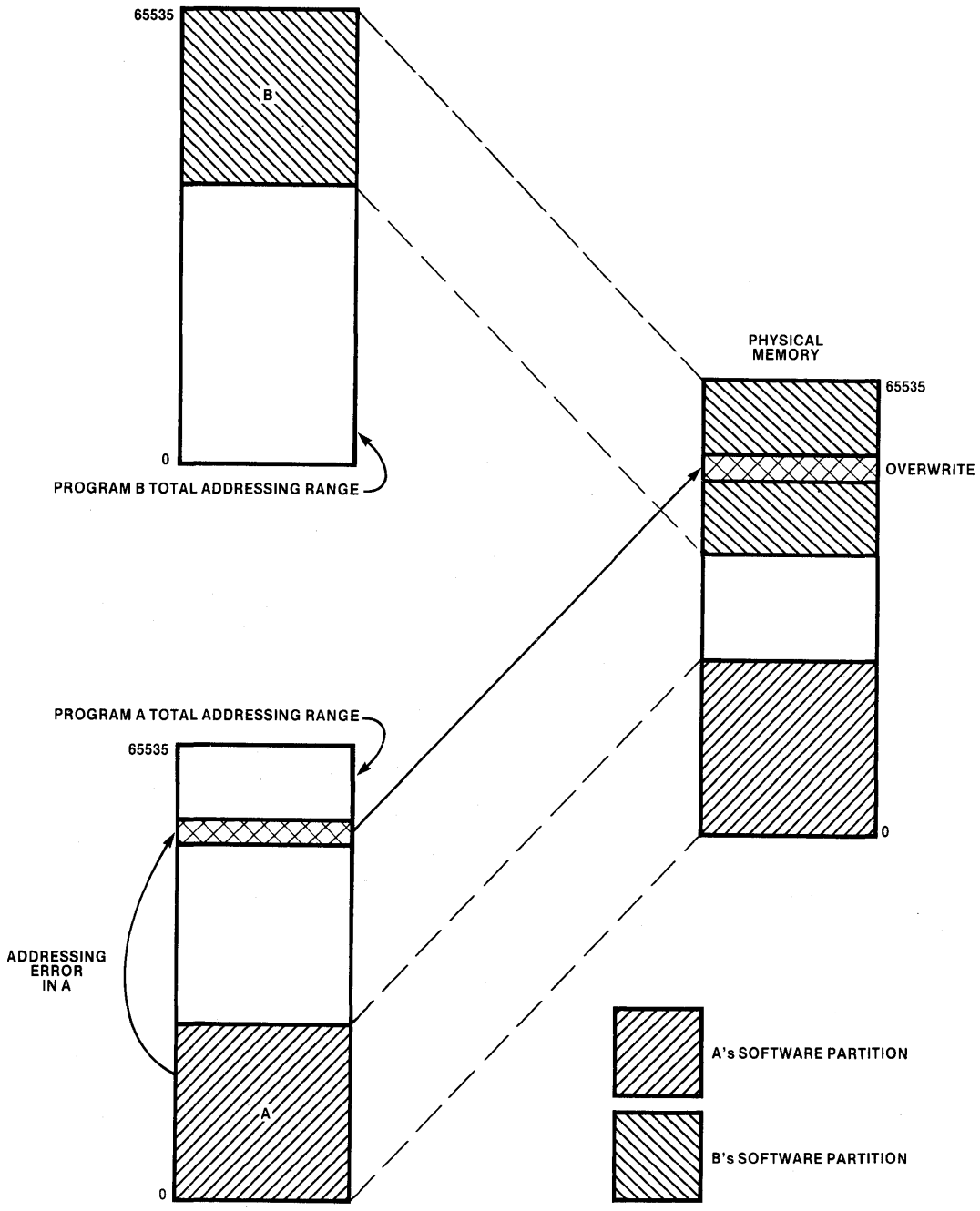


Figure 2-2. Vulnerability of Unmapped Memory

171 821-07

2.2.1 Mapping Linear Memory

Mapping is basically the process of translating logical addresses into physical addresses. In the 8080 system we considered above, logical addresses are simply equated to physical addresses; but by exploiting mapping a logical address can be assigned to an arbitrary physical address. Thus mapping is a mechanism for relocating the logical address space within the physical address space.

Figure 2-3 shows a very simple mapping operation. The entire logical address space of a program (locations 0-65535 in this case) is mapped onto physical locations 30000-95535. Thus, a program reference to location 50000 actually fetches data from physical location 80000 (50000 + 30000).

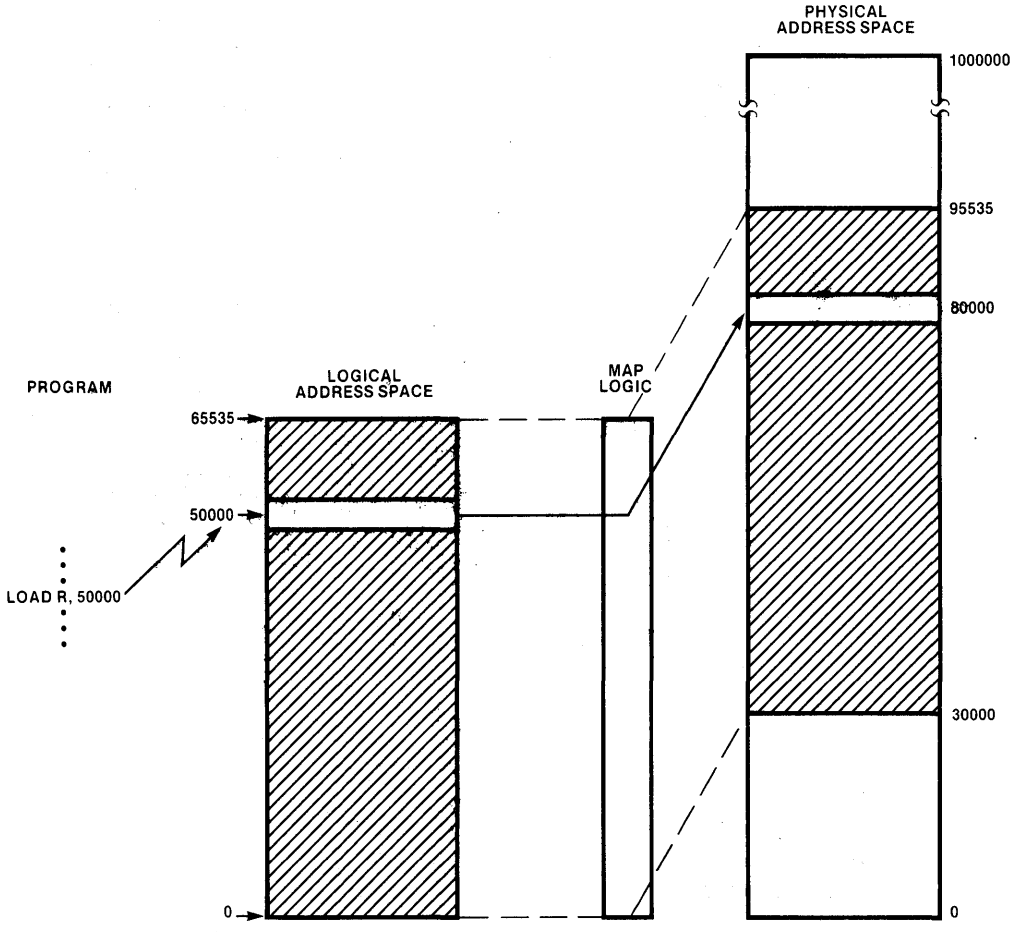


Figure 2-3. A Simple Mapping Scheme

171821-08

The utility of this operation may not be obvious, but it is extremely useful in multiprogram systems. (In fact, mapping was developed for such systems.) With this kind of mechanism, which is typical of most minicomputers and some advanced microcomputers, each program has its own logical address space which is completely independent of any other program. Thus many programs can share physical memory without any possibility of interfering with each other. The mapping unit fits all the logical address spaces into one physical memory, but the process is transparent to the programs. See figure 2-4 for a multiprogram mapped environment.

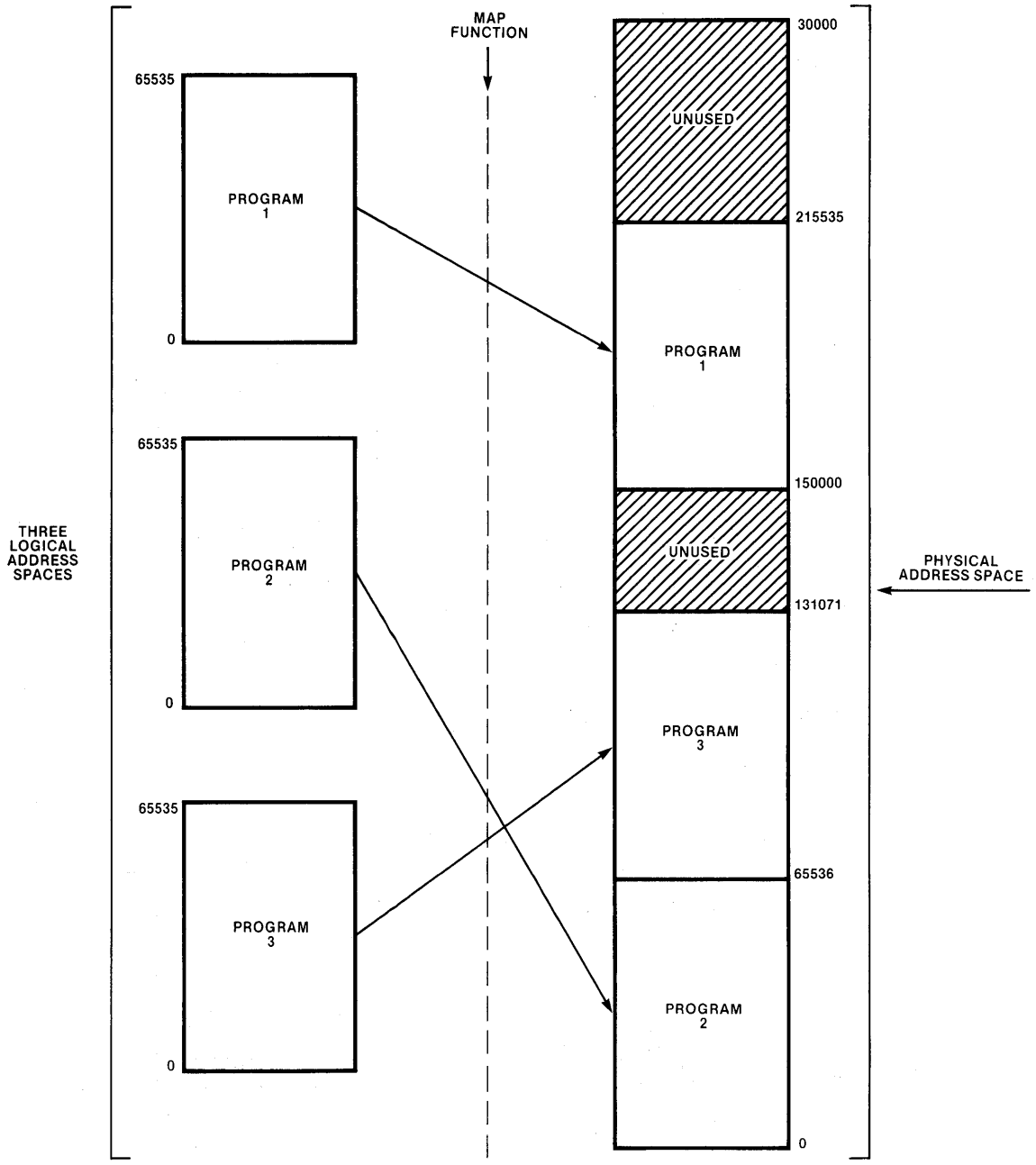


Figure 2-4. A Multiprogram Mapped Environment

171821-09

2.2.2 Page-Based Mapping

Instead of mapping the entire logical address space as a unit, as in figure 2-4, more advanced address translation mechanisms map smaller, fixed-size "pages" of logical address space onto pages of physical memory. Thus, a large program need not be relocated into one contiguous chunk of physical memory, which might be hard to find in a multiprogramming environment, but rather into several smaller sections of memory, which are more likely to be available. (It is often much easier to find twenty 1K pages than one 20K block.)

2.2.3 Access Rights Based on Pages

The page mechanism can also provide the basis for memory protection within a logical address space. Each page can have attributes associated with it (called *access rights*) that indicate how the page can be accessed. These attributes can allow reads only, reads and writes, or they can prevent any access at all. See figure 2-5 for a diagram of such a paged mapping mechanism.

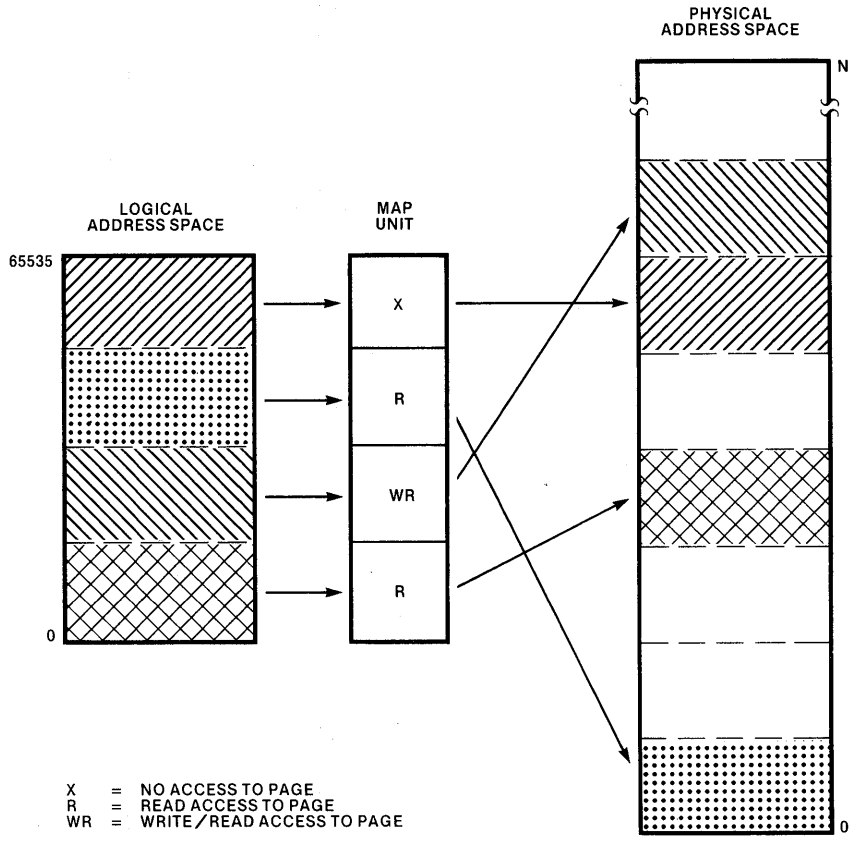


Figure 2-5. Page-Based Protection 171821-10

2.2.4 Virtual Memory

In many computer systems, the logical address space is far larger than the physical memory. *Virtual memory* is a mechanism for getting around the limits on physical memory size. Under a virtual memory system, it appears to users as if the entire logical address space were available for storage. But, in fact, at any given moment only a few pages of the logical address space are mapped onto physical space. The other pages are not present in main memory at all; instead the information in these pages is stored on a secondary-storage device, such as a disk, whose cost-per-bit is more economical.

Every time a missing page is accessed, operating system software loads the missing page from disk and stores on disk a page that has not been referenced recently. The user will have the illusion of a gigantic, although slower, physical memory.

2.3 Segmented Memory

While many systems employ linear memory architecture, with memory management mechanisms of varying complexity, a number of new processors (the Intel 8086 and the Zilog Z8000) and some older systems (the Burroughs B5000 and the Multics processor) have abandoned linear memory architecture altogether. Instead they use a form of logical memory organization called *segmented memory*.

The basic motivation for segmented memory is that programs are not written as one linear sequence of instructions and data, but rather as parcels of code and parcels of data. For example, there could be a main code section and many separate procedures. Data could be organized into arrays, or arrays of arrays, linked lists, or any number of complex data structures. Moreover, these modules of code and data could come in many different sizes.

Segmented memory architectures were developed to support this logical structure. The logical address space is broken down into many linear address spaces, each with a specified size or length. Each of these linear address spaces is called a *segment*. Each item within a segment is accessed by a two component address; the first component (the *segment selector*) specifies the segment itself, while the second component (the *displacement*) specifies the offset from the base of the segment to the item being selected.

Each segment can be used to hold a meaningful program module or data module. Thus, a program might have the main procedure in a segment, each additional procedure in its own segment, and perhaps each major data structure in its own segment. With a segmented architecture, the structure of logical address space reflects the logical organization of the program. (See figure 2-6.)

In contrast, a linear address space is by definition logically structureless. As discussed in the previous section, protection mechanisms for linear memory are usually based on fixed-length pages, whose size is determined by hardware criteria and which have no necessary relationship to the logical structure of programs.

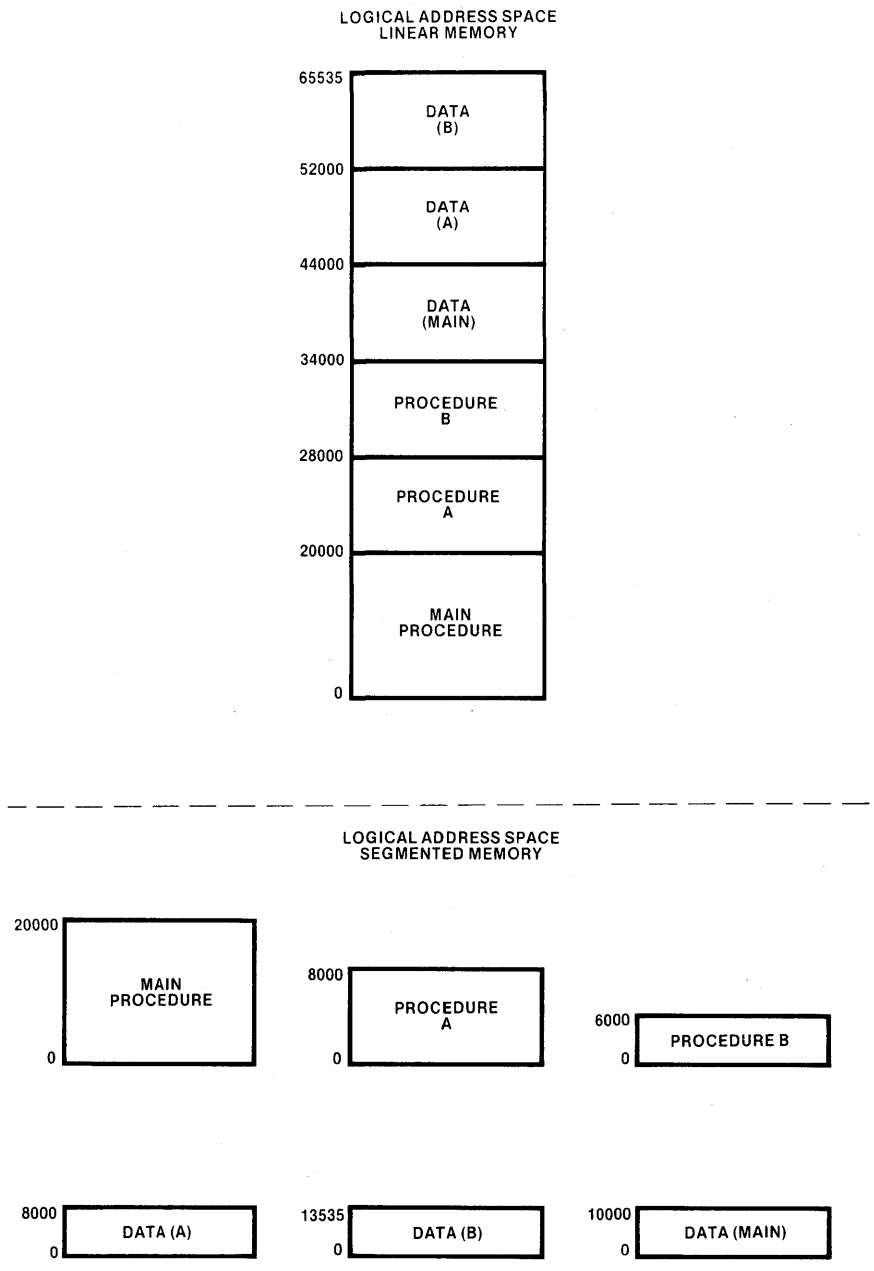


Figure 2-6. Comparison of Segmented and Linear Memory 171821-11

The problem with decomposing the logical address space into pages is that the protection mechanism cannot protect the program modules precisely; it either protects too little or too much. The basic flaw is that the page length is fixed for hardware reasons; thus it is not flexible enough for the logical structure of programs.

By contrast, any protection mechanism provided for segments naturally accrues to meaningful pieces of a program. Since each segment has a specified length, it is easy to give even a small instruction sequence its own segment and protect it from other programs, provided there are enough segments available. (Since programs can consist of hundreds or even thousands of modules, this last criterion makes clear the importance of having a large number of segments in the architecture, if segmentation is to be used properly.)

Virtual memory mechanisms can also be implemented for segmented architectures. In this case the segment is the memory unit that is swapped to and from the secondary storage device.

2.3.1 Mapping Segmented Memory

Segmented architecture is also compatible with logical to physical address translation. In this case, the segment is the unit that is mapped onto physical memory, instead of the whole logical address space or some arbitrary fixed-size unit.

Mapping in this case is implemented by a *segment table* that holds a *segment descriptor* for each segment. A segment descriptor contains the starting physical address and the length of a segment. The segment selector component of a logical address is used as an index to select a segment descriptor in the segment table. Then the displacement is added to the segment starting address to produce the physical address of the operand being referenced. (See figure 2-7.) The displacement is easily checked by the hardware to ensure that the reference does not exceed the length of the segment.

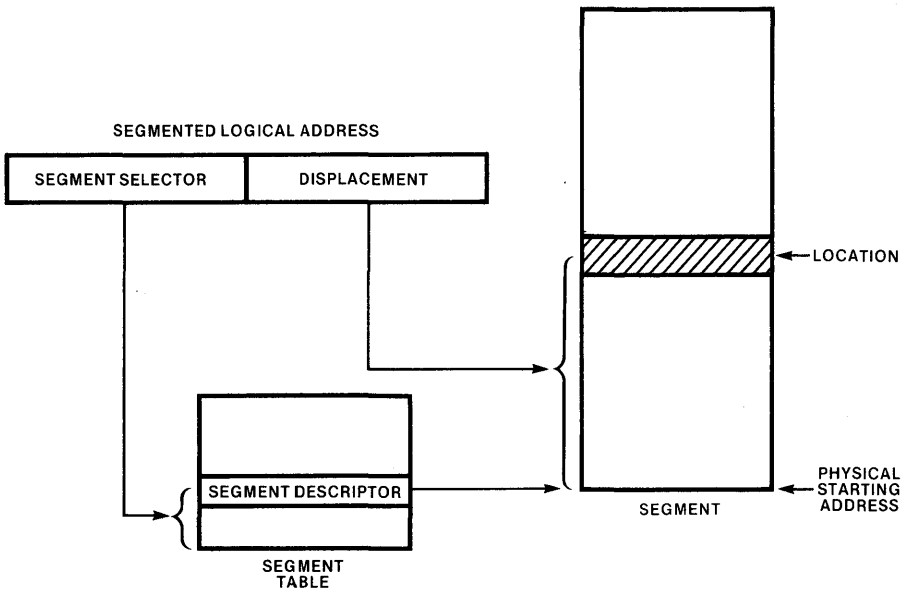


Figure 2-7. Segmented Mapped Memory 171821-12

2.3.2 Segment Types and Access Rights

Many computers with segmented architectures (such as the Z8000) include *segment type* attributes in the segment descriptors. For example, the Z8000 Memory Management Unit supports stack segments, which are defined by a particular bit in the segment descriptor. Whenever an access is made to a stack segment, the hardware checks to make sure that a stack push or pop operation is occurring. If the wrong kind of access is made to a stack segment (an instruction fetch, for example), the hardware signals a fault.

In addition, the segment descriptors often contain the access rights attributes for each segment. Notice that the access rights are therefore associated with particular segments, instead of to the program modules that reference the segments. If a segment is read-only, for example, it is read-only for *all* modules that reference it. This association of access rights with segments is a disadvantage, because we may wish to give different modules access to the same segment, but with different access rights.

2.3.3 Access Control

Another disadvantage with the segment mapping mechanism we have described is that it is difficult to limit the access by one program to another program's segments. Since the segment table contains *all* the segment descriptors, any program can access any segment simply by indexing through the segment table. However, the usual cure for this problem, multiple segment tables, may be worse than the disease itself. Under this scheme each program is given its own segment table, with each table referencing only the segments that the program needs. But this scheme implies that whenever a segment is relocated in physical memory, all the programs that share the segment will have to update their segment descriptors. This could require hundreds of entries throughout memory to be changed. There are great advantages in keeping all the segment descriptors in one central location.

2.4 Structured Memory

The iAPX 432 has a segmented memory. However, the differences between the iAPX 432 and other segmented machines are great enough to justify the use of a new term--structured memory--for the iAPX 432 memory architecture.

First, the iAPX 432 can address a very large number of segments -- 2^{24} -- or approximately 16 million segments, far more than most other computers. Furthermore, each segment can be up to 2^{16} bytes long. The total virtual address space is therefore 2^{40} bytes, that is to say, more than one *trillion* bytes!

Second, the 432 uses a two-step mapping process that separates segment relocation from access control. This two-stage mapping also allows a division of protection features into segment-specific protection (segment type checking) and user-specific protection (access rights). The two-step process is unique to the iAPX 432 and is responsible for some of its most powerful features.

2.4.1 Two-Level Mapping and Access Control

The iAPX 432 mapping is based on the simple, one-step segment mapping process described in section 2.3.1 and illustrated in figure 2-7, but another step has been added.

Each independently translated program module is supplied at run-time with a collection of segment numbers (i.e., values that can be used as indices into the segment table to select segment descriptors) for all the segments it may need to access during execution (and no others). This collection describes the *access environment* of the module, and the entire collection is stored in a set of *access segments*. Access segments form the other step in the two-step mapping process.

The access mapping step works like this: The segment selector part of a logical address is used as an index into an access segment to select one of the segment numbers. The segment number itself then acts as an index into the segment table to select a segment descriptor. This two-level process is illustrated in figure 2-8. The segment numbers are actually contained in thirty-two bit entities called *access descriptors*, which also contain access rights data.

The access environment is defined by a set of four access segments that are associated with this particular invocation of the program module (see section 4.4.2, *Context Objects*). Since the access environment restricts the logical address space available to the module, we can think of the access environment as defining the *effective address space*. Figure 2-9 illustrates the relationship among the logical address space, the effective address space, and the physical address space.

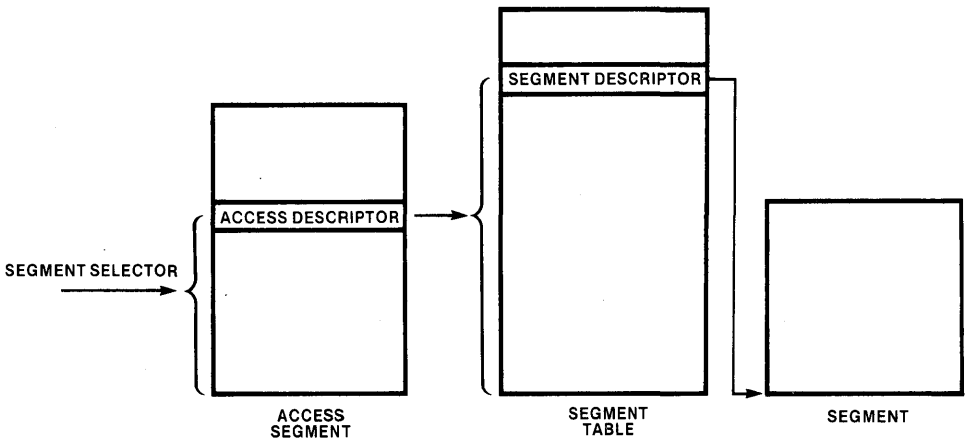


Figure 2-8. Two-Level Mapping 171821-13

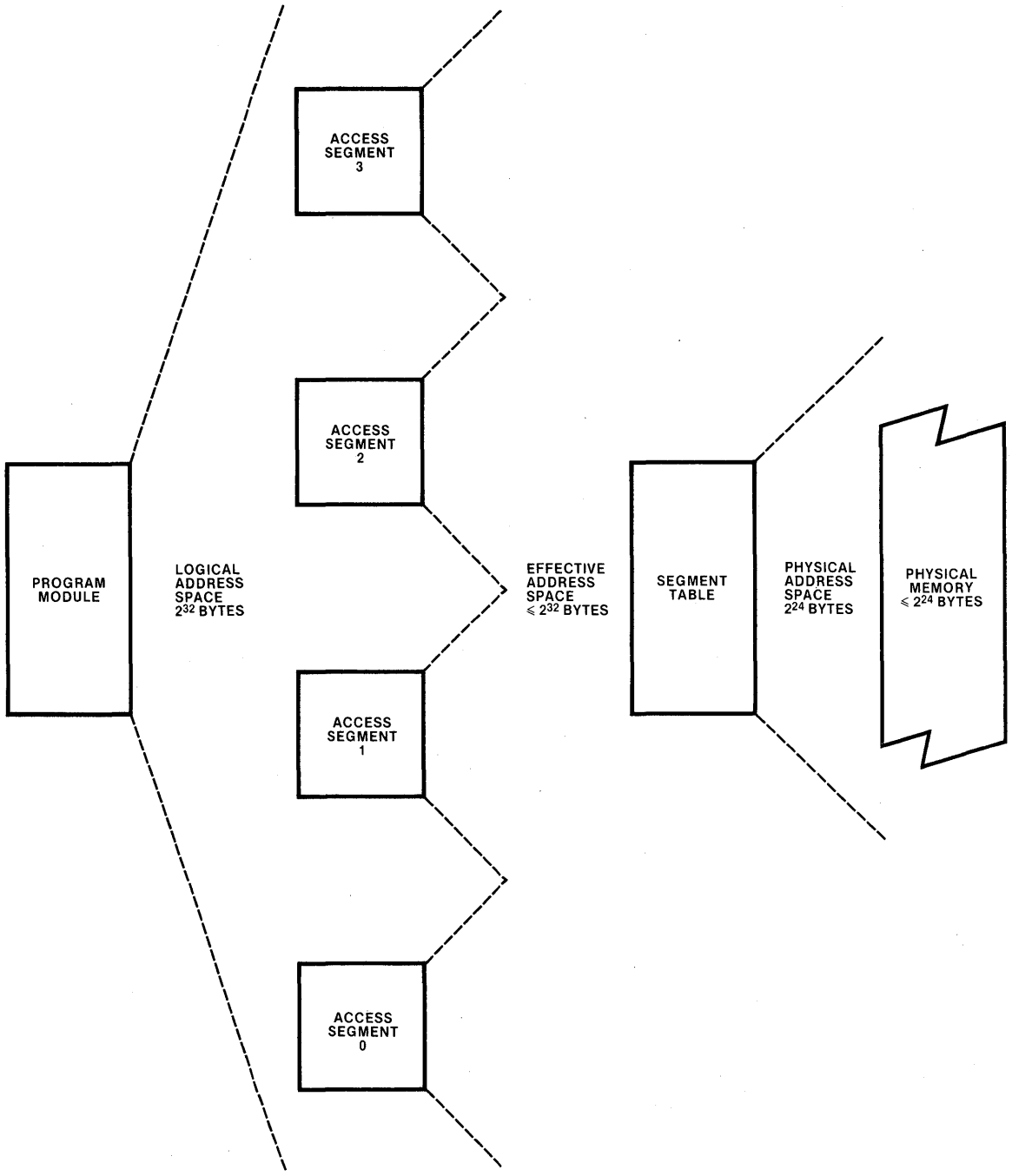


Figure 2-9. iAPX 432 Address Spaces 171821-14

The two-stage mapping process has three major advantages over a one-step map:

First, it takes fewer bits in an address to specify a particular segment. Sixteen million segments mean 16 million (2^{24}) entries in the segment table, so an index into the segment table (the segment number) must be 24 bits wide. This is far too many bits to carry around in every logical address. But only those segments in the access environment can be referenced. And, since the number of segments in the access environment will generally be much smaller than the total number of segments in memory, fewer bits are needed to specify one of them. In fact, the iAPX 432 can use as few as four bits to specify a segment.

Second, the two-step mapping makes it possible to restrict the number of segments accessible by a given program or program module. Whereas, as we indicated in the previous section, the one-step mapping allows any program to address any segment in memory, simply by indexing through the segment table.

Third, as we shall see in the next section, it allows the separation of access rights information from the segment descriptors. Access rights are instead contained in the access segments and are thus associated with program modules.

One potential problem with a two-step mapping is access time. A memory access requires both an access descriptor and a segment descriptor, each of which may have to be fetched from memory. The iAPX 432 avoids this problem by maintaining an internal associative cache to speed up the address translation process. The most recently used segment descriptors, access descriptors, and the addresses of a number of commonly accessed items (e.g., the segment table) are all stored on the chip. The result is that most accesses go directly to the addressed item.

2.4.2 Segment Types and Access Rights

In the iAPX 432, as in all computers with segmented memory, all information is contained in segments. It is apparent that if the processor could tell what *type* of information each segment contained, it could provide a powerful protection mechanism. For example, if it were known that a particular segment contained access descriptors, and a program inadvertently tried to execute the contents of the segment as code, the error could be detected, and corrective action taken before any damage was done.

Several different segment types are recognized by the iAPX 432 hardware. The two fundamental types (called the two *base types*) are data segments and access segments. Data segments are used to contain both instructions and operands (i.e., everything but access descriptors). Access segments contain access descriptors only. Type information is contained in the segment descriptors and is checked whenever an attempt is made to access a segment. If the access is not allowed for that type, a fault occurs.

The hardware maintains an absolute distinction between access segments and data segments. The contents of access segments may not be manipulated in any way by instructions which operate on data segments. Access segments have their own set of instructions which manipulate access descriptors.

Each base type is subdivided into several hardware-recognized *system types*. For example, data segments can be subcategorized as instruction segments or stack segments, to name only two possibilities. The hardware checks for inappropriate accesses to system types as well as to base types. Thus, an attempt to execute the contents of a stack segment can be detected and aborted.

The access segment is the appropriate place to put access rights information, not the segment table (compare 2.3.2), since these rights should be associated with a program module, not directly with a segment. Since access rights are stored independently of the segment descriptors, several modules can share the same segment, each with a different access right to it. See figure 2-10 for an illustration of this case.

An access rights field is part of the access descriptor. Every reference to a segment from an instruction will be checked against an access right for that segment, contained in the access descriptor. If a reference is detected that is not allowed, a fault will occur.

2.4.3 Virtual Memory and Dynamic Storage Allocation

Each program module has its own access environment, and within that environment instructions can potentially generate 2^{32} unique addresses. Since a program can dynamically modify its current access environment (for example, by calling different procedures, see section 4.4.5), the address space available to the program over the lifetime of its execution is equal to the maximum total number of segments in memory times the maximum length of each segment. This enormous figure -- 2^{40} bytes -- is called the *virtual address space*.

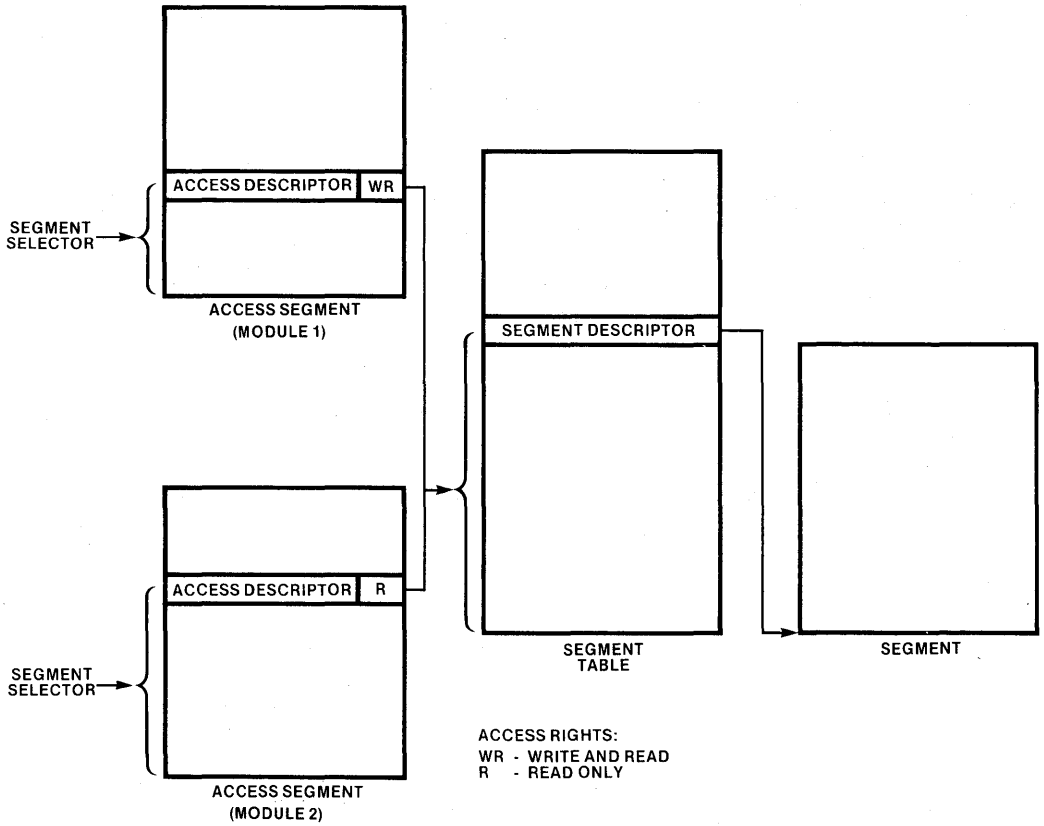


Figure 2-10. Shared Segments

171821-15

As described in section 2.2.3, virtual memory is a way of implementing a large logical address space with a much smaller physical space. Since no iAPX 432 user will attach 1 trillion bytes of main memory to his system, a virtual memory mechanism must be implemented if the gigantic virtual address space is to be used. Each segment descriptor contains the following four 1-bit fields, which are maintained by the hardware and can be used by the operating system to implement virtual memory:

- The *valid* field, which indicates whether or not the segment is currently present in main memory.
- The *storage allocated* field, which indicates whether any memory has been associated with this descriptor.
- The *accessed* field, which indicates whether this segment has been accessed by some executing process.
- The *altered* field, which indicates whether or not the information contained in the segment has been modified by some executing process.

The operating system can use the *valid* bit and the *storage allocated* bit to detect when a physical segment is not present in memory, and it can use the *accessed* and *altered* bits to decide which of the currently present segments should be swapped out or simply overwritten by the new segment. In addition, several fields in the segment descriptor can be used by the operating system to record other useful information about the segment (e.g., frequency of use) that can also be used in the swapping algorithm.

The amount of memory allocated to a module is not necessarily fixed at compile time; it may be changed dynamically. The mechanism for implementing dynamic storage allocation is described in section 4-6. The operating system can use this dynamic storage allocation mechanism to make virtual memory more efficient by only allocating segments as they are needed, thus preventing unreferenced segments from using up memory. The user can also make use of the dynamic storage allocation mechanism to reserve storage as needed.

2.4.4 Complexes of Segments

The segment table contains entries for access segments as well as data segments, so the ultimate target address specified by an access descriptor can be another access segment. A logical address in an instruction therefore can reference an access descriptor as well as an item of data. This access descriptor will in turn point to another segment, which, of course, can also be an access segment. The journey from the initial access segment, through all the intervening access segments, to the final data segment is called an *access path*.

By following these access paths, we can move through elaborate complexes of segments, in which access segments point to other access segments, and so on. To conveniently diagram these segment complexes, the two-step mapping process will be symbolized by a single arrow, which will be called an *object reference* (see figure 2-11). Figure 2-11 also shows a large segment complex.

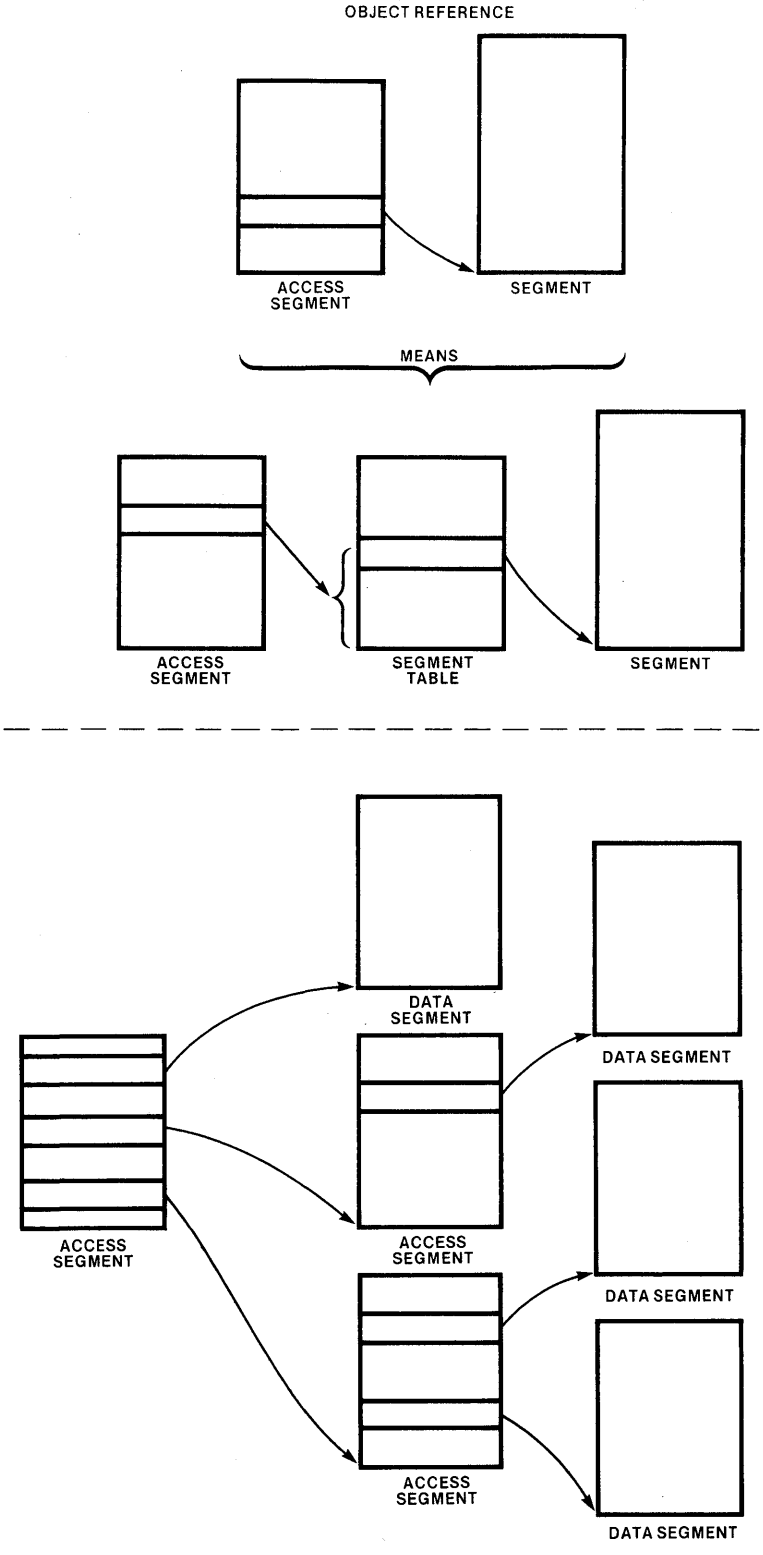


Figure 2-11. A Complex of Segments 171821-16

A complex of segments can be conceptually viewed as a single entity called an *object*. The iAPX 432 hardware recognizes a number of these objects and can manipulate them with instructions. Objects can consist of a single segment, a complex of segments, or a contiguous subsection of a segment (called a *refinement* of the segment). Objects are used to support the advanced features of the iAPX 432 architecture that are described in Chapter 4.

Viewing an object as a single entity, we can consider the segment descriptor that points to the root segment of the object as a descriptor for the whole object. We will often refer to such segment descriptors as *object descriptors* and refer to the segment table as the *object table*. The object table actually consists of several data segments. See Chapter 2 of the *iAPX 432 General Data Processor Architecture Reference Manual* for more details.

2.5 Summary

The iAPX 432 memory architecture has the following features:

- Two-step mapping process which separates the relocation mechanism from the access control mechanism.
- A unique access environment for each invocation of a program module.
- 2^{40}-byte virtual address space with mechanisms for implementing virtual memory.
- Two basic types of segments: access segments and data segments; they are recognized by the hardware and the distinction between them is rigorously enforced.
- Access rights associated with program modules not with segments.
- Mechanisms for constructing and accessing complicated data structures consisting of several segments.

CHAPTER 3
DATA MANIPULATION

3.1 What are Data Types?

All information in a computer's memory is stored as a pattern of ones and zeros. What these bits represent depends upon the interpretation given to them by the computer. The same bit pattern that encodes an ADD instruction might represent the number 17562 when interpreted as an integer. All computers have some basic data representations that are recognized by the hardware. These basic representations are called hardware-recognized *data types*.

Hardware-recognized data types have two additional characteristics. First, each instance of a type (e.g. each integer) can be addressed *as a unit* in memory. Even if the datum is several bytes long, it has a single address. Second, associated with each type is a set of operations that can be performed on the type. The instruction set of a computer is basically just the combination of all the hardware recognized operators with all the data types that can act as operands.

Early computers (and first generation microprocessors) had very simple instruction sets, with only a few operators and a few hardware-recognized data types. The arithmetic instructions were usually limited to addition and subtraction of integers. All other data types (e.g. real numbers) were manipulated by using special subroutines. Subroutines also had to be written to perform more complex operations (e.g. multiplication and division).

For instance, to multiply two real numbers on a computer lacking hardware multiplication and real data types, a number of steps had to be taken. A binary pattern for real numbers had to be devised that could be stored in the computer's memory and could adequately represent the range and accuracy desired. Second, a software subroutine had to be written that could perform the multiplication and store the result in memory. The software routine had to accomplish the multiplication using only simple operations, such as additions and subtractions of integers.

As the cost of hardware decreased, it became feasible to expand the instruction set to include operations and data types that had previously been handled by the software. Typically, real numbers, characters, and bits were added to the data types supported by the instruction set. The net result was that overall performance was increased, especially for data-type-intensive applications (e.g. scientific number crunching) and less memory was required, since whole software routines could be replaced by single instructions.

Ideally, any new hardware-supported data type should have been supported by a complete set of operators. For example, if the Add operation existed, Subtract should also have existed, and usually Multiply and Divide. Unfortunately, this ideal was not always met. Operators were often added in an *ad hoc* manner, and all the relevant operators were not supplied for each data type. The results were instruction sets that were very difficult to use.

3.2 iAPX 432 Operators and Primitive Data Types

The iAPX 432 provides a comprehensive set of operators for manipulating several different hardware recognized data types. We will call them "primitive" data types because they are used to construct more complex data structures. All required operators are available for eight primitive data types, which may be divided into four classes: character, ordinal, integer, and real. Figure 3-1 shows the formats of each of the eight types.

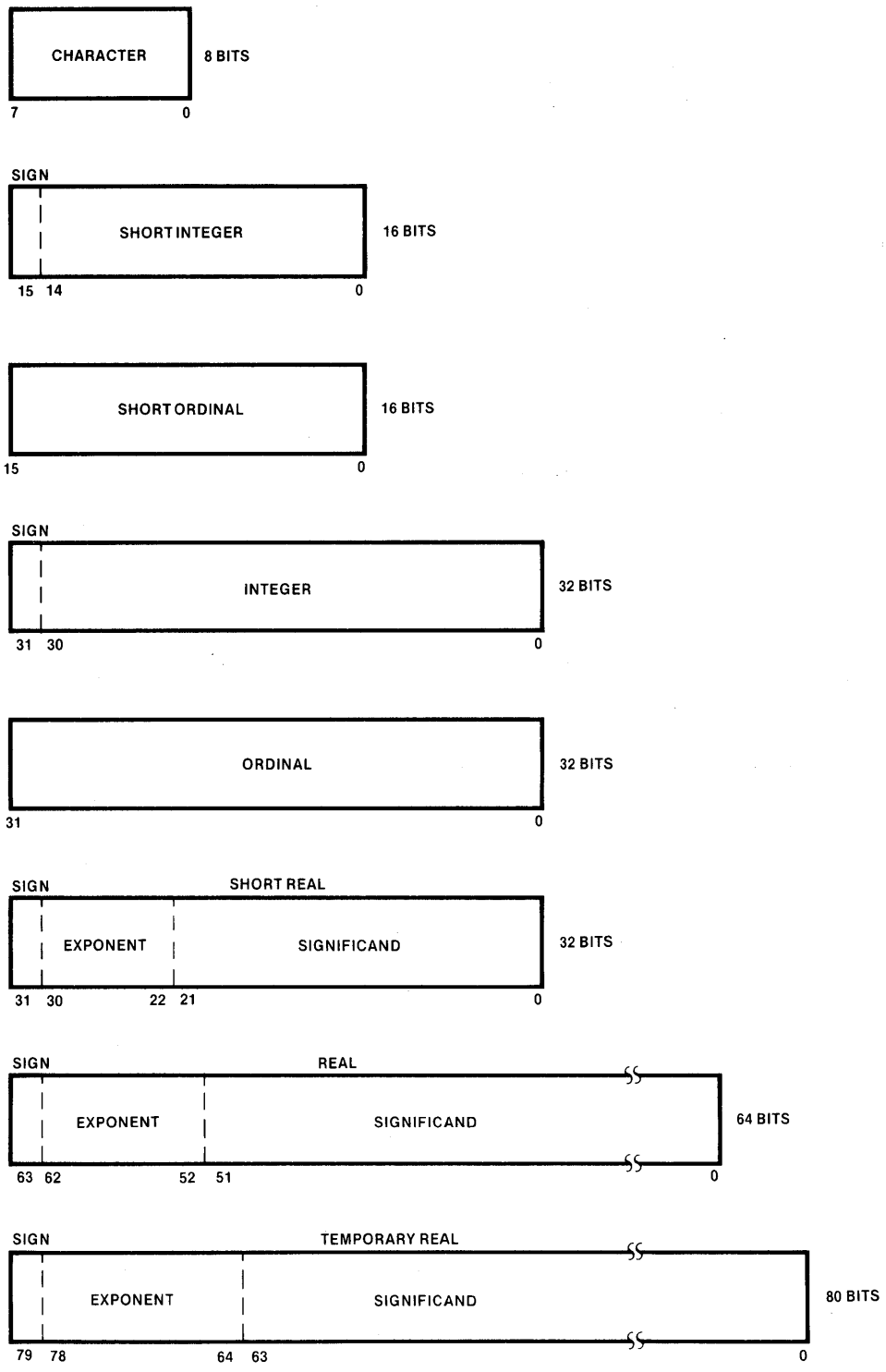


Figure 3-1. Primitive Data Types 171821-17

The operators for these data types can be divided into several broad groups: arithmetic operators (e.g. Add, Subtract, Multiply, Divide), logical operators (e.g. AND, OR, NOT, XOR), relational operators (e.g. Equals, Greater Than, Less Than), conversion operators (e.g. Convert to Character, Convert to Integer), move operators (e.g. Move, Zero, Save), and bit-field manipulation operators (e.g. Extract Bit Field, Insert Bit Field). Table 3-1 provides a chart showing all hardware recognized data types and all the operators that can be used with them. The sections that follow describe briefly the four data type classes and the operators that go with each class.

Table 3-1. iAPX 432 Operators and Data Types

	OPERATOR	CHARACTER	SHORT ORDINAL	ORDINAL	SHORT INTEGER	INTEGER	SHORT REAL	REAL	TEMPORARY REAL
MOVE OPERATORS	MOVE	X	X	X	X	X	X	X	X
	SAVE	X	X	X	X	X	X	X	X
	ZERO	X	X	X	X	X	X	X	X
	ONE	X	X	X	X	X			
LOGICAL OPERATORS	AND	X	X	X	—	—	—	—	—
	OR	X	X	X	—	—	—	—	—
	XOR & XNOR	X	X	X	—	—	—	—	—
	COMPLEMENT	X	X	X	—	—	—	—	—
ARITHMETIC OPERATORS	ADD	X	X	X	X	X	*	*	X
	SUBTRACT	X	X	X	X	X	*	*	X
	MULTIPLY		X	X	X	X	*	*	X
	DIVIDE		X	X	X	X	*	*	X
	REMAINDER		X	X	X	X			X
	INCREMENT	X	X	X	X	X	—	—	—
	DECREMENT	X	X	X	X	X	—	—	—
	NEGATE	—	—	—	X	X	X	X	X
	ABSOLUTE VALUE	—	—	—			X	X	X
	SQUARE ROOT								X
BIT FIELD OPERATORS	EXTRACT		X	X	—	—	—	—	—
	INSERT		X	X	—	—	—	—	—
	SIGNIFICANT BIT		X	X	—	—	—	—	—
RELATIONAL OPERATORS	EQUAL	X	X	X	X	X	X	X	X
	NOT EQUAL	X	X	X	X	X			
	EQUAL ZERO	X	X	X	X	X	X	X	X
	NOT EQUAL ZERO	X	X	X	X	X			
	GREATER THAN	X	X	X	X	X	X	X	X
	GREATER THAN OR EQUAL	X	X	X	X	X	X	X	X
	POSITIVE	—	—	—	X	X	X	X	X
	NEGATIVE	—	—	—	X	X	X	X	X
CONVERSION OPERATORS	CONVERT TO: CHARACTER	—	X						
	SHORT ORDINAL	X	—	X					
	ORDINAL		X	—		X			X
	SHORT INTEGER				—	X			
	INTEGER			X	X	—			X
	SHORT REAL						—		X
	REAL							—	X
	TEMPORARY REAL		X	X	X	X	X	X	—

Key

X = This operator is available for the given data type.

* = This operator is available for the given data type *and* for operations where one of the operands is a temporary real.

— = This operator is not available and would not be useful if it were.

(blank) = This operator is not available.

3.2.1 Characters

Characters require one byte of memory and can represent three kinds of data: ASCII characters, unsigned integers in the range 0-255, and the boolean values TRUE or FALSE (which are stored as xxxxxxx1 and xxxxxxx0, where x means either 1 or 0). Character operators include move operators, logical operators, simple arithmetic operators (add, subtract, increment, and decrement), relational operators, and the operator Convert Character to Short Ordinal.

3.2.2 Ordinals

Ordinals are unsigned integers; they are availiable in two sizes: 16-bit (Short Ordinals) and 32-bit (Ordinals). They have values in the ranges 0 to 2^{16}-1 or 0 to 2^{32}-1. Ordinals are commonly used as indices into vectors and arrays and to hold bit fields. Ordinal operators include moves, arithmetic operators (including multiplication and division), logical operators, operators for manipulating bit fields, relational operators, and several conversion operators.

3.2.3 Integers

Integers are available in two sizes: 16-bit (Short Integers) and 32-bit (Integers). They have values in the ranges -2^{15} to 2^{15}-1 or -2^{31} to 2^{31}-1. Integer operators include moves, arithmetic operators, relational operators, and several conversion operators.

3.2.4 Reals

Real numbers have three components: an exponent, a significand (mantissa), and a sign bit. Reals are available in three sizes: 32-bit (Short Reals), 64-bit (Reals), and 80-bit (Temporary Reals). Each increase in size provides more precision (larger significand) and greater range (larger exponent). Short Reals and Reals are typically used as input and final-result operands in floating-point calculations. Temporary reals are intended for use as intermediate results, thus preserving accuracy and reducing the risk of overflow or underflow in multi-step calculations. Real operators include moves, arithmetic operators, relational operators, and several conversion operators. Table 3-2 lists the range and precision of the three sizes of reals.

Table 3-2. Range and Precision of Short Real, Real, and Temporary Real

Type	Range	Precision
Short Reals	2^{-126}—2^{127}	7 decimal digits (approx)
Reals	2^{-1022}—2^{1023}	15 decimal digits (approx)
Temporary Reals	2^{-16383}—2^{16383}	19 decimal digits (approx)

3.3 Structured Data Types

The data types discussed in section 3.1 are called *primitives*. By contrast, we will use the term *structured data types* for ordered aggregates of primitives. The iAPX 432 facilitates access to two structured data types that are commonly used in high-level languages:

- **Arrays.** An array consists of a number of components, each of the same data type. Thus we speak of arrays of integers, arrays of characters, etc.
- **Records.** A record consists of a number of components (usually called fields), that can be of different data types. Thus, a record might consist of characters, integers, and real numbers. (For instance, an employee record.)

These structured types are not supported by a set of hardware operations, but the iAPX 432 does provide a mechanism that allows structured types to be manipulated easily. Each of the primitive types may be accessed through several *addressing modes*, and these facilitate the selection of individual elements from arrays and records (see section 3.4.1). The addressing mode used to reference an operand is determined by the way the logical address is formed in the instruction that references it.

3.4 Instructions

The iAPX 432 instructions specify the operator and the operands that it acts on. Up to three operands may be required for some operators. Instructions are contained in special hardware-recognized *instruction segments* in memory (see section 2.3). The processor views an instruction segment as a continuous string of bits called the instruction stream. Instructions may contain a variable number of bits and may begin at any bit within the stream. (In the current implementation, the processor reads an instruction segment in units of 32 bits.) The general instruction format of the iAPX 432 consists of four fields arranged in the format shown in figure 3-2.

The first two instruction fields encountered by the processor in the instruction stream are called the *class* field and the *format* field. These two fields specify how many operands are in the instruction and information on how they are to be accessed. The last two fields are the *reference* field and the *operator* field. The operator field specifies the operator (e.g. ADD, SUBTRACT, NEGATE). The reference field contains the logical addresses of up to three operands. As shown in figure 2-7 of section 2.3, each logical address has two parts, a segment selector and a displacement. The segment selector identifies the segment that contains the operand, while the displacement locates the operand within the segment.

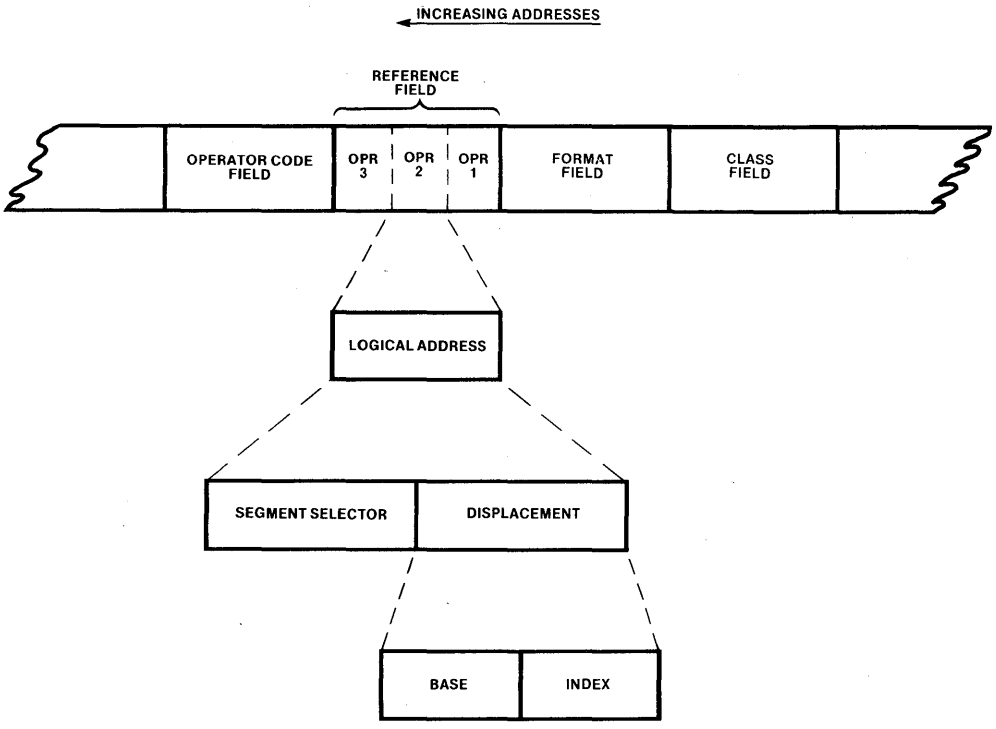


Figure 3-2. iAPX 432 Instruction Format (3 operands) 171821-18

3.4.1 Addressing Modes

The displacement really consists of two subcomponents: a base value and an index value. (See figure 3-2.) Each subcomponent can reference its value either *directly* (the subcomponent contains the value itself) or *indirectly* (the subcomponent contains a pointer to the value, which is located elsewhere in memory). A direct reference by both subcomponents is equivalent to a single-component displacement and can be used to access non-structured data (scalars). Indirect references can be combined with direct references to easily access three structured data types. The four combinations of direct and indirect reference are called *addressing modes*:

- Base and Index Direct — used to access *scalars*
- Base Indirect, Index Direct — used to access *records*
- Base Direct, Index Indirect — used to access *static arrays* (arrays whose starting location, the base, is established at compile time)
- Base and Index Indirect — used to access *dynamic arrays* (arrays whose base can be established during execution)

Figure 3-3 illustrates the way the logical address is formed in each of the four addressing modes and the structured data type that is accessed by each mode.

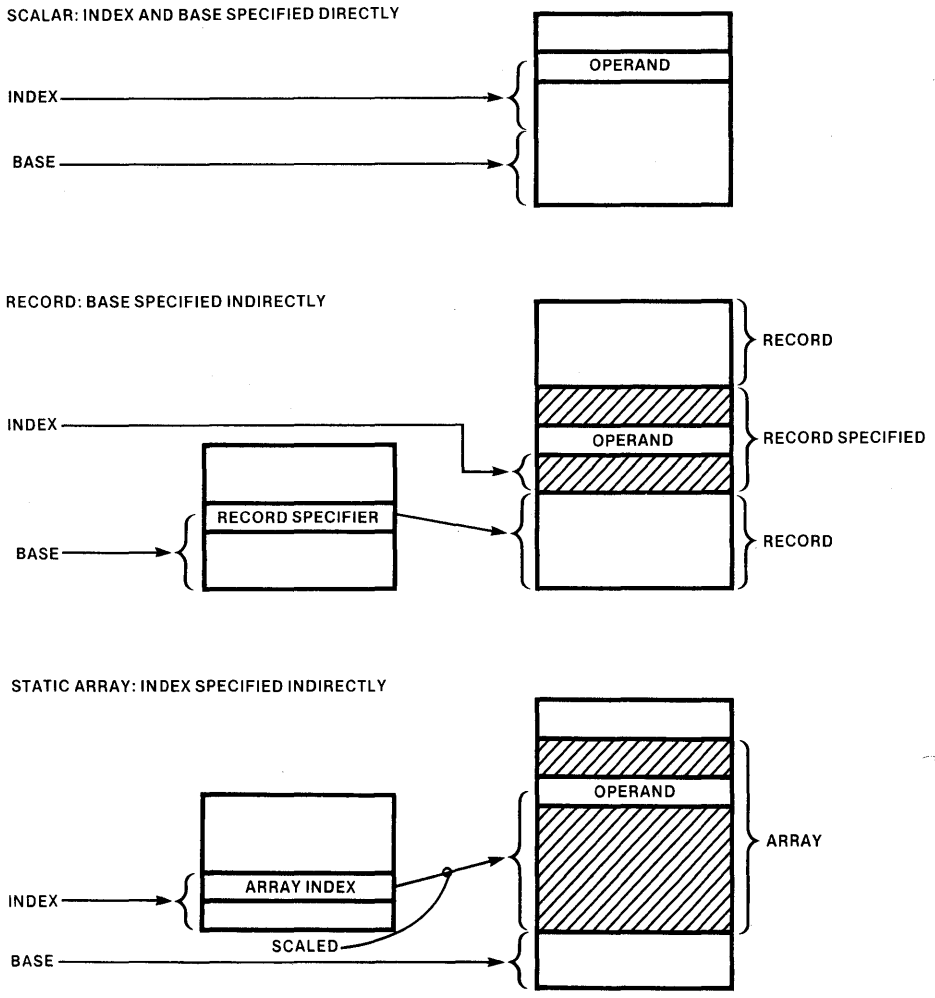


Figure 3-3. Addressing Modes and Structured Data Types (1 of 2) 171821-19

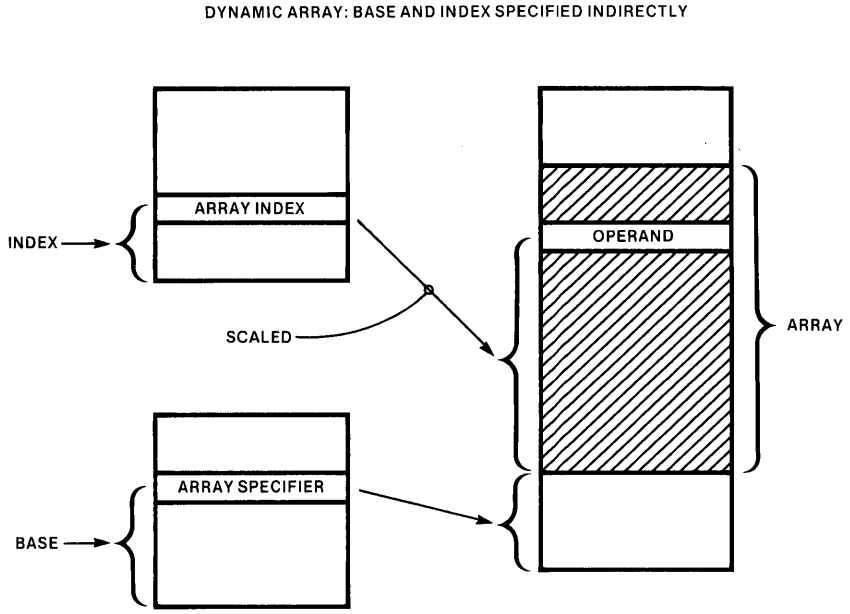


Figure 3-3. Addressing Modes and Structured Data Types (2 of 2) 171821-20

The processor automatically *scales* an index value by multiplying it by 1, 2, 4, 8, or 16 depending on whether the operand it points to occupies a byte, double byte, word, double word, or extended word. The compiler is thus freed from having to perform this calculation.

The segment selector component of the logical address can also be specified indirectly. This additional capability means that large, multi-segment arrays can be implemented easily.

It is important to stress that not only are all required operators available for every data type, but also all four addressing modes are available for any operand in an instruction. This means that iAPX 432 instruction set is completely *symmetric*. Symmetry (sometimes called regularity) is defined essentially as the degree to which all addressing modes exist for every operand and all required operators exist for every data type in the instruction set.

Symmetry is especially important for easy translation of high-level languages. If the instruction set is not symmetric (some operators are not provided for some data types, or some addressing modes are not available with some operators) then the compiler writer is faced with many special cases where software has to make up for the holes in the instruction set. Thus, the compiler is more difficult to write and more complicated.

The symmetry of the iAPX 432 instruction set and the powerful operators allow the efficient encoding of high level language statements. For example, each of the following statements is encoded by a single iAPX 432 instruction:

A = B * C	— A three-address multiply, all scalars
A [I] = B [J] / C [K]	— A three-address divide, all static array elements
P.Q = A [I] * C	— Three addresses, mixed type, "element Q of record P is assigned the product of static array A, element I, and scalar C"

3.4.2 Operand Stack

When an instruction references an operand, it can explicitly specify a logical address, in the manner described in the previous section, or it can access the top of the operand stack. The operand stack is a special data segment that is maintained by the hardware for expression evaluation. An access to an operand on the stack is called an implicit reference. Items are added to or removed from the "top" of the stack on a last-in-first-out basis. The current stack top is pointed to by a hardware-maintained stack pointer. The following arithmetic expressions, which can be performed by single iAPX 432 instructions, illustrate the flexibility of stack usage (the symbol "$" signifies that the operand is on the stack):

A = $ + B	— Add the value on the top of the stack to the value in B, then put the result in A
$ = A * $	— Multiply the value in A by the value on the top of the stack, then push the result on the stack
$ = $ + $	— Add the top two items on the stack and leave the result at the top of the stack

See figure 3-4 for an illustration of a case where the last example would occur.

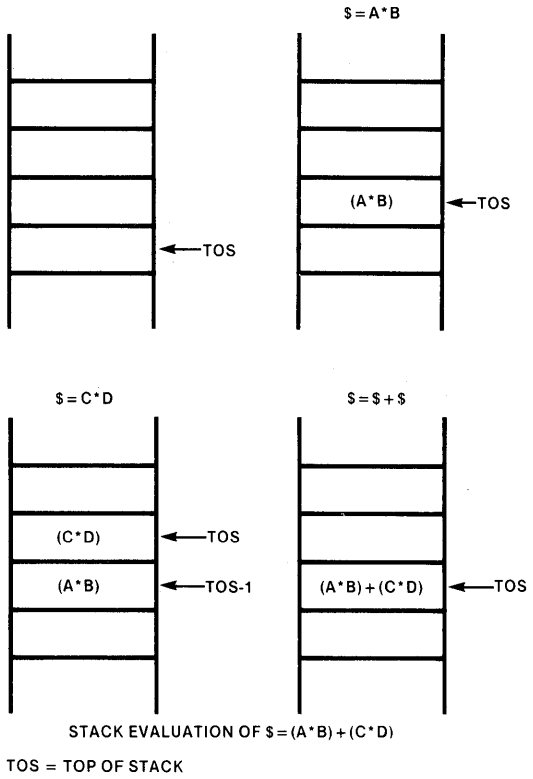


Figure 3-4. Using the Operand Stack 171821-21

Using implicit stack references instead of explicit memory references to address operands allows shorter and faster instructions. The iAPX 432, however, allows both stack arithmetic and memory-to-memory arithmetic. Research has shown that pure stack-oriented machines (i.e. machines where all arithmetic operations are between the top two elements of the stack) are not as fast at evaluating expressions, nor do they have as dense a code, as machines which allow both types of arithmetic.* But even a pure stack machine is faster and more efficient at evaluating expressions than a machine with no stack.

3.4.3 Instruction Encoding

From studies of program behavior, it is known that certain instructions are used more often than others. For instance, a push is far more frequently executed than a halt. Therefore, it makes sense to distribute the binary encoding of instructions in such a way that frequently used instructions take fewer bits to specify than infrequently occurring instructions. Most machines, however, have an instruction size that is always some multiple of the basic unit of memory storage (frequently 8 or 16 bits). This fixed-length instruction encoding tends to be inefficient and frequently forces unfavorable compromises in the design of the instruction set. Since a large percentage of a computer's time is used in fetching instructions, an improvement in instruction bit density would result in faster execution of programs and a decrease in program storage requirements.

The iAPX 432 instruction set has been designed to be very compact; the instructions are bit-variable in length and are not constrained to start or end on byte or word boundaries. If an operand reference is used more than once in an instruction, it need be specified only once. For example, in the common expression

$$A = A + B$$

the A operand need be specified only once. The format field in the instruction indicates how many times a particular operand is used.

The bit encodings were based on the frequency-of-use of operators and operand references. The shortest instruction is 6 bits long and the longest is 344 bits long.

3.5 Summary

The iAPX 432 data manipulation instruction set

- supports all required operators for 8 primitive data types
- has four addressing modes for every operand
- allows explicit or implicit access to every operand
- supports efficient coding of high-level language statements
- allows easy access to elements in arrays and records
- has a frequency-of-use bit encoding that provides compact code

*Glenford J. Myers, *Advances in Computer Architecture*, (New York: 1978).

CHAPTER 4
PROGRAMMING ENVIRONMENT SUPPORT

4.1 The Software Crisis

In his 1972 Turing Award lecture, E. W. Dijkstra, father of the "structured programming" movement, described the software crisis in the following terms:

> As long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem; and now that we have gigantic computers, programming has become an equally gigantic problem . . . . The increased power of the hardware . . . made solutions feasible that the programmer had not dared to dream about a few years before. And now, a few years later, he *had* to dream about them and, even worse, he had to transform such dreams into reality!

Decreasing hardware costs and increasing power led to more ambitious programming projects. But it proved difficult to make the programs reliable, and even more difficult to make the programming projects come in on schedule and within budget. There were four fundamental problems: the sheer complexity of many large systems, the increasing demand for system security, the increasing use of concurrent programming, and finally the demand for systems whose processing power can be expanded without affecting the existing software.

4.1.1 Modularity

Modern computer applications often require hundreds of thousands of lines of code; some data processing installations use systems that represent several million lines. Such enormous software packages are far too complicted for one person to understand completely; if they are to be developed and maintained, some form of program modularization must be used. It should be possible for different modules to be developed by different people, relatively independently, and for bugs to be fixed in one module without affecting other modules in unpredictable ways. It would also be desirable to be able to add functions by simply incorporating additional modules, without having to change the existing program.

Even though modularization seems like an easy idea to grasp, it has proved surprisingly difficult to implement correctly. A program can be modularized in a number of different ways, and some of these ways make the programming job even more difficult than if no modules were used. Consider, as an analogy, a diamond cutter. His job is to "modularize" an uncut diamond. If he chooses the correct modularization, he will create valuable gems. But if he makes a mistake, he will get a pile of dust.

The basic problem in modularization is to define correctly the interface between modules. Under many modularization schemes, some modules need to know the internal implementation details of other modules in order to operate. What is needed is a set of criteria for decomposing programs into modules in a way that minimizes the need for outside knowledge of the internal representations.

4.1.2 Security

Modern society has entrusted an enormous amount of sensitive data to computer systems, from bank accounts, credit records, and medical histories to the specifications for hydrogen bombs. Most of this information is, unfortunately, not very

securely stored. No computer system has ever successfully resisted determined efforts to defeat its security mechanisms, and many commercially available systems barely put up a fight.

Security is based on limiting access to information. The basic principle is "need to know"; users, programs, and modules within programs should have access to only that information which they absolutely need. Most modern operating systems, however, have a very crude access limitation scheme. Usually a distinction is made between a few privilege levels, and programs in the same level can do the same things. This scheme almost always results in programs being given too many privileges, since they cannot operate if they are given too few. What is needed is a method that grants each module exactly those privileges that it needs to execute properly, and no more.

4.1.3 Concurrency

As computers take on more complex tasks, they are called upon to model more and more features of the real world. One fundamental fact about the real world is that actions generally happen concurrently rather than in strict sequence. For example, in industrial control applications several jobs are usually being performed at the same time, and critical developments in one job can interrupt the work in another. Consequently, software designers have developed models of concurrent processing in order to handle these real-world situations. Concurrent processing is also often used to multiplex several programs on one processor, so that shared resources, such as memory, I/O, and the processor itself, can be more efficiently used.

Implementing concurrency requires mechanisms that permit the sharing of common resources and also mechanisms for bringing together concurrently executing modules to exchange information (communication) or to coordinate their action (synchronization). Several problems have made concurrent systems difficult to implement. The basic problem is defining the unit of concurrency. The problem is somewhat similar to the modularization problem described in section 4.1.1, except here the problem is how to modularize in the time domain. What is needed are criteria for decomposing programs into modules that can be run concurrently, and also mechanisms for communication, synchronization, and resource sharing.

4.1.4 Expandability

Project managers have always found it difficult to predict accurately just how much memory and processing power their project will require. As the project evolves, requirements tend to change (usually upward). Furthermore, over the lifetime of the product new features and functions are often added, placing more demands on power and memory. The standard approach computer manufacturers have taken to address this well-known phenomenon is to develop a family of computer systems that have a range of processing power.

Unfortunately, such a family takes years to develop, and the procesor with exactly the right performance is seldom available when it is needed. The user has to buy either too much or too little. Moreover, the operating system software is often different on different members of the family. What users need is a system that can be quickly and easily expanded in power without requiring any software changes.

The low cost of microcomputers has always presented the seductive possibility that computers might someday be constructed out of a large number of identical microprocessor components. If a system needs more power, just add another microprocessor. But the software difficulties of such multiprocessor systems have been daunting. The basic problems have been in sharing common resources and in

defining a unit of work in such a way that it could be sent to an arbitrary processor. This unit of work needs to be totally independent of particular processors and the communication between units of work has to be handled in an entirely processor-independent manner.

4.2 New Software Methodologies

In recent years, progress has been made in all four problem areas: modularity, security, concurrency, and expandability. By the late 1970s computer scientists involved in programming language and operating systems research believed that, in fact, solutions had been found to many of these problems.

4.2.1 Type Managers

A consensus seems to have emerged that the best way to modularize programs is by applying the principle of "information hiding." The term is due to D. L. Parnas, who has identified a number of criteria for correctly decomposing programs into modules. Another name for the principle is "data encapsulation." The basic idea is that a module should contain a collection of related procedures and the data structures they operate on. Procedures outside the module should not have access to the implementation details inside the module. The data can be referenced from outside the module only by a call to one of the procedures inside the module. Many names have been given to the modules that result when this principle is applied—abstract data types, extended types, Parnas modules—but in this document the term *type manager* will be used exclusively.

Until recently, the full power of the concept could not be realized, since most programming languages did not provide any direct support for the type manager construction. Users had to build their own type manager structures on top of the language, thus no compile-time checking facilities or enforcement mechanisms existed for type managers. Moreover, since no standard format for type managers existed, there was not necessarily any compatibility among different users' representations of the type manager concept, even if they used the same language.

Several modern programming languages, however, support the type manager construction (or closely related concepts): In Intel's Object Programming Language (OPL) and in Concurrent Pascal the structure is called a "class," CLU uses the term "cluster," Alphard calls its version a "form," and the new Department of Defense standard language, Ada, uses the term "package."

At the same time that programming language research was focusing on the concept of a type manager as the solution to the modularization problem, operating system research was defining a very similar structure, the *protection domain*, as the solution to the security problem. (We will treat protection domain and type-manager as synonyms.) Operating system researchers concentrated on the *data* in these domains, instead of the procedures (which were the focus of the language research), since they were concerned with security more than modularization. They gave the name *objects* to the data items associated with domains, and they called the whole information hiding methodology "object-oriented design methodology" or "the object model."

4.2.2 Processes

The fundamental concept that underlies concurrency is the *process*. The process is the module of concurrency, just as the type manager is the module of the static organization of a program. A program can be constructed out of a single process or

several processes that communicate with each other. The use of processes originated in multi-programming operating systems, where each job was a process. But in recent years multiprocessing has evolved into a general methodology for implementing concurrency, a methodology we will call the *process model*. In fact, just as Ada has the notion of a "package," corresponding to a type manager, it also has the concept of a "task," which corresponds to a process.

Since a process is an abstraction of a processor in action, breaking up programs into processes is a start toward multiple processor systems. In such systems, a process is a natural unit to allocate to an available processor, because a process is naturally processor independent. But before multiple processor systems can be implemented, several other concepts must be formalized and made processor independent as well. In particular, models have to be developed that abstract the details of the interprocess communication and synchronization mechanisms.

Much recent operating system research has focused on interprocess communication and synchronization, and progress here has been rapid. Software structures such as "communication ports" and "messages" have been developed to separate the logical character of the communications mechanism from the physical implementation. A communications port is an abstraction of a "mailbox," a place where messages can be sent or picked up. Processes send messages to ports and wait at ports to receive messages from other processes. Process scheduling can be handled in a similarly processor-independent fashion by defining the concept of a "dispatching port," where processes wait to be allocated to processors.

The object model and the process model are independent, but they can be combined very simply and elegantly. Processes and ports will be objects, and the procedures that manipulate these objects can be grouped into type managers. The HYDRA operating system for the C.mmp multiple-processor system (16 PDP-11 minicomputers), a research project at Carnegie-Mellon University, demonstrated just this combination of the object model with multiprocessing.*

4.2.3 Architectural Support

What does it mean to say that an architecture supports a software methodology? At a minimum, it means that the architecture simply provides mechanisms to help users follow the methodology; in this case we will say that the architecture is *oriented* toward the methodology. But in a more general sense it could mean that the design of the architecture itself also follows the principles of the methodology, in which case we will say that the architecture is *based* on the methodology.

For example, an *object-oriented* architecture helps users to write programs that use type managers and reflect the philosophy of information hiding. But if the architecture's design is also an example of the use of that philosophy, then we will say that it is an *object-based* architecture. An object-based architecture can be oriented to both the object model and the process model; it can support type managers, but it can also support multiprocessing, as the HYDRA system suggested. In fact, an object-based architecture provides a uniform approach toward supporting all system services.

If an architecture, a high-level language, and an operating system are all based on the same methodology, then the boundaries between them begin to blur. It becomes hard to tell where one begins and another leaves off. Functions can be moved from the operating system to the architecture, or from the language to the operating system. Basically, the whole system has a kind of geometrical integrity, constructed as it is out of a set of common building blocks.

*William Wulf, Roy Levin, and Samuel P. Harbison. *HYDRA/C.mmp: An Experimental Computer System* (New York: 1981).

The choice of which functions to provide in the architecture and which in system software is dependent on many factors—the size of the microcode, the number of gates on the chip, the presence of time-critical bottlenecks in certain frequently-used operations, and the need for security, among others. Management of any given type of object can be a responsibility shared by hardware and software. Time-critical functions will be placed in hardware while less frequently used or extremely complex functions will be left to software.

For example, type manager modularization results in many more intermodule control transfers than do more conventional modularization schemes. If the mechanism of these transfers results in high overhead, it will not be feasible to use this methodology. The same holds true for process scheduling and dispatching. The frequent process switches envisioned by the process model would be prohibitively expensive on a conventional architecture. These factors suggest that intermodule control transfer mechanisms and process scheduling and dispatching mechanisms should be provided by the architecture.

For reasons of flexibility, it is essential to maintain a separation of *policy* and *mechanism*. Resource management policies should be in software, while mechanisms should be in hardware. For example, scheduling policies (e.g. the choice between a round-robin system or a priority system) should be specified in software, since each application may have different scheduling requirements. But all scheduling policies require an efficient scheduling and dispatching mechanism (i.e. the ordering of available processes according to the policy, and the selection of one of these processes for execution), so this mechanism should be in hardware.

4.3 iAPX 432 Object-Based Architecture

The iAPX 432 architecture provides the mechanisms which make it feasible for users to write programs modularized into type managers and the mechanisms to support multiprocessing and other system services. The iAPX 432 can provide these mechanisms because it has an *object-based architecture*. This means that the designers of the iAPX 432 architecture themselves followed the object methodology described in section 4.2.1.

Therefore, since the object methodology is based on programming with type managers, it makes sense to ask what structures in the architecture correspond to type managers.

4.3.1 System Objects

In the iAPX 432 architecture, the structure that corresponds to a type manager is the group of all hardware operators that manipulate certain complexes of segments (see section 2.4.4). These hardware manipulated segment complexes are called *system objects*. System objects may consists of one segment, a refinement of one segment (see section 2.4.4), or several segments, but they are manipulated as a unit by their instructions.

System objects provide the architectural mechanisms that support type manager modularization and system services. For example, system objects are used to implement process scheduling and interprocess communications (in fact, an iAPX 432 process is a system object, as is a communication port and a dispatching port—see section 4.5 for more details).

System objects provide mechanisms that allow users to build their own type managers and define their own objects. Like system objects, user objects can be represented by one segment, a refinement of a segment, or an entire segment com-

plex. In fact, the only real difference between system objects and user objects is that the operations on system objects are implemented primarily in the microcode, whereas user objects are manipulated by software. User objects can be given hardware-enforced protection features that are similar to the ones associated with system objects.

For example, a user could create a type manager containing a telephone directory object and two procedures (look up telephone number, look up address) that manipulate this object. The iAPX 432 architecture offers mechanisms that make it impossible for any other procedure to access the directory object directly. The only way another procedure could access the telephone directory is by calling one of the two procedures in the type manager. The directory object is thus totally protected against accidental or malicious damage, and no procedure outside the type manager needs to know the internal representation of the directory.

4.3.2 Object Protection

The two-level mapping of the iAPX 432's structured memory (see section 2.4.1) provides the mechanisms for restricting access to objects. Recall the definition of object references in section 2.4.4 in terms of access descriptors and segment descriptors. These object references form the basis of object protection.

Object A is said to have an object reference for object B if an access segment in object A's segment complex contains an access descriptor for the root segment of B's complex. See figure 4-1 for an illustration.

The access rights mechanism of segmented addressing provides one level of object protection, the segment type mechanism provides another (see section 2.4.2), but the basic protection comes simply from controlling the dispersal of object references. The only way someone can get access to an object is by acquiring an object reference for it. If a procedure doesn't have an object reference for an object, the object

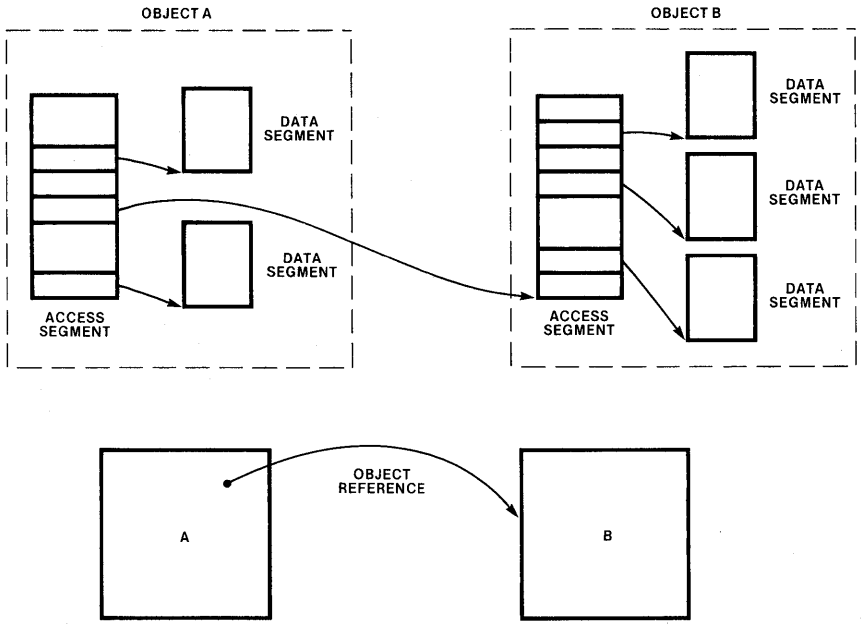


Figure 4-1. Objects and Object References 171821-22

simply doesn't exist for the procedure. In operating system research, an object reference is called a *capability*, and the whole object-oriented access mechanism is called *capability-based addressing*. Figure 4-2 shows a mulitiple-object environment and the object references of each object.

4.4 Architectural Support for Type Managers

Two aspects of the programming environment of a type manager are candidates for architectural support: its *static structure* and its *dynamic behavior* during execution. The iAPX 432 architecture provides a system object called a *domain* that implements the static structure, while the dynamic behavior is handled by a system object called a *context*.

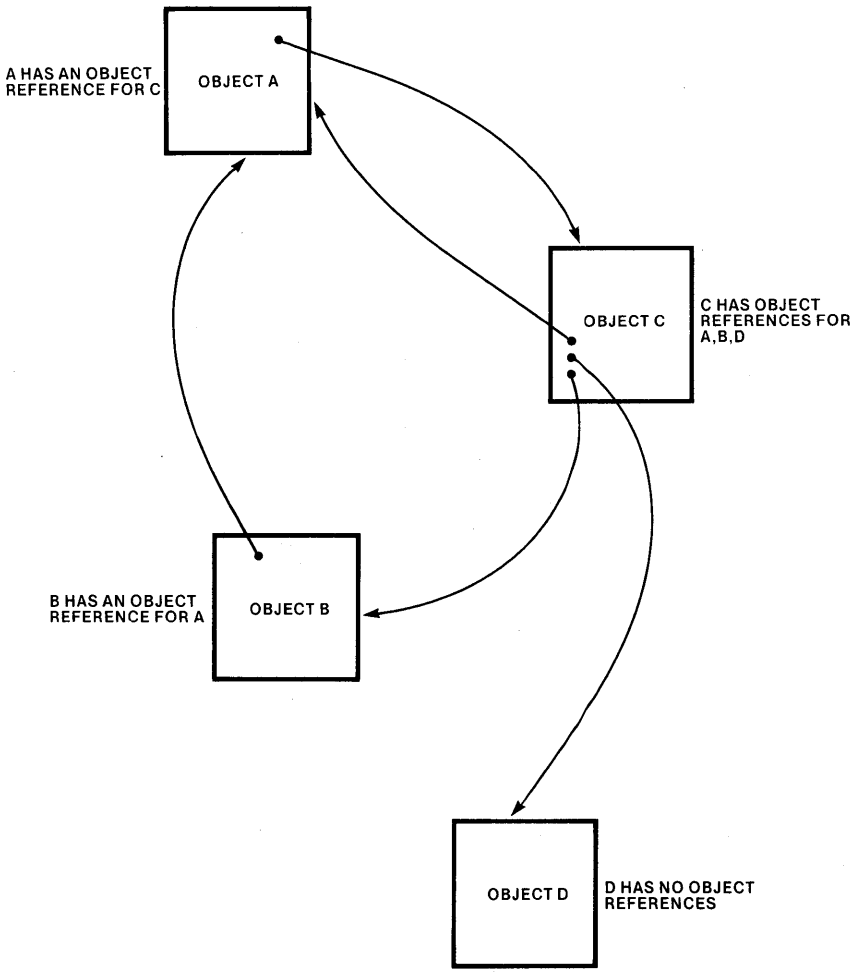


Figure 4-2. Object References 171821-23

4.4.1 Domain Objects

A type manager is represented by a segment complex called a *domain object*. The root access segment in a domain object, that is, the *domain access segment*, contains object references for all the other objects in the domain. These other objects include instruction objects (represented by instruction segments) that contain the type manager's procedures, and data objects (represented by data segments or segment complexes) containing the data that is encapsulated inside the type manager.

The domain access segment distinguishes its *public* object references from its *private* references. The public object references are contained in a refinement of the total access segment; all other references are private. Access to the list of public references will be made available to procedures outside the domain itself, but the private references are accessible only by procedures within the domain. Thus the private objects are effectively "hidden" inside the domain. This mechanism supports the information hiding philosophy of the object-oriented design methodology. Figure 4-3 shows the segments that make up a simple domain object. Notice that references to the instruction segments have been placed in the public portion of the domain access segment, while the data segment complex is hidden in the private portion.

4.4.2 Context Objects

A *context* object is the iAPX 432 hardware-recognized object that supports the run-time environment of a procedure in a type manager. Before a called procedure can be executed it must be supplied with a data structure containing run-time information that is unique to this particular instance of execution (e.g., the new instruction pointer value and the return address to the calling procedure). All this information is

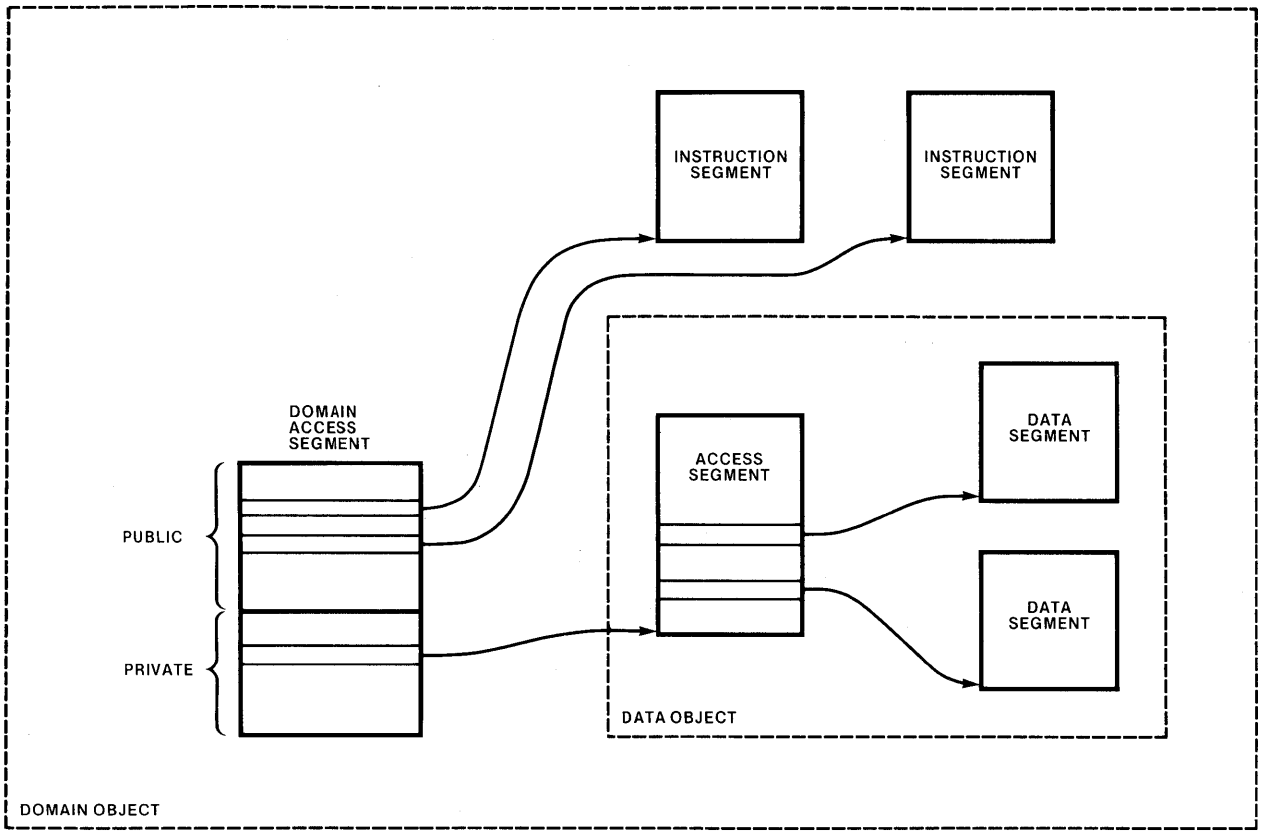


Figure 4-3. A Domain Object

171821-24

contained in a context object. It has much the same function (although in a much more protected fashion) that the *activation record* or *stack frame* has in stack-based programming language implementations.

A context object is represented by a segment complex, the root of which is called the *context access segment*. The context access segment contains object references for the following objects:

- an operand stack for expression evaluation;
- the domain access segment of the context's type manager;
- four access segments that define the instantaneous *access environment* of the called procedure (i.e. all the objects that may be referenced by the procedure);
- a message (if there is one) from the calling procedure;
- the context object of the calling procedure;
- a data segment containing local constants needed by the called procedure; and
- a data segment containing the current instruction pointer, stack pointer, and status information.

By convention, we will group into the context object the context access segment, the operand stack, the context data segments, and the four access segments that define the access environment (called the four *entry access segments*). For reasons which will become clear in the next section, one of the four entry access segments is the context access segment itself. Figure 4-4 shows the segments and object references included in a context object.

4.4.3 Calling Contexts

A new context object is automatically created when a procedure in one type manager calls a procedure in another type manager. The new context is automatically destroyed when the called procedure returns control to the calling procedure. A call instruction may reference any instruction object in the public part of any domain in the calling access environment. See figure 4-5 for an example of a call.

When a call is made, the new context has a different access environment from the calling context, so the access environment of the called procedure will be different from the access environment of the calling procedure. For example, notice that in figure 4-5 the access environment of the calling context includes only the public part of the called domain, whereas the new context's access environment includes the entire domain. Each invocation of a procedure can be given an access environment that includes only those objects it needs to access. Thus the call context mechanism helps to insure the security of the whole program.

Along with the call instructions, the iAPX 432 instruction set includes two groups of branch instructions: intrasegment branches and intersegment branches. The range of the former group is limited to a single segment, while the latter group can specify branches to any instruction segment in the domain (i.e. to any procedure in the current type manager). Branch instructions do not change the access environment.

4.4.4 Using the Inside Representation of Type Managers

The domain objects we have considered so far include both procedures and the data objects they act on. We will call this kind of domain object, in which data objects are in the domain, the *inside representation* of type managers.

The telephone directory example from section 4.3.1 can be easily implemented using the inside representation. Object references for the two procedures are placed in the public part of the domain, while the telephone directory object itself is in the private portion of the domain. Thus the procedures can be called from outside the domain, but the directory is inaccessible from outside. Whenever anyone needs to access the directory, a call must be made to one of the two procedures, which performs the operation on the directory, then returns to the caller.

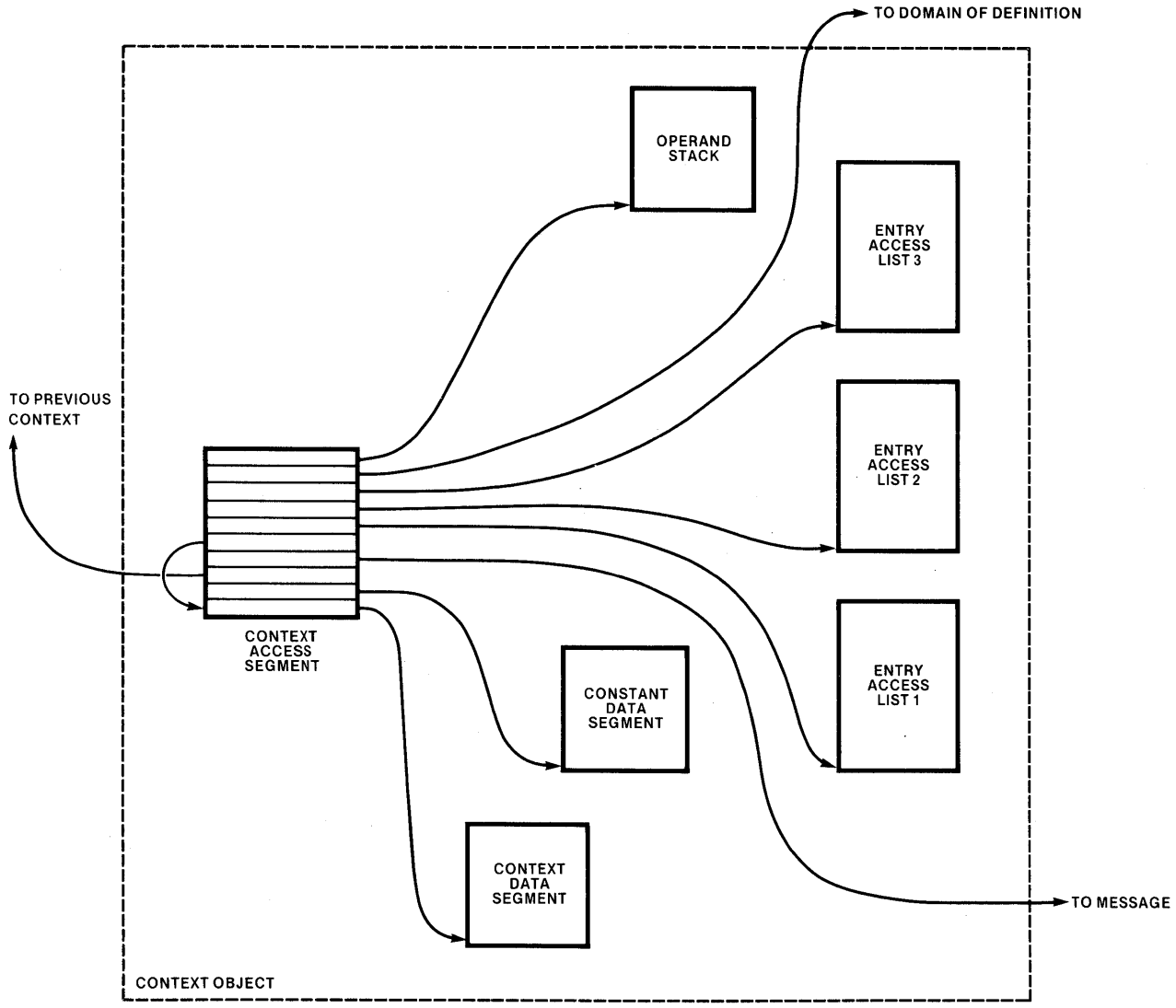


Figure 4-4. A Context Object 171821-25

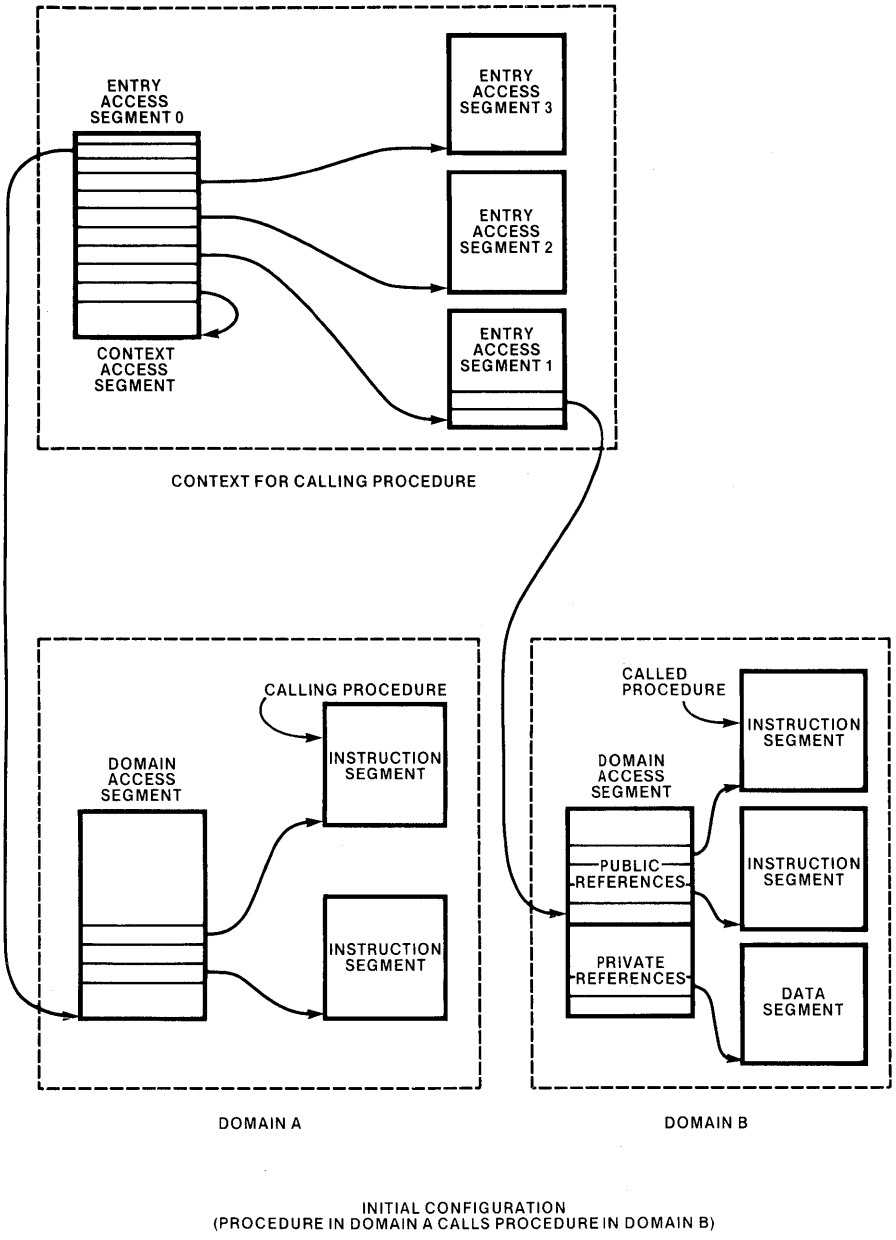


Figure 4-5. Calling a Context (1 of 2) 171821-26

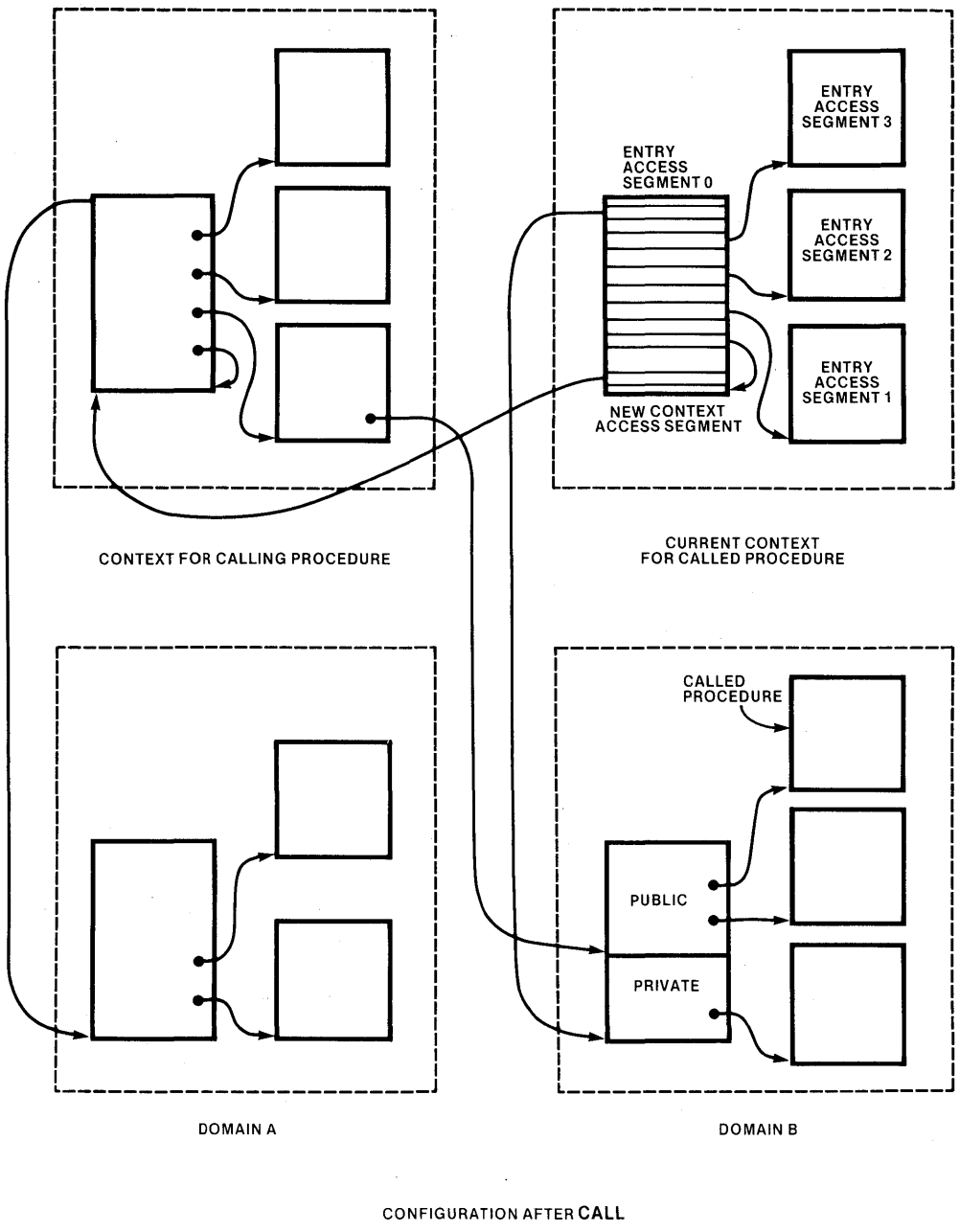


Figure 4-5. Calling a Context (2 of 2)

171821-27

Figure 4-6 displays the calling sequence for the inside representation in detail. The first drawing in figure 4-6 shows the situation that results after a call has been made to the "look up number" procedure in the telephone directory type manager. Notice that the telephone directory object is not in the immediate access environment of the procedure, because the directory is referenced only by the domain access segment, which is not one of the four entry access segments.

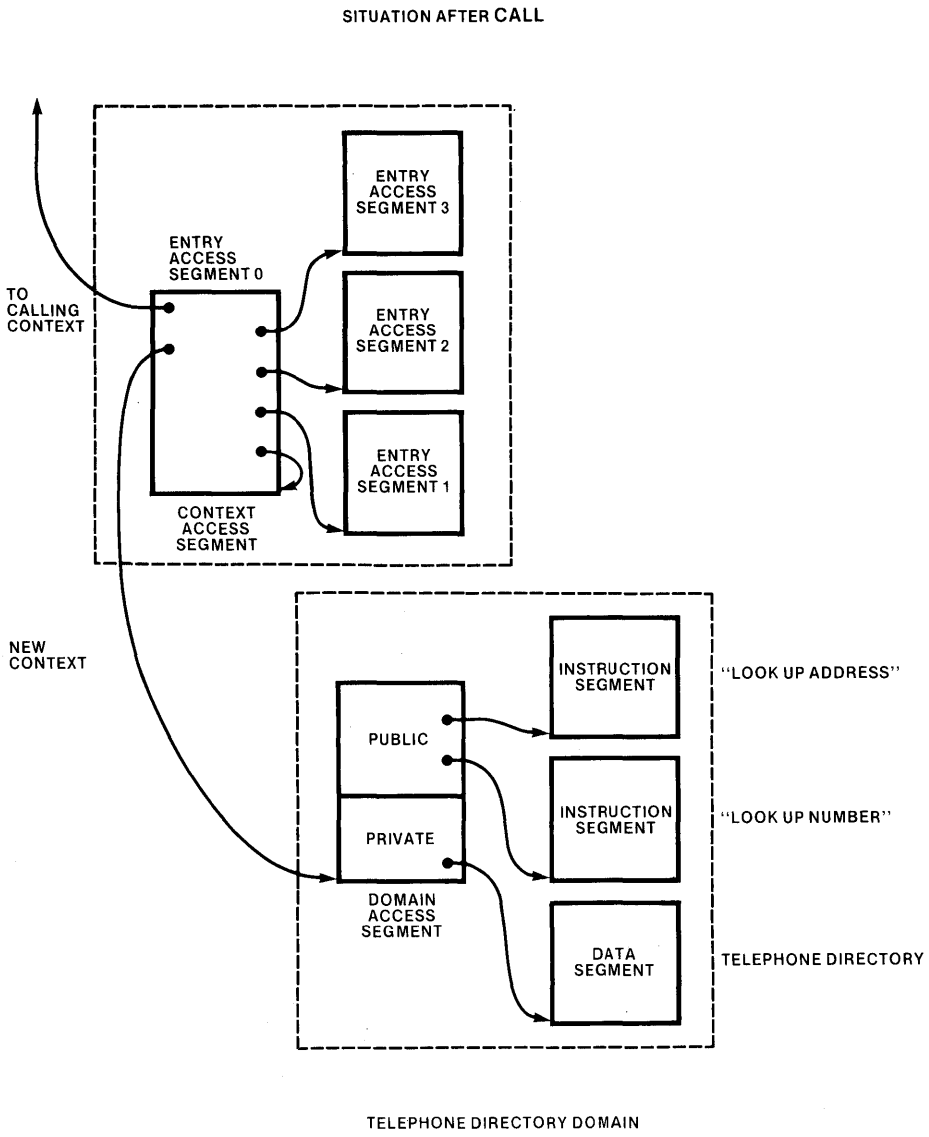


Figure 4-6. Changing the Access Environment (1 of 2) 171 821-28

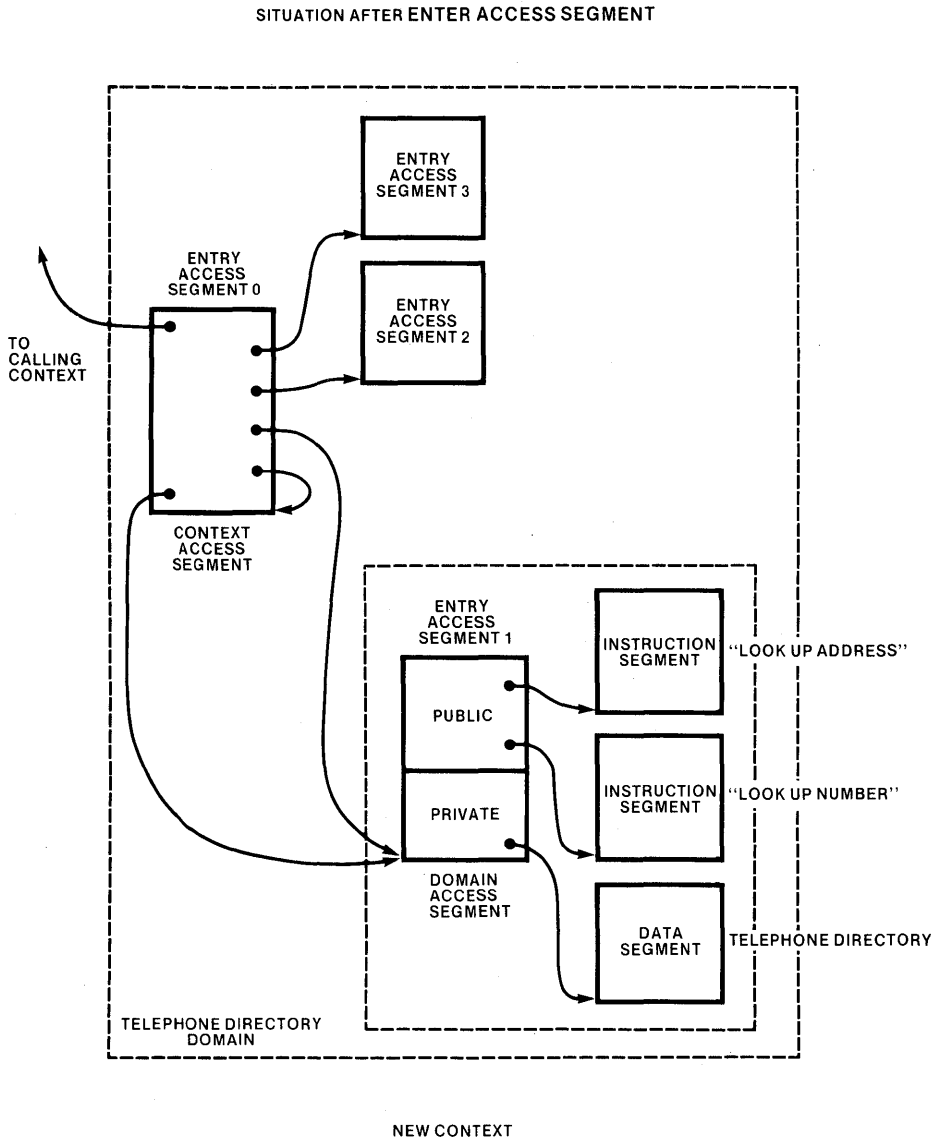


Figure 4-6. Changing the Access Environment (2 of 2) 171821-29

In order to bring the directory object into the access environment, the domain access segment must be made into one of the entry access segments. The iAPX 432 instruction set includes an instruction, ENTER ACCESS SEGMENT, which allows the user to make any access segment in the immediate access environment into an entry access segment. Since the domain access segment is referenced by entry access segment 0 (the context access segment itself), the domain access segment is in the access environment. The second drawing in figure 4-6 shows the result of executing an ENTER ACCESS SEGMENT instructon. The domain access segment has been made entry access segment 1, and the telephone directory object is now in the access environment. The procedure can now act on the object.

This example illustrates why the context access segment is one of the four entry access segments.

4.4.5 Using the Outside Representation of Type Managers

The alternative to the inside representation is for data to be kept outside the domain object; only procedures will be found inside. Whenever a procedure in the type manager is called, a reference to the data object must be passed into the domain. We will call this the *outside representation* of type managers.

The outside representation is actually more common than the inside representation, since the outside representation can be used with multiple instances of the same type of object. For example, the type managers that control system objects use the outside representation. There is only one type manager for each type of system object, but many different objects of one type can be controlled by the same type manager. The outside representation is also used for large data objects that must be passed to several type managers.

As an example of the outside representation, consider another telephone directory type manager. This type manager contains only the two procedures, look up number and look up address, and no data object. Whenever a procedure is called, a reference to some telephone directory object must be passed to the procedure. Figure 4-7 illustrates this example. The calling procedure uses the instruction CALL WITH MESSAGE, and sends the directory object as a message to the called procedure.

The problem with this simple use of the outside representation is that the data object has no protection. Any procedure with a reference for the object can manipulate it. Ideally, only procedures in the telephone directory type manager should be able to manipulate directory objects. System objects solve this problem through the mechanism of *type checking*; the only operators that are allowed to access a system object of some system type are the operators in one particular type manager. Each time an operation is attempted on a system object, the system type field in the corresponding object descriptor (see sections 2.4.2 and 2.4.4) is checked to see if the operation is allowed.

The iAPX 432 architecture provides a way to give a similar type checking mechanism to user objects.

4.4.6 User Defined Types

The iAPX 432 architecture includes several system objects and operations that can be used to create a protection mechanism for user-defined objects that is similar to the hardware protection mechanism provided for system objects by system type checking. This protection mechanism is called *user-defined type checking*.

The type definition object (TDO) is used to label user objects so they can have type checking performed on them, in much the same way that the system type fields in the object descriptors label system objects. All references to a typed object are made through a special kind of descriptor, called a *type descriptor*, in the object table (see section 2.4.4). This type descriptor points to both the typed object and the TDO.

The iAPX 432 instruction set includes several instructions for creating typed objects and for accessing them. Two varieties of types can be created, *public types* and *private types*. Public types provide no hardware protection; they merely serve to label user objects for the software. Private types, on the other hand, offer a protection mechanism. Successfully accessing a privately-typed object requires a reference to the TDO as well as a reference to the object itself. See Chapter 7 of the *iAPX 432 General Data Processor Architecture Reference Manual* for more information on how user typing is implemented.

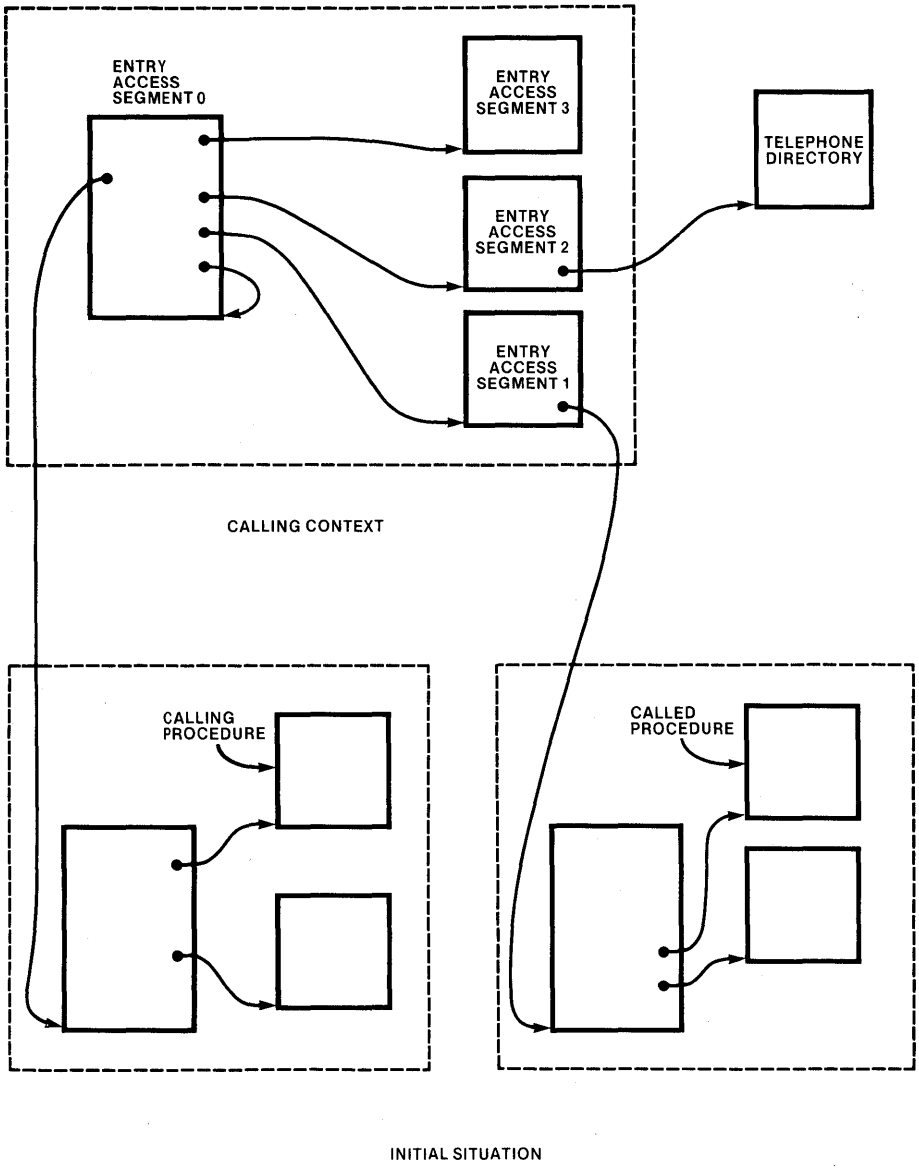


Figure 4-7. The Outside Representation (1 of 2) 171821-30

Figure 4-8 shows the same example as figure 4-7, but now the telephone directory object has been given a user-defined private type (shown by the TDO object). The domain has a matching TDO, so procedures in the domain can access the directory. No other procedures can access the directory. Notice that both a CALL CONTEXT WITH MESSAGE instruction and an ENTER ACCESS SEGMENT instruction are used in the example. The first instruction passes the typed object into the type manager. The second instruction puts the TDO object in the access environment, where it can be used in an access to the directory object.

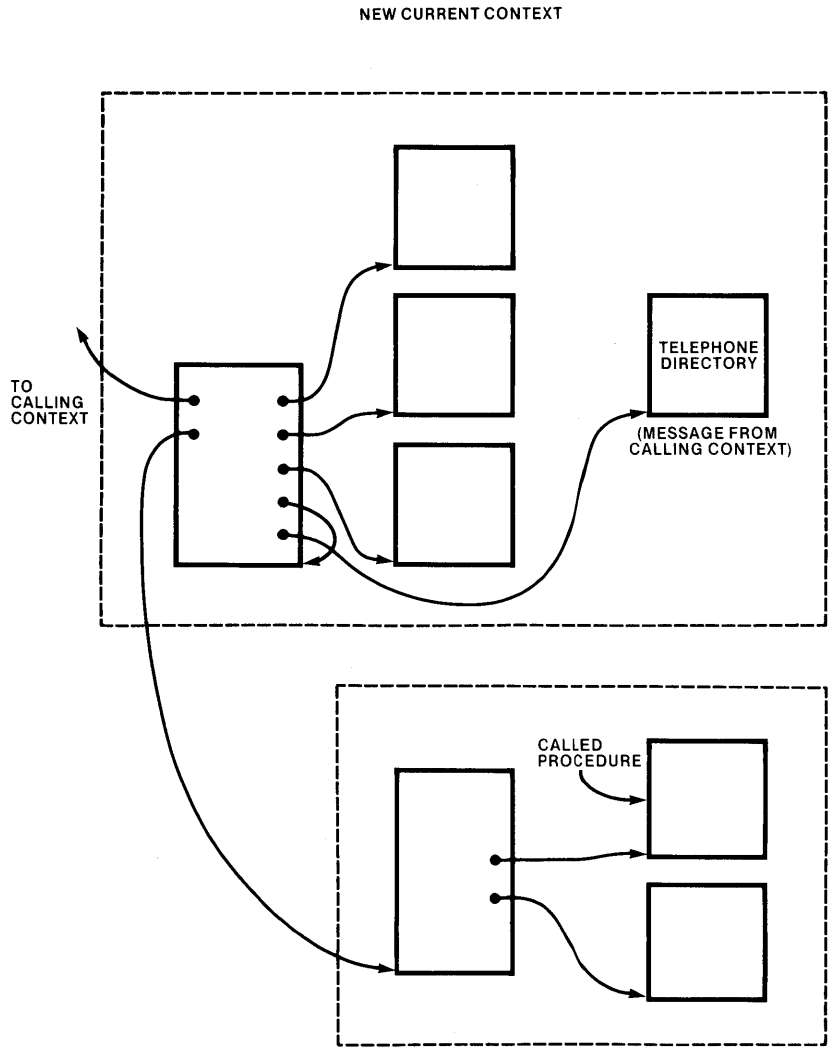


Figure 4-7. The Outside Representation (2 of 2) 171821-31

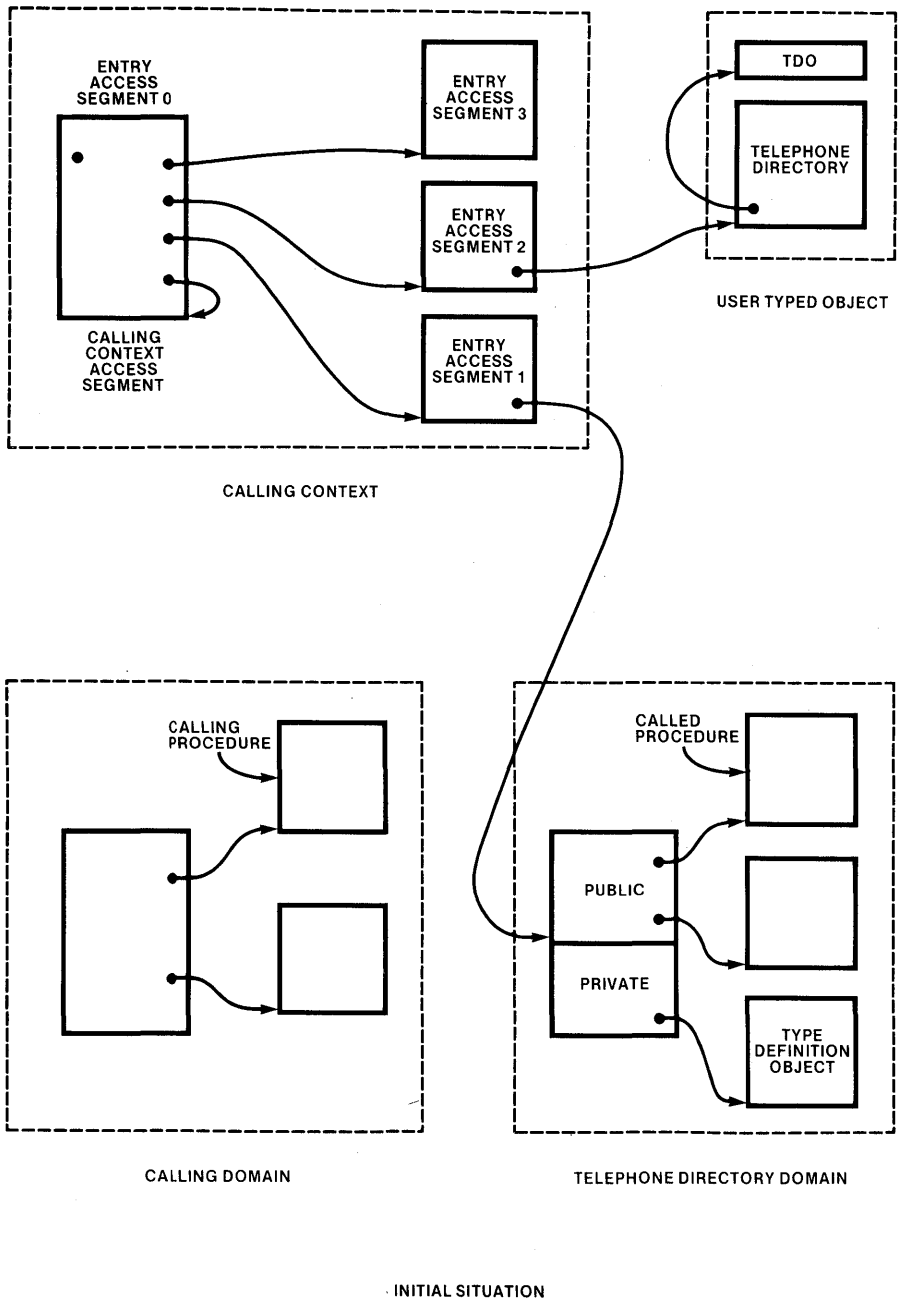
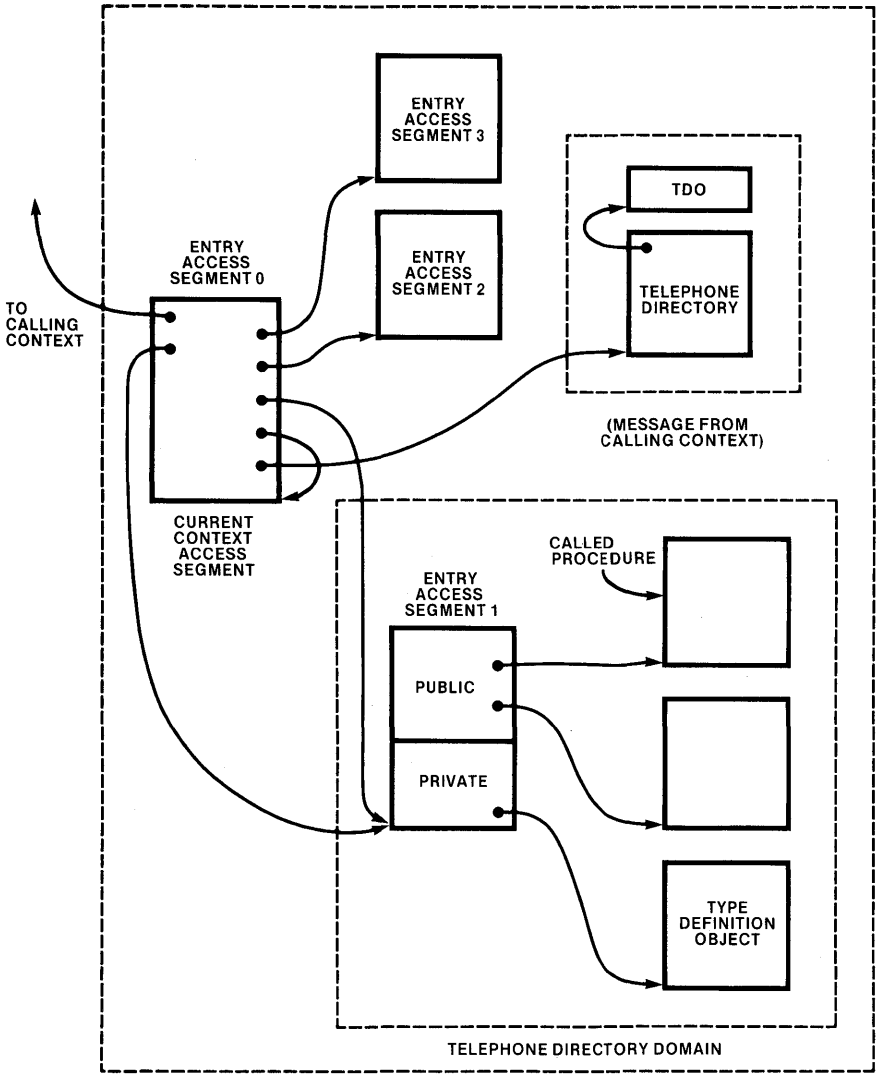


Figure 4-8. User Defined Type 171821-32



SITUATION AFTER CALL CONTEXT WITH MESSAGE AND ENTER ACCESS SEGMENT

Figure 4-8. User Defined Type (Cont'd.) 171821-33

4.5 Architectural Support for System Services: The Silicon OS

The iAPX 432 architecture implements a number of resource management mechanisms in the hardware. We call these hardware-supplied mechanisms the *Silicon Operating System*. (It is worth stressing again, as we did in section 4.2.3, that only the *mechanisms* are supplied in the hardware; the resource management policies are established by software.)

We can divide the Silicon OS into two unequal parts, one concerned with the management of *memory* resources, the other with the management of *processor* resources. By far the largest fraction is concerned with processor management. In the following sections we will explore the objects that implement dynamic storage allocation (one aspect of memory management) as well as process scheduling, dispatching and interprocess communication.

4.5.1 Process Objects and Processor Objects

A process is the unit of concurrent execution; it may also be defined as the unit of code that can be scheduled to run on a processor. The iAPX 432 architecture supports concurrent programming with several system objects: *process* objects for every process in the system, *processor* objects for every processor in the system, and *processor carrier* objects, which link processor objects to process objects. We will not describe the complete segment content of these objects, as we did for domains and contexts, but users can refer to the *iAPX 432 General Data Processor Architecture Reference Manual* for details.

We have now mentioned all the system objects that are needed for a minimal iAPX 432 system. Figure 4-9 shows the basic objects and the object references needed for a simple system. The system includes a processor object, a processor carrier object, a process object, a context object, and a domain object.

4.5.2 Storage Resource Objects

Storage Resource Object (SROs) are used to implement the dynamic storage allocation mechanisms (see section 2.4.3) of the Silicon Operating System. The SROs and the operations associated with them perform the actual binding of an unallocated descriptor in the object table to a newly created segment in memory. Several instructions, including CREATE ACCESS SEGMENT and CREATE DATA SEGMENT, require a reference to an SRO.

A special SRO associated with the process object is used implicitly whenever a call instruction is executed, since these instructions create a new context object, which must be allocated storage when it is created. Unless an SRO is used explicitly, it does not have to be included in the system. Therefore we have not included it in the simple system shown in figure 4-9.

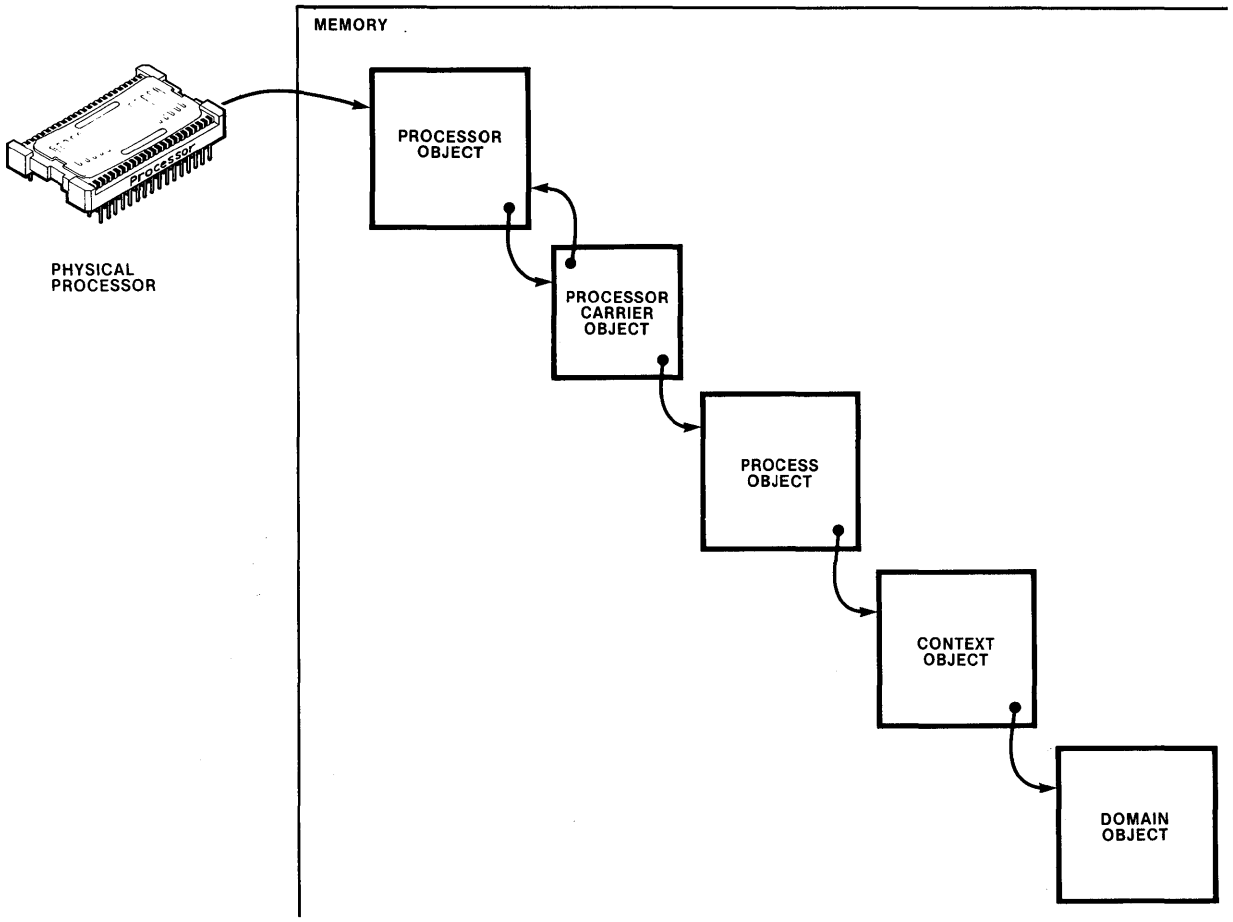


Figure 4-9. Basic System Objects 171821-34

4.5.3 Simple Interprocess Communication Without Blocking

In a multi-process environment, procedures in different processes often need to exchange information. Since the processes are asynchronous, the first one may be ready to communicate before or after the second is ready. Therefore, some kind of interprocess synchronization must be provided. In the iAPX 432 Silicon OS, a system object called a *communications port* provides the synchronization mechanism that allows asynchronous processes to communicate.

The information communicated is sent in a *message object*, which has no defined system type and in fact may be any object. For example, as we shall see in section 4.5.5, an entire process can be sent as a message.

When a procedure in one process wants to send a message to a procedure in another process, the sender specifies a message object and a communictions port as the destination of the message, then executes a SEND instruction. Similarly, the receiver specifies the same communications port as the source of a message and executes a RECEIVE instruction. Neither process need be aware of the other; neither process

knows, when it communicates with the port, if the other process is also ready to communicate. The port object contains a fixed-length buffer that holds a queue of object references for the message objects that have been sent to the port and are waiting for a receiver. As processes execute receive instructions on the port, messages are removed from the head of the queue.

Figure 4-10 shows simple examples of messages being sent and received via communications ports. Notice that the message buffer in the communications port is neither full nor empty, so both the send and receive instructions can be executed without difficulty. The message object being sent must be in the access environment of the sending context. When a message is received, its object reference is placed in the "message from calling context" slot in the context access segment.

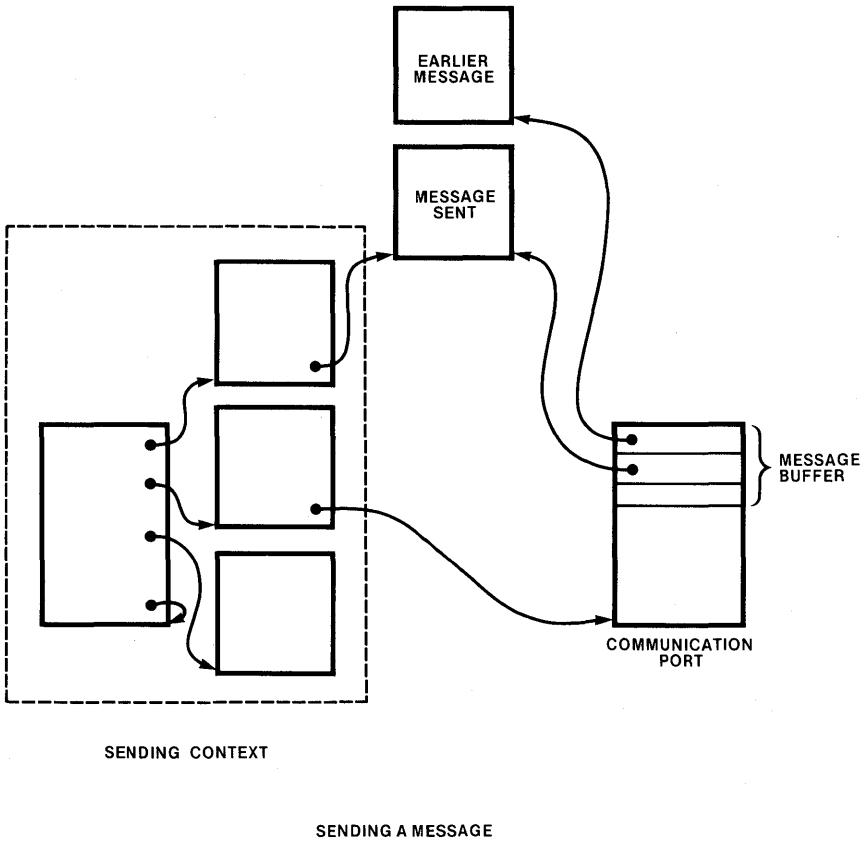


Figure 4-10. Sending and Receiving Messages 171821-35

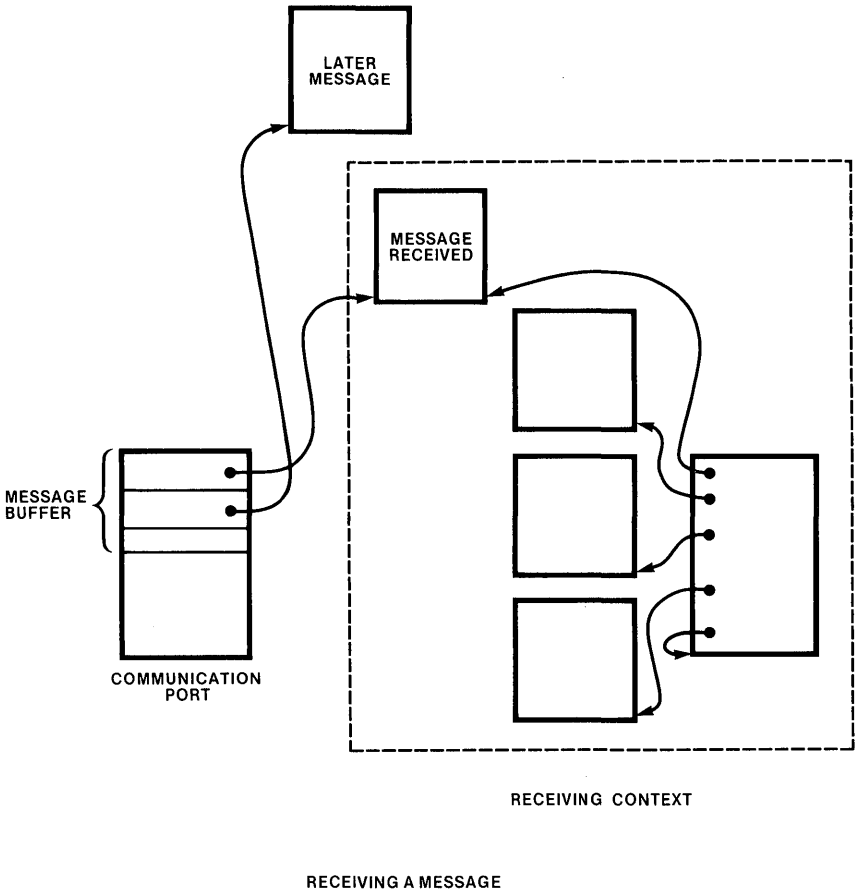


Figure 4-10. Sending and Receiving Messages (Cont'd.) 171821-36

4.5.4 Conditional and Surrogate Communication

As we shall see in the next section, when a process executes a simple SEND instruction on a port whose message buffer is full, or when it executes a simple RECEIVE on an empty port, the process is suspended while it waits for space in the buffer or for a message. Often, however, it will be unacceptable for a process to become suspended in this manner. The conditional and surrogate communication operators (CONDITIONAL SEND, CONDITIONAL RECEIVE, SURROGATE SEND, SURROGATE RECEIVE) can be used to avoid this suspension and also to implement more advanced forms of communication.

Conditional sends have exactly the same effect as simple sends, if the message buffer in the port is not full. Similarly, conditional receives are identical to simple receives, if the port has a message waiting. However, if these conditions are not satisfied, the conditional operations are not performed, whereas the simple operations are blocked and the process is suspended (see section 4.5.5). The conditional operations can be retried at a later time.

Surrogate sends and receives are more complicated than either simple or conditional operations. In surrogate operations, two ports are specified, along with a special system object called a *carrier*. When a SURROGATE SEND instruction is executed, for example, the message is sent to the first port, if the buffer is not full. If the buffer is full, the carrier object, containing a reference to the message, is enqueued

in a linked list at the port. When space becomes available in the buffer, the message object reference is inserted in the port, and the carrier is resent to the second port. Similarly, a SURROGATE RECEIVE can cause its carrier to become enqueued at a port waiting for a message. When the message is received, it and its carrier are resent to the second port. Figure 4-11 shows a queue of carriers and messages at a port.

Surrogate operations allow sophisticated communications mechanisms to be implemented. For example, priority communications channels can be implemented by specifying a priority-based port protocol and encoding priority information in the carrier object. Parallel communications channels can be multiplexed onto one port, in priority order, by executing a SURROGATE RECEIVE on several ports, but specifying the same second port in each case. A message sent to any of the several ports will be resent to the same second port. Thus, by executing a receive on the second port, one process can monitor several prioritized channels. (See Chapter 4 of the *iAPX 432 General Data Processor Architecture Reference Manual* for more information.)

4.5.5 Process Blocking, Scheduling, and Dispatching

When a process attempts to execute a simple SEND instruction on a communications port with a full buffer, or to execute a RECEIVE instruction on a communications port with an empty buffer, the process will become *blocked*. When a process is

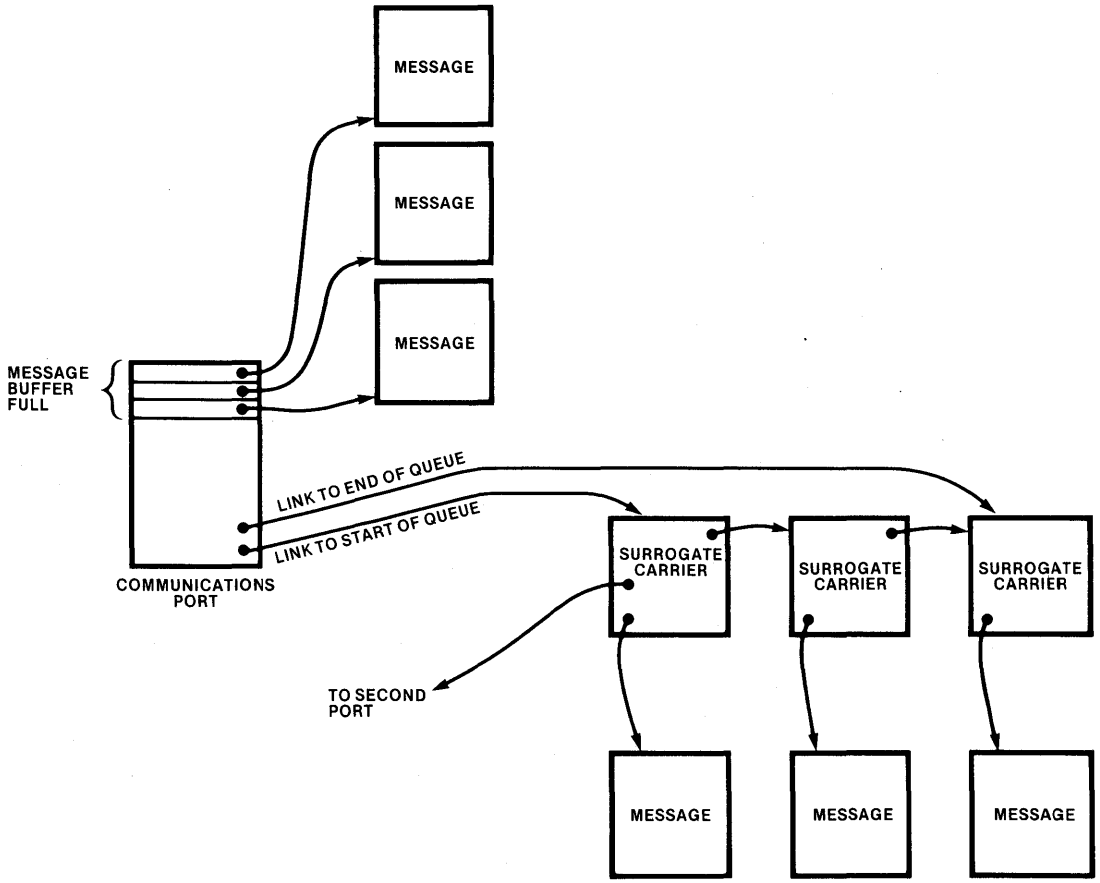


Figure 4-11. Surrogate Sending

171821-37

blocked it is enqueued at the communications port through its *process carrier*, a special carrier object associated with the process object itself. When the port buffer is available again, the process will be unblocked, and *resent* to another port object called a *dispatching port*. At the dispatching port the hardware automatically performs *scheduling* and *dispatching*.

Scheduling is the determination of the execution order of each process in a multiprocess system. Although the scheduling *policy* (e.g. round-robin or priority-based) is set by software, the *implementation* of the policy (the ordering mechanism itself) is performed by the iAPX 432 hardware. Dispatching is the assignment of a physical processor to execute a process.

Besides process objects, process carrier objects, and dispatching port objects, the hardware also uses processor objects and processor carrier objects (see section 4.5.1) to implement scheduling and dispatching. A dispatching port binds processor carriers to process objects just as a communications port binds processes to messages. (In fact, communications ports and dispatching ports are really just two slightly different varieties of a generalized port object.) When the dispatching port is created, the software specifies parameters that define the scheduling policy. (For example, one parameter defines the maximum period of time a process can run on a processor before rescheduling occurs.) The hardware executes this policy automatically for all available processes.

A process can become available for scheduling for a variety of reasons: it can be blocked while waiting to send or receive a message; it can use up its time slice on a processor; a fault can occur which causes the process to suspend itself; or it can deliberately suspend itself by executing a DELAY instruction.

Figure 4-12 shows the complete sequence of process blocking at a communications port, unblocking and resending to the dispatching port, scheduling (i.e. enqueueing at the dispatching port), and dispatching to an available processor.

Several dispatching ports can exist in a system (even a single-processor system). In this case, the ports usually implement different scheduling policies or else are dedicated to different functions. (One port might be dedicated to scheduling processes that have faulted, for example.)

4.5.6 Multiple-Processor Systems

The iAPX 432 architecture can support more than one active processor in the same system. Dispatching on a multiple processor iAPX 432 system becomes the assignment of a process to an available processor. The ability to add processors to a system and, without changing the software, improve the performance of a multi- process environment is one of the most important and powerful features of the iAPX 432.

The iAPX 432 architecture supports multiple processor dispatching as a simple extension of single-processor dispatching. Merely adding another processor object and processor carrier object for each additional physical processor enables the system to expand its power with no other changes required. Two processor carrier objects reference the same dispatching port, but otherwise the mechanism is the same. Multiple dispatching ports may be provided, as in the single-processor case.

Figure 4-13 shows a two-processor system running six processes. Two processes are executing and four are waiting at the dispatching port for a free processor.

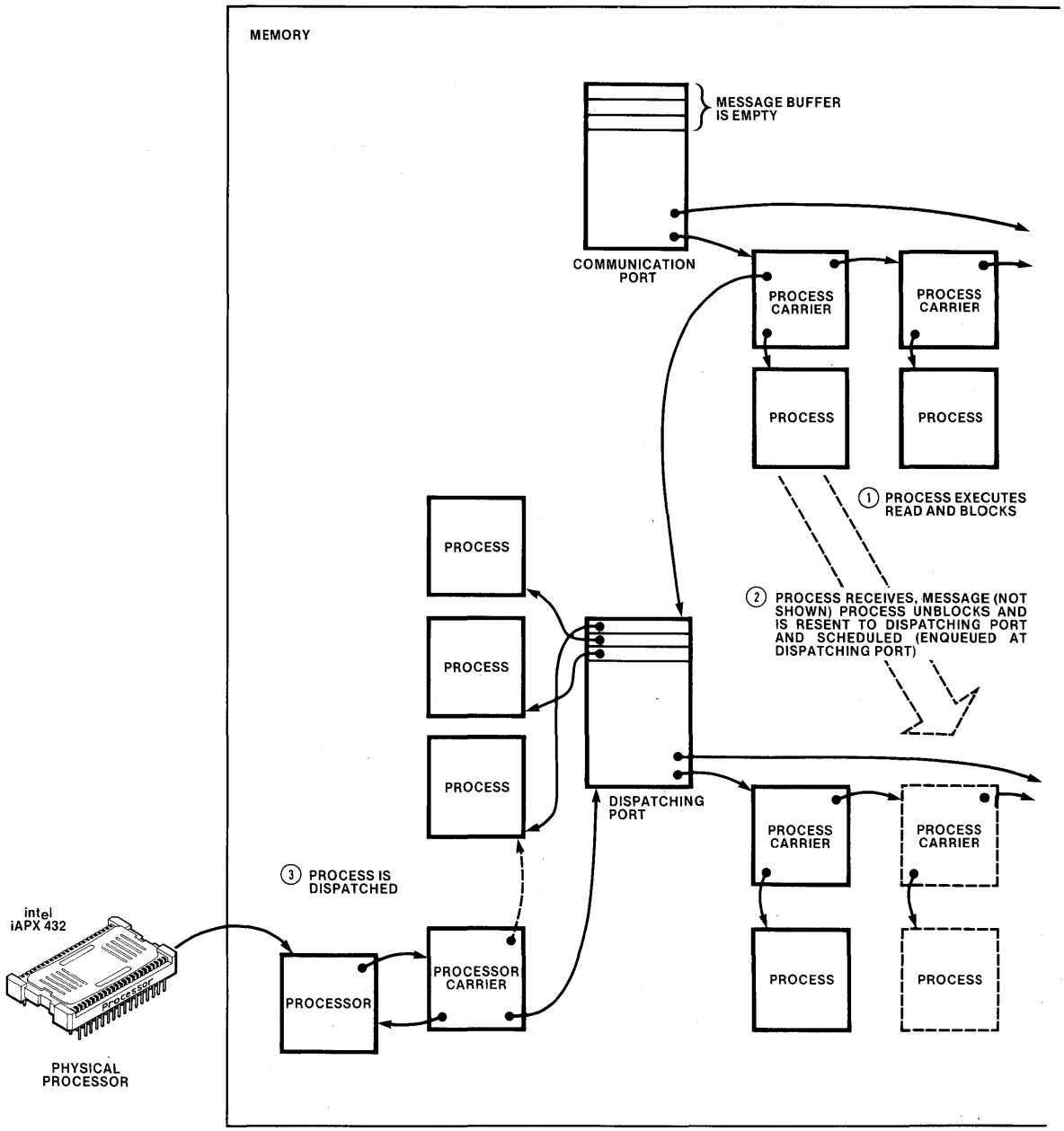


Figure 4-12. Resending, Scheduling, and Dispatching 171821-38

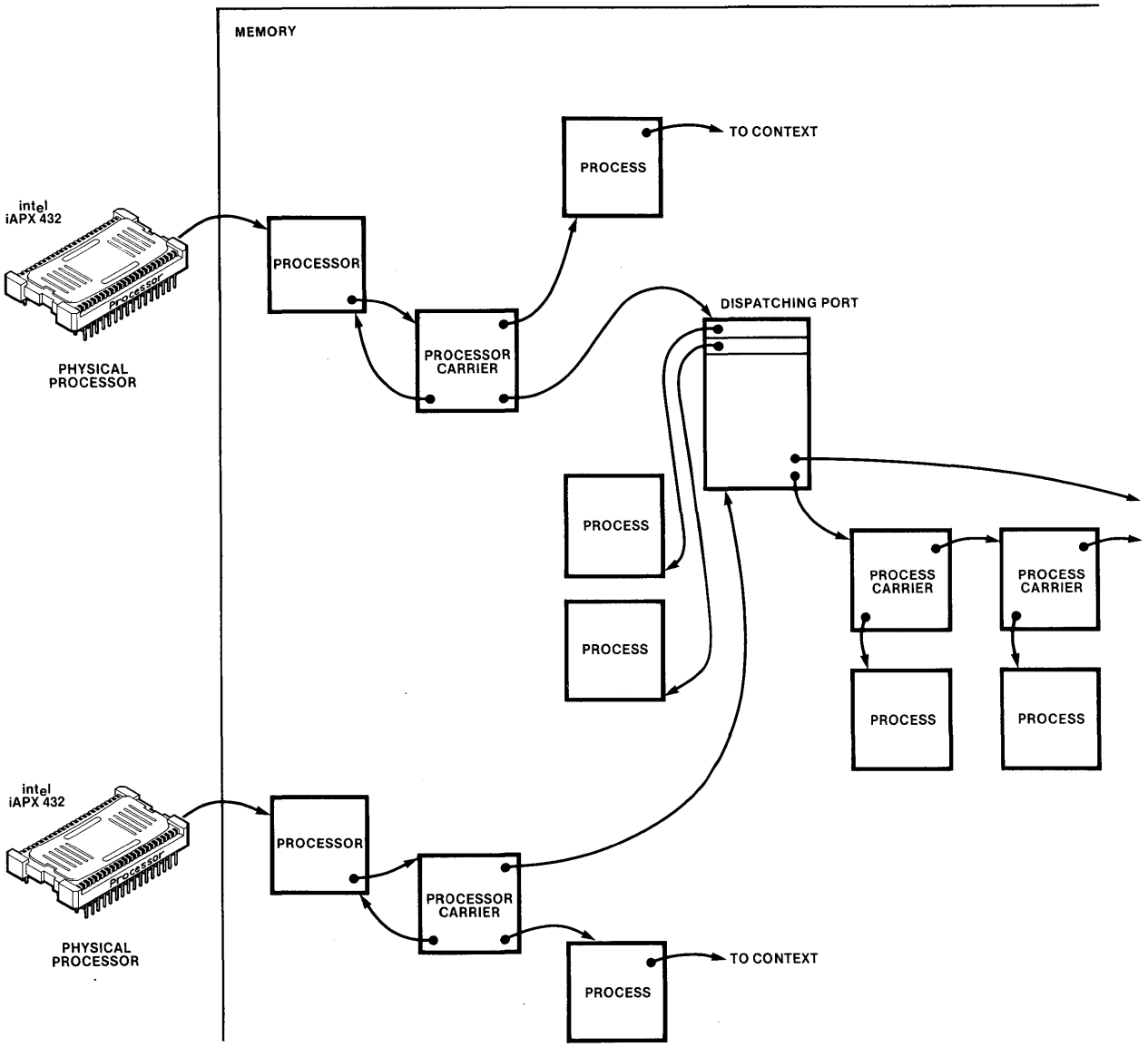


Figure 4-13. Multiple Processor System

171821-39

4.6 Summary

- The object-oriented design methodology has great promise for solving many of the current problems in software design and maintenance.
- This methodology is based on information hiding as the criterion for modularization.
- Type managers are the modules that result when the methodology is actually applied. They contain all procedures that directly manipulate objects of a given type.
- Domain objects and context objects are used to implement the static structure and dynamic behavior of type managers. The type definition object is used to give hardware protection to user objects.
- Multi-process operation can be supported by the object-based architecture of the iAPX 432, which also provdies a uniform approach to all system services (the Silicon Operating System).
- The Silicon Operating System provides a full set of hardware recognized system objects that provide support for concurrent processing (process objects, processor objects, dispatching ports, carriers), interprocess communication (communication ports), and dynamic storage allocation (storage resource objects).

CONCLUSIONS

The goal of reliable software will remain distant until computer architecture is significantly modernized to narrow the semantic gap between language concepts and hardware concepts. Given the high annual toll of people-hours lost to unreliable software, we architects have something bordering on a moral responsibility to modernize.

Peter J. Denning
Chairman of the Computer
Science Department,
Purdue University

Objects, type managers, concurrent processes, and communication ports may be new to the world of computers, but these concepts are very familiar in the real world of mailboxes, locks, keys, things, and events. For too long programmers have been forced to think in concepts that are tied closely to the hardware but are distant from the world of applications. Programmers usually find that object-oriented design using modern languages, such as Ada, quickly becomes second nature, because it is so much like the anthropomorphic way we all think. These modern languages and methodologies can make programming more of a human activity, and thus capable of being performed correctly by human beings.

But the new languages and methodologies have to be supported by the architecture in order to work effectively. Unfortunately, most architectures deal with concepts that are as far below the level of Ada as Fortran is below the real world. Glenford Myers called this conceptual incompatibility between modern languages and conventional architectures the "semantic gap," a gap which the iAPX 432 architecture has bridged.

The conceptual gap between Ada and the iAPX 432 architecture is very small, because the designers of the iAPX 432 architecture followed the same object-oriented design methodology that users should follow when writing Ada programs. The same methodology is also used in iMAX, the Multifunction Applications Executive that Intel provides with Ada to handle the interface between user programs and the Silicon OS. The object methodology is thus found at every level of the iAPX 432 system.

REQUEST FOR READER'S COMMENTS

Intel Corporation attempts to provide documents that meet the needs of all Intel product users. This form lets you participate directly in the documentation process.

Please restrict your comments to the usability, accuracy, readability, organization, and completeness of this document.

1. Please specify by page any errors you found in this manual.

2. Does the document cover the information you expected or required? Please make suggestions for improvement.

3. Is this the right type of document for your needs? Is it at the right level? What other types of documents are needed?

4. Did you have any difficulty understanding descriptions or wording? Where?

5. Please rate this document on a scale of 1 to 10 with 10 being the best rating. _____

NAME _____ DATE _____

TITLE _____

COMPANY NAME/DEPARTMENT _____

ADDRESS _____

CITY _____ STATE _____ ZIP CODE _____

Please check here if you require a written reply. ☐

WE'D LIKE YOUR COMMENTS . . .

This document is one of a series describing Intel products. Your comments on the back of this form will help us produce better manuals. Each reply will be carefully reviewed by the responsible person. All comments and suggestions become the property of Intel Corporation.

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 79 BEAVERTON, OR

POSTAGE WILL BE PAID BY ADDRESSEE

Intel Corporation
SSO Technical Publications Dept.
3585 SW 198th Ave.
Aloha, OR 97007

AL3-2-485

INTEL CORPORATION, 3585 S.W. 198th Avenue, Aloha, Oregon 97007 • (503) 681-8080

Printed in U.S.A./Y81/5K/0881/AP